Applied
Microsystems
Corporation

XICE Debugger Supplement for Motorola 68000, 68EC000, 68HC000 and 68302 Processors and the EL1600 Emulator

May 1993
P/N 922-17320-02

Contents

Chapter 3
XICE Tutorial

Chapter 4
Using Breakpoints and the Event System

Chapter 5
Using Performance Analysis

Chapter 6
Using the Time Stamp Module

Chapter 7
Simulated I/O

Chapter 1
XICE for 68000/EC000/HC000 and 68302

XICE is Applied Microsystems' integrated debugger for use with Applied in-circuit emulators. It is a part of a complete embedded development toolchain. Other toolchain components include the XRAY simulator, XRAY monitor, MCC and CCC cross-compilers, and ASM cross-assembler, linker, and object module librarian.

This manual provides information that is specific to using XICE with Applied emulators for the Motorola 68000/68HC000/68EC000 and 68302 processors. It supplements the descriptions of core debugger commands found in the *XRAY68K Documentation Set.*

Note

Two emulator hardware configurations support the 68000 processor. One consists of an emulation board, probe module and probe tip; the other consists of an emulation board and probe tip. The emulation board and probe tip configuration is also used by the 68HC000 and 68EC000 emulators. See Chapter 1 of your *Hardware Setup and Reference Guide* to identify the configuration of your 68000. The two configurations have minor differences in operation. Unless otherwise noted, all references to 68000 in this supplement refer to both versions of 68000 and to 68HC000 and 68EC000.

This version of XICE is based on the latest version of Microtec Research XRAY68K. It fully supports the output of the 4.3 MCC 68K compiler and 6.9 ASM68K assembler. Although it may function properly with earlier versions of MCC and ASM, Applied Microsystems does not guarantee full backwards compatibility.

Overview

This supplement provides the information you need to use XICE with an EL 1600 emulator for a Motorola 68000/ 68HC000/68EC000 or 68302 microprocessor.

- Chapter 1 highlights several key concepts, including register support, using trace, operations during run, and software performance analysis. The "Command considerations" section describes exceptions and any XRAY command that is not fully supported in XICE.
- Chapter 2 provides an alphabetical reference of XICE commands.
- Chapter 3 provides a set of mini-tutorials on code preparation, emulation, and XICE setup and use.
- Chapter 4 covers using the standard breakpoints and the optional advanced event system.
- Chapter 5 explains the performance analysis capabilities of XICE.
- Chapter 6 explains installation and use of the Timestamp module.
- Chapter 7 describes simulated I/O functions.

For detailed information on running under X windows on Sun workstations see Appendix A of your *XRAY68K User's Guide.*

For detailed information on starting the debugger see the *XICE Installation Guide.*

Conventions

This manual uses the following conventions:

When you see	This means
bold type	The name of a control software configuration or executable file, a keyword or command, or a key that you should press.
italics	A variable, or a file name. Sometimes italics are used for emphasis the first time a key word or concept is introduced.
<F7>	Press the F7 function key.
[*option*]	Optional item. You do not have to select the option. You do not enter the brackets.
{*this* | *that*}	You must choose either this or that. The vertical bar means you have a choice. The curly braces indicate that you may only choose one item. Do not enter the braces.
{*item*}...	You may select one or more of the items.
`run`	Screen output or example code.

Support services

Applied Microsystems provides a full range of support services. New software is covered by a 90-day warranty that includes full applications phone support. Additional support agreements are available to extend the initial warranty and to provide additional services.

If you encounter trouble installing or using your software, consult your manuals to verify that you are using appropriate procedures. See also the *Appendix B, Troubleshooting*, in the *XICE Installation Guide*. It covers the most frequently encountered operational problems.

If the problem persists, call Customer Support at 1-800-ASK-4AMC or 1-206-882-2000 (in Washington State). Customers outside the United States should contact their sales representative or local Applied Microsystems office.

Have your software serial number available. This number displays during boot-up and is also included on the label of your master media.

XICE screens

Figure 1-1 and Figure 1-2 show C-source-level and assembly-level XICE screens.

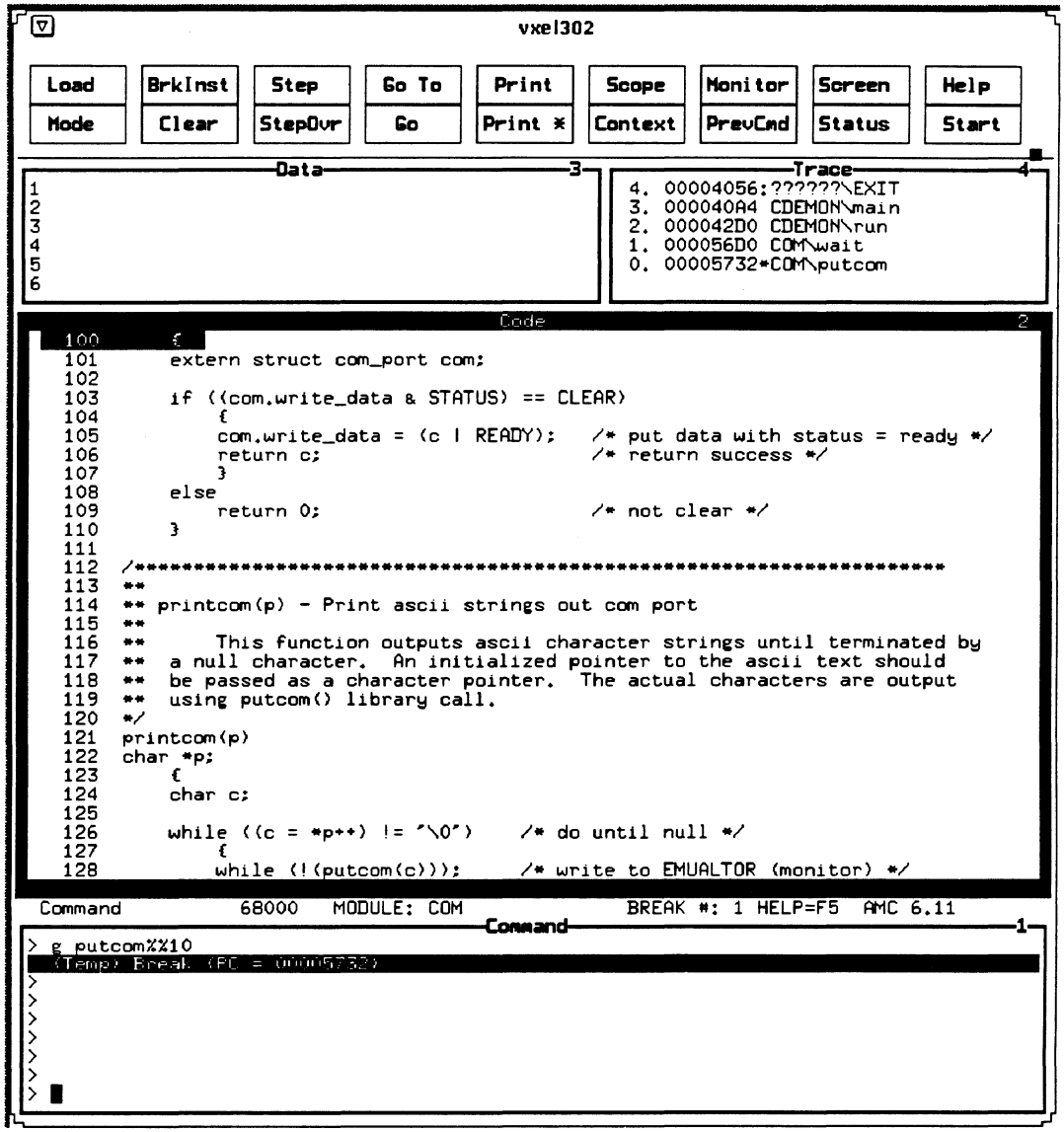

Figure 1-1 XICE source-level screen

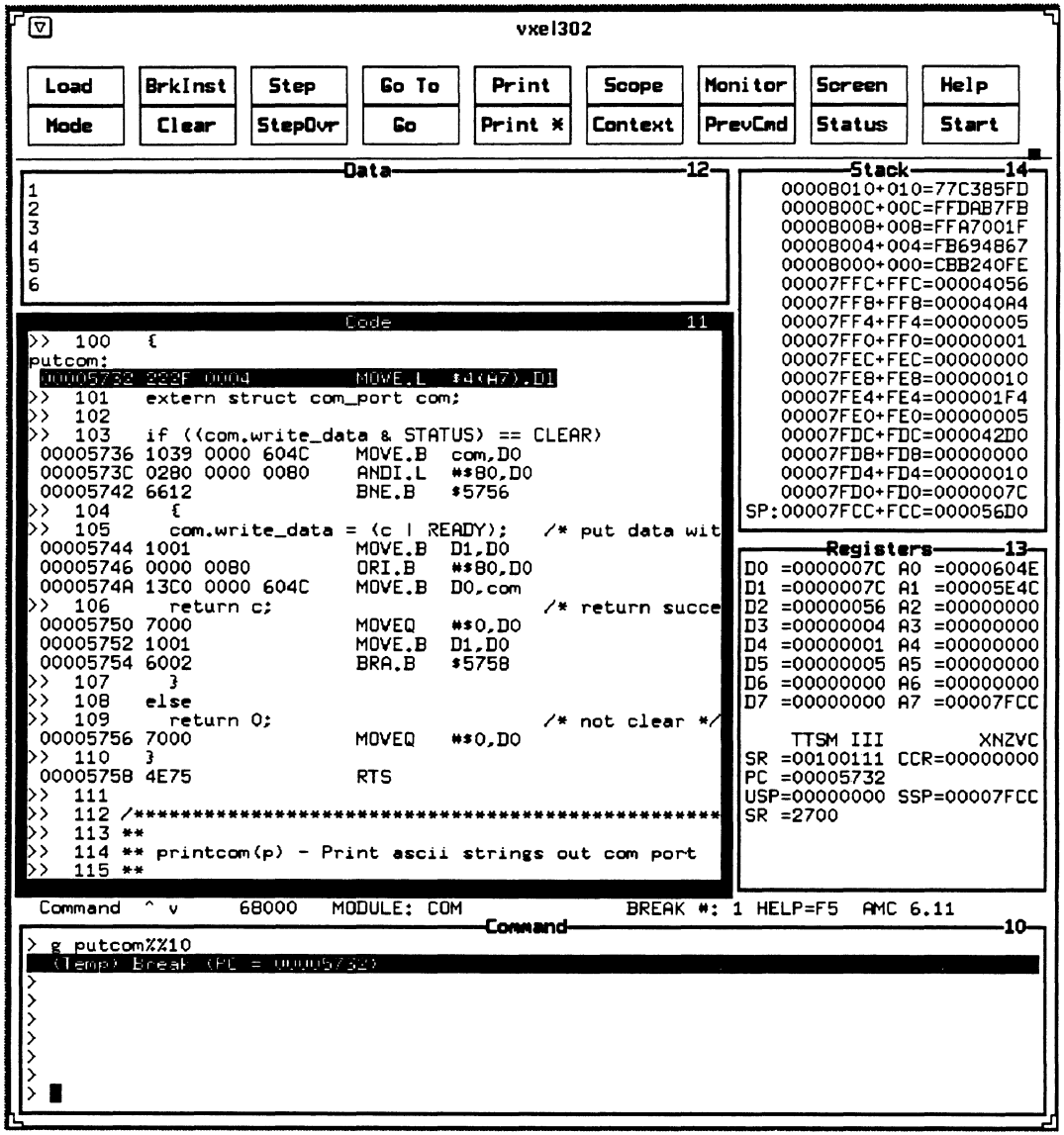

Figure 1-2 XICE assembly-level screen

Command considerations

Chapter 3 of the reference manual in the *XRAY Documentation Set* describes the core set of commands for both XRAY and XICE. The information that follows covers any XICE exceptions to the information in the reference manual.

BREAKACCESS	Cannot be used when event system triggers are armed. Number limited when BREAKINSTRUCTION also used. See "Basic breakpoints" in Chapter 3 for an overview.
BREAK-COMPLEX	Variable EVTMODE in xice.cfg must be ON. Restricts event system to one armed trigger. See Chapter 2.
BREAK-INSTRUCTION	Requires assignment of trap vector and scratch space. See EXVEC and SCRATCH in Chapter 2 and "Instruction Breakpoints" in Chapter 4 for an explanation.
BREAKREAD	Cannot be used when event system triggers are armed. Number limited when BREAKINSTRUCTION also used. See "Basic breakpoints" in Chapter 3 for an overview.
BREAKWRITE	Cannot be used when event system triggers are armed. Number limited when BREAKINSTRUCTION also used. See "Basic breakpoints" in Chapter 3 for an overview.
CLOCK	Not supported in XICE.
CPU	Not supported in XICE.
HOST	Causes erratic behavior on some PCs.
ICE	Not supported in XICE.
INPORT	Chapter 7 describes the use of INPORT for simulated I/O.
INTERRUPT	Not supported in XICE.

NOICE	Not supported in XICE.
NOINTERRUPT	Not supported in XICE.
NOME	Memory is mapped in blocks of 2K. If you start or end your mapping request at points that are not at multiples of 2K, XICE adjusts the request to meet the 2K requirement.
OPTION	The options CPU and SPEED are not supported. The option VPCOLOR is supported only for IBM-PCs and compatibles.
OUTPORT	Chapter 7 describes the use of OUTPORT for simulated I/O.
OVERLAY	Not supported by the 68000 or 68302 emulator.
RAM	Memory is mapped in blocks of 2K. If you start or end your mapping request at points that are not at multiples of 2K, XICE adjusts the request to meet the 2K requirement.
RESTORE	Not supported in XICE.
ROM	Memory is mapped in blocks of 2K. If you start or end your mapping request at points that are not at multiples of 2K, XICE adjusts the request to meet the 2K requirement.
SAVE	Not supported in XICE.
SEARCH	Not supported on XICE.
SETSTATUS	The following identifiers are not supported in XICE: QUALIFY and TRACE.
STATUS	The following identifiers are not supported in XICE: QUALIFY and TRACE.
TRACE	Not supported in XICE.

Register support

The following are the 68000 and 68302 registers that are supported by XICE. You can enter them in either upper case or lower case, but they are displayed in the case shown.

Register	Name
A0-A7	Address Registers
D0-D7	Data Registers
FP	
PC	Program Counter
SP	
SSP	Supervisor Stack Pointer
SR	Status Register
USP	User Stack Pointer

The following additional registers, which are related to just the 68302, are also supported by XICE:

Register	Name
BAR	Base Address Register
BCR	Byte Count Register
BR0	Base Register 0
BR1	Base Register 1
BR2	Base Register 2
BR3	Base Register 3
CMR	Channel Mode Register
CR	Command Register

Register	Name
CSR	Channel Status Register
DAPR	Destination Address Pointer Register
DSR1	SCC1 Data Sync Register
DSR2	SCC2 Data Sync Register
DSR3	SCC3 Data Sync Register
FCR	Function Code Register
GIMR	Global Interrupt Mode Register
IMR	Interrupt Mask Register
IPR	Interrupt Pending Register
ISR	Interrupt In-Service Register
OR0	Option Register 0
OR1	Option Register 1
OR2	Option Register 2
OR3	Option Register 3
PACNT	Port A Control Register
PADAT	Port A Data Register
PADDR	Port A Data Direction Register
PBCNT	Port B Control Register
PBDAT	Port B Data Register
PBDDR	Port B Data Direction Register
SAPR	Source Address Pointer Register
SCCE1	SCC1 Event Register
SCCE2	SCC2 Event Register
SCCE3	SCC3 Event Register

Register	Name
SCCM1	SCC1 Mask Register
SCCM2	SCC2 Mask Register
SCCM3	SCC3 Mask Register
SCCS1	SCC1 Status Register
SCCS2	SCC2 Status Register
SCCS3	SCC3 Status Register
SCM1	SCC1 Mode Register
SCM2	SCC2 Mode Register
SCM3	SCC3 Mode Register
SCON1	SCC1 Configuration Register
SCON2	SCC2 Configuration Register
SCON3	SCC3 Configuration Register
SCR	System Control Register
SIMASK	Serial Interface Mask Register
SIMODE	Serial Interface Mode Register
SPMD	SCP, SMC Mode and Clock Control Register
TCN1	Timer Counter Register
TCN2	Timer Counter Register
TCR1	Timer Capture Register
TCR2	Timer Capture Register
TER1	Timer Event Register
TER2	Timer Event Register
TMR1	Timer Mode Register

Register	Name
TMR2	Timer Mode Register
TRR1	Timer Reference Register
TRR2	Timer Reference Register
WCN	Watchdog Counter
WRR	Watchdog Reference Register

Chip select registers (68302 only)

You must configure the 68302 chip select registers to match the target system before starting up XICE. The EL 1600 requires that the chip select registers be programmed to respond in all function code spaces to permit target memory operations to work correctly. Memory operations (also referred to as peeks and pokes) include actions such as displaying memory, downloading code and data, loading the reset vectors, and fill and block moves.

You can configure the chip select registers in the **xice.cfg** file. The *XICE Installation Guide* describes how to preset registers using **xice.cfg**. The *El 1600 Hardware Setup and Reference Guide* also provides information on setting the chip select registers.

It is important to set up the chip select registers correctly; otherwise, it is possible to inadvertently program a DTACK over the memory location of emulator internal memory. Then, if the processor supplies a DTACK for emulator internal memory, an emulator crash can occur.

Once the chip select registers are loaded initially, the switches RIRR and UIR determine when they are restored or updated. RIRR set to ON restores the registers after an emulator reset. UIR set to ON updates the registers after an emulation break occurs or whenever you make a change to any of the register values.

If you do not set RIRR to ON, the emulator will not reload the register values after a reset.

If you do not leave UIR set to ON, the emulator's copy of the internal registers will not be updated until the next RUN to PAUSE transition.

Using trace

The XICE trace capability allows you to view either raw bus cycles or disassembled source code interleaved with the high-level code.

In addition to the softswitches, the following commands set up trace:

PPT	controls the tracing of peeks and pokes
TCEBRK	controls the tracing of breakpoints
TED	controls whether external or internal DMA is traced
TID	controls whether external or internal DMA is traced
TRCFRAME	sets the trace cycle numbers
TRCINT	specifies how traced timestamp information is displayed
TRCMODE	sets trace mode (assembly and source, assembly only, or source only)
TSTAMP	controls whether timestamp or LSA is traced
TUNITS	sets the timestamp units

The commands to use trace are as follows:

DRT	display raw trace
DT	display trace, both assembly and disassembled source level
DTB	display trace backwards, both assembly and disassembled source level
DTF	display trace forwards, both assembly and disassembled source level
TRCCLR	clear trace buffer
TRSRCH	search for specified pattern in trace

The commands to set up and use trace are described in Chapter 2 of this supplement. When you use a trace command, the trace information is displayed in the command viewport. Raw trace is formatted as shown in Figure 1-3.

```
}          }          }          }          }          FLAGS X-break¶
MEM      R-read     MEM        B-byte     MEM        T-target          B-BERR¶
ACCESS:  W-write    BOUNDARY:  W-word     LOCATION:  O-overlay         M-illegal¶
                                                                        mem access
FRAME  ADDRESS  DATA  IPL  FCn  MEM  DMA IRC  FLAGS  --LSA BITS-------¶
¶
   16  0010F8  E548  000  SP   RW0                   11111111 11111111¶
   15  0010FA  D041  000  SP   RW0                   11111111 11111111¶
   14  0010FC  E548  000  SP   RW0                   11111111 11111111¶
   13  0010FE  207C  000  SP   RW0                   11111111 11111111¶
   12  001100  0000  000  SP   RW0                   11111111 11111111¶
   11  001102  1310  000  SP   RW0                   11111111 11111111¶
   10  001104  2F30  000  SP   RW0                   11111111 11111111¶
    9  001106  0000  000  SP   RW0                   11111111 11111111¶
    8  001108  4EB9  000  SP   RW0                   11111111 11111111¶
    7  001310  0000  000  SD   RW0                   11111111 11111111¶
    6  001312  0001  000  SD   RW0                   11111111 11111111¶
    5  00110A  0000  000  SP   RW0                   11111111 11111111¶
    4  000FE2  0001  000  SD   WW0                   11111111 11111111¶
    3  000FE0  0000  000  SD   WW0                   11111111 11111111¶
    2  00110C  1188  000  SP   RW0                   11111111 11111111¶
    1                                 BREAK¶
```

Figure 1-3 Raw trace format

Raw trace display

The raw trace display columns shown in Figure 1-3 are:

FRAME	The decimal count of the line in the trace buffer. Line 0 corresponds to the most recently traced cycle.
ADDRESS	The hex value of the address bus.
DATA	The hex value of the data bus.
IPL	This column lists the bustime information if the BUSTIME is set to ON. If BUSTIME is set to OFF, this column lists the current interrupt level. The range for the bustime information is 3 to 9. Raw trace displays a plus mark (+) if bustime is outside of this range.
FCn	The memory space accessed: UD user data space UP user program space SD supervisor data space SP supervisor program space PU CPU space 000 reserved memory space 011 reserved memory space 100 reserved memory space
MEM	The memory type accessed, its boundary, and its location: R read B byte T target W write W word O overlay For example, WBT indicates a byte wide write to target memory and RWO means a word wide read from overlay.
VPA(68000)	State of valid peripheral access pins.

DMA (68302)	A direct memory access. For the 68302, this information indicates internal DMA cycles if TID is set to ON, or external DMA cycles if TED is set to ON. (TED and TID relate to the 68302 only.) If both TID and TED are set to ON, trace will show the DMA cycles for both TED and TID but there will be no way to determine which was a result of internal DMA and which was a result of external DMA.
VMA(68000)	State of valid memory access pins.
IAC (68302)	CPU internal access pin.
FLAGS	Flags set: X cycles for which the emulator break bit is asserted B bus error M illegal memory access
LSA BITS	LSA BITS displays the state of each pin of the LSA during that bus cycle. This column does not appear if you set TSTAMP to ON but is replaced by timestamp information.
TIMESTAMP	The timestamp information is recorded as the interval between successive bus cycles, if TRCINT is set to INTERVAL, or relative to the bus cycle number specified by the command TRCFRAME, if TRCINT is set to OFFSET. This column does not appear if you set TSTAMP to OFF but is replaced by LSA information.

Disassembled trace display

Figure 1-4 shows the disassembled trace format. The information that follows this figure defines the information in each field.

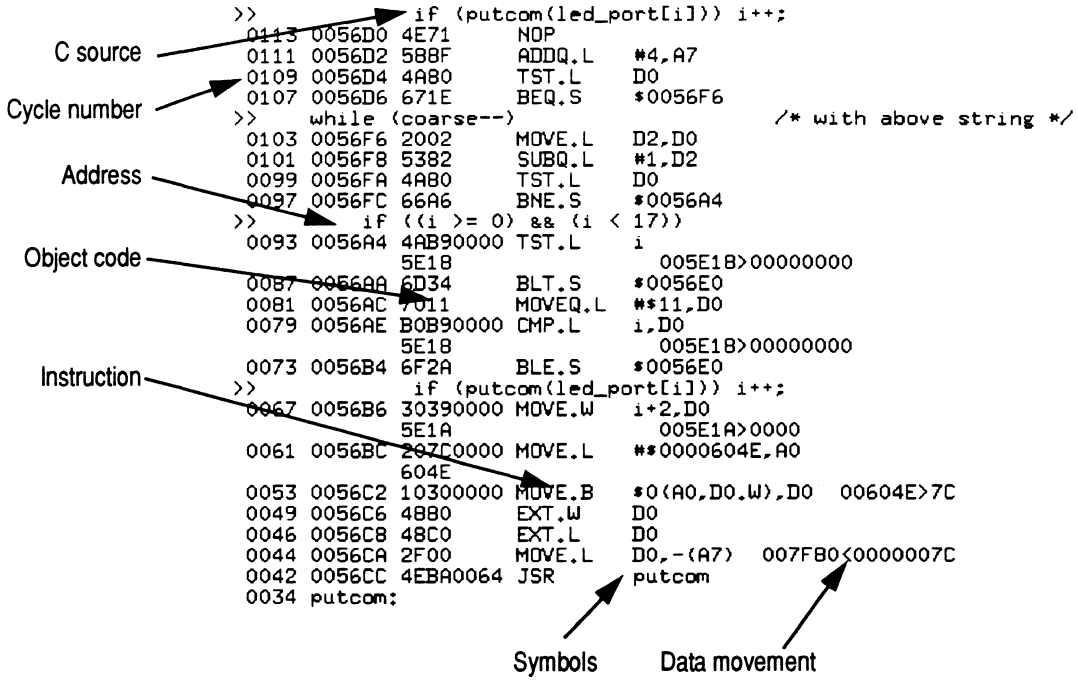

Figure 1-4 Disassembled trace format

C source	If trcmode is set to BOTH, C source is interleaved into disassembly.
Cycle number	An index of the bus cycle in the trace buffer. The most recently traced cycle is 1. This number corresponds to the frame number in the raw trace display.
Address	Address of instruction in memory.

Object code	Numeric representation of assembly code.
Instruction	Assembly language instruction in text form.
Symbols	English text representation of addresses, operands and data.
Data movements	The data cycles that occurred as a result of the instruction: • address < data data written to address • data > address data read from address

Operations during run

In normal operating mode, XICE does not permit additional operations while the emulator is running. The DRUN, DSTOP, and DUPDATE commands provide this additional functionality.

The DRUN (dynamic run) command executes the target program and continues execution until it is stopped by DSTOP, a breakpoint, an error, or a halt instruction. The purpose of this mode is to allow you to interact with the emulator and debugger dynamically, while the emulator is running. In DRUN you can examine and qualify trace, set and change events and breakpoints, examine and change memory and perform most other interactive functions with the emulator.

The DUPDATE command allows you to specify how frequently the emulator is polled and the viewports updated during DRUN (dynamic run) mode. However, dynamic commands are no longer accepted.

Chapter 2 describes each command in detail. Chapter 3 includes a short tutorial.

Software performance analysis

The performance analysis features of XICE for the EL1600 68000 and 68302 development system allow you to:

- determine which areas of a program use the most CPU time
- identify bottlenecks in time critical applications
- monitor the effects of programming changes made to improve throughput.

Using statistical performance analysis, these features sample instruction fetch bus cycles at regular intervals using the event system; determine what function was active when a sample was taken; keep a tally of the number of samples falling within each function; and report the sample information. The report is in a user-specifiable format.

Chapter 5 provides an overview of performance analysis.

File formats and converters

Although XICE requires IEEE695 object format to enable symbolic debugging, support for other formats is built into XICE, and additional converters are available. See the descriptions in Chapter 2 for UPL, DNL, UPLFMT, and DNLFMT for supported non-IEEE695 formats, procedures, and limitations. Contact your Applied Microsystems representative for information about additional converters.

Chapter 2
XICE Command Supplement

Core debugger commands are covered in the *XRAY Reference Manual*. This chapter lists commands that are specific to XICE and are not listed in the *XRAY Reference Manual* or that are XICE-specific implementations of core XRAY commands. These commands are entered in the command viewport at the XICE prompt. Any combination of upper-case and lower-case letters can be used in commands.

ASM — single line assembler

Works with

■ EL 1600 ❐ CodeTAP

Syntax

```
ASM [address]
```

Description

The ASM command allows you to enter assembly code. If an address is included, assembly will begin at that point. If no address is included assembly will begin at the last assembly address. A carriage return with no assembly instruction terminates the assembler.

If loaded, symbols can be used in instructions, for example

```
call main
```

The prompt for the single line assembler is the address followed by a colon, e.g.,

```
00000000:
```

Notes

The line assembler does not accept symbols that are not already in the symbol table.

In single-line assembler mode, you can change assembly locations with the ORG command. For example,

```
org 100
```

will change the assembly location to 100.

Example

```
-ASM 0X100
 00000100: move.l (A7), A0
 00000102: add.l #6, D0
 00000108: move.l #0, (A0)
 0000010e:
```

BPSPACE— specify breakpoint space

Works with

■ EL 1600 ❒ CodeTAP

Syntax

```
BPSPACE [USER|SUPERVISOR|ANY]
```

Description

This command allows you to explicitly specify the space for breakpoints. The choices are USER, SUPERVISOR, or ANY.

The default is ANY. If no argument is given, the state of the switch is displayed.

Notes

You can also specify a default for BPSPACE in XICE.CFG with SW_BPSPACE.

Example

```
BPSPACE ANY
```

Related Commands

XICEVARS

BREAKACCESS—sets an access breakpoint

Works with

■ EL 1600 ❐ CodeTAP

Abbreviation

BA

Syntax

```
BREAKACCESS [address | address_range] [;macro_name()]
```

Description

address	Specifies the address of the breakpoint.
address_range	Specifies the address range for the breakpoint. A breakpoint will be set at each statement within the address range.
macro_name()	Specifies a macro to be executed when the breakpoint is encountered. Note that the macro name must include a set of parentheses: macro1(). When a macro is executed, the macro controls whether program execution stops or continues.

The **BREAKACCESS** command sets an access breakpoint at the specified memory location(s). An access breakpoint halts program execution each time the target program attempts to read from or write to the specified memory location(s). Memory locations can contain code or data.

Use an ampersand (&) to reference symbolic addresses rather than just the symbol names. Using a symbol name alone returns its value, not the address.

The debugger performs the following functions when an access breakpoint is encountered:

1. Completes the execution of the instruction at that location.
2. Suspends program execution.
3. Executes a macro (if one was specified when the breakpoint was set). Depending on the macro, the debugger will do one of the following:
4. If the macro return value is true (nonzero), the debugger resumes execution at the instruction immediately after the break.
5. If the macro return value is false (zero), the debugger returns to command mode and displays break information.
6. If a macro was not specified, the debugger returns to command mode and displays break information.

XICE automatically assigns a breakpoint number to each breakpoint; this number is used to reference or clear the breakpoint. The Break viewport displays breakpoint numbers, breakpoint locations, breakpoint types, and breakpoint command arguments. In high-level mode, the line number and module name containing the breakpoint are also displayed. The breakpoint type for the **BREAKACCESS** command is **ACCESS**.

Breakpoints can be removed with the **CLEAR** command.

If the **BREAKACCESS** command is specified without parameters, the debugger displays the Break viewport.

Notes

Access breakpoints are set to break on a read (**br**), a write (**bw**), or a read or write (**ba**) of the breakpoint address. These breakpoints are implemented using emulator hardware and may be set in RAM or ROM.

You cannot set an access breakpoint when any event system triggers are armed. Disarming the triggers will allow you to set access breakpoints. Likewise, setting access breakpoints causes the event system to be disabled. Clearing the access breakpoints will allow you to arm the event system triggers.

With instruction breakpoints, the break occurs before the instruction at the specified address is executed. With access breakpoints, the break begins on the cycle in which the access occurs but may continue or "skid" several cycles after access of the breakpoint location.

Cannot be used when event system triggers are armed. Number limited when BREAKINSTRUCTION also used. See "Basic breakpoints" in Chapter 3 for an overview.

Examples

Example	Meaning
BA @sieve\\flags	Sets a breakpoint at the address of the variable array `flags` in the root named @sieve.
BA flags..flags+10	Sets breakpoints starting at the address of the array `flags` and ending `10` bytes after the address of `flags`.
BA &flags[0]	Sets a breakpoint at the address of the array element `flags[0]`.

BA #17;readprime()	Sets a breakpoint at line `17` and executes the macro `readprime` when the breakpoint is encountered.
BA prime	Sets a breakpoint at the address referred to by the value in variable `prime`. This command is correct if `prime` is a pointer. The breakpoint is set at the location specified by the variable prime. For example, if the value of prime is 0x0123, a breakpoint is set at the address 0x0123. This command may not be correct if prime is a scalar, since the value in prime is treated as an address and the breakpoint is set at that address rather than at the address of the variable prime.
BA &prime	Sets a breakpoint at the address of the variable `prime` regardless of its type. This command is correct if `prime` is a scalar; it sets a breakpoint at the address of the variable `prime`.

	If `prime` is a pointer, the breakpoint is set at the address of the pointer rather than at the address it is pointing to (i.e., prime).
BA 20	Sets a breakpoint at address `20`.
BA flags..flags+9;when (*flags == 1)	Sets breakpoints starting at the address of the array `flags` and ending `9` bytes after the address of `flags`, but the predefined `when` macro stops debugger execution when the first element of flags is equal to `1`.

Related Commands

BREAKINSTRUCTION, BREAKREAD, BREAKWRITE, CLEAR, GO, GOSTEP, STEP, STEPOVER

BREAKCOMPLEX — ties a macro to an event system break

Works with

■ EL 1600 ❐ CodeTAP

Syntax

```
BC trig{n}[;macro_name]
```

Description

BC is used to associate a macro with an event system breakpoint. Complex breakpoints are used to halt program execution and then execute the specified macro. The argument *trig{n}* refers to a trigger that you must set using the command TRIG following setting up the BC command. If a macro is specified, it is executed each time a break is encountered. Execution continues if the macro returns non-zero.

BC (with an attached macro) works the same way as the other breakpoint commands (BREAKACCESS, BREAKREAD, BREAKWRITE, and BREAKINSTRUCTION).

The debugger performs the following functions when a complex breakpoint is encountered:

1. Completes the execution of the instruction at that location.
2. Suspends program execution.
3. Executes a macro (if one was specified when the breakpoint was set). Depending on the macro, the debugger will do one of the following:
4. If the macro return value is true (non-zero), the debugger resumes execution at the instruction immediately after the break.
5. If the macro return value is false (zero), the debugger returns to command mode and displays break information.

6. If a macro was not specified, the debugger returns to command mode and displays break information.

All breakpoints are automatically assigned a breakpoint number by the debugger, which is used to reference or clear the breakpoint. The break viewport displays breakpoint numbers, breakpoint locations, breakpoint types, and breakpoint command arguments. In high-level mode, the line number and module name containing the breakpoint are also displayed.

Breakpoints are removed with the CLEAR command.

Notes

You must set the variable EVTMODE to ON to use BC.

While EVTMODE is set to ON, XICE is restricted to only one event system breakpoint armed at a time even though STAT TRIG may display other triggers as armed.

Examples

`bc trig(2); done()`	Executes the macro done each time trigger 2 is true. Breaks execution when done returns a zero.

Related Commands

BREAKACCESS, BREAKINSTRUCTION, BREAKREAD, BREAKWRITE, CLEAR, EV, EVTMODE, TRIG

BREAKINSTRUCTION — sets an instruction breakpoint

Works with

■ EL 1600 ❐ CodeTAP

Abbreviation

```
B
```

Syntax

```
BREAKINSTRUCTION [address | address_range] [;macro_name]
```

Description

address	Specifies the address of the breakpoint.
address_range	Specifies the address range for the breakpoint. If you specify a range of instruction breakpoints, they will be set on the first byte of each instruction or (for high-level code) the first instruction of each line.
macro_name	Specifies a macro to be executed when the breakpoint is encountered. Note that the macro name must include a set of parentheses: macro1(). When a macro is executed, the macro controls whether program execution stops or continues.

The **BREAKINSTRUCTION** command sets an instruction breakpoint at the specified memory location(s). An instruction breakpoint halts program execution each time the target program attempts to execute an instruction at the specified memory location(s).

The debugger performs the following functions when an instruction breakpoint is encountered:

1. Suspends program execution before the instruction at the breakpoint address is executed.
2. Executes a macro (if one was specified when the breakpoint was set). Depending on the macro, the debugger will do one of the following:
3. If the macro return value is true (nonzero), the debugger resumes execution starting at the instruction where the break occurred and displays break information.
4. If the macro return value is false (zero), the debugger returns to command mode without executing the instruction where the break occurred.
5. If a macro was not specified, the debugger returns to command mode without executing the instruction where the breakpoint was encountered.

XICE automatically assigns a breakpoint number to each breakpoint; this number is used to reference or clear the breakpoint. The Break viewport displays breakpoint numbers, breakpoint locations, breakpoint types, and breakpoint command arguments. In high-level mode, the line number and module name containing the breakpoint are also displayed.

The breakpoint type for the **BREAKINSTRUCTION** command is **INST**.

Breakpoints can be removed with the **CLEAR** command.

If the **BREAKINSTRUCTION** command is specified without parameters, the debugger displays the Break viewport.

Notes

See Chapter 4 for more detailed explanation of breakpoint use.

Execution breakpoints consume event system resources and affect what is possible using other features. In general, the emulator manages these resources and warns you when it makes adjustments and presents an error when resources are exhausted or when you attempt something that creates a conflict. So you need not concern yourself with more than the following general guidelines.

- If you use them in addition to the event system, note that instruction breakpoints consume an address and a status resource in each event group. Limit event system address/ status resource consumption to no more than one address and one status comparator in each group. Set up the event and trigger statements, but leave them disarmed until you are ready to use them.
- You can specify an address range. XICE handles ranges by breaking them into multiple individual single-point breakpoints. Thus, if you specify that a breakpoint should be for a range of 20 addresses, you may set only 12 additional breakpoints.
- If you use them with access breakpoints, note the limitations explained in the BREAKACCESS command description.
- When XICE is invoked, it performs a read of the area designated for SCRATCH if SW_SCRATCH is specified in *xice.cfg*. If SCRATCH is set to an area of memory that does not return a DTACK at the end of the read, it will hang XICE. In such a case, comment out the default address for SCRATCH (0x9ff0) in the *xice.cfg* file. Then specify the address for SCRATCH before setting any breakpoints by using the SCRATCH command or preferably in an include file when you invoke XICE.

Examples

Example	Meaning
B #20	Sets a breakpoint at line number 20.
B 2210h..2216h	Sets breakpoints starting at address 2210 and ending at address 2216 (hexadecimal), assembly-level mode only.
BREAKI #1..#4	Sets breakpoints starting at line number 1 and ending at line number 4.
B #15..#18;FOO()	Sets breakpoints starting at line number 15 and ending at line number 18. Executes macro FOO after each line.
B SIEVE\#28	Sets a breakpoint at line number 28 in the module SIEVE.
B #10;when(i==3)	Sets a breakpoint at line number 10 and stops only if variable i is equal to 3.
B 0x93	Sets a breakpoint at address 93 (hexadecimal), assembly-level mode only.

Related Commands

BREAKACCESS, BREAKREAD, BREAKWRITE, CLEAR, GO, GOSTEP, STEP, STEPOVER

BREAKREAD — sets a read breakpoint

Works with

■ EL 1600 ❒ CodeTAP

Abbreviation

BR

Syntax

BREAKREAD [*address* | *address_range*] [;*macro_name*]

Description

address	Specifies the address of the breakpoint.
address_range	Specifies the address range for the breakpoint. A breakpoint will be set at each statement within the address range.
macro_name	Specifies a macro to be executed when the breakpoint is encountered. Note that the macro name must include a set of parentheses: macro1(). When a macro is executed, the macro controls whether program execution stops or continues.

The **BREAKREAD** command sets a read breakpoint at the specified memory location(s). A read breakpoint halts program execution each time the target program attempts to read data from the specified memory location(s).

Use an ampersand (&) to reference symbolic addresses rather than just the symbol names. Using a symbol name alone returns its value, not the address.

The debugger performs the following functions when a read breakpoint is encountered:

1. Completes the execution of the instruction at that location.
2. Suspends program execution.
3. Executes a macro (if one was specified when the breakpoint was set). Depending on the macro, the debugger will do one of the following:
4. If the macro return value is true (nonzero), the debugger resumes execution at the instruction immediately after the breakpoint.
5. If the macro return value is false (zero), the debugger returns to command mode and displays breakpoint information.
6. If a macro was not specified, the debugger returns to command mode and displays updated breakpoint information.

XICE automatically assigns a breakpoint number to each breakpoint; this number is used to reference or clear the breakpoint. The Break viewport displays breakpoint numbers, breakpoint locations, breakpoint types, and breakpoint command arguments. In high-level mode, the line number and module name containing the breakpoint are also displayed. The breakpoint type for the **BREAKREAD** command is **READ**.

Breakpoints can be removed with the **CLEAR** command.

If the **BREAKREAD** command is specified without parameters, the debugger displays the Break viewport.

Notes

See BREAKACCESS for restrictions.

Examples

Example	Meaning
BR 0x300	Sets a breakpoint at address `300` (hexadecimal).
BR @sieve\\flags	Sets a breakpoint at the address of the variable array `flags` in the root named @sieve.
BR flags..flags+10	Sets breakpoints starting at the address of the array `flags` and ending `10` bytes after the address of `flags`.
BR 20h..30h;FOO()	Sets breakpoints from address `20h` (hexadecimal) to `30h` and executes the macro `FOO` on every breakpoint between these addresses.
BR &flags[0]	Sets a breakpoint at the address of array element `flags[0]`.
BR prime	Sets a breakpoint at the address referred to by the value in variable `prime`. This command is correct if `prime` is a pointer. The breakpoint is set at the location of the variable `prime`. For example, if the value of `prime` is 0x0123, a breakpoint is set at the address 0x0123.

This command may not be correct if prime is a scalar, since the value in prime is treated as an address and the breakpoint is set at that address rather than at the address of the variable prime.

BR &prime

Sets a breakpoint at the address of the variable `prime` regardless of its type.

This command is correct if `prime` is a scalar; it sets a breakpoint at the address of the variable `prime`.

If `prime` is a pointer, the breakpoint is set at the address of the pointer rather than at the address it is pointing to (i.e., prime).

BR &count; when (k<30)

Sets a breakpoint at the address of count and only stops when the value of k is less than `30`.

Related Commands

BREAKACCESS, BREAKINSTRUCTION, BREAKWRITE, CLEAR, GO, GOSTEP, STEP, STEPOVER

BREAKWRITE — Sets a Write Breakpoint

Works with

■ EL 1600 ❐ CodeTAP

Abbreviation

BW

Syntax

BREAKWRITE [*address* | *address_range*] [;*macro_name*]

Description

address	Specifies the address of the breakpoint.
address_range	Specifies the address range for the breakpoint. A breakpoint will be set at each statement within the address range.
macro_name	Specifies a macro to be executed when the breakpoint is encountered. Note that the macro name must include a set of parentheses: macro1(). When a macro is executed, the macro controls whether program execution stops or continues.

The **BREAKWRITE** command sets a write breakpoint at the specified memory location(s). A write breakpoint halts program execution each time the target program attempts to write data to the specified memory location(s).

The debugger performs the functions listed below when a write breakpoint is encountered:

1. Completes the execution of the instruction at that location.
2. Suspends program execution.
3. Executes a macro if one was specified when the breakpoint was set. Depending on the macro, the debugger will do one of the following:
4. If the macro return value is true (nonzero), the debugger resumes execution at the instruction immediately after the breakpoint.
5. If the macro return value is false (zero), the debugger returns to command mode and displays breakpoint information.
6. If a macro was not specified, the debugger returns to command mode and displays viewport information and breakpoint information.

XICE automatically assigns a breakpoint number to each breakpoint; this number is used to reference or clear the breakpoint. The Break viewport displays breakpoint numbers, breakpoint locations, breakpoint types, and breakpoint command arguments. In high-level mode, the line number and module containing the breakpoint are also displayed. The breakpoint type for the **BREAKWRITE** command is **WRITE**.

Breakpoints can be removed with the **CLEAR** command.

If the **BREAKWRITE** command is specified without parameters, the debugger displays the Break viewport.

Notes

See BREAKACCESS for restrictions.

Related Commands

BREAKACCESS, BREAKINSTRUCTION, BREAKREAD, CLEAR, GO, GOSTEP, STEP, STEPOVER

Examples

Example	Meaning
BW @sieve\\flags	Sets a breakpoint at the address of the variable array `flags` in the root named @sieve.
BW flags..flags+10	Sets breakpoints starting at the address of the array `flags` and ending `10` bytes after the address of `flags`.
BW 0x100;FOO()	Sets a breakpoint at address `0x100` and executes the macro `FOO` on the break.
BW &flags[0]	Sets a breakpoint at the address of array element `flags[0]`.
BW prime	Sets a breakpoint at the address referred to by the value in variable `prime`. This command is correct if prime is a pointer. The breakpoint is set at the location specified by the variable prime. For example, if the value of prime is 0x0123, a breakpoint is set at the address 0x0123. This command may not be correct if prime is a scalar, since the value in prime is treated as an address and the breakpoint is set at that address rather than at the address of the variable prime.

BW &prime	Sets a breakpoint at the address of the variable `prime` regardless of its type. This command is correct if `prime` is a scalar; it sets a breakpoint at the address of the variable `prime`. If `prime` is a pointer, the breakpoint is set at the address of the pointer rather than at the address it is pointing to (i.e., prime).
BREAKW 100h	Sets a breakpoint at address `100h` (hexadecimal).

BROWSE — display class inheritance information

Works with

■ EL 1600 ❐ CodeTAP

Syntax

BROWSE *SYMBOL_NAME*

Description

This command displays the inheritance relationships of a C++ class. It shows the base classes (parents) and the derived classes (children) of the given class.

The symbol name that you specify may be the name of a class, object, or class member.

Notes

Appendix G of the *XRAY Reference Manual* covers C++ features.

Example

```
BROWSE COUT
```

BTE — enable or disable emulator bus timeout

Works with

■ EL 1600 ❐ CodeTAP

Syntax

```
BTE [ON|OFF]
```

Description

This switch controls the emulator bus timeout. The valid arguments are ON and OFF.

When set to ON, during RUN mode the EL 1600 will automatically time out in the length of time specified by the switch FTO if the target fails to supply a DTACK signal. It performs the timeout by sending a DTACK signal to the CPU and then executing a break. This ensures that the emulator will not hang after an invalid memory location has been accessed.

When set to OFF, the EL 1600 waits for the target to supply a DTACK signal, and may wait indefinitely.

The default for BTE is OFF. If no argument is given, the state of the switch is displayed.

Notes

BTE is only valid during RUN mode.

See FTO for the length of time until the timeout.

You can also enable or disable the emulator bus timeout in XICE.CFG with SW_BTE.

Example

```
BTE OFF
```

Related Commands

FTO

BUS — show state of processor bus signals

Works with

■ EL 1600 ❒ CodeTAP

Syntax

```
BUS
```

Description

This command displays information about the processor, and the state of the processor's pins. The signals are displayed as 0 or 1.

0	The signal is inactive.
1	The signal is active.

Notes

The information for internal emulator memory (EIM) is displayed for the 68302 only. See the *EL 1600 Hardware Setup and Reference Manual* for further information on EIM.

Example

```
> bus
 PINS:
                    BERR= 0
                    VCC= 1
                    TRST= 0
                    PRST= 0
```

IPL0= 0

IPL1= 0

IPL2= 0

BGT= 0

DBF= 0

HLT= 0

CLK= 1

8 Bit Mode= 0

EIM switch= 0xff0000

FSI count: 0

BUSTIME — capture bus timing in trace buffer

Works with

■ EL 1600 ❒ CodeTAP

Syntax

```
BUSTIME [ON|OFF]
```

Description

This switch specifies whether bus timing information or interrupt level information should be recorded in trace. If BUSTIME is set to ON, bus timing information is recorded in the trace under the heading IPL. If BUSTIME is set to OFF, the current interrupt level information is recorded in trace rather than the bus timing information.

The default for BUSTIME is OFF. If no argument is given, the state of the switch is displayed.

Notes

You can also set BUSTIME in XICE.CFG with SW_BUSTIME:ON and SW_BUSTIME:OFF.

Example

```
BUSTIME ON
```

CAS — continuous address strobe

Works with

■ EL 1600 ❐ CodeTAP

Syntax

```
CAS [ON|OFF]
```

Description

This switch specifies whether the target sees address strobes while the emulator is paused. If CAS is set to ON, the target sees address strobes while the emulator is paused. If CAS is set to OFF, the target does not see address strobes when the emulator is paused.

The default for CAS is OFF. If no argument is given, the state of the switch is displayed.

Notes

You can also set CAS in XICE.CFG with SW_CAS:ON and SW_CAS:OFF.

Example

```
CAS OFF
```

CRC — calculate a CRC for a range of memory

Works with

■ EL 1600 ❐ CodeTAP

Syntax

```
CRC address_range [/address_space]
```

Description

This command performs a CRC over the specified range, where *range* is of the form *start..end. Address_space* specifies the memory space from which the target reads are to be performed. If no memory space is specified, it defaults to the memory read space specified by the command SPACE.

Notes

Memory space is processor-specific. The valid values for 68000 family processors are as follows:

Address Space	Description
SC0	Reserved memory space
UD or SC1	User data space
UP or SC2	User program space
SC3	Reserved memory space
SC4	Reserved memory space
SD or SC5	Supervisor data space
SP or SC6	Supervisor program space
CPU or SC7	CPU space

Example

CRC 0..0xFFFF /SD	Calculates a CRC in supervisor data space from 0 to 0xFFFF

DBP — disable bus error on peek/poke

Works with

■ EL 1600 ❒ CodeTAP

Syntax

```
DBP [ON|OFF]
```

Description

This command controls whether bus errors are reported on peeks and pokes. If DBP is set to ON, the bus error signals detected from the target system are displayed. If DBP is set to OFF, target system bus errors are ignored during peeks and pokes.

The default for DBP is ON. If no argument is given, the state of the switch is displayed.

Notes

You can also set DBP in XICE.CFG with SW_DBP:ON and SW_DBP:OFF.

Example

```
DBP ON
```

DIAG 0 — simple target ram test

Works with

■ EL 1600 ❒ CodeTAP

Syntax

```
DIAG 0,address_range[#count][;memory_space][/access_-
size]
```

Description

Target diagnostic 0 performs a simple target RAM test on the range of target memory you specify.

The optional parameter *#count* indicates the number of passes to make. The valid choices for *#count* are #0 and #1. #0 will cause the test to continue until you enter ctrl-C. #1 will cause one iteration of the test. If you do not set *#count*, it defaults to #1.

The optional parameter *;memory_space* specifies the memory space in which the test is performed. Memory space codes are the same as for the CRC command. If you do not set *;memory_space*, it defaults to the value set with the SPACE DIAG command.

Memory space is processor-specific. The valid values for 68000 family processors are as follows:

Code Space	Description
SC0	Reserved memory space
UD or SC1	User data space
UP or SC2	User program space
SC3	Reserved memory space

Code Space	Description
SC4	Reserved memory space
SD or SC5	Supervisor data space
SP or SC6	Supervisor program space
CPU or SC7	CPU space

The optional parameter */access_size* specifies the size in which memory is accessed. It may be a **/1** (byte), **/2** (word), or **/4** (longword). If you do not set */access_size*, it defaults to the value set with the SIZE DIAG command. For access size **/4**, the emulator actually uses two word accesses since the 68000 and 68302 processors cannot directly access memory using longwords.

Example

```
DIAG 0, 0..0x0fff #6
```

DIAG 1 — complex target ram test

Works with

■ EL 1600 ❒ CodeTAP

Syntax

```
DIAG 1,address_range[#count][;memory_space][/access_-
size]
```

Description

Target diagnostic 1 performs a complex target RAM test on the range of target memory you specify.

The optional parameter *#count* indicates the number of passes to make. The valid choices for *#count* are #0 and #1. #0 will cause the test to continue until you enter ^C. #1 will cause one iteration of the test. If you do not set *#count*, it defaults to 1.

The optional parameter *;memory_space* specifies the memory space in which the test is performed. Memory space codes are the same as for the CRC command. If you do not set *;memory_space*, it defaults to the value set with the SPACE DIAG command.

Memory space is processor-specific. The valid values for 68000 family processors are as follows:

Code Space	Description
SC0	Reserved memory space
UD or SC1	User data space
UP or SC2	User program space
SC3	Reserved memory space

Code Space	Description
SC4	Reserved memory space
SD or SC5	Supervisor data space
SP or SC6	Supervisor program space
CPU or SC7	CPU space

The optional parameter */access_size* specifies the size in which memory is accessed. It may be a **/1** (byte), **/2** (word), or **/4** (longword). If you do not set */access_size*, it defaults to the value set with the SIZE DIAG command. For access size **/4**, the emulator actually uses two word accesses since the 68000 and 68302 processors cannot directly access memory using longwords.

Example

```
DIAG 1, 0..0xfff #0
```

DIAG 2 — continuous read from target memory

Works with

■ EL 1600 ❒ CodeTAP

Syntax

```
DIAG 2,address[;memory_space][/access_size]
```

Description

Target diagnostic 2 performs a continuous read from the target memory address you specify. This test continues until you enter ctrl-C.

The optional parameter *;memory_space* specifies the memory space in which the test is performed. Memory space codes are the same as for the CRC command. If you do not set *;memory_space*, it defaults to the value set with the SPACE DIAG command.

Memory space is processor-specific. The valid values for 68000 family processors are as follows:

Code Space	Description
SC0	Reserved memory space
UD or SC1	User data space
UP or SC2	User program space
SC3	Reserved memory space
SC4	Reserved memory space

Code Space	Description
SD or SC5	Supervisor data space
SP or SC6	Supervisor program space
CPU or SC7	CPU space

The optional parameter */access_size* specifies the size in which memory is accessed. It may be a **/1** (byte), **/2** (word), or **/4** (longword). If you do not set */access_size*, it defaults to the value set with the SIZE DIAG command. For access size **/4**, the emulator actually uses two word accesses since the 68000 and 68302 processors cannot directly access memory using longwords.

Notes

None

Example

```
DIAG 2, 0x0 /2
```

DIAG 3 — continuous write to target memory

Works with

■ EL 1600 ❐ CodeTAP

Syntax

```
DIAG 3,address=data[;memory_space][/access_size]
```

Description

Target diagnostic 3 performs a continuous write of *data* to the target memory address you specify. This test continues until you enter ctrl-C.

The optional parameter *;memory_space* specifies the memory space in which the test is performed. Memory space codes are the same as for the CRC command. If you do not set *;memory_space*, it defaults to the value set with the SPACE DIAG command.

Memory space is processor-specific. The valid values for 68000 family processors are as follows:

Code Space	Description
SC0	Reserved memory space
UD or SC1	User data space
UP or SC2	User program space
SC3	Reserved memory space
SC4	Reserved memory space

Code Space	Description
SD or SC5	Supervisor data space
SP or SC6	Supervisor program space
CPU or SC7	CPU space

The optional parameter */access_size* specifies the size in which memory is accessed. It may be a **/1** (byte), **/2** (word), or **/4** (longword). If you do not set */access_size*, it defaults to the value set with the SIZE DIAG command. For access size **/4**, the emulator actually uses two word accesses since the 68000 and 68302 processors cannot directly access memory using longwords.

Example

```
DIAG 3, 0x1000 = 0xcf2617 /2
```

DIAG 4 — write alternating pattern to target location

Works with

■ EL 1600 ❒ CodeTAP

Syntax

```
DIAG 4,address=data[;memory_space][/access_size]
```

Description

Target diagnostic 4 writes an alternating pattern of *data* to the target memory address you specify. It uses the data given as one pattern and the data given, inverted bit-wise, as the alternate pattern. This test continues until you enter ctrl-C.

The optional parameter *;memory_space* specifies the memory space in which the test is performed. Memory space codes are the same as for the CRC command. If you do not set *;memory_space*, it defaults to the value set with the SPACE DIAG command.

Memory space is processor-specific. The valid values for 68000 family processors are as follows:

Code Space	Description
SC0	Reserved memory space
UD or SC1	User data space
UP or SC2	User program space
SC3	Reserved memory space
SC4	Reserved memory space

Code Space	Description
SD or SC5	Supervisor data space
SP or SC6	Supervisor program space
CPU or SC7	CPU space

The optional parameter */access_size* specifies the size in which memory is accessed. It may be a **/1** (byte), **/2** (word), or **/4** (longword). If you do not set */access_size*, it defaults to the value set with the SIZE DIAG command. For access size **/4**, the emulator actually uses two word accesses since the 68000 and 68302 processors cannot directly access memory using longwords.

Example

```
DIAG 4, 0x10 = 0x5555 /2
```

DIAG 5 — write rotated to target memory

Works with

■ EL 1600 ❐ CodeTAP

Syntax

```
DIAG 5,address=data[;memory_space][/access_size]
```

Description

Target diagnostic 5 performs a continuous write of *data* to the target memory address you specify, and after each write the data value is rotated left by one bit. For example with data=01, one complete rotation is as follows:

01
02
04
08
10
20
40
80
01

This test continues until you enter ctrl-C.

The optional parameter *;memory_space* specifies the memory space in which the test is performed. Memory space codes are the same as for the CRC command. If you do not set *;memory_space*, it defaults to the value set with the SPACE DIAG command.

Memory space is processor-specific. The valid values for 68000 family processors are as follows:

Code Space	Description
SC0	Reserved memory space
UD or SC1	User data space
UP or SC2	User program space
SC3	Reserved memory space
SC4	Reserved memory space
SD or SC5	Supervisor data space
SP or SC6	Supervisor program space
CPU or SC7	CPU space

The optional parameter */access_size* specifies the size in which memory is accessed. It may be a **/1** (byte), **/2** (word), or **/4** (longword). If you do not set */access_size*, it defaults to the value set with the SIZE DIAG command. For access size **/4**, the emulator actually uses two word accesses since the 68000 and 68302 processors cannot directly access memory using longwords.

Example

```
DIAG 5, 0 = 0x1212 ;UD
```

DIAG 6 — write then read target memory

Works with

■ EL 1600 ❒ CodeTAP

Syntax

```
DIAG 6,address=data[;memory_space][/access_size]
```

Description

Target diagnostic 6 performs a continuous write of *data* to the target memory address you specify, and after each write then reads the same location. This test continues until you enter ctrl-C.

The optional parameter *;memory_space* specifies the memory space in which the test is performed. Memory space codes are the same as for the CRC command. If you do not set *;memory_space*, it defaults to the value set with the SPACE DIAG command.

Memory space is processor-specific. The valid values for 68000 family processors are as follows:

Code Space	Description
SC0	Reserved memory space
UD or SC1	User data space
UP or SC2	User program space
SC3	Reserved memory space
SC4	Reserved memory space

Code Space	Description
SD or SC5	Supervisor data space
SP or SC6	Supervisor program space
CPU or SC7	CPU space

The optional parameter */access_size* specifies the size in which memory is accessed. It may be a **/1** (byte), **/2** (word), or **/4** (longword). If you do not set */access_size*, it defaults to the value set with the SIZE DIAG command. For access size **/4**, the emulator actually uses two word accesses since the 68000 and 68302 processors cannot directly access memory using longwords.

Example

```
DIAG 6,0x1000 = 0x24 /1
```

DIAG 7 — continuous read from target memory range

Works with

■ EL 1600 ❐ CodeTAP

Syntax

```
DIAG 7,address_range[;memory_space][/access_size]
```

Description

Target diagnostic 7 performs a continuous read from the target memory range you specify. This test continues until you enter ctrl-C.

The optional parameter *;memory_space* specifies the memory space in which the test is performed. Memory space codes are the same as for the CRC command. If you do not set *;memory_space*, it defaults to the value set with the SPACE DIAG command.

Memory space is processor-specific. The valid values for 68000 family processors are as follows:

Code Space	Description
SC0	Reserved memory space
UD or SC1	User data space
UP or SC2	User program space
SC3	Reserved memory space
SC4	Reserved memory space

Code Space	Description
SD or SC5	Supervisor data space
SP or SC6	Supervisor program space
CPU or SC7	CPU space

The optional parameter */access_size* specifies the size in which memory is accessed. It may be a **/1** (byte), **/2** (word), or **/4** (longword). If you do not set */access_size*, it defaults to the value set with the SIZE DIAG command. For access size **/4**, the emulator actually uses two word accesses since the 68000 and 68302 processors cannot directly access memory using longwords.

Example

```
DIAG 7, 0..0x1f /2
```

DIAG 8 — write incremental value to target memory

Works with

■ EL 1600 ❐ CodeTAP

Syntax

```
DIAG 8,address[;memory_space][/access_size]
```

Description

Target diagnostic 8 writes an incrementing value to the target memory address you specify. This test continues until you enter ctrl-C.

The optional parameter *;memory_space* specifies the memory space in which the test is performed. Memory space codes are the same as for the CRC command. If you do not set *;memory_space*, it defaults to the value set with the SPACE DIAG command.

Memory space is processor-specific. The valid values for 68000 family processors are as follows:

Code Space	Description
SC0	Reserved memory space
UD or SC1	User data space
UP or SC2	User program space
SC3	Reserved memory space
SC4	Reserved memory space

Code Space	Description
SD or SC5	Supervisor data space
SP or SC6	Supervisor program space
CPU or SC7	CPU space

The optional parameter */access_size* specifies the size in which memory is accessed. It may be a **/1** (byte), **/2** (word), or **/4** (longword). If you do not set */access_size*, it defaults to the value set with the SIZE DIAG command. For access size **/4**, the emulator actually uses two word accesses since the 68000 and 68302 processors cannot directly access memory using longwords.

Example

```
DIAG 8, 0x1000 ;SD /1
```

DIAG 9 — continuous stream of reset pulses

Works with

■ EL 1600 ❐ CodeTAP

Syntax

```
DIAG 9
```

Description

Target diagnostic 9 sends a continuous stream of reset pulses to the target CPU. This test continues until you enter ctrl-C.

Example

```
DIAG 9
```

DNL — download hex file to target

Works with

■ EL 1600 ❒ CodeTAP

Syntax

```
DNL "filename" [,offset]
```

Description

DNL allows you to download a hex file from the host to the target in the format specified by the DNLFMT command.The contents of the file will be downloaded to the memory locations specified in the file. If you wish to download the file to memory locations different from those specified in the file, enter a value for *offset*. The *offset* value will be added to the address of each record to determine the actual download address. The default value for *offset* is 0.

The MAP, OVERLAY, SPACE, and SIZE commands affect how memory is accessed by DNL. Memory read-after-write verification is controlled by the setting of the VERIFY switch.

Notes

Quotation marks are optional if the file name consists of alphanumeric characters or a period. File names that contain a leading slash must be in double quotation marks (e.g., "/root"). File names that contain a leading backslash must be in single quotation marks (e.g., '\root').

Example

```
DNL my.file ,0x1000
```

DNLFMT — specify download format

Works with

■ EL 1600 ❐ CodeTAP

Syntax

```
DNL format
```

Description

DNLFMT is used to specify the format for hex file downloads using the DNL command. Recognized formats are:

INTEL	Intel hex format. Extended segment address records and extended linear address records are supported.
SREC	Motorola S1, S2, S3-records with Microtec extensions.
XTEK	Extended Tektronics hex format.

The default format of DNLFMT is SREC. The command XICEVARS displays the status of this variable as well as all the XICE variables.

Notes

Symbols are not supported for these formats.

Example

```
DNLFMT SREC
```

Related Commands

DNL, UPL, UPLFMT

DNL_GAP — specify maximum bytes between blocks

Works with

■ EL 1600 ❐ CodeTAP

Syntax

```
DNL_GAP [1-1024]
```

Description

DNL_GAP specifies the maximum number of bytes allowed between two cached download blocks before they are considered discontiguous. Download speed is faster for contiguous blocks than for discontiguous blocks. Specifying a larger number will improve download speed, but may cause some locations to be overwritten if there are discontiguous blocks that are smaller than the value of DNL_GAP. A lower value will avoid this. If no argument is given, the current value is displayed.

The default value of DNL_GAP is 1. The command XICEVARS displays the status of this variable as well as all the XICE variables.

Example

```
DNL_GAP 4
```

Related commands

DNL, DNLFMT

DOWN (DOW) — move current scope

Works with

■ EL 1600 ❐ CodeTAP

Syntax

```
DOWN [number_of_levels]
```

Description

The UP and DOWN commands allow you to move the current scope up or down the runtime stack. This is especially helpful when debugging recursive functions. It is not a good idea to go down farther than you have gone up.

Example

```
DOW 5
```

Related commands

UP

DRT — display raw trace

Works with

■ EL 1600 ❐ CodeTAP

Syntax

```
DRT [start|start..end]
```

Description

This command displays raw trace information. If you wish to limit the display, you may specify a range of bus cycles.

DRT by itself displays the last page of trace. DRT with a range displays trace for the specified range. DRT with a start number displays trace from the specified frame forward.

Notes

The columns DMA and IAC relate only to the 68302. See section 1 of this supplement for a description of the information in each of the columns in the trace display.

Example

```
drt
                                                                  FLAGS: X-break
 MEM      R-read     MEM        B-byte    MEM        T-target            B-BERR
 ACCESS: W-write     BOUNDARY: W-word     LOCATION: O-overlay            M-illegal
                                                                         mem access
 FRAME   ADDRESS   DATA   IPL   FCn  MEM  DMA IAC  FLAGS   --LSA BITS-------

    44   005732   22..   000   SP   RBO                   11111111 11111111
    43   005733   ..2F   000   SP   RBO                   11111111 11111111
    42   007FA4   00..   000   SD   WBO                   11111111 11111111
    41   007FA5   ..00   000   SD   WBO                   11111111 11111111
    40   007FA6   56..   000   SD   WBO                   11111111 11111111
    39   007FA7   ..D0   000   SD   WBO                   11111111 11111111
    38   005734   00..   000   SP   RBO                   11111111 11111111
    37   005735   ..04   000   SP   RBO                   11111111 11111111
    36   005736   10..   000   SP   RBO                   11111111 11111111
    35   005737   ..39   000   SP   RBO                   11111111 11111111
    34   007FA8   00..   000   SD   RBO                   11111111 11111111
    33   007FA9   ..00   000   SD   RBO                   11111111 11111111
    32   007FAA   00..   000   SD   RBO                   11111111 11111111
    31   007FAB   ..7C   000   SD   RBO                   11111111 11111111
    30   005738   00..   000   SP   RBO                   11111111 11111111
    29   005739   ..00   000   SP   RBO                   11111111 11111111
    28   00573A   60..   000   SP   RBO                   11111111 11111111
    27   00573B   ..4C   000   SP   RBO                   11111111 11111111
    26   00573C   02..   000   SP   RBO                   11111111 11111111
    25   00573D   ..80   000   SP   RBO                   11111111 11111111
    24   00604C   8D..   000   SD   RBO                   11111111 11111111
```

Related Commands

DT, DTB, DTF

DRTMR — enable dynamic refresh of memory (68302 only)

Works with

■ EL 1600 ❒ CodeTAP

Syntax

```
DRTMR [NONE | TMR1 | TMR2 | SCC1 | SCC2 | SCC3]
```

Description

This switch controls the dynamic refresh of memory. The following arguments are allowed for DRTMR:

NONE	do not allow a DRAM refresh
TMR1	use TMR1 to trigger a DRAM refresh
TMR2	use TMR2 to trigger a DRAM refresh
SCC1	use SCC1 to trigger a DRAM refresh
SCC2	use SCC2 to trigger a DRAM refresh
SCC3	use SCC3 to trigger a DRAM refresh

The default for DRTMR is NONE.

Notes

This command is not used by XICE for the 68000.

You can also set DRTMR in XICE.CFG with SW_DRTMR:ON or SW_DRTMR:OFF.

Example

```
DRTMR NONE
```

DRUN — start dynamic run mode

Works with

■ EL 1600 ❒ CodeTAP

Syntax

```
DRUN
```

Description

The DRUN (dynamic run) command executes the target program and continues execution until it is stopped by DSTOP, a breakpoint, an error, or a halt instruction. The purpose of this mode is to allow the user to interact with the emulator and debugger dynamically, while the emulator is running. In DRUN the user can examine and qualify trace, set and change events and breakpoints, examine and change memory and perform most other interactive functions with the emulator.

During DRUN the breakpoint and event systems are active but the emulator is not polled regularly for status. This can result in the emulator breaking execution with no notification to the user. Because most commands force polling of the emulator, unless there is no user interaction, the emulator status will be made known to XICE. When a break in execution is detected, the user will be notified and DRUN will be exited. The DUPDATE command can be used to force regular polling of the emulator.

During DRUN the XICE version number on the XICE status line is replaced by the word DRUN.

Notes

When trace is requested from the emulator, the trace system is disabled during the time trace is uploaded to the host. This has an unavoidable side effect of also disabling the break system during the same period.

Example

```
DRUN
```

Related Commands

DSTOP, DUPDATE

DSTOP — stop dynamic run mode

Works with

■ EL 1600 ❐ CodeTAP

Syntax

```
DSTOP
```

Description

The DSTOP command stops the DRUN (dynamic run) command and breaks program execution.

Notes

None

Example

```
DSTOP
```

Related Commands

DRUN, DUPDATE

DT — display trace

Works with

■ EL 1600 ❐ CodeTAP

Syntax

```
DT [start|start..end]
```

Description

This command displays disassembled trace (bus cycles), showing either assembly instructions, source lines or both depending on the value you set for TRCMODE. You may specify the *start..end* range of bus cycles anywhere from 0 to 8K.

TRCMODE values are as follows:

ASM	causes an assembly instruction only display
SRC	causes a source line only display
BOTH	causes an interleaved high level source and assembly display

If no argument is given, DT shows the last instruction executed.

Notes

The emulator displays the message UNATTACHED BUS DATA when there is data on the bus that does not match up to an instruction. Unattached bus data may be caused by an external device putting data on the bus or, for CPUs that have cache, running with cache which allows data movements that do not match fetched instructions.

The disassemblers require a continuous trace stream to be able to disassemble correctly. For this reason, if you have PPT set to ON, or if you use the event system to qualify trace, DT will not be able to function correctly. Failures will range from incorrect information being displayed to crashing XICE.

Example

DT 10..50	Disassembles trace cycles 10 to 50.

Related commands

DTB, DTF, TRCMODE

DTB —display disassembled trace backwards

Works with

■ EL 1600 ❐ CodeTAP

Syntax

```
DTB
```

Description

This command displays disassembled trace backwards (away from cycle 0), one page at a time starting from the most recent cycle count. It shows either assembly instructions, source lines or both depending on the value you set for TRCMODE.

TRCMODE values are as follows:

ASM	causes an assembly instruction only display
SRC	causes a source line only display
BOTH	causes an interleaved high level source and assembly display

Notes

The emulator displays the message UNATTACHED BUS DATA when there is data on the bus that does not match up to an instruction. Unattached bus data may be caused by an external device putting data on the bus or, for CPUs that have cache, running with cache which allows data movements that do not match fetched instructions.

The disassembler requires a continuous stream of at least 128 lines of raw trace for a good disassembly. For this reason, if you have PPT set to ON, if you use the event system to qualify trace, or if you attempt using DTB with fewer than 128 lines of

raw trace you will not be able to accurately disassemble the trace.The results will range from an error message being displayed to incorrect information being displayed to XICE crashing.

Example

```
DTB¶
  ¶
 )         )          )          WORKING  ...)..¶
  ¶
  ¶
  0048 004050 4EB90000 JSR      main¶
              408E¶
  0045 main:¶
 >>    inital();                              /* Initialize Variables */¶
  0045 00408E 4EBA0018 JSR      inital¶
  0041 inital:¶
 >>    pattern = ONE_ON;                      /* 0___ ____ */¶
  0041 0040A8 13FC00FE MOVE.B   #$FE,pattern¶
              00005F44¶
 >>    speed = MEDIUM;                        /* SLOW - MEDIUM - FAST */¶
  0035 0040B0 23FC0000 MOVE.L   #$000001F4,speed¶
              01F400005E58¶
 >>    direct = left;                         /* left to right */¶
  0029 0040BA 7001     MOVEQ.L  #$1,D0¶
  0026 0040BC 23C00000 MOVE.L   D0,direct¶
              5FC0¶
 >>}¶
  0023 0040C2 4E75     RTS¶
 >>    inital();                              /* Initialize Variables */¶
  0017 004092 4E71     NOP¶
 >>    step();                                /* Single Step Loop */¶
  0016 004094 4EBA002E JSR      step¶
  0014 step:¶
 >>{¶
  0014 0040c4 2F02     MOVE.L   D2,-(A7)¶
 >>    for (loops = 5; loops != 0; loops--)/* repeat output 5 times */¶
  0011 0040C6 7405     MOVEQ.L  #$5,D2¶
 >>        outled(0xFE);                      /* 0111 1111 */¶
  0010 0040C8 487800FE PEA.L    $0000FE¶
  0006 0040CC 4EB90000 JSR      outled¶
              407C               007FF2<00FE¶
                 ¶
```

Related Commands

DT, DTF, TRCMODE

DTF — display disassembled trace forward

Works with

■ EL 1600 ❒ CodeTAP

Syntax

```
DTF
```

Description

This command displays disassembled trace forward (toward cycle 0), one page at a time starting from the most recent cycle count. It shows either assembly instructions, source lines or both depending on the value you set for TRCMODE.

TRCMODE values are as follows:

ASM	causes an assembly instruction only display
SRC	causes a source line only display
BOTH	causes an interleaved high level source and assembly display

Notes

The emulator displays the message UNATTACHED BUS DATA when there is data on the bus that does not match up to an instruction. Unattached bus data may be caused by an external device putting data on the bus or, for CPUs that have cache, running with cache which allows data movements that do not match fetched instructions.

The disassembler requires a continuous stream of at least 128 lines of raw trace for a good disassembly. For this reason, if you have PPT set to ON, if you use the event system to qualify trace, or if you attempt using DTF with fewer than 128 lines of raw trace you will not be able to accurately disassemble the trace. The results will range from an error message being displayed to incorrect information being displayed to XICE crashing.

Example

```
> dtf¶
  ¶
  ¶
}          }          }          WORKING  ...}..¶
  ¶
  ¶
  0054 START:¶
  0054 004044 2E7C0000 MOVE.L   #$00008000,A7¶
               8000¶
  0051 00404A 2C7C0000 MOVE.L   #$00000000,A6¶
```

Related commands

DT, DTB, TRCMODE

DUPDATE —specify polling frequency in dynamic run mode

Works with

■ EL 1600 ❐ CodeTAP

Syntax

```
DUPDATE [1...]
```

Description

The DUPDATE command allows you to specify how frequently the emulator is polled during DRUN (dynamic run) mode. The value entered represents the number of polls per minute. Whenever the emulator is polled the screen viewports are updated and the user is notified if emulation has broken.

DUPDATE is entered while in DRUN mode; in DUPDATE mode, commands from the user are no longer accepted. To exit DUPDATE and return to DRUN, enter Ctrl-C.

If no argument is given, DUPDATE defaults to 20. Above 100 polls per minute, increasing the polling rate will have no increasing effect.

Notes

None

Example

```
DUPDATE 30
```

Related Commands

DRUN, DSTOP

EMUVARS — display emulator variable values

Works with

■ EL 1600 ❐ CodeTAP

Syntax

```
EMUVARS
```

Description

This command displays the current values and descriptions for all the emulator softswitch variables.

Example

```
                    EMULATOR SOFTSWITCHES
                    ---------------------
BTE         OFF     Enable (ON)* vs. disable (OFF) bus timeout
BUSTIME     OFF     Capture bus timing (ON) vs. interrupt level (OFF)* in trace
CAS         OFF     Address strobe active during run (OFF)* vs. run and pause (ON)
DBP         ON      Enable (ON)* vs. disable (OFF) bus error detect on peek/poke
DRTMR       NONE    Use TMR1, TMR2, SCC1, SCC2, SCC3, NONE* to trigger DRAM refresh
EXVEC       15      Soft breakpoint execution trap number (0-15*)
FAST        ON      Enable on emulation (ON)* vs. disable (OFF) fast interrupts
FRZ         OFF     Assert (ON) vs. do not assert (OFF)* FRZ while paused
FTO         OFF     Enable (ON) vs. disable (OFF)* fast bus timeout
MWARN       ON      Warn user (ON)* vs. ignore (OFF) out of range memory accesses
OVE         0x66    Memory spaces overlay will respond to (0-0xFF). Default: 0x66
OVS         1       Overlay speed (0-7). Default: 0
PPT         OFF     Enable (ON) vs. disable (OFF)* tracing of peek/poke cycles
RFS         OFF     Enable (ON) vs. disable (OFF)* software refresh
RFSADR      0       Software refresh address. Default: 0
RFSASP      5       Address space for software refresh (0-7). Default: 5
RFSMSK      0       Don't care mask for software refresh. Default: 0
RIRR        ON      Restore (ON)* vs. do not restore (OFF) registers on reset
SCRATCH     0xfff0  Start address of breakpoint scratch area. Default: 0
SIA         0       Special interrupt vector address. Default: 0
SLO         OFF     Wait then enable (ON) vs. disable (OFF)* slow interrupts
TAD         OFF     Tri-state (ON) vs. do not tri-state (OFF)* address bus in pause
TCEBRK      OFF     Trace (ON) vs. do not trace (OFF)* execution break cycles
TED         OFF     Enable (ON) vs. disable (OFF)* trace/overlay for external DMA
TID         OFF     Enable (ON) vs. disable (OFF)* trace/overlay for internal DMA
UIR         ON      Enable (ON)* vs. disable (OFF) auto-update of chip select regs
```

Related commands

BUS, EVTVARS, MEMVARS, XICEVARS

EV — define an event

Works with

■ EL 1600 ❒ CodeTAP

Syntax

```
EV{n} = [event_definition|CLEAR]
```

Description

The EV command supports the EL 1600 event system capability. EV sets up an event definition and TRIG defines the action(s) to take place, once the trigger is armed, each time the event definition is met. For example,

trig{5} = ev{1},break	Sets trigger 5 to cause a break when event 1 is true.
trig{5} = arm	Arms trigger 5.

An event name is shown as `ev{n}` where *n* is the number identifying this event (the curly braces are required punctuation). The event number *n* must be between 1 and 32.

An event definition is the specification of a possible state of the trace frame (the address, data, and status buses) along with the state of other event resources such as counters, during that trace frame's bus cycle. An event is true when all of the terms within the event are true at the same time (i.e., the same single bus cycle).

The following general rules relate to setting up the event definition:

- The logical operators for equality (==), or inequality (!=) are used to set the values. E.g., `ev{1} = stat==rd`

- Event terms can be used only once in any one event definition.
- Each testable condition must be separated from the next by a comma.
  E.g., `ev{1} = addr==0x1, data==0x2, stat==word`
- Addresses can be specified as ranges that are denoted by (..).
  E.g., `ev{1} = addr==0x0000..0xffff`. Note, however, that you cannot have two comparators of the same type in a single event statement.
  E.g., `ev{1} = addr==0x1, addr==0x2` will not work.
- Don't care masks can be used to exclude parts of data bus information.
  E.g., `ev{1} = data==0x0034 &=0x00ff`
  defines an event that would be valid whenever the 8 least significant bits of the data bus are 0x34.
- The counters start at 0 each time you GO. They require specific values. You may not use ranges or don't care masks with the counter.
- LSA bits may be set using don't care masks.
  E.g., `ev{31} = LSA==0x2&=0x3`

The information that follows lists the elements available for setting up an event definition for the 68000 and 68302.

Condition	Definition
ADDRESS	The value that appears on the address bus.
COUNT	The counter value.
DATA	The value that appears on the data bus.
STATUS	The type of bus activity (e.g. instruction fetch, read, write, interrupt acknowledge).
LSA	The value of the LSA bits.

The information that follows lists the valid 68000 and 68302 STATUS mnemonics that can be used in an event definition either in their positive form as listed below or in their negative form by prepending a NOT_.

Mnemonic	Definition
BERR	Bus error
BRK	Break signal asserted. This status is useful for determining the skid, i.e., the number of bus cycles between the time the break occurs and emulation stops.
BYTE	Byte access
DMA	Pod DMA signal state (68302 only)
IAC	CPU IAC pin state (68302 only)
OVL	Overlay access
RD	CPU read
TAR	Target access
VIO	Violation error
WORD	Word access
WR	CPU write

The information that follows lists additional valid 68000 and 68302 STATUS mnemonics that can be used in an event definition. However, these mnemonics may only be used as shown below, in other words, unlike the list of mnemonics above, they do not have a negative form.

Mnemonic	Definition
CPU	Access to CPU space
INTR0	An interrupt 0 is pending
INTR1	IP0-IP2 is set to 1 (active low)
INTR2	IP0-IP2 is set to 2 (active low)
INTR3	IP0-IP2 is set to 3 (active low)
INTR4	IP0-IP2 is set to 4 (active low)
INTR5	IP0-IP2 is set to 5 (active low)
INTR6	IP0-IP2 is set to 6 (active low)
INTR7	IP0-IP2 is set to 7 (active low)
SD	Access to supervisory data space
SP	Access to supervisory program space
UD	Access to user data space
UP	Access to user program space

Examples:

```
ev{1} = addr==0x13400..0x134FF, stat==up|rd
```

In the example above, the event expression `ev{1}` is true if any address in the 256 byte block 0x13400..0x134FF is read in user fetches.

```
ev{1} = addr==0x13400, count==4
```

The expression `ev{1}` is true for all accesses of address 0x13400 when the counter has reached the count of 4.

```
ev{1} = clear
```

Clears the event definition for ev{1} for reuse.

Related commands

EVTMODE, EVTGRP, EVTCLR, EVTARM, GROUP, TRIG

EVTARM — enable or disable automatic trigger arming

Works with

■ EL 1600 ❒ CodeTAP

Syntax

```
EVTARM [ON|OFF]
```

Description

This switch specifies whether triggers are automatically armed when they are defined using the TRIG command. If EVTARM is set to ON, triggers are automatically armed when defined. If EVTARM is set to OFF, triggers are not automatically armed when defined.

The default is ON. If no argument is given, the state of the switch is displayed.

Notes

Triggers are only active when they have been armed.

You can also set EVTARM in XICE.CFG with SW_EVTARM:ON or SW_EVTARM:OFF.

Example

```
EVTARM ON
```

Related Commands

EV,EVTMODE, EVTCLR,TRIG, XICEVARS

EVTCLR — clear event system

Works with

■ EL 1600 ❐ CodeTAP

Syntax

```
EVTCLR
```

Description

This command clears all events and triggers set up in the event system and resets the event state variables to their initial values.

Example

```
EVTCLR
```

Related Commands

EV, EVTMODE, EVTARM, TRIG, XICEVARS

EVTGRP — specify event group

Works with

■ EL 1600 ❒ CodeTAP

Syntax

```
EVTGRP [1|2|3|4]
```

Description

This command specifies the event group for an event when arming triggers. Whenever a trigger is armed, either automatically (when EVTARM is set to ON) or explicitly with a TRIG command, it is armed in a particular event group. This group is specified by the EVTGRP variable.

Event group 1 is the default. If no argument is given, the state of the switch is displayed.

The emulator allows four event groups. For any one group there may not be more than the following comparators:

- 2 address comparators (specifying an address range counts as a single address comparator)
- 2 data comparators
- 2 status comparators
- 1 LSA comparator
- 1 counter

Notes

You can also set EVTGRP in XICE.CFG with SW_EVTGRP:1, SW_EVTGRP:2, SW_EVTGRP:3, or SW_EVTGRP:4.

Example

```
EVTGRP 2
```

Related commands

GROUP, EV, EVTCLR, EVTARM, TRIG, EVTVARS

EVTVARS — display internal debugger variable values

Works with

■ EL 1600 ❒ CodeTAP

Syntax

EVTVARS

Description

This command displays the current value and description for the emulator event state variable GROUP. It indicates the event group that the emulator is currently in and is a read-only value. It cannot be changed.

Example

```
> evtvars

                        WORKING  ...    ...

                                EVENT STATE VARIABLES
                                ---------------------
 GROUP    1          Event group that the emulator is currently in.  <READ-ONLY>
>
```

Related Commands

EMUVARS, MEMVARS, XICEVARS

EXPLAIN — explain error message

Works with

■ EL 1600 ❒ CodeTAP

Syntax

```
EXPLAIN
```

Description

This command provides additional information about the last emulator-related error message reported.

Notes

This command only supports emulator-related error messages. If no emulator-related error messages have been generated, EXPLAIN states that the error is not emulator related.

Example

```
EXPLAIN
```

EXVEC — software breakpoint execution trap number

Works with

■ EL 1600 ❐ CodeTAP

Syntax

```
EXVEC [number]
```

Description

The EXVEC switch specifies the software execution breakpoint trap number, where *number* may be from 0 to 15. The number that you enter for EXVEC should be the same as the trap number in your Vector Table.

The default for EXVEC is 15. If no argument is given, EXVEC shows the current setting for the switch.

Notes

The trap number that you specify for EXVEC must be dedicated to XICE exclusively. This is because XICE uses that trap to implement software breakpoints. If your program also uses that trap, XICE will report a spurious break every time your program executes that trap.

XICE sets the vector for the specified trap to point to code that it has placed in the scratch area to handle software breakpoints. The target system may not change the vector of the specified trap once XICE has initialized it, nor may the target system modify the code XICE has placed in the scratch area.

To install a software breakpoint, XICE must be able to modify the opcode at the desired break address. If that address is in ROM, you will have to map that section of code to emulator

overlay RAM. You may map the overlay in read-only mode, which prevents the target system from writing into the area but still allows XICE to modify the opcode.

You can also set EXVEC in XICE.CFG with SW_EXVEC:*number*.

Example

```
EXVEC 15
```

Related commands

BREAKINSTRUCTION, GOSTEP

FAST — fast interrupt emulation control

Works with

■ EL 1600 ❐ CodeTAP

Syntax

```
FAST [OFF|ON]
```

Description

The FAST switch controls the fast interrupt enable. When FAST is set to ON, interrupts are enabled immediately upon entering RUN mode. When FAST is set to OFF, interrupts are disabled.

The default is OFF. If no argument is given, FAST shows the current setting for the switch.

Notes

You can also set FAST in XICE.CFG with SW_FAST:ON or SW_FAST:OFF.

If you set both FAST and SLO to ON, FAST has precedence over SLO. The following table shows the results for the possible switch setting combinations for FAST and SLO. This table applies to target-generated interrupts passed to the target processor when the emulator is running.

SLO	FAST	Result While in RUN Mode	Result While in PAUSE Mode
ON	ON	Interrupts immediately enabled.	Interrupts immediately enabled upon return to RUN mode.
ON	OFF	Interrupts enabled after approximately160 clock cycles.	Interrupts enabled after approximately160 clock cycles after return to RUN mode.
OFF	ON	Interrupts immediately enabled.	Interrupts immediately enabled upon return to RUN mode.
OFF	OFF	Interrupts generated by the target system will be inhibited from reaching the emulator.	Interrupts generated by the target system will be inhibited from reaching the emulator.

Example

```
FAST ON
```

Related commands

SLO

FRZ — freeze peripheral activity (68302 only)

Works with

■ EL 1600 ❐ CodeTAP

Syntax

```
FRZ [ON|OFF]
```

Description

The FRZ switch enables or disables peripheral activity during PAUSE mode. When FRZ is set to ON, the FRZ pin is asserted in PAUSE mode, which disables any peripheral activity. When FRZ is set to OFF, peripheral activity is not blocked during PAUSE.

The default is OFF, which allows peripheral activity during PAUSE mode. If no argument is given, FRZ shows the current setting for the switch.

Notes

You can also set FRZ in XICE.CFG with SW_FRZ:ON or SW_FRZ:OFF.

This command is not used by XICE for the 68000.

Example

```
FRZ ON
```

FTO — fast bus timeout

Works with

■ EL 1600 ❐ CodeTAP

Syntax

```
FTO [ON|OFF]
```

Description

The FTO switch controls the length of time for the bus timeout. If FTO is set to ON, a bus timeout occurs in 112 clock cycles. If FTO is set to OFF, a bus timeout requires 28,672 clock cycles, which is approximately 2 milliseconds.

The default for FTO is OFF. If no argument is given, FTO shows the current setting for the switch.

Notes

If BTE is set to OFF, this switch has no effect regardless of its setting.

You can also set FTO in XICE.CFG with SW_FTO:ON or SW_FTO:OFF.

Example

```
FTO OFF
```

Related Commands

BTE

GROUP — display active event group

Works with

■ EL 1600 ❒ CodeTAP

Syntax

```
GROUP
```

Description

This command displays which of the four event groups was active at the last refresh or break.

Example

```
:-GROUP
 Current setting is 2
```

Related Commands

EVTGRP, EV, EVTVARS

HWCONFIG — display hardware name and version

Works with

■ EL 1600 ❐ CodeTAP

Syntax

```
HWCONFIG
```

Description

This command displays the name and version of all hardware and software being used by the emulator.

Example

```
HWCONFIG
Current emulator configuration is:
    EL1600 Ethernet controller, version 1.01
    EL1600 1M Overlay, version 0.01
    EL1600 Dynamic T & B Board, version 0.01
    EL1600 68000 SCSI Shell(00), version 1.02
```

Related Commands

BUS

MEMVARS — display memory access variable values

Works with

■ EL 1600 ❒ CodeTAP

Syntax

```
MEMVARS
```

Description

This command displays the current values and descriptions for all the memory access variables.

Notes

The possible values for SPACE, SIZE, and OVERLAY are provided in the descriptions of each of these commands.

Example

```
memvars
                MEMORY ACCESS VARIABLES
                -----------------------
   Operation    Space    Size
   ---------    -----    ----
   CODE:          SP       1
   COMP1:         SD       1
   COMP2:         SD       1
   COPYFROM:      SD       1
   COPYTO:        SD       1
   DIAG:          SD       1
   FILL:          SD       1
   READ:          SD       1
   SEARCH:        SD       1
   STACK:         SD       1
   TEST:          SD       1
   WRITE:         SD       2
■
```

Related commands

EMUVARS, EVTVARS, XICEVARS

MWARN — control address out-of-range warnings

Works with

■ EL 1600 ❐ CodeTAP

Syntax

```
MWARN [ON|OFF]
```

Description

This switch is used to protect your target hardware from unwanted accesses that may be used by the emulator during target writes due to the inability of the 68000 and 68302 to access target memory with byte accesses. For example, if you request a word write to an odd address such as 0x1, the emulator will read the word at 0x0, OR in the new data value for address 0x1 and write it back. Then it will read the word at 0x2, or in the data value at 0x2, and write it back. If you wish to be informed when these types of accesses occur at the upper and lower boundaries, set this switch ON.

The default for MWARN is ON. If no argument is given, MWARN shows the current setting for the switch.

Notes

You can also set MWARN in XICE.CFG with SW_MWARN:ON or SW_MWARN:OFF.

Example

```
MWARN ON
```

NETERR — specify timeout warning delay

Works with

■ EL 1600 ❐ CodeTAP

Syntax

```
NETERR [seconds]
```

Description

NETERR is the approximate amount of time, in seconds, that XICE waits for a response from the emulator before issuing a warning message after a code packet has been sent during download.

Example

`NETERR 90`	delays 90 seconds before issuing a warning message.

See also

NETFAIL

NETFAIL — specify download abort timeout

Works with

■ EL 1600 ❐ CodeTAP

Syntax

```
NETFAIL [seconds]
```

Description

NETFAIL is the approximate amount of time, in seconds, that XICE waits for a response from the emulator before generating a timeout error and aborting the download process after a NETERR warning message has been displayed.

Example

`NETFAIL 90`	delays 90 seconds before aborting downloads

See also

NETERR

NULL_TGT — enable null target mode (68000/HC000/EC000)

Works with

■ EL 1600 ❒ CodeTAP

Syntax

```
NULL_TGT [ON|OFF|AUTO]
```

Description

This switch enables and disables null target mode. In null target mode probe tip signals to the target are disconnected and an internal clock is used. This mode allows you to operate the emulator without a target. The arguments cause the following actions:

ON	Enables null target mode.
OFF	Disables null target mode.
AUTO	Null target mode is selected automatically when emulator detects absence of target power.

The default for NULL_TGT is AUTO. If no argument is given, NULL_TGT shows the current setting for the switch.

Notes

Null target mode is available only for 68000/HC000/EC000 emulators which use a probe tip configuration, with no probe module. It is not available for the 68302 or for 68000's which use a probe tip/probe module configuration. See your *EL 1600 Hardware Setup and Reference Guide* to identify the configuration of your 68000.

If you invoke XICE with no target connected to the emulator the SW_NULL_TGT switch in the XICE.CFG should be set to ON or AUTO. XICE invocation will fail if you have no target or no target power and are not in null target mode.

When running code in null target mode, the overlay memory board must be set up to return DTACK. Use OVS to specify this operation.

You can also set NULL_TGT in XICE.CFG with SW_NULL_TGT:AUTO or SW_NULL_TGT:ON or SW_NULL_TGT:OFF.

Example

```
NULL_TGT ON
```

Related Commands

OVS

OVE — overlay memory spaces

Works with

■ EL 1600 ❐ CodeTAP

Syntax

```
OVE [0..0xFF]
```

Description

This switch specifies which memory spaces overlay responds to. Overlay can respond to multiple spaces. To specify multiple spaces, OR the masks given below to create a number that is between 0 and 0xFF.

Memory address space is processor-specific. The valid values for the 68000 and 68302 processors are as follows:

Value	Address Space	Description
0x01	SC0	Reserved memory space
0x02	UD or SC1	User data space
0x04	UP or SC2	User program space
0x08	SC3	Reserved memory space
0x10	SC4	Reserved memory space
0x20	SD or SC5	Supervisor data space
0x40	SP or SC6	Supervisor program space
0x80	CPU or SC7	CPU space

The default for OVE is 0x66, which represents the address spaces for Supervisor Program Space (0x40), Supervisor Data Space (0x20), User Data Space (0x02), and User Program Space (0x04) all ORed together (0x66).

If no argument is given, OVE shows the current setting for the switch.

Notes

You can also set OVE in XICE.CFG with SW_OVE:*value*.

Example

```
OVE 0x66
```

OVS — set emulator overlay speed

Works with

■ EL 1600 ❐ CodeTAP

Syntax

```
OVS [number]
```

Description

This switch specifies the number of wait states (from 0 to 7) to be inserted before the overlay memory supplies a DTACK signal to terminate the cycle.

The valid OVS settings are as follows:

0	DTACK supplied by target memory
1	No delay, address strobe returned to the processor as DTACK (DSACK)
2	+1 cycle delay
3	+2 cycles delay
4	+3 cycles delay
5	+4 cycles delay
6	+5 cycles delay
7	+6 cycles delay

The default for OVS is 0. If no argument is given, the state of the switch is displayed.

Notes

The CPU accepts the first DTACK it receives, either from the target or from the internally generated DTACK using the OVS setting.

The 68000 and the 68302 may not be able to run out of overlay memory without wait states since overlay RAM may not respond as quickly as target RAM. If you observe erratic overlay operation, set OVS to 2 or more to insert a delay.

The 20 MHz 68302 requires that OVS be set at 2 or greater. Otherwise, XICE may report illegal switches on initialization.

If you are using the emulator with a null target, you must set OVS to a non-zero value for proper operation with overlay memory.

You can also set the emulator overlay speed in XICE.CFG with SW_OVS:*n*, for example SW_OVS:1.

Example

```
OVS 1
```

Related Commands

MEMVARS

PERFACT — enable performance analysis data gathering

Works with

■ EL 1600 ❐ CodeTAP

Syntax

```
PERFACT [STATISTICAL|OFF]
```

Description

PERFACT turns statistical performance analysis data gathering on and off. If PERFACT is on, the emulator will periodically upload data during run, and process the data for display. The interval between data uploads is determined by the PERFINT switch.

Notes

For more information refer to Chapter 5, *Using Performance Analysis.*

Example

```
PERFACT STATISTICAL
```

Related Commands

PERFINT

PERFCLR — remove performance analysis data

Works with

■ EL 1600 ❒ CodeTAP

Syntax

```
PERFCLR
```

Description

PERFCLR purges all accumulated performance analysis data from the system. The event system and address exclusion setups are not disturbed.

Accumulated performance analysis data is automatically cleared when a LOAD is executed.

Notes

For more information refer to Chapter 5, *Using Performance Analysis*.

Example

```
PERFCLR
```

Related Commands

PERFDATA — display performance analysis symbol data

Works with

■ EL 1600 ❐ CodeTAP

Syntax

```
PERFDATA [symbol|string]
```

Description

PERFDATA displays the address range and number of samples for symbols which appear in the accumulated performance analysis data.

Note that the address range is derived from the trace data and is typically a subrange of the actual addresses for that function (as shown by PRINTSYMBOL, for example).

Notes

For more information refer to Chapter 5, *Using Performance Analysis.*

Example

```
PERFDATA main
main:
    Address range: 0x0000566E..0x00005700
    Hits: 5789
```

Related Commands

PERFDEPTH — maximum number of lines of PA output

Works with

■ EL 1600 ❐ CodeTAP

Syntax

```
PERFDEPTH [0...]
```

Description

When running Performance Analysis on a large program, the high number of symbols encountered can cause the PA display to exceed the depth of the screen. Quite often, many of the last symbols displayed are of little interest because they did not occur often. PERFDEPTH can be used to limit the display to only the more frequently encountered symbols.

The default is zero; all available lines display. Setting PERFDEPTH to any non-zero number limits the display to the specified number of lines.

Notes

You can also set the symbol display line limit in xice.cfg using the softswitch SW_PERFDEPTH:number.

For more information refer to Chapter 5, *Using Performance Analysis*.

Example

```
PERFDEPTH 20
```

PERFDISP — display performance analysis information

Works with

■ EL 1600 ❐ CodeTAP

Syntax

```
PERFDISP
```

Description

PERFDISP displays performance analysis information. The format of the display is specified by PERFFORMAT. If it is set to display all data, the display is in the format: *symbol: percentage of samples in function: number of samples in function: histogram*, where *symbol* is a function or subroutine in your program. Any of percentage, samples, or histogram may be left out, although at least one will always be present. For more information refer to Chapter 5, *Using Performance Analysis.*

Example

```
PERFDISP

Hits used: 40928  (40928 total, 0 excluded)
FUNCTION      PERCENT       HITS
-----------------------------------------------
func9:           30.8   (  12605) ***************
func8:           17.1   (   6999) ********
main:            16.6   (   6794) ********
func7:           12.4   (   5075) ******
func6:           11.1   (   4543) *****
func5:            6.2   (   2538) ***
func4:            3.1   (   1269) *
func3:            1.6   (    655) *
func2:            0.8   (    327) *
func1:            0.3   (    123) *
```

Related commands

PERFMODE, PERFFORMAT

PERFEX — exclude addresses from performance analysis

Works with

■ EL 1600 ❐ CodeTAP

Syntax

```
PERFEX [address|address range|symbol[#distance]]
```

Description

PERFEX excludes certain addresses from the performance analysis data. If an address or address range is specified, those addresses are excluded. If a symbol name is given, PERFEX searches forward for the next symbol and excludes up to that symbol. This can be used to exclude a function. Normally the limit for this search is given by PERFTOL, but it may be overridden with #distance. Exclusion ranges are automatically merged when they overlap or are contiguous.

If no arguments are given, PERFEX displays all exclusions in effect and the names of symbols within those exclusions.

For recommendations on using PERFEX effectively, see the chapter on using performance analysis in this supplement.

Notes

For more information refer to Chapter 5, *Using Performance Analysis.*

Example

```
PERFEX 0x5800..0x5900

PERFEX wait
Symbol getcom found at 0x00005704.Excluding from
0x0000566E to 0x00005703.

PERFEX
Current address exclusion ranges are:
    0x0000566E..0x00005703     wait
    0x00005800..0x00005900     sort, shuffle
```

Related Commands

PERFEXCLR, PERFTOL

PERFEXCLR — clear performance analysis exclusions

Works with

■ EL 1600 ❐ CodeTAP

Syntax

```
PERFEXCLR [address|address range|symbol[#distance]]
```

Description

PERFEXCLR clears address range exclusions set with PERFEX. If an address or address range is specified, any exclusion of those addresses is cleared. If a symbol name is given, PERFEXCLR searches forward for the next symbol and clears any exclusions up to that symbol. This can be used to clear the exclusion of a function. Normally the limit for this search is given by PERFTOL, but it may be overridden with #distance.

If no arguments are given, PERFEXCLR clears all exclusions in effect. All address range exclusions are automatically cleared when a LOAD is executed.

For more information refer to Chapter 5, *Using Performance Analysis.*

Example

```
PERFEXCLR 0x5800..0x5900

PERFEXCLR wait
Symbol getcom found at 0x00005704.Clearing from
0x0000566E to 0x00005703.
```

Related Commands

PERFEX, PERFTOL

PERFFORMAT — format of performance analysis display

Works with

■ EL 1600 ❐ CodeTAP

Syntax

```
PERFFORMAT [ST*ANDARD|PE*RCENT|HI*TS|BA*R|PH|PB|HB|-
PHB|ALL]
```

Description

PERFFORMAT governs the display of performance analysis data according to the chart below.

	percent	hits (samples)	histogram
PERCENT	■		
HITS		■	
BAR			■
PH	■	■	
PB	■		■
STANDARD	■		■
HB		■	■
PHB	■	■	■
ALL	■	■	■

Notes

For more information refer to Chapter 5, *Using Performance Analysis*.

Example

```
PERFFORMAT ALL
```

Related commands

PERFDISP, SW_PERFFMT_STAT

PERFINT — specify performance analysis time interval

Works with

■ EL 1600 ❒ CodeTAP

Syntax

```
PERFINT [1 - 120]
```

Description

PERFINT controls the time in seconds between uploads of performance analysis data from the emulator

The default for PERFINT is 3.

Notes

You can also set the performance analysis data gathering time interval in XICE.CFG with SW_PERFINT:*number*.

For more information refer to Chapter 5, *Using Performance Analysis*.

Example

```
PERFINT 3
```

PERFMODE — control performance analysis data display

Works with

■ EL 1600 ❐ CodeTAP

Syntax

```
PERFMODE [A*LWAYS | D*EMAND]
```

Description

PERFMODE controls whether performance analysis data is displayed every time it is uploaded from the emulator (ALWAYS), or stored for display at a later time (DEMAND), using PERFDISP.

The default for PERFMODE is DEMAND.

Notes

You can also set the performance analysis display mode in XICE.CFG with SW_PERFMODE:*option*.

For more information refer to Chapter 5, *Using Performance Analysis*.

Example

```
PERFMODE D
```

Related Commands

PERFDISP

PERFTOL — specify symbol search distance

Works with

■ EL 1600 ❐ CodeTAP

Syntax

```
PERFTOL [ 1... ]
```

Description

Because most addresses do not fall exactly on the beginning of a symbol, it is necessary to search backward to determine to which symbol a traced address belongs. PERFTOL specifies the maximum distance to search before giving up and labelling the address NO_SYMBOL.

PERFTOL also controls how far forward PERFEX will search when trying to exclude a symbol.

The default for PERFTOL is 2000. If no argument is given, the state of the switch is displayed.

Notes

You can also set the symbol search distance in XICE.CFG with SW_PERFTOL:*number*.

For more information refer to Chapter 5, *Using Performance Analysis*.

Example

```
PERFTOL 2000
```

Related Commands

PERFEX

PPT — peek/poke trace

Works with

■ EL 1600 ❐ CodeTAP

Syntax

```
PPT [OFF|ON]
```

Description

This switch controls the tracing of emulator peek/poke cycles made while in PAUSE mode. If PPT is set to ON, peek/poke cycles while in PAUSE are traced. If PPT is set to OFF, peek/poke cycles while in PAUSE are not traced.

When PPT is ON, XICE may be unable to perform trace disassembly in certain circumstances. This switch can, however, be useful for capturing cycles generated during a DIAG test.

The default for PPT is OFF. If no argument is given, the state of the switch is displayed.

Notes

You can also set PPT in XICE.CFG with SW_PPT:ON or SW_PPT:OFF.

If this switch is ON, the DT, DTB, and DTF commands could give erroneous information.

Example

```
PPT OFF
```

Related Commands

DTB, DTF

RAMACCESS — locate a range of RAM memory

Works with

■ EL 1600 ❐ CodeTAP

Syntax

```
RAMACCESS [address|address_range] [{, | =}
{COPY|MAP|TARGET|UNKNOWN} [=bank_range]
```

Description

This command specifies a range of memory locations that can be accessed during execution of the target program. If no parameters are specified, the memory map is displayed in the command viewport. The MAP and COPY options map the specified range to the emulator's overlay memory. TARGET and UNKNOWN map memory to target or as unknown.

Mappings have a 2K minimum granularity. If the beginning and end of an attempted mapping do not fall on 2K boundaries, the emulator automatically adjusts the start and end addresses and returns a warning that the mapping has been adjusted.

Example

`ram 0x1000..0x1fff=target`	maps the range to target memory
`ram 0x2000..0x2fff=copy`	copies contents of the range from target memory to emulator overlay memory

Related Commands

ROMACCESS

RESET — reset processor and target to initial state

Works with

■ EL 1600 ❐ CodeTAP

Syntax

```
RESET
```

Description

The XICE **RESET** command asserts both HALT and RESET simultaneously to the microprocessor in the probe module or on the probetip. Once in pause mode, the processor executes a RESET instruction that resets the external target hardware.

If you want to reset the external target hardware without resetting the processor, you must execute a RESET instruction in code. Or you can install a reset button on your target for this purpose.

Since memory is not re-initialized upon use of the **RESET** command, variables are not reset to their original values.

Notes

None

Example

```
RESET
```

Related Commands

RESTART

RFS — control software refresh

Works with

■ EL 1600 ❐ CodeTAP

Syntax

```
RFS [ON|OFF]
```

Description

This switch enables memory refreshes during PAUSE mode. If RFS is set to ON, memory is refreshed during PAUSE mode. If RFS is set to OFF, memory is not refreshed during PAUSE mode.

If you set RFS to ON, you must also specify the memory area to be refreshed using the following commands:

RFSADR	specifies the memory area to be refreshed
RFSMSK	specifies any mask to be applied to the memory area to be refreshed
RFSASP	specifies the memory space to be refreshed

If you change the values for any of the above switches, the change takes effect immediately.

The default for RFS is OFF. If no argument is given, the state of the switch is displayed.

Notes

The switch PPT specifies tracing peek/poke cycles during PAUSE mode. Setting both RFS and PPT to ON will result in incorrect trace information.

You can also set RFS in XICE.CFG with SW_RFS:ON or SW_RFS:OFF.

Example

```
RFS OFF
```

Related Commands

RFSADR, RFSASP

RFSADR — refresh software addresses

Works with

■ EL 1600 ❐ CodeTAP

Syntax

```
RFSADR [address]
```

Description

This switch specifies the address for memory refreshing during PAUSE mode. It is only active if RFS is set to ON. The related command RFSASP specifies the address space for memory refreshes and RFSMSK specifies any masks to be applied.

The default address for RFSADR is 0x0. If no argument is given, the state of the switch is displayed.

Notes

You can also set RFSADR in XICE.CFG with SW_RFSADR:*address*.

Example

```
RSFADR 0x0
```

Related Commands

RFSASP, RFSMSK

RFSASP — refresh software address space

Works with

■ EL 1600 ❐ CodeTAP

Syntax

```
RFSASP [space]
```

Description

This switch specifies the address space for memory refreshing during PAUSE mode. It is only active if RFS is set to ON. The related command RFSADR specifies the address for refreshes and the command RFSMSK specifies any mask on the refresh.

The 68000 and 68302 address spaces are as follows:

Code	Address Space	Description
0	SC0	Reserved memory space
1	UD or SC1	User data space
2	UP or SC2	User program space
3	SC3	Reserved memory space
4	SC4	Reserved memory space
5	SD or SC5	Supervisor data space
6	SP or SC6	Supervisor program space
7	CPU or SC7	CPU space

The default space for RFSASP is 5. If no argument is given, the state of the switch is displayed.

Notes

You can also set RFSASP in XICE.CFG with SW_RFSASP:[*space*].

Example

```
RFSASP 5
```

Related commands

RFSADR, RFSMSK

RFSMSK — refresh software mask

Works with

■ EL 1600 ❐ CodeTAP

Syntax

```
RFSMSK [address_mask]
```

Description

This switch specifies the address mask for memory refreshing during PAUSE mode. It is only active if RFS is set to ON. The related command RFSASP specifies the address space for refreshes and RFSADR specifies the address for refreshes.

The default mask for RFSMSK is 0. If no argument is given, the state of the switch is displayed.

Notes

You can also set RFSMSK in XICE.CFG with SW_RFSMSK:*address_mask*.

Example

```
RFSMSK 0x0
```

Related Commands

RFS, RFSADR, RFSASP

RIRR — control register restoration on reset (68302 only)

Works with

■ EL 1600 ❐ CodeTAP

Syntax

```
RIRR [ON|OFF]
```

Description

This switch controls the restoration of the CPU's chip select and control registers (SCR, BAR, WRR, BR0, OR0, BR1, OR1, BR2, OR2, BR3, and OR3). If RIRR is set to ON, the emulator's internal copy of the chip select and control registers is written to the CPU whenever you use the command RESET to reset the emulator. If RIRR is set to OFF, internal registers are not restored to the CPU on RESET.

The default for RIRR is ON. If no argument is given, the state of the switch is displayed.

Notes

Timer and serial communication controller registers may be saved depending on the setting you use for the switch DRTMR.

You can also set RIRR in XICE.CFG with SW_RIRR:ON or SW_RIRR:OFF.

This command is not used by XICE for the 68000.

Example

```
RIRR OFF
```

Related Commands

DRTMR, UIR

ROMACCESS — locate a range of ROM memory

Works with

■ EL 1600 ❐ CodeTAP

Syntax

```
ROMACCESS [address|address_range] [{, | =}
{COPY|MAP|TARGET|UNKNOWN} [=bank_range]
```

Abbreviation

ROM

Description

This command specifies a range of memory locations that cannot be written to during execution of the target program. If no parameters are specified, the memory map is displayed in the command viewport. The MAP and COPY options map the specified range to the emulator's overlay memory. TARGET and UNKNOWN map memory to target or as unknown.

Mappings have a 2K minimum granularity. If the beginning and end of an attempted mapping do not fall on 2K boundaries, the emulator automatically adjusts the start and end addresses and returns a warning that the mapping has been adjusted.

Example

`rom 0x1000..0x1fff=target`	maps the range to target memory
`rom 0x2000..0x2fff=copy`	copies contents of the range from target memory to emulator overlay memory

Related Commands

RAMACCESS

RUN_POLL — set number of polls per second during run

Works with

■ EL 1600 ❒ CodeTAP

Syntax

```
RUN_POLL [n]
```

Description

This command controls how many times per second the emulator is polled while in RUN mode. Valid values are 1-20.

The default for RUN_POLL is 5. If no arguments are given, the current value is displayed.

Notes

A lower number will slightly reduce response time, but will also reduce network traffic and CPU load. You can set polls per second in XICE.CFG with SW_RUN_POLL:*num*.

Example

`RUN_POLL 10` causes 10 polls per second

Related Commands

XICEVARS

RUN_TIME — set maximum run time

Works with

■ EL 1600 ❐ CodeTAP

Syntax

```
RUN_TIME [n]
```

Description

This command controls the maximum time (in seconds) the emulator will be allowed to run before emulation is broken.

The default is 0, which will allow the emulator to stay in run indefinitely. If no arguments are given, the current value is displayed.

Notes

RUN_TIME controls the time the emulator is actually running. When PERFACT is set to STATISTICAL, the emulator is not always in run due to the overhead associated with performance analysis. Therefore the elapsed time may be several seconds longer than the actual run time.

Example

RUN_TIME 3	causes emulator to run for 3 seconds before breaking

SCRATCH — breakpoint scratch area address

Works with

■ EL 1600 ❒ CodeTAP

Syntax

```
SCRATCH [address]
```

Description

This command specifies the starting address in RAM for the 8 bytes of scratch memory in the supervisor program space needed for software breakpoints. This area must be specified in order to use execution breakpoints and it must be an area that is unused by the program being debugged. If no argument is given, the current address is displayed.

If you do not have any spare RAM in the target system, you may set the scratch space in unused memory and map overlay memory to that area.

When XICE is invoked, it performs a read of the area designated for SCRATCH if SCRATCH is specified. If SCRATCH is set to an area of memory that does not return a DTACK at the end of the read, it will hang XICE. In such a case, comment out the default address for SCRATCH (0x9ff0) in the *xice.cfg* file. Then specify the address for SCRATCH using this command before setting any breakpoints, or create an include file or start-up file for invoking XICE.

Notes

You can also set SCRATCH in XICE.CFG with SW_SCRATCH.

Example

```
SCRATCH 0x0
```

SIA — special interrupt vector

Works with

■ EL 1600 ❐ CodeTAP

Syntax

```
SIA [address]
```

Description

This command specifies the address of the special interrupt vector for forced special interrupts, which are one of the actions that an event system trigger may use.

The default address for SIA is 0. If no argument is given, the current address is displayed.

Notes

You can also set the special interrupt vector in XICE.CFG with SW_SIA:*address*.

Example

```
SIA 0xFFFF00
```

SIZE — set the size for memory accesses

Works with

■ EL 1600 ❐ CodeTAP

Syntax

```
SIZE [<memory_access_type>] [1|2|4]
```

Description

This command allows you to examine and set the size to be used for memory accesses. The memory access type you specify must be one of the following:

CODE	All accesses of code, including fetches
COMP1	Memory for first argument of COMPARE
COMP2	Memory for second argument of COMPARE
COPYFROM	Source memory for a COPY
COPYTO	Destination memory for a COPY
DIAG	Memory to use with DIAG
FILL	Memory for use with FILL
READ	Memory for generic reads (DUMP, CRC, etc.)
SEARCH	Memory for use with SEARCH
STACK	Memory accesses for the stack
TEST	Memory for use with TEST
WRITE	Memory for generic writes (SETMEM, etc.)

The code for the sizes are as follows:

1	1 byte
2	2 bytes (word)
4	4 bytes (long)

If you do not specify a size argument, the current state of SIZE for the specified memory access type is displayed. If you do not specify a memory access type, the current state of SIZE for all memory access types is displayed.

Notes

You can also set SIZE in XICE.CFG with SW_SIZE:"[<*memory_access_type*>] [1|2|4]". For example, SW_SIZE:"CODE 2" would set the size for all accesses of code, including fetches, to 2 bytes (word) long. Note that you must enclose the two arguments in double quotes, e.g., SW_SIZE:"CODE 2".

Example

```
SIZE CODE 4
```

Related Commands

MEMVARS

SLO — slow interrupt emulation control

Works with

■ EL 1600 ❐ CodeTAP

Syntax

```
SLO [ON|OFF]
```

Description

This switch enables inserting a delay before allowing an interrupt when entering RUN mode. If SLO is set of ON, XICE will insert a 160 clock cycle delay before allowing an interrupt. If you set both FAST and SLO to ON, FAST takes precedence, which means that interrupts are enabled immediately upon entering RUN mode. If SLO is set to OFF, the setting for FAST determines whether interrupts are enabled.

The default for SLO is OFF. If no argument is given, the current address is displayed.

Notes

You can also set SLO in XICE.CFG with SW_SLO:ON and SW_SLO_OFF.

If you set both FAST and SLO to ON, FAST has precedence over SLO. The following table shows the results for the possible switch setting combinations for FAST and SLO. This table applies to target-generated interrupts passed to the target processor when the emulator is running.

SLO	FAST	Result While in RUN Mode	Result While in PAUSE Mode
ON	ON	Interrupts immediately enabled.	Interrupts immediately enabled upon return to RUN mode.
ON	OFF	Interrupts enabled after approximately160 clock cycles.	Interrupts enabled after approximately160 clock cycles after return to RUN mode.
OFF	ON	Interrupts immediately enabled.	Interrupts immediately enabled upon return to RUN mode.
OFF	OFF	Interrupts generated by the target system will be inhibited from reaching the emulator.	Interrupts generated by the target system will be inhibited from reaching the emulator.

Example

```
SLO ON
```

Related commands

FAST

SPACE — set the space for memory accesses

Works with

■ EL 1600 ❐ CodeTAP

Syntax

```
SPACE [<memory_access_type>] [address_space]
```

Description

This command allows you to examine and set the space to be used for memory accesses. The memory access type you specify must be one of the following:

CODE	All accesses of code, including fetches
COMP1	Memory for first argument of COMPARE
COMP2	Memory for second argument of COMPARE
COPYFROM	Source memory for a COPY
COPYTO	Destination memory for a COPY
DIAG	Memory to use with DIAG
FILL	Memory for use with FILL
READ	Memory for generic reads (MEMGET, CRC, etc.)
SEARCH	Memory for use with SEARCH
STACK	Memory accesses for the stack
TEST	Memory for use with TEST
WRITE	Memory for generic writes (MEMSET, etc.)

If you do not specify an address space, the current state of SPACE for the specified memory access type is displayed. If you do not specify a memory access type, the current state of SPACE for all memory access types is displayed.

Notes

Memory address space is processor-specific. The valid values for the 68000 and 68302 processors are as follows:

Address Space	Description
SC0	Reserved memory space
UD or SC1	User data space
UP or SC2	User program space
SC3	Reserved memory space
SC4	Reserved memory space
SD or SC5	Supervisor data space
SP or SC6	Supervisor program space
CPU or SC7	CPU space

You can also set SPACE in XICE.CFG with SW_SPACE:"[<*memory_access_type*>] [*address_space*]". For example, SW_SPACE:"CODE UP" would set the size for all accesses of code, including fetches, to 2 bytes (word) long. Note that you must enclose the two arguments in double quotes, e.g., SW_SIZE:"CODE SC2".

Example

```
SPACE READ UP
```

Related commands

MEMVARS

STI — enable or disable step-through interrupts

Works with

■ EL 1600 ❒ CodeTAP

Syntax

```
STI [ON|OFF]
```

Description

This switch enables or disables step-through interrupts. If STI is ON, the emulator will recognize an interrupt during a STEP operation and STEP through the interrupt service routine. If STI is OFF, the emulator will ignore interrupts during a STEP operation.

The default for STI is OFF. If no argument is given, the state of the switch is displayed.

Notes

You can also set STI in XICE.CFG with SW_STI:ON or SW_STI:OFF.

Example

```
STI OFF
```

TAD — control tri-state of address bus

Works with

■ EL 1600 ❒ CodeTAP

Syntax

```
TAD [ON|OFF]
```

Description

This switch specifies whether the address bus is tri-stated while in PAUSE mode. If TAD is set to ON, the address bus is tri-stated while the emulator is PAUSED and during peeks and pokes. If TAD is set to OFF, addresses generated during PAUSE mode are output to the target system.

The default for TAD is OFF. If no argument is given, the state of the switch is displayed.

Notes

You can also set TAD in XICE.CFG with SW_TAD:ON or SW_TAD:OFF.

Example

```
TAD ON
```

TCEBRK — control tracing of breakpoints

Works with

■ EL 1600 ❐ CodeTAP

Syntax

```
TCEBRK [ON|OFF]
```

Description

This switch controls the capture of software instruction breakpoint execution in the trace buffer. If TCEBRK is set to ON, the breakpoint cycles are recorded in the trace and are marked with an X in the column FLAGS. If TCEBRK is set to OFF, the breakpoint cycles are not recorded in the trace.

The default for TCEBRK is OFF. If no argument is given, the state of the switch is displayed.

Notes

You can also set TCEBRK in XICE.CFG with SW_TCEBRK:ON or SW_TCEBRK:OFF.

Example

```
TCEBRK OFF
```

TED — control trace/overlay for external DMA (68302 only)

Works with

■ EL 1600 ❐ CodeTAP

Syntax

```
TED [ON|OFF]
```

Description

This switch enables or disables tracing external DMA cycles (which are generated by the target hardware following the BR, BGF, or BGACK protocol) and enables or disables external DMA accesses to overlay. If TED is ON, external DMA cycles are traced and external DMA can access overlay. If TED is OFF, external DMA cycles are not traced and external DMA cannot access overlay. Also, if TED is OFF external DMA cycles cannot access internal registers or dual-port locations.

The default for TED is OFF. If no argument is given, the state of the switch is displayed.

Notes

The switch TID enables tracing of internal DMA cycles. External DMA can run while the emulator is in PAUSE mode; however, these cycles will not be traced, nor may they access overlay, regardless of the TED setting.

You can also set TED in XICE.CFG with SW_TED.

Example

```
TED OFF
```

Related commands

TID

TID — control trace/overlay for internal DMA (68302 only)

Works with

■ EL 1600 ❐ CodeTAP

Syntax

```
TID [ON|OFF]
```

Description

This switch enables or disables tracing internal DMA cycles and enables or disables internal DMA access to overlay. If TID is ON, internal DMA cycles are traced and internal DMA can access overlay. If TID is OFF, internal DMA cycles are not traced and internal DMA cannot access overlay.

The default for TID is OFF. If no argument is given, the state of the switch is displayed.

Notes

The switch TED enables tracing of internal DMA cycles.

You can also set TID in XICE.CFG with SW_TID:ON or SW_TID:OFF.

This command is not used by XICE for the 68000.

Example

```
TID OFF
```

Related commands

TED

TRCCLR — clear trace buffer

Works with

■ EL 1600 ❐ CodeTAP

Syntax

```
TRCCLR
```

Description

This command deletes the information in the trace buffer.

Example

```
TRCCLR
```

TRCFRAME — trace cycle number

Works with

■ EL 1600 ❐ CodeTAP

Syntax

```
TRCFRAME [cycle_number]
```

Description

This command specifies the trace cycle number to be the time 0 for offset timestamps in a raw trace display.

The default cycle is cycle 0. If no argument is given, the state of the switch is displayed.

Notes

You can also set TRCFRAME in XICE.CFG with SW_TRCFRAME: *cycle_number*.

Example

TRCFRAME 125	Cycle 125 will be time 0 in the next raw trace display.

Related commands

XICEVARS

TRCINT — trace interval

Works with

■ EL 1600 ❐ CodeTAP

Syntax

```
TRCINT [INTERVAL|OFFSET]
```

Description

This command specifies how timestamps are displayed in raw trace. If TRCINT is set to INTERVAL, timestamps are displayed as the time interval between successive bus cycles. If TRCINT is set to OFFSET, timestamps are the time relative to the bus cycle number specified by the command TRCFRAME.

The default for TRCINT is OFFSET. If no argument is given, the state of the switch is displayed.

Notes

You can also set TRCINT in XICE.CFG with SW_TRCINT:INTERVAL or SW_TRCINT:OFFSET.

Example

```
TRCINT INTERVAL
```

Related Commands

XICEVARS

TRCMODE — trace mode

Works with

■ EL 1600 ❒ CodeTAP

Syntax

```
TRCMODE [ASM|SRC|BOTH]
```

Description

This command specifies the type of information displayed by the commands **DT**, **DTB**, and **DFT**. The default is BOTH. The valid arguments are:

ASM	displays assembly instructions only
SRC	displays source level instructions only
BOTH	displays source and assembly instructions interleaved together

Notes

You can also set TRCMODE in XICE.CFG with SW_TRCMODE:ASM, SW_TRCMODE:SRC, or SW_TRCMODE:BOTH.

Example

```
TRCMODE ASM
```

Related Commands

DT, DTB, DTF, XICEVARS

TRIG — set status trigger

Works with

■ EL 1600 ❒ CodeTAP

Syntax

```
trig{n} = [event{n},action|CLEAR|ARM|DISARM]
```

Description

The TRIGGER command defines the action for XICE to take after an event or a number of events is true. A trigger may be used to control event system resources (timers, counters, etc.), tracing, breaking, or other analyzers. A trigger must be armed for any action to be taken.

The following general rules relate to setting a trigger:

- A trigger may list up to 8 events separated by vertical bars (|'s).

  E.g., `trig{1} = ev{1} | ev{32}, grp2` sets a trigger

  so that any time either event 1 or event 32 is true, XICE will switch its monitoring to only group 2 events.
- Triggers are active only when they have been armed, either explicitly by using the command `trig {n} = arm`, or automatically when they are defined if the XICE switch EVTARM is set to ON.
- If arming a trigger fails (for example, if there are not enough resources to fulfill the request), the trigger will be defined but not armed.
- While armed, you cannot change the definition of the trigger or the event(s) specified by the trigger.

- In any one event group, you may not arm triggers using more than the following comparators:
  - 2 address comparators (specifying an address range counts as a single address comparator)
  - 2 data comparators
  - 2 status comparators
  - 1 LSA comparator
  - 1 counter
- You cannot arm triggers for an event breakpoint if you are already using BA breakpoints in your code.
- A given trigger `trig {n}` can be active (armed) in only one group at a time. However, an event can be used in more than one trigger and more than one group at a time.
- By not specifying a trigger number for an arm or disarm command, all defined triggers can be armed or disarmed at once. E.g., trig = arm will arm all the triggers that have been set.
- You cannot use the TRIG command to set up a break followed by a macro. However, you can set the switch SW_EVTMODE in XICE.CFG to ON and use the command BREAKCOMPLEX (BC) to do so. When EVTMODE is ON, however, you will not be able to use break as an action in TRIG commands.

To change the group that triggers will be armed in, use the EVTGRP command. To list the status of one or more triggers, use the STATUS command. To clear all the triggers at once, use the EVTCLR command.

Possible actions that can be specified using the TRIG command are:

Action	Definitions
BREAK	Breaks emulation as a result of specified conditions, or by default if no action is requested.
CNT	Count only qualified cycles.
FSI	Force a special interrupt. FSI provides a way to jump to a specified address when a specific event is detected. It can allow you to patch your code fast and can also allow you to write soft shutdown routines for machinery that cannot be halted using a simple breakpoint. You must set up the SIA (special interrupt address) switch prior to using an FSI action. You may also see some unusual cycles in the trace memory at the address where the FSI occurred. These are internal cycles that are traced as the execution address is changed. These internal cycles are not purged from the trace memory. The FSI routine residing at the SIA address should terminate with a return from exception (RET) instruction. Execution resumes at the address immediately following the instruction that caused the FSI. If this is a soft shutdown, you will probably define a breakpoint at the RET instruction.
GRP1	Switch to group 1 events.
GRP2	Switch to group 2 events.
GRP3	Switch to group 3 events.
GRP4	Switch to group 4 events.

Action	Definitions
RCT	Resets the counter.
TGR	The trigger signal is an output that is available from the BNC connector on the back panel of the emulator chassis and from pin 19 of the optional LSA pod. When a trigger event is detected, the trigger signal is asserted and remains so for the duration of the specified bus cycle. If a trigger event is specified for more than one consecutive bus cycle, the signal stays high for the duration of the consecutive bus cycles. You can use the trigger signal as a pulse for triggering other diagnostic equipment. You can also use it in conjunction with a counter or timer for timing subroutines or use it with the optional timestamp pod for timing subroutines. See the command TUNITS for further information on using the optional timestamp capability.
TOC	Toggle the current setting for the counter. No count until first TOC.
TOT	Toggle the current setting for tracing or not tracing. Trace uses the following rules: If you do not specify TOT, trace is on. If you specify TOT, when you change event groups, trace goes off and when it encounters the next TOT, it goes ON.If you specify TOT, every time the emulator goes into RUN, trace is OFF until the first TOT is encountered.
TRC	Trace this bus cycle.

Examples

```
TRIGGER{1} = ev{4} | ev{5}, TOT
TRIGGER{2} = ev{6}, TOT
```

The above command for trigger 1 toggles the current setting for trace when either `event{4}` or `event{5}` is true. The command for trigger 2 toggles the setting for trace when `event{6}` occurs.

Related commands

BC, EV, EVTARM, EVTCLR, EVTGRP, EVTMODE, STATUS

TSRCH — search trace memory for patterns

Works with

■ EL 1600 ❐ CodeTAP

Abbreviation

TS

Syntax

```
TSRCH [trace range],[addr=value],[data=
value],[stat=value],[lsa=value]
```

Description

Lets you search trace memory for a specified pattern. The syntax is similar to that for the "EV" command, with an optional starting point, or search range. Output is in "DRT" format.

The "value" can be a simple value (0x1000), a range (0x1000..0x2000), or a value with a "care" mask (0x1000&=0xf000).

For the status comparator, values can also be entered mnemonically using the mnemonics recognized by the event system. Mnemonics can be logically combined. See the EV command description on page 2-71 for valid status mnemonics.

Examples

ts addr=0x1000	Search for all occurrences of address 0x1000
ts 200,addr=0x1000..0x2000,data=0x300	Start at cycle 200, search for cycles with address range of 1000 to2000, and data = 0x300
ts 200..300,stat=sd|wr	Search traces cycles 200 to 300 for writes to supervisor data space
ts lsa=0x100&=0x300	Search for bit 8 = 1, bit 9 = 0 in the LSA field, ignore other bits

Related Commands

DRT, EV

TSTAMP —show timestamp or LSA in trace

Works with

■ EL 1600 ❐ CodeTAP

Syntax

```
TSTAMP [ON|OFF]
```

Description

This switch controls whether the raw trace display reports timestamp or LSA data. If you set TSTAMP to ON, the DRT command shows timestamp information. If you set TSTAMP to OFF, the DRT command shows LSA information.

If you have a timestamp unit attached to your emulator, you should set TSTAMP to ON; otherwise, it should be set to OFF.

The default for TSTAMP is OFF. If no argument is given, the state of the switch is displayed.

Notes

You can also set TSTAMP in XICE.CFG with SW_TSTAMP:ON or SW_TSTAMP:OFF.

For more information about using time stamp, see Chapter 6.

Example

```
TSTAMP OFF
```

Related Commands

DRT, XICEVARS

TUNITS —timestamp units

Works with

■ EL 1600 ❐ CodeTAP

Syntax

```
TUNITS [0x0..0xF]
```

Description

This switch determines the timestamp units displayed in the raw trace display. You should set this switch if you have a timestamp module attached to your emulator and you have set TSTAMP to ON. Otherwise, you should leave it set to 0.

The valid values for TUNITS are as follows:

Unit	Time Base	Effect of TGR on Timestamp Counter	Useful Measurements
0x0	.1 µs	Any TGR high causes the timestamp counter to be reset to 0. No manual reset is required in this mode for either absolute or relative timestamping.	Elapsed time
0x1	1 µs		
0x2	.01 ms		
0x3	.1 ms		
0x4	1 ms		
0x5	.1 µs	While the TGR is held high by the Event Monitor System, the timestamp counter counts. Manual reset is required in this mode for absolute timestamping, but not for relative timestamping.	Elapsed time
0x6	1 µs		
0x7	.01 ms		
0x8	.1 ms		
0x9	1 ms		

Unit	Time Base	Effect of TGR on Timestamp Counter	Useful Measurements
0xA	.1 µs	In this mode, a long TGR signal from the Event Monitor System resets the counter. After that, successive short TGR signals turn the counter on and off. Manual reset stops the counter and sets it to zero	Elapsed time
0xB	1 µs		
0xC	.01 ms		
0xD	.1 ms		
0xE	1 ms		
0xF	NA	This setting is used to count occurrences. Each time the TGR signal goes high, the timestamp counter is incremented. Manual reset is required	Count occurrences

The default for TUNITS is 0. If no argument is given, the state of the switch is displayed.

Notes

The setting that you use for TUNITS must match the settings you use for the physical switches on the timestamp module itself.

You can also set TUNITS in XICE.CFG with SW_TUNITS:*units*.

For more information about using time stamp, see Chapter 6.

Example

```
TUNITS 0x0
```

Related Commands

XICEVARS

UIR — update internal chip select registers (68302 only)

Works with

■ EL 1600 ❐ CodeTAP

Syntax

```
UIR [ON|OFF]
```

Description

This switch controls whether the emulator's copy of the CPU's internal chip select and control registers are automatically updated. Each time the emulator makes a transition from RUN to PAUSE mode, the internal registers SCR, WRR, BAR, BR0, OR0, BR1, OR1, BR2, OR2, BR3, and OR3 are automatically updated. If UIR is set to OFF, the EL 1600's internal copy of the chip select registers are not automatically updated.

If you modify the chip select registers while the emulator is paused, you must have UIR set to ON to have the modifications update the internal copy of the registers. IF UIR is ON, the modified version of the internal chip select copy will be loaded into the 68302 chip select registers after each RESET command until either they are modified during a RUN, in which case these modifications will be saved during the RUN to PAUSE transition, or until you change the setting for UIR to OFF.

The default for UIR is ON. If no argument is given, the state of the switch is displayed.

Notes

We recommend leaving UIR set to ON.

Timer and serial communication controller registers may be saved depending on the setting you use for the switch DRTMR.

See also the information in section 1 on configuring the chip select registers and the command description for RIRR. RIRR controls resetting the internal chip select registers after a reset.

You can also set UIR in XICE.CFG with SW_UIR:ON or SW_UIR:OFF.

This command is not used by XICE for the 68000.

Example

```
UIR ON
```

Related commands

DRTMR, RIRR

UP — move the current scope

Works with

■ EL 1600 ❐ CodeTAP

Syntax

```
UP [number_of_levels]
```

Description

The UP and DOWN commands allow you to move the current scope up or down the runtime stack. This is especially helpful when debugging recursive functions. It is not a good idea to go down farther than you have gone up.

Example

```
UP 5
```

Related commands

DOWN

UPL — upload hex data to host

Works with

■ EL 1600 ❐ CodeTAP

Syntax

```
UPL "filename",address_range
```

Description

UPL is used to upload data from the target to a host file in the format specified by UPLFMT. The address range is the address range of the data to be uploaded.

The MAP, OVERLAY, SPACE, and SIZE commands affect how memory is accessed by UPL.

Notes

Quotation marks are optional if the file name consists of alphanumeric characters or a period. File names that contain a leading slash must be in double quotation marks (e.g., "/root"). File names that contain a leading backslash must be in single quotation marks (e.g., '\root').

Example

```
UPL my.file ,0x8000..+0x3fff
```

Related Commands

DNL, DNLFMT, UPLFMT, MAP, OVERLAY, SIZE, SPACE, VERIFY

UPLFMT — specify upload format

Works with

■ EL 1600 ❐ CodeTAP

Syntax

UPLFMT *format*

Description

UPLFMT is used to specify the format for hex file uploads using the UPL command. Recognized formats are:

INTEL	Intel hex format. Extended segment address records and extended linear address records are supported.
SREC	Motorola S3-records with Microtec extensions.
XTEK	Extended Tektronics hex format.

Notes

The default format is SREC. The command XICEVARS displays the status of UPLFMT and all other XICE variables. Symbols are not supported for these formats.

Example

```
UPLFMT AMC
```

Related Commands

DNL, DNLFMT, UPL

VERIFY —memory read-after-write verify switch

Works with

■ EL 1600 ❐ CodeTAP

Syntax

```
VERIFY [ON|OFF]
```

Description

This switch enables or disables memory read-after-write verification. If VERIFY is set to ON, memory is verified after being written. If VERIFY is set to OFF, memory is not verified after being written.

The default for VERIFY is ON. If no argument is given, the state of the switch is displayed.

Notes

You can also set VERIFY in XICE.CFG with SW_VERIFY:ON or SW_VERIFY:OFF.

Example

```
VERIFY ON
```

Related commands

XICEVARS

XICEVARS — display internal debugger variable values

Works with

■ EL 1600 ❐ CodeTAP

Syntax

```
XICEVARS
```

Description

This command displays the current values and descriptions for all the internal debugger variables.

Example

```
                        XICE INTERNAL VARIABLES
                        -----------------------
BPSPACE   ANY         Set processor space in which inst. breakpoints will be set
DNLFMT    SREC        External file format for hex/binary downloads
DNL_GAP   1           Number of bytes to allow between cached download blocks
UPLFMT    SREC        External file format for hex/binary uploads
EVTARM    ON          Arm (ON)* vs. do not arm (OFF) triggers automatically
EVTGRP    1           Event group to use when arming triggers. Default: 1
RUN_POLL  5           Frequency of host polling of emulator (1-20). Default: 5
RUN_TIME  0           Maximum time to stay running (0: forever)
TRCFRAME  0           Cycle number for time 0 timestamp alignment. Default: 0
TRCINT    OFFSET      Display of raw trace timestamps: (INTERVAL) vs. (OFFSET)*
TRCMODE   BOTH        Trace display is assembly (ASM), source (SRC), or both (BOTH)*
VERIFY    ON          Memory verification enabled (ON) vs. disabled (OFF)*
PERFACT   OFF         If ON, P.A data will be collected and processed.
PERFFORMATOFF          Format of P.A. data
PERFMODE  DEMAND      Display P.A. data whenever uploaded or  display on demand
PERFDEPTH0            How many lines of P.A. output to display

TSTAMP    OFF         Interpret raw trace field as timestamp (ON) or LSA (OFF)*
TUNITS    0           Timestamp units for raw trace display (default of 0)
■
```

Related commands

EMUVARS, EVTVARS, MEMVARS

Chapter 3
XICE Tutorial

Overview

This chapter introduces the XICE debugger. It covers the basics needed to prepare the sample code for an embedded system application, introduces the new user to the XICE debugger interface, and demonstrates the use of many commands commonly used in a debug session.

User-entered commands

Throughout the tutorial, commands that you should enter are prefaced by a **">"** prompt. Examples, headed by **For example only**, should not be entered by the user during the tutorial.

Example of a command for you to enter

```
>context
```

XICE allows most commands to be abbreviated. The abbreviated command is used whenever possible.

An example of the abbreviated form of the "context (con)" command

```
>con
```

All the XICE commands used in the tutorial are covered in depth in the *XICE Supplement* or the *XRAY Documentation Set for 68xxx Family.*

Tutorial program

The program *cdemon.x* (SUN version) or *cdemon.abs* (PC version) is used throughout the tutorial. Further references to the tutorial code will use the SUN version, *cdemon.x*. If you are using a PC, substitute *cdemon.abs* for *cdemon.x* references.

The *cdemon.x* program, located in the *demo* directory, is compiled and ready to be loaded by the debugger, allowing you to skip the tools section and go directly to the XICE section if you wish to do so.

Embedded systems considerations

Developing code for the embedded system environment, as opposed to the native operating system environment, is in some ways analogous to leaving home for the first time. Many of the chores that may have been taken care of by parents, like laundry or dishes, must now be done without such support. Likewise, in the embedded systems environment, many of the functions that were performed by the operating system (locating code in memory, communicating with I/O) must now be taken care of by the programmer. These, and other, embedded systems considerations are treated in depth in the "Embedded Environments" chapter of the MCC compiler manual and the "Software Development Cycle" section of the ASM68K manual.

Preparing code for debugging

Environment variables

Before invoking the compiler, assembler, and linker ensure the path and environment variables have been set up. To check the path and environment variables you can type **env** if your are on a UNIX workstation or type **set** if you are using a PC. If they are not set up or you are not sure they are correct, please refer to the *XICE Installation Guide* and set them up at this time.

Makefile for cdemon demonstration code

The Sun makefile, named *makefile*, for *cdemon.x* is located in the *demo* directory. It shows the flags (options) necessary to produce the symbolic information used by XICE to display source, evaluate expressions and symbolic references, and to display type information.

The makefile also generates the linker command file, *ieee.cmd*, which includes flags for placing symbolic information in the object module, designating output file format, and commands for locating code and data.

A batch file performing functions similar to the makefile is used for the PC version of cdemon.

Makefile line to generate a linker command file line

```
echo FORMAT IEEE >> ieee.cmd
```

The assembler, compiler, and linker invocations are assigned the variable names AS, CC, and LD respectively. The variable names are substituted for the invocations later on in the makefile, to cut down on typing time and errors and to improve readability.

Makefile line associating a variable with the compiler invocation

```
CC=mcc68k
```

Makefile line associating a variable with the compiler options

```
CCFLG= -g -nOc -nOl -nOR -c -nQ
```

```
# makefile for cdemon demonstration program¶
# commands necessary to assemble, compile, and link¶

CC=mcc68k¶
AS=asm68k ¶
LD=lnk68k¶
CCFLG= -g -n0c -n0l -n0R -c -n0¶

all : cdemon.x¶
cdemon.x: alib.o cdemon.o clib.o data.o ieee.cmd¶
}     $(LD) -c ieee.cmd - cdemon.x -m>cdemon.map¶

ieee.cmd : makefile¶
}     echo CHIP}        68000}                >ieee.cmd¶
}     echo FORMAT}      IEEE}                 >>ieee.cmd¶
}     echo NAME}        CDEMON}               >>ieee.cmd¶
}     echo LIST}        d,s,t,x}              >>ieee.cmd¶
}     echo PUBLIC}      STACKTOP=0,08000h}    >>ieee.cmd¶
}     echo ORDER}       startup,code,strings,zerovars,vars
>>ieee.cmd¶
}     echo SECT}        MMIO_LO=3000h}        >>ieee.cmd¶
}     echo SECT}        MMIO_HI=0a000h}       >>ieee.cmd¶
}     echo SECT}        VECTORS=0}            >>ieee.cmd¶
}     echo SECT}        startup=4000h}        >>ieee.cmd¶
}     echo LOAD}        alib}                 >>ieee.cmd¶
}     echo LOAD}        cdemon}               >>ieee.cmd¶
}     echo LOAD}        clib}                 >>ieee.cmd¶
}     echo LOAD}        com}                  >>ieee.cmd¶
}     echo LOAD}        data}                 >>ieee.cmd¶
}     echo LOAD}        mcc68kab.lib}         >>ieee.cmd¶
# main application module¶
cdemon.o : cdemon.c¶
}     $(CC) $(CCFLG) cdemon.c¶

# common c code application functions module¶
clib.o : clib.c¶
}     $(CC) $(CCFLG) clib.c¶

# comport support communications module¶
com.o : com.c¶
}     $(CC) $(CCFLG) com.c¶

# simple blackjack cardgame module¶
data.o : data.c¶
}     $(CC) $(CCFLG) data.c¶

# common assembly code application functions module¶
alib.o : alib.s¶
}     $(CC) $(CCFLG) alib.s ¶
```

Figure 3-1 Cdemon makefile

Compiler flags for symbolic debugging

The following MCC68K compiler command line flags produce linkable object modules containing symbolic information for the debugger.

```
mcc68k -g -nOc -nOl -nOR -c -nQ
-g (generate line number and tracing info)(not default)
```

The remaining data options select other functions.

```
-nOc (disable stack pop optimizations)
-nOl (disables local optimizations such as code hoist-
ing)
-nOR (disables use of registers for variables)
-c    (make object file but don't link it to make an
executable file)
-nQ  (display any informational messages)
```

Linker command file for cdemon

In our example the linker command file, *ieee.cmd*, is created by the makefile lines that "echo" the actual linker commands into the command file. This takes advantage of UNIX and DOS's ability to append (>>) echoed "statements" into a file. The resulting linker command file is shown below.

```
PUBLIC STACKTOP=8000h
ORDER  startup,code,strings,zerovars,vars
SECT MMIO_LO=3000h
SECT MMIO_HI=0a000h
SECT VECTORS=0
SECT startup=4000h
LOAD alib
LOAD cdemon
LOAD clib
LOAD com
LOAD data
LOAD mcc68kab.lib
```

Figure 3-2 Linker command file

Locating target code and data

In embedded systems applications, code and data are usually located by the linker at compile time instead of by an operating system loader at run time. The linker command file *ieee.cmd* is a good place to examine how this may be accomplished. The lines of the linker command file pertinent to locating code and data are presented and explained below.

```
PUBLIC STACKTOP=8000h (defines value of the external
definition of   STACKTOP)
ORDER startup,code,strings,zerovars,vars (overrides
linker's default ordering of assigning base addresses
to segments)
SECT MMIO_LO=3000h
SECT MMIO_HI=0a000h
SECT VECTORS=0
SECT startup=4000h
```


Linker switches for symbolic debugging

The following linker commands and flags produce an output file that XICE can symbolically debug. The linker is invoked in the command file mode.

```
lnk68k -c ieee.cmd -m -o cdemon.x >cdemon.map
-c ieee.cmd (use ieee.cmd as the linker command file)
-m (write a memory/symbol map to standard out)
>cdemon.map (redirect memory/symbol map to file cde-
mon.map)
-o cdemon.x (name the linker output file cdemon.x)
FORMAT IEEE (produce MRI-extended IEEE-695 output file
format)
LIST d,x,s,t (list of linker flags)
     d (external definition symbols in object)
     x (external definition symbol table to listing)
     s (local symbols in object)
     t (local symbol table to listing)
The remaining commands perform other functions.
NAME cdemon (names final output module cdemon when gen-
erating ieee)
```

Additional information

Additional information on subjects covered in this section is available in the compiler and assembler manuals.

Using the XICE debugger user interface

The XICE debugger is a "windowed" user interface. The windows in XICE are called "viewports". The DOS version has scrollable, zoomable viewports and a keyboard command line interface. The Sun version is similar but offers mouse support primarily in the form of command buttons, point-and-click temporary breakpoints and variable evaluation, and "cut and paste" features. Both versions include on-line help.

Brief information about navigating the user interface, including function keys, mouse, viewports, line editing, and control keys, can be found in the on-line help. Detailed explanations of the features of user interface can be found in the *XRAY User's Guide*.

Environment variables

Before invoking the debugger ensure the path and environment variables (XRAY and XRAYLIB) have been set up. To check the path and environment variables you can type **env** if your are on a UNIX workstation or type **set** if you are using a PC. If they are not set up or you are not sure they are correct, please refer to the *XICE Installation Guide* and set them up now.

Debugger configuration file - *xice.cfg*

For the tutorial, you need to modify two parameters, called softswitches, in the debugger's configuration file, *xice.cfg*, before invoking the debugger. The modifications can be made using an editor like "vi" or "ed". *Appendix A* in the *XICE Installation Guide* provides a detailed description of the *xice.cfg* file.

Note

Before modifying *xice.cfg*, you should make a backup copy using the UNIX "**cp *xice.cfg xice_bak.cfg***" or the DOS "**copy *xice.cfg xice_bak.cfg***" command.

The tutorial assumes the emulator is plugged into the factory supplied "null" target, or that NULL_TGT is enabled for 68000/HC000/EC000 probe-tip-only configuration emulators. This is referred to as "null" target mode. The tutorial program will be loaded into emulator memory configured (or mapped) to replace (or overlay) the desired address spaces of target memory. Memory can be mapped by commands in the *xice.cfg* file, in an include file, or at the XICE command line level. We will begin with the default condition of memory already mapped by the "MAP" command in the *xice.cfg* file. Modifying the memory map at the XICE command line is covered later.

Open the *xice.cfg* file using your text editor

At this time you should open the *xice.cfg* file using your text editor.

First *xice.cfg* modification - "scratch" memory softswitch

```
/* SW_SCRATCH:0x9ff0 */
```

The **SW_SCRATCH** statement sets aside 8 bytes of memory starting at address 0x9ff0 as instruction breakpoint "scratch" memory.

The debugger requires 8 bytes of "scratch" RAM located in supervisor program space to support the TRAP instruction used for BI's (break instructions), temporary breakpoints, and high-level single stepping. The address of the "scratch" RAM is set by the **SW_SCRATCH** softswitch in the *xice.cfg* file. If the **SW_SCRATCH** line is "commented out" using C language comment field notation, "/*.. */", you must uncomment this line for the tutorial.

Uncomment the SW_SCRATCH line:

```
/SW_SCRATCH:0x9ff0 /
```

The TRAP instruction number is determined by the **SW_EXVEC** softswitch in the *xice.cfg* file. The TRAP instruction selected is reserved for emulator use and should not be used in your code. The default setting for **SW_EXVEC** selects TRAP 15.

Second *xice.cfg* modification - bus timeout softswitch

The bus timeout switch, **SW_BTE**, determines whether or not emulation breaks and returns control to the debugger if it enters a hung condition, i.e. waiting for a DTACK signal to be returned from non-existent memory. The timeout is fixed at 1 second. To enable the timeout set **SW_BTE:ON**.

The softswitch should be turned on while debugging hardware or software.

Modify the *xice.cfg* line for enabling bus timeout

```
SW_BTE:ON \
```

Now you should close the *xice.cfg* file saving the changes.

Close the *xice.cfg* file using your text editor

Debugger invocation

The tutorial is meant to be run from the *bin* directory of the toolchain. If you are not in the *bin* directory of the toolchain, **cd** to there now.

The emulator control software, called shell code, is located in the file with the extension *.shl*. When starting a debug session, you should invoke the debugger using the "force reload of shell code" option, **-e boot**. This ensures the emulator control software is correct for the version of the debugger you are

using. If you update the debugger, be sure to install its corresponding updated shell code file. Using the debugger with mismatched shell code will have unexpected results.

Invoke the debugger with forced reload of shell code

```
executable_name -e boot
```

As the debugger comes up, you will see messages displayed sequentially on the XICE screen, "Initializing", "NOTE: Downloading operating system to emulator, please wait", and "Loading file: xxxxx.shl". Once the debugger is up and running you may exit the debugger by typing an abbreviated version of quit followed by yes, "**q y**".

Exit the debugger

```
:q y
```

Include files - introduction

An include file is simply a file containing debugger commands that will be executed when the file is loaded by the debugger. It is similar to a UNIX shell script or a DOS batch file. For example, the supplied include file, *cdemon.inc*, simply loads the appropriate *cdemon* absolute file from the **demo** subdirectory.

You can invoke the debugger using the "include file" option, **-i *filename.inc***. Those with UNIX systems may want to background the program in order to free up a shell or make it easier to "kill" the program if necessary.

Invoke the debugger with the include file option

```
executable_name -i cdemon.inc
```

As the debugger comes up you will again see the "Initializing" message. The "Loading file: xxxxx.shl" message will not appear since we did not choose to reload the shell code. You will see a "Reading Absolute File: none" message followed by the two include file commands being echoed in the debugger command window.

At this point, the tutorial code, *cdemon.x* or *cdemon.abs*, has been loaded, and the debugger is ready for commands.

Required memory

Although set up for the Applied Microsystems Demonstrator module, the tutorial can work with the "null" target shipped with every new system, or with NULL_TGT enabled, or with your own target system. It requires either overlay or target RAM at addresses 0x0..0xffff. The default *xice.cfg* automatically maps overlay RAM for *cdemon*. If overlay is not available, map target RAM as follows (mapping memory is described on page S3- 32):

Map 0x0 to 0xffff as target RAM:

```
ram 0x0..0xffff=target
```

The specific memory map for *cdemon.x* is as follows:

Section	Address range
startup	00004000..0000408D
MMIO_LO	00003000..00003001
MMIO_HI	0000A000..0000A001
VECTORS	00000000..00000013
vars	00004B8C..00004BA5
code	0000408E..000049C5
zerovars	000049FC..00004B8B
strings	000049C6..000049F9

Include file for 68302 setup

When you emulate while plugged into a 68302 target, it is extremely important to correctly setup the System Integration Block (SIB) to match your target's memory map and DTACK requirements. An incorrectly programmed SIB can potentially cause the emulator to hang. The hardware portion of SIB setup is covered in the EL1600 emulator manual.

The 68302 tutorial depends on the default register settings and overlay memory mapping specified in the *xice.cfg* shipped with the software; these set up the SIB compatible with the default hardware settings. If you have not changed the default register setting in xice.cfg, you may proceed with the tutorial.

If you have changed *xice.cfg*, you may want to create an include file that sets up the CPU's System Integration Block, speeds the download, suppress memory related warning messages, loads the program *cdemon.x* or *cdemon.abs*, and switches the debugger into source level display mode.

The following provides an example of such a file:

Note: Comment fields begin with a semi-colon (;). The actual commands appear here in bold type.

```
;cdemon.inc
;Set memory access size to word(2); increases speed of the "load"
size write 2
; set "error" to ignore errors and continue include file execution
error=continue
; The following are 68302 specific commands
; Other CPU's should ignore the error messages
;set BAR register to Base Address 0xe00000, CFC=0 to ignore FC's
setreg @BAR=0x0e00
;set OR0 to internal 6 wait-state DTACK, R/W, any FC in 64k block
setreg @OR0=0xdfe0
;set BR0 enable chip select 0(CS0), any FC, Base Address 0
setreg @BR0=0xc001
; return "error" to its default value of abort include for errors
error=abort
;"mwarn off" suppresses memory warning messages during the "load"
mwarn off
load '../demo/cdemon.x'
;Turn "mwarn" back on
mwarn on
;Switch to "source-level" mode
mode high
```


Figure 3-3 Sample UNIX include file for 68302 registers

Using Help

The on-line help information displayed by the debugger is in the ASCII file with the suffix *.hlp*.

Help is available in scrollable menu form or may be invoked with a subject argument.

Using the help menu

Typing "**help**" or "h" invokes the help information menu of XICE topics and commands. The are two levels of help, the top menu level of alphabetized subjects and the lower level of actual help information on each subject.

Invoke the help menu

```
>help
```

Scroll up and down through the menu by using the up and down arrow keys (depicted as "^" and "v" on the bottom of the help menu). A carriage return selects the subject highlighted by the cursor and displays the next level of help information.

Scroll down and up using the up/down arrow keys

```
>v
>^
```

Scroll down to and select the command 'DEFINE'

```
>v
>Return
```

At this point you can scroll through the subjects at the "help information" level. A carriage return or down arrow (CR/v) goes to either the second page of information on the current subject or to the next subject if there's only one page. The up arrow scrolls back through the previous subjects. Pressing Escape (Esc) closes the help window and returns you to the Command viewport.

For example only -Invoking help on the step command

You can also invoke help for a specific command:

```
h step
```

Exiting help

An "Esc" exits help and returns you to the XICE command line. If you want to leave the help information displayed on the screen while you type in a command use the F7 key to exit help instead of Esc.

Exit help and return to the command line

```
>Esc
```

Additional error message information

If an error message pops up for any reason, you may be prompted by the debugger to type **explain** to receive additional information about the error.

Navigating XICE windows (viewports)

Activating viewports (selecting XICE windows)

An active viewport is indicated by a shaded box surrounding the viewport. You can activate any XICE viewport in either of two ways. One way is to use the viewport activate command, **vactive** (**va**), with the viewport's number as the argument. The viewport's number is located in the upper right hand corner of each viewport.

Another way, perhaps faster, is to simply scroll through the viewports using the **F1** function key.

Activate the code viewport

```
>va 2
```

Zooming and unzooming viewports (enlarging and reducing XICE windows)

The **zoom** (**z**) command toggles viewports between their normal and enlarged sizes.

A viewport can be enlarged in either of two ways. You can enlarge and activate a viewport by using the **zoom** command similar to the way the **vactive** command is used. Or, you can use the **F4** function key to enlarge an active viewport.

Zoom or **F4** the enlarged window again to return it to normal size.

Zoom the code viewport

```
-z 2
```

Return the code viewport to normal size

```
-z 2
```

Scrolling viewports

You can scroll up and down in an active viewport by using the up and down arrow keys. For a standard SUN or PC keyboard, use the Home and PgUp keys to go to the beginning or end of a viewport.

Modifying and saving debugger start-up options and viewports

A set of default debugger display and execution options is read in when XICE is invoked. The options control functions such as breakpoint alignment, default radix, viewport display and others. These options can be changed and saved.

In addition to the options, you can also save the size and position information for all the predefined and user defined viewports. The viewport information and options are saved to a file created by the **startup** command and restored when you invoke the debugger.

Detailed functional descriptions of all the options are in the *XRAY Documentation Set* under the **option** command.

Displaying debugger options

Use the **option** (**op**) command to display the current set of debugger options. It will zoom the command window to display the options.

Display the current debugger options and unzoom the command window

```
:-op
:-F4
```

Modifying debugger options

Use the **option** command with an "option name=value" argument to change an option.

We will not modify any options, but an example is shown below.

For example only - Modifying the debugger default radix

```
op radix=hex
```

Saving the options and viewports

Use the **startup** command to save options and viewport information to the start-up file. You can choose the name of the start-up file. This allows different users to save personal setups in their own unique start-up files. The **startup** command used with no argument saves the information to a file named *startup.xry*.

To use your saved options, use the **-s** debugger command line switch with your start-up file name as the argument when you invoke the debugger.

We will not save any options now. An example of saving the information to the default start-up file, *startup.xry*, is shown below.

For example only - Saving options and viewports to default file *startup.xry*

```
startup
```

Recording a debug session

Recording commands

Sometimes it may be useful to record the commands used during a debug session. The **log** command opens a file and saves the command line input into the named file. The "log" file can be used as an "include" file which can be loaded and executed by the debugger command **include** to recreate a debug session.

Recording commands and their output

The **journal** command records both the commands and their output into a file. This will be demonstrated later in the tutorial where we use the command to save the contents of the emulator's "bus cycle trace" memory.

Convenience features

Command history

XICE has a command "history" feature similar to that of the UNIX C Shell. You can display a list of executed commands or recall a specific command from the list by using the command **history** (**hi**).

For example only - Using history

```
hi          (display list of executed commands)
hi step<ret> (recall step command if on history list)
hi step<ret><ret> (recall and execute step command)
```

Another way to re-execute a past command is simply to use the **up/down** arrow keys to scroll through past commands until you reach the desired one, then press **<Return>** to execute the command.

Command aliasing

XICE allows you to assign a different name to a debugger command. Use the **alias** command to do this. Preferred aliases can be placed in an "include" file and loaded at the start of your debug session

For example you may want to rename the **load** command to **dnl**.

For example only - Renaming load to ld

```
alias load=ld
```

Note

Although XICE allows you to define any alias, you cannot successfully invoke an alias for a command that is itself an alias. Many XICE commands are aliases of XRAY simulator commands. To determine whether a command is already an alias, invoke help for the command. If it is an alias, the help screen says "alias of SS . . . ." If you create and invoke an alias for such a command, an "Unknown Command Entered" error is returned.

Using debugger functions

Getting oriented with the code

When starting a debug session you will want to get oriented with the code, particularly if the code is not your own. The following commands will help you do this.

Displaying available modules

A quick display of the names of the source modules available for debugging is a good place to start. Use the **printsymbol** (**ps**) command with the **/m** flag and * argument to display all module names. Of course with very large programs containing many modules, this may be impractical.

The command will zoom the command window and display the names of cdemon's seven modules along with "type" and address information for each module.

Display module names and unzoom the command window

```
>ps /m *
>F4
```

Printsymbol is an important and versatile command with many other options for displaying symbols and subsets of symbols.

Current viewing (scope) and execution context

The debugger is capable of viewing a module that is not the current execution module. The current execution module is the module that the program counter (PC) is focused on. If you were to execute a **step** command the debugger would execute the source line pointed to by the PC. Use the **context** (**con**) command to display the current "viewing" and "execution" modules.

Display context (current viewing and execution modules)

```
·con
```

Note the current viewing module line, CDEMON, will have "(view)" at the end of it, while the current execution module line, address 4044, will end in "(PC)".

Changing scope

The viewing context can be changed by using the **scope** (**sc**) command. This will cause the source for the module to be displayed in the CODE viewport. It also allows access to the module's symbols and line numbers without having to type the qualifying module or procedure name, saving a considerable amount of typing.

The **scope** command is case sensitive.

Change the current scope to the module DATA

```
>sc DATA
```

Display the current context

```
·con
```

You will see the current viewing module is now "DATA", while "4044" remains the current execution module address.

Return to scoping the current execution module CDEMON

```
>sc
```

Selecting assembly or high-level debugging mode

You might be debugging a small piece of code that controls some I/O device and decide you need to work with your code at the assembly level. Use the **mode** (**m**) command to quickly toggle between high-level and assembly-level debugging modes.

Change to assembly-level debugging mode

```
>-m
```

The code viewport now displays assembly-level code and its corresponding high-level source code, if any.

Return to high-level debugging mode

```
>-m
```

Other C source operations

There are other debugger commands that display source without changing scope (**list**), evaluate expressions (**cexpression**), find the next occurrence of a given string in the source (**next**), and display parameters passed to procedures (**expand**). These commands are covered in both the on-line help and the *XRAY Documentation Set*.

Checking the state of the debugger and emulator

When starting a debug session you should take a quick look at the state of the debugger and emulator. This is particularly true if someone else has been using the emulator between your sessions. Also, you should examine the state of the debugger and emulator any time you get unexpected results from breakpoints or event system setups.

The following commands will allow you to view and modify the parameters that control the state of the debugger and emulator.

Displaying debugger status

Use the **status all** command to display the debugger's current version, directory, log file, journal file, startup file, and loaded absolute file. Have this information at hand if you ever call for factory support.

Display debugger status

```
>stat all
```

The status information should be displayed in the VIEW(ALL) viewport located directly above the CODE viewport.

Displaying emulator hardware configuration

Use the **hwconfig** command to display the emulator's hardware configuration and the version of emulator control (shell) code. Have this information at hand if you ever call for factory support. The display will vary depending on your particular configuration.

Display the emulator's hardware configuration and shell code versions

```
·hwconfig
EL1600 Enhanced SCSI Controller, version 0.01
EL1600 1M Overlay, version 0.01
EL1600 Dynamic T & B Board, version 0.01
EL1600 68302 SCSI Shell(00), version 2.00
```

Softswitches, options, variables, and "double-argument" commands

Three "display variable" commands - **emuvars**, **xicevars**, and **memvars** - show the state of most of the parameters controlling debugger and emulator functionality. Display the remaining parameters with the **evtvars** and **option** commands.

These parameters are referred to as "softswitches" (short for software switches), options, variables, commands, and "double argument" commands. To help remember the names of the commands to display these parameters, you should think of them collectively as "variables".

The variables can all be modified at the command line or in the *xice.cfg* file, with the notable exception of **evtmode,** which can only be modified in the *xice.cfg* file. Also note the state of the variable **evtmode** is not accessible at the XICE command line. You must look in the *xice.cfg* file for this information.

The *xice.cfg* file contains a commented list of the variables along with a brief description of each. This list is a useful reference when you are modifying the *xice.cfg* file. You may want to print a copy of the *xice.cfg* file to use as a quick reference guide.

In the tutorial we will only display the variables. Detailed functional descriptions of the variables can be found in chapter 2 and in the *XICE Installation Guide.*

Activate the COMMAND viewport

```
>va 1
```

You need to **zoom** the command viewport to see all the variables.

```
Zoom the COMMAND viewport
>z 1
```

Displaying emulator variables

The emulator softswitches control how the emulator treats CPU signals and functions like DMA, timers, chip selects, refresh, and interrupts.

Display the emulator variables

```
>emuvars
```

You should see a list of the emulator's softswitches showing their state and a brief functional description.

Displaying XICE variables

The XICE variables control tracing and event system functions.

Display the XICE debugger variables

```
>xicevars
```

You should see a list of XICE's internal variables showing their state and a brief functional description.

Displaying XICE memory operation variables

The variables controlling memory access operations are called "double argument" commands in the *xice.cfg* file.

Display the debugger memory operation variables

```
>memvars
```

You should see a 12x5 matrix showing the twelve memory operations and how each is configured for the four memory access variables

The twelve memory operations are code, comp1, comp2, copyfrom, copyto, diag, fill, read, search, stack, test, and write.

The four memory access variables are shown with a brief description below.

space **sp**/sd/up/ud/cpu/sc0,3,4	(specify memory space)
size **1**	(specify access size 1 "byte", 2 "word", 4 "longword", ANY)
overlay **on**/off	(specify overlay(on) or target(off) memory accesses)
address **phys**/logical	(specify physical or logical addressing)

Checking the state of the target

Checking the CPU bus

Use the bus command to display the state of the CPU's control lines as sampled at the probetip.

Display the CPU bus status

```
>bus
```

Return the COMMAND viewport to normal size

```
>F4
```

Controlling the Emulator and CPU

Changing the contents of a CPU register

While debugging your code, you may find a register holding a different value than what you expected. The **setreg** command allows you to directly modify the contents of a CPU register. This lets you replace the questionable value with the expected value and test the results.

Go to assembly mode and change the contents of the PC register to 0x100

```
>m
>setreg @pc=0x100
```

Note the PC register value displayed in the REGISTERS viewport has changed to 0100. (You must zoom (Z 14) the REGISTERS viewport on the IBM-PC.)

Return the contents of the PC register to 0x4044 and return to high level mode

```
>setreg @pc=0x4044
>m
```

Other 68000 and 68302 register controls

These are the softswitches that control the DMA (**ted**, **tid**), timer (**drtmr**), interrupt (**fast, slo**), refresh (**drtmr**), and chip select (**rirr**, **uir**) registers. These softswitches are covered in Chapter 2 of this supplement.

Memory control

Displaying memory addresses and variables

The **dump** (**du**) command displays the contents of memory at a given address or range of addresses in both hexadecimal and ASCII format.

As do most XICE commands, **dump** also accepts a symbol name as an address argument. This allows us to **dump** the contents of the tutorial's "memory mapped" output port, "led_port", without recalling the port's numerical address. (Note: We could have found the address of led_port with the **printsymbol** command.)

Dump the contents of "led_port"

```
>du led_port
```

Another way of viewing the contents of "led_port" is to use the **printvalue** (**p**) command. The **printvalue** command displays the values of expressions according to their type. The **printsymbol** (**ps**) command will show "led_port" is an array of signed char, so **printvalue** will display character values found at "led_port".

Display symbol information about "led_port"

```
>ps led_port
```

Print the value at "led_port"

```
>p led_port
```

You may want to keep a continuous display of a variable's value on the screen. The **monitor** (**mon**) command opens the DATA viewport and displays the selected variable. The display is updated during every "run to pause" transition.

Monitor the variable "led_port"

```
>mon led_port
```

Modifying memory

Modify memory with the **setmem** command. Setmem has a switch for byte, word, and longword data arguments. We will use **setmem** with the longword switch, /l, to modify the contents of "led_port", and then view "led_port" with the **dump** and **printvalue** commands.

Set memory at "led_port", then display the new contents

```
>setmem/l led_port=0x58494345
>du/l led_port
>p led_port
```

Using the single line assembler

There may be times when you need to make a small change to an assembly module, perhaps just to try something out. Use the debugger's built in line assembler to make your patch and avoid a time consuming "exit debugger, edit code, assemble and link, download, and try-it-out" debugging cycle. The line assembler is invoked with the **asm** command.

Below we assemble a simple "nop" loop. To exit the assembler type a **carriage return** on an empty assembler line.

Assemble a "nop" loop beginning at address 0x9000

```
asm 0x9000
00009000: nop
00009002: nop
00009004: jmp 9000
0000900a: <Return>
```

Using the memory disassembler

The memory disassembler can only be used with XICE in the "assembly mode".

Disassemble the "nop" loop at 0x9000. The disassembled memory is displayed in the CODE viewport.

Go to assembly mode and disassemble memory at 0x9000

```
·m
·dis 0x9000
00009000 4E71            nop
00009002 4E71            nop
00009004 4EF9 0000 9000 jmp $009000
```

Return to high level mode

```
·m
```

Other memory operations

There are other memory commands available that fill memory with a given value (**fill**), copy the contents of one block of memory to another (**copy**), compare the contents of two memory blocks (**compare**), and search through memory for a pattern (**search**). These commands are covered in the *XRAY Documentation Set.*

Using overlay memory

Overlay memory is emulator memory that can replace target memory by overlaying it, or be used where target memory resources do not exist. Assigning overlay memory to address ranges and access types chosen by the user is called "mapping" overlay.

Note

Overlay has a minimum granularity of 2K. If a mapping does not begin and end on a 2K boundary, the emulator automatically adjusts the mapping in both directions to the next 2K boundary and issues a warning that it has adjusted the original mapping.

Displaying the memory map

Use the **ramaccess** (**ram**) command with no arguments to display the current overlay vs. target memory map.

Display current overlay vs. target memory map

```
>ram
```

You should see a display of the type of memory (ram, rom, or nomem), who owns it (emulator or target), its address range, and how much emulator memory remains.

Mapping overlay memory as RAM

Use the **ramaccess** command with a range argument to map overlay memory as read/write memory (RAM). Memory mapped as RAM is fully accessible to the executing program and to the user.

We will not alter the memory map in the tutorial, so an example is shown below.

For example only - Mapping overlay from 0x1000 to 0x2000 as RAM

```
ram 0x1000..0x2000
```

Mapping overlay memory as ROM

Use the **romaccess** (**rom**) command with a range argument to map overlay memory as read only memory (ROM).

Memory mapped as ROM will cause a "write violation" break if written to by the executing program. However, the user can still write to this memory using any debugger memory write command such as **setmem** or **fill**.

An example is shown below.

For example only - Mapping overlay from 0 to 0xfff as ROM

```
rom 0x0..0xfff
```

Mapping memory as illegal access space

Memory can be mapped as illegal or non-existent by the **nomemaccess** (**nomem**) command.

Memory mapped as "nomem" causes an "illegal access" break when the executing program tries to access it for any reason. This is useful when implemented as a built-in bounds checker for the executing program. If there are areas in the memory map that nothing should access, map them as **nomem**.

Memory mapped as "nomem" is also unavailable to the user, even through debugger memory commands such as **setmem** or **fill**.

An example is shown below.

For example only - Mapping 0x1000 to 0x2000 as illegal memory

```
nomem 0x1000..0x1fff
```

Mapping overlay memory back to target memory

Use the **ramaccess** command with the **=target** argument to reassign overlayed memory to the target.

An example is shown below.

For example only - Returning overlay memory 0x1000 to 0x2000 to target

```
ram 0x1000..0x1fff=target
```

Copying target memory contents to overlay memory

The **ramaccess** command, when used with **=copy**, first maps overlay memory over the range argument and then copies the contents of target memory into that range of overlay memory.

Use this command when you need to copy the contents of your target ROM or PROM into overlay memory for patching, to avoid having to burn a new ROM.

An example is shown below.

For example only - Copying the contents of target memory into overlay

```
ram 0x9000..0x9fff=copy
```

Basic breakpoints

Basic breakpoints include access and instruction breakpoints. Simple to set up and use, they let you stop emulation at a predetermined location in the program.

Setting access breakpoints

Access breakpoints are set to break on a read status (**br**), a write status (**bw**), or don't-cared for a read or write (**ba**) of the breakpoint address. These breakpoints are implemented using emulator hardware and may be set in RAM or ROM.

You cannot set an access breakpoint when any event system triggers are armed. Disarming the triggers will allow you to set access breakpoints. Likewise, setting access breakpoints causes the event system to be disabled. Clearing the access breakpoints will allow you to arm the event system triggers.

Macros can be attached to access breakpoints. You may have multiple ba, br, and, bw breakpoints set, each with its own macro attached. Macros are illustrated on page S3-42.

Set a write access breakpoint at "led_port"

```
>bw led_port
```

Setting instruction breakpoints

Instruction breakpoints (**bi**) are software implemented breakpoints. They use the TRAP instruction to temporarily replace the instruction at the breakpoint address. These breakpoints can only be set in RAM. There must also be RAM located at the "scratch" address to accommodate the TRAP instruction's interrupt service routine.

A **bi** uses one each of the emulator's two address and status comparators. The remaining address and status comparators are available for access breakpoints. This means if you set a **bi** you can additionally set 3 single-address access breakpoints of the same type (all **br**'s, for example) or one range access breakpoint.

Set an instruction breakpoint at outled

```
>bi outled
```

Clearing breakpoints

Use the **clear** (**cl**) command to clear all breakpoints. Use **clear** with a number or range argument to clear a particular breakpoint or group of breakpoints.

Clear breakpoints 1 and 2

```
>cl 1..2
```

Program execution and related commands

The following commands control resetting the CPU, restoring the program start address, and executing the program in real-time or in steps at a time.

Resetting the processor

Use the **reset** (**rese**) command to restore the processor to its initial reset state.

Reset the CPU

```
>rese
```

Restoring the program start address

Use the **restart** (**rest**) command to reset the program counter to the program's starting address. For *cdemon.x* this returns us to "start:" in the ALIB module, address 0x4044.

Restore the program start address

```
>rest
```

Starting and stopping program execution

Use the **go** (**g**) command to start or continue program execution. The program will execute until a breakpoint is reached, an error occurs, or the user stops emulation with a **CTRL-C.**

Use the **go** command with an address and a passcount to execute until the address is seen "passcount" number of times. The command sets a temporary breakpoint at the address and counts each occurrence of the breakpoint.

Execute until "outled" is seen four times

```
>g outled%%4
```

After a few moments, the emulator will break and display the PC value at the time execution stopped.

Stepping through the program

Stepping refers to executing code a number of lines at a time. Single stepping executes either one source line or one assembly line of code at a time, depending on which mode you are in. To single step use the **step** (**s**) command without a number argument.

For example only - Executing one line of code and five lines of code

```
s
s 5
```

Use the **stepover** (**so**) command if you want to single step but do not want to step through called routines. This command will execute the entire called routine then stop.

Use the **gostep** (**gos**) command if you want to step continuously until a specific condition is met. The condition is defined by a macro you attach to the **gostep** instruction. For instance, **gostep** can be used to step until a register holds a particular value.

For example only - Stepping until a condition defined in "my_macro" is met

```
gos my_macro()
```

Capturing and displaying execution trace history

The trace capture feature lets the user observe exactly how the code executed. Raw trace consists of CPU bus level information including address, data, status, and an optional 16 channels of logic state or timestamp information. Disassembled trace is displayed as assembly, source, or a mixture of both. Raw and disassembled trace are both displayed in the COMMAND viewport.

In high level mode (source mode) the TRACE viewport displays a trace of the "procedure calling chain". Don't confuse this with the raw and disassembled trace discussed above. The TRACE viewport is covered in depth in chapter 2 of the *XRAY Documentation Set.*

Clearing trace memory

You may want to clear the trace memory buffer of previous trace information before running your code. This ensures all information in the trace buffer will be newly acquired. Use the **trcclr** command to clear the trace memory buffer.

Clear the trace memory buffer

```
>trcclr
```

Capturing trace in run mode and pause mode

Unless conditionally tracing with the event system, the trace capture feature is always enabled for run mode. Every time you use **go** or **step**, the bus information generated is captured in the trace buffer.

An emulator softswitch, **ppt**, controls the capture of additional information. With **ppt on** you can capture bus cycle information generated by XICE memory read and write commands such as **setmem**, **fill**, **copy**, **diag**, and others. This

trace information can assist you in diagnosing general memory problems or memory errors that may have shown up in one of XICE's ram diagnostic tests (**diag**).

Also, with **ppt on**, cycles generated by XICE memory commands or by downloading code with the **load** command are included in trace memory. The **load** command cycles can be a valuable source of troubleshooting information when a download fails for some reason. You can examine the last cycle in trace memory and determine if the download went to valid RAM memory, nonexistent memory, or ROM, for example.

To collect trace information we will **restart** and then **go** until we reach the function "outled".

Restart, then go to "outled"

```
>rest
>g outled
```

Displaying raw trace history

Use **drt** for displaying raw bus cycle information and optional logic state and timestamp information.

Display the captured raw trace information

```
>drt
```

Zoom the COMMAND viewport and examine the raw trace. You can scroll up and down in the viewport with the up/down arrow keys to view trace that may have scrolled off the display.

Zoom the COMMAND viewport

```
>z 1
```

The FRAME numbers on the far left of the trace are used to reference when in trace history the information occurred. The lower line numbers are the last cycles captured prior to a "break" in emulation. The "break" at trace line 1 occurred when we reached the function "outled".

The other raw trace columns show the address (ADDRESS), the data (DATA), Interrupt Priority Level (IPL), Function Code (FCn), type of memory cycle (MEM), flags set (FLAGS), and logic state information (LSA BITS) when each bus cycle was captured.

Additionally, trace for the 68302 shows a column of information on DMA cycles (DMA), and on the CPU internal access pin (IAC). For the 68000, these columns are replaced by information on the state of valid peripheral access pins (VPA), and the state of the memory access pins (VMA).

Searching trace history for a pattern (emulator only)

To search trace memory for patterns, use the **tsrch** (**ts**) command. The command syntax is similar to the **ev** command. You may qualify the search with combinations of address, data, and lsa patterns, and status. You may also specify a starting line number in trace history.

Search trace history for an occurrence of a write to address 0xfe

```
>ts addr=0xfe,stat=wr
```

Displaying disassembled trace history

Use **dtb** (display trace backwards) for displaying the trace buffer information formatted in assembly or high-level mode, or as an interleaving of both modes. The **dtf** (display trace forwards) command performs the same trace display function, but in a different direction. Use the **dt** command with a start address to begin disassembling at a particular line in trace.

The **trcmode** XICE variable controls the disassembled trace display mode. The variable's default (**both**) causes an interleaving of assembly and source.

Display the trace information in disassembled format

```
>dtb
```

The numbers on the far left of the disassembled trace correspond directly to the FRAME numbers on the far left of the raw trace display. They are useful when correlating a line of disassembled trace to its bus cycle equivalent line in raw trace.

Observe the call to the function "outled", JSR outled, and the source line for the function "outled", outled(0xFE).

Saving trace to a file

You may need a hardcopy of trace or a copy of trace on disk for later referencing. Or, you may have a problem that requires factory support. The Applications department might request a hardcopy of trace memory to assist in solving the problem.

Earlier we discussed the **journal** (**jou**) command, which records both the commands and their output into a file. You can use the **journal** command to save a partial or entire trace disassembly into a file. The example below shows how to save a trace memory display to a file.

For example only - Saving part of a raw trace to a file named *trace.raw*

```
jou on="trace.raw"
drt 0..42
jou off
```

jou on="trace.raw" creates a file named *trace.raw* as the journal file. **drt 0..42** displays raw traces lines 0 through 42. This display goes to both the COMMAND viewport and the journal file. **jou off** causes journaling to cease and closes the journal file.

Using the **/a** option with **jou** allows you reopen and append to an existing file.

Executing XICE commands in run mode (dynamic run mode-drun)

The **drun** command lets you use XICE commands without stopping program execution (run mode).

For instance, you may want to examine trace history (**drt** or **dtb**) while executing your program. If you enter run mode using **drun** instead of **go**, you can use the **drt** command to display the trace history, gathered up to the point where you entered the **drt** command, while the target program continues to run.

To exit dynamic run mode use the **dstop** command.

Restart, enter dynamic run mode, then display raw trace history

```
>rest
>drun
>drt
```

Zoom command window, view raw trace, then unzoom

```
>z 1
>z 1
```

Use the **dupdate** command in dynamic run mode to cause the code, register, stack/trace, and data viewports to be updated at the interval given in the command's argument. The default argument value is 20 polls per minute.

Note that you cannot enter commands while in this mode, and that real time operation is sacrificed during the command's polling process.

Switch to assembly level, enter dupdate mode, and then observe the stack and register viewports being updated (60 times per minute)

```
>m
>dupdate 60
```

Exit dupdate mode (control-c), exit dynamic run mode (dstop), then switch back to high-level mode

```
·Control-c
·dstop
·m
```

Logic state and timestamp options

Logic state probe and timestamp probe options are available from Applied Microsystems. If you have either of these options, see Chapter 6 of this supplement for information on how to use the timestamp option and the following XICE variables. These variables control capturing and displaying logic state and timestamp information.

XICE variables for logic state and timestamp options

```
tstamp (display lsa or timestamp info on/off)
trcframe (cycle number for time 0 timestamp in display)
trcint (display interval or offset timestamp)
tunits (timestamp units for raw trace 0x0)
```

Debugger macros

Macros provide an efficient means of executing repetitive tasks or generalizing a task that originally acted on only a specific item. XICE uses the same C-like sequence of expressions, statements, and debugger commands as XRAY to define and invoke macros. Chapter 4 of the *XRAY Reference Manual* in the *XRAY Documentation Set* is devoted to explaining how to generate your own macros and to use the predefined macros that come with XICE. The following section demonstrates briefly how to create a macro and then save it into an "include" file that can be executed by the debugger.

Creating a macro

Use the **define** (**def**) command to create a macro. This puts XICE in the macro define mode. Notice the prompt will change from a greater-than sign (>) to a colon (:) when you enter this mode. A period (.) in the first column of the line exits macro definition mode.

Most of the commands found in this supplement are Applied's emulator-specific aliases of "set status (ss)" commands. When these commands are used in a macro you must precede and follow the command "$," preface the command with "ss," and usually follow the command with a comma to accommodate the XICE command line parser. To see whether a command is an alias of a "set status" command, invoke on-line help for the command in question. Then press **<return>**. There will be an "alias of SS command_name" message in the upper right corner of the help screen, i.e. for **drt** you will see "alias of SS DRT". The macro we will create demonstrates using the aliased command **drt**.

Define a macro named dmp_trc

```
>def dmp_trc()
:{
:$
:ss drt,
:$
:;
:}
:.
```

Displaying a macro

Use the **show** command to display a macro.

Display the macro dmp_trc

```
>sh dmp_trc
```

Deleting a macro

Use the **delete** command to delete a macro. An example follows.

For example only - deleting a macro

```
del big_mac
```

Saving a macro to a file

After you determine that your macro works, you may want to save it to a file for later use. The procedure to do this consists of XICE commands to open a file and assign it a viewport number (fopen), display the macro (show) into the file, and close the file (vclose). The resulting file can be used as an include file that recreates the macro.

Save a macro to a file

```
>fopen 50, "dmp_trc.inc"
>show dmp_trc,50
>vclose 50
```

Fopen should have created a file named *dmp_trc.inc* in the current directory. The file contains the commands necessary to create the macro **dmp_trc**, placed there by the **show** command.

Assigning a macro to a breakpoint

A macro can be assigned to a breakpoint by setting a breakpoint and following it with "; your_macro()". The macro is executed when the breakpoint occurs.

Assign macro "dmp_trc" to a write access breakpoint at "led_port"

```
>bw led_port;dmp_trc()
```

Restart, then go till the breakpoint is reached

```
>rest
>g
```

When the breakpoint at "led_port" occurred, emulation stopped and raw trace information was immediately displayed in the COMMAND viewport.

Return the COMMAND viewport to normal size

```
>F4
```

Access breakpoints and event system breakpoints are mutually exclusive. We need to clear the write access breakpoint before continuing on to the event system.

Clear write breakpoint number 1

```
-cl 1
```

Using the event system

Sometimes running to a basic breakpoint and examining trace history does not provide information specific enough to debug your target's code or hardware. Also, you may want the emulator to perform some action other than breaking when the conditions become true.You may need to trigger an oscilloscope after a complex set of CPU bus cycle conditions become true, or to trace only certain types of bus cycle information under certain conditions. For example, the conditions might be the fifth write that a specific subroutine makes to a certain I/O location.

The event system supplies the mechanism to define conditions and take actions by creating **event** statements composed of logically combined conditions and binding these to **trigger** statements that perform the actions. This mechanism allows the emulator to perform various actions based on events of complexity far surpassing that of simple breakpoints.

This section will help you get started using the event system. Comprehensive user information and descriptions of all available conditions and actions are in Chapter 4, "Using Breakpoints and the Event System."

General information

The event system is implemented with emulator hardware and can be used in both RAM and ROM regions.

Setting access breakpoints disables the event system. Clearing the access breakpoints will allow you to arm event system triggers.

Step 1: Setting up event statements

The first step in setting up an event statement is deciding what condition(s) you need to include. For most simple address and status conditions you probably need only an access breakpoint. We will start out here with those conditions however, to keep the first event statement simple.

Define an event statement for a write to "led_port"

```
>ev{1}=addr==led_port,stat==wr
```

ev{1}= begins the definition of event number 1.
addr==led_port defines "led_port" as the address of interest.
stat==wr defines the access to "led_port" as a write.

Displaying event status

Use **stat ev** to display the event in the VIEW(EV) viewport.

```
>stat ev
```

The first line in the VIEW(EV) viewport indicates there were 32 events available. We used one event, leaving 31 currently available.

The second and third lines display the event we set up, EV{1}. Notice that the address is displayed in both numerical form and by its symbolic name, "led_port".

The remaining lines indicate that no other events are defined.

If we had defined many events you would need to zoom the viewport to display them. The following demonstrates zooming and scrolling the VIEW(EV) viewport.

Zoom the VIEW(EV) viewport, scroll around, then unzoom

```
>z 24
><down arrow>
><up arrow>
>z 24
```

Step 2: Setting up trigger statements (assigning events to actions)

Next, you choose what action(s) are taken when the conditions defined in the event statement(s) occur. The event statement is tied to the action(s) by forming a trigger statement. When the event becomes true the actions happen.

We'll bind the simple action of stopping emulation to our event statement by defining trigger statement 1.

Define a trigger binding the break action to ev{1}

```
>trig{1}=ev{1},break
```

Arming triggers

Trigger statements must be "armed" before they are considered active. The easiest way to arm triggers is to leave the "auto-arm" variable **evtarm** at its default state of **on**. With **evtarm on** the triggers are automatically armed as soon as they are defined.

Triggers are armed for a particular "group". The **evtgrp** command determines what group a trigger statement is armed for when the trigger statement is defined. If you don't need to use an additional "group" of triggers leave **evtgrp** set to **1**. In this tutorial we leave **evtgrp** at its default value of **1**. This means the trigger we set up is armed for group 1.

Displaying trigger status

Use **stat trig** to display the trigger statement in the VIEW(TRIG) viewport.

Display trigger status and zoom the VIEW(TRIG) viewport

```
>stat trig
>z 24
```

The first line in the VIEW(TRIG) viewport indicates there were 16 triggers available. We used one, leaving 15 currently available.

The second line shows trigger statement 1 (TRIG {1}) is associated with (=) event statement 1 (EV{1}).

The third line indicates trigger 1 is armed for group 1.

The fourth line shows the action to take.

The fifth line shows the actual trigger statement definition.

The remaining lines indicate the other triggers are undefined, unarmed, and available (CLEAR).

Unzoom the VIEW(TRIG) viewport

```
>z 24
```

Step 3: Using the trigger statement

The trigger statement we set up is armed and ready for use. As soon as we **go** into "run" mode, the event system will unobtrusively monitor the executing environment until the event statement conditions are met. When this happens the actions in the trigger statement occur.

Restart, then run the program until the event systems takes over

```
>rest
>g
```

The emulator will "break" and display the current PC value. In the CODE viewport you should see a highlighted source line indicating where the break occurred. The symbol "led_port" should also be on this line.

Assigning a macro to an event system "break" action (breakcomplex)

Use the **breakcomplex** (**bc**) command to tie a macro to an event system "break" action. Since it severely limits the event system when in effect (only one armed trigger allowed), this command should only be used when you absolutely need to tie a macro to an event system "break" action. Use a macro tied to a basic breakpoint whenever possible.

To use **bc** the **evtmode** variable must be **on**. This variable's default is **off.** Unlike most other XICE variables, **evtmode** can only be changed by exiting the debugger, modifying the SW_EVTMODE command in the *xice.cfg* file, and then re-invoking the debugger.

You must set up for a **bc** in a carefully prescribed order.

1. Set up the event statement.
2. Tie the macro to the trigger statement and the bc.
3. Define the trigger statement with an event statement but no action. (No action is used in the trigger statement since the "break" action is built into the **bc** command.)

An example demonstrating the exact order of commands necessary to tie a macro to an event system "break" trigger follows. Assume **evtmode** is **on**.

For example only - Attaching the macro "dmp_trc" to an event system trigger

```
ev{1}=addr==led_port,stat==wr
bc trig{1}; dmp_trc()
trig{1}=ev{1}
```

Additional event system features

In addition to the simple conditions and actions illustrated here, the event system possesses many advanced features such as groups, counters, timers, flags, "soft shutdown", conditional tracing, trigger generation, and others. These features are covered in Chapter 4 of this supplement.

Scope loops and diagnostics

Built in scope loops and memory diagnostic programs are included with the debugger in the form of **diag** commands. These programs save you from writing your own routines to test memory or to stimulate memory for "scoping" or logic analysis.

Another diagnostic, named **crc**, calculates the CRC-16 (cyclic redundancy check) over the desired range.

Memory and IO read/write scope loops

Diagnostic numbers 2 through 8 are used to perform reads and/or writes of selected memory with patterns chosen by the user.

For example only - scope loop, continuous read of address 0x4000

```
diag 2,0x4000
```

Memory diagnostics, simple and complex

Diagnostic numbers 0 and 1 perform simple and complex diagnostics on the selected memory.

For example only - complex memory test of address range 0x2000 to 0x2400

```
diag 0,0x2000..0x2400
```

Cyclic Redundancy Check

Use the crc command with a range argument to perform a CRC-16 of the specified range. The command will return a hex value for the CRC.

For example only - cyclic redundancy check of address range 0x2000 to 0x2400

```
crc 0x2000..0x2400
```

Chapter 4
Using Breakpoints and the Event System

Overview

This chapter covers the breakpoint and conditional control features of XICE when used with the EL 1600 event system.

Basic breakpoints are tools for interrupting emulation or simulation in order to inspect trace for insight into code execution. Typically, they take two forms: access breakpoints and execution breakpoints.

The EL 1600 event system provides XICE with additional flexibility both in what can cause the emulator to intervene in code execution and in what actions can occur. These features provide powerful capabilities for debugging and integration.

Because all break capabilities and event conditionals are implemented using event system resources, the following sections explain both in terms of event system resource use. The initial sections explain the automatic invocation of event resources to provide basic access and execution breakpoints. Later sections explore using the full power of the event system to control emulation using sets of conditionals that you define.

Emulator and simulator versions of the debugger

This chapter deals only with XICE in-circuit debugger extensions to the XRAY simulated event system. The commands mentioned here are documented in the alphabetical command reference, Chapter 2, of this supplement.The XICE version uses emulator hardware resources to exercise and respond to actual processor and other target resources. For information about the XRAY simulator's event system, refer to the SET STATUS command description in the *XRAY Reference Manual.*

Breakpoint and event system commands

The following XICE commands comprise those for basic breakpoints and the event system.

Basic Breakpoint Command Names	Mnemonic	Function
BREAKACCESS	BA	Set a breakpoint on access to specified address(es)
BREAKINSTRUCTION	BI or B	Set a breakpoint on specified instruction
BREAKREAD	BR	Set a breakpoint on a read at specified address
BREAKWRITE	BW	Set a breakpoint on a write at specified address
Event System Command Names	**Mnemonic**	**Function**
BREAKCOMPLEX	BC	Set a breakpoint that calls a named macro.
EV	EV	Define event statement
EVTARM	EVTARM	Enable/disable automatic trigger arming
EVTCLR	EVTCLR	Clear event system
EVTGRP	EVTGRP	Specify event group
EVTVARS	EVTVARS	Display all event state variables
STATUS EVENT	STAT EV	Display event statement(s)
STATUS TRIGGER	STAT TRIG	Display trigger statement(s)
TRIG	TRIG	Define trigger statement

Working with basic breakpoints

You use a breakpoint to examine behavior of the target under certain controlled conditions. Typically you set breakpoints either on accesses to specific memory locations or on specific instructions in code. This is very helpful in isolating bugs when troubleshooting hardware and software in the target environment.

Setting basic breakpoints

There are two categories of basic breakpoints: access and instruction breakpoints. You use an access breakpoint (**BA**) to break on reads and/or writes to data and program locations. By specifying the type of access, you can limit the access break to break exclusively on read (**BR**) or write (**BW**) accesses. When you want to stop program execution on a particular instruction in your code, you use an instruction breakpoint (**BI**) or a temporary "go-until" breakpoint.

You set up basic breakpoints by typing the appropriate breakpoint command at the prompt. The debugger manages the event resources, assigns a number to each breakpoint for reference, and displays them in the breakpoint viewport (va 25).

Displaying breakpoints

Breakpoints are displayed in viewport 25. Using the **VACTIVE** command (**VA 25**), you can see the current breakpoints. This viewport also opens when you set a breakpoint, issue a breakpoint command without an argument, or use the command OPTION BREAK ON. (OPTION BREAK SWAP returns the window to normal mode.)

Clearing breakpoints

Breakpoints may be cleared using the **CLEAR** command. You can clear an individual breakpoint by giving its number or all breakpoints by not specifying a number.

Instruction breakpoints (BI, GO_*instruction*)

XICE uses a software execution breakpoint mechanism to provide up to 32 instruction execution breakpoints (BI), as well as temporary breakpoints (GO_*instruction*) and high-level single stepping. The break occurs immediately before the instruction actually executes.

Use the following syntax:

Example	Meaning
BI #20	Sets a breakpoint at line number 20. May require module name.
BI SIEVE\#28	Sets a breakpoint at line number 28 in the module SIEVE.
BI 0x2210..0x2216	Sets breakpoints starting at address 2210 and ending at address 2216 (hexadecimal), assembly-level mode only.
BREAKI #1..#4	Sets breakpoints starting at line number 1 and ending at line number 4.
BI #15..#18;FOO()	Sets breakpoints starting at line number 15 and ending at line number 18. Executes macro FOO after each line.

BI #10;when(i==3)	Sets a breakpoint at line number 10 and stops only if variable i is equal to 3.
BI 0x93	Sets a breakpoint at address 93 (hexadecimal), assembly-level mode only.
BI step	Sets a breakpoint at the address of step.

Ranges

You can specify an address range when defining a basic breakpoint (**BI**). XICE handles ranges by breaking them into multiple individual single-point breakpoints. Thus, if you specify that a breakpoint should be for a range of 20 addresses, you may set only 12 additional breakpoints.

Setting one-time breakpoints

A temporary or one-time breakpoint is an execution breakpoint attached to the current XICE GO instruction. Temporary breakpoints are commonly used to skip over a section of code or a subroutine. You can either click on an instruction in the code window (platforms with mouse support only) or enter the address or line number on which to break as a suffix to the GO command.

go 0x1234	Tells the emulator to run until it sees address 0x1234 on the bus

The same software execution breakpoint mechanism is used; the emulator runs to it and clears the breakpoint after breaking. If a macro or other operator is attached, it acts just as it would if attached to permanent breakpoints.

Trace display

When you view disassembled trace after an instruction breakpoint is reached, the display shows the instruction break at the point you defined. However, the raw trace display may

show extra pre-fetches that result from filling the CPU pipeline. The break has occurred at the specified point, as is shown by the CPU program counter (PC).

Initial setup

The execution breakpoint mechanism replaces the actual instruction in memory with a TRAP *n* instruction and sets an access breakpoint to stop execution on instruction fetches in the interrupt service routine. Consequently, these breakpoints *will* use target or overlay resources and can only be used in writable memory. So some setup is necessary before the initial use of execution breakpoints and temporary breakpoints.

In most cases, setup is virtually transparent, even for code that resides in PROM, if you use overlay memory and the defaults in *xice.cfg*:

- Dedicate a TRAP for XICE to use.

  The default set in *xice.cfg* is TRAP 15; if your code makes no use of this trap, skip to the next bullet.

  You must allocate exclusive use of one of the 16 TRAP vectors to XICE. Use the EXVEC command, or set a default with the SW_EXVEC softswitch in *xice.cfg*.

  The assigned trap in the vector table must be in writable memory so that it can be modified to point to the scratch area. This memory may be target RAM or emulator overlay ROM or RAM. If your vector table resides in target ROM or PROM, use the emulator's overlay memory to map the vector area as ROM; then the vector can be modified by XICE, but the area is protected from target writes during program execution. For example, if the vector table resides at 0x0000..0x0013 in target PROM, enter the following to map it to overlay ROM:

  rom 0x0000..0x0013

- Define scratch space in writable memory.

The default 8-byte scratch space set in *xice.cfg* starts at 0x9ff0; if your program never makes use of this area, skip to the next bullet.

XICE needs 8 bytes of writable memory for its breakpoint routine. This SCRATCH memory area may be either target RAM or emulator overlay RAM or ROM. If you do not have any unused target RAM, pick an area of target memory outside the boundaries of your memory map, and map overlay to that area.

Set the SW_SCRATCH softswitch in *xice.cfg* to the scratch area's starting address or allocate scratch space using the SCRATCH command before using a BI or temporary breakpoint. If the area you select does not return DTACK, see the guidelines in "Working within the limits" .

- Ensure that the breakpoint falls on an instruction boundary.

  If you use an instruction's label, XICE always places the breakpoint in the correct place. If you use a hex address in assembly mode or a line number in source mode, XICE does not check for alignment. In some instances, code corruption can result if you do not place the breakpoint on the start of the instruction.

- Ensure that the instruction resides in writable memory.

  The memory location on which you are placing the instruction breakpoint must be in writable memory so that the actual instruction can be replaced by the trap instruction. This may be either target RAM or emulator overlay memory. If the code resides in PROM, use the emulator's overlay memory to map the area as ROM; the code space can then be modified by XICE, but the area is protected from target writes during program execution.

  For example, to overlay the *wait* instruction from target PROM to ROM overlay, enter:

  rom wait=copy

For example, this is what takes place if you set an execution breakpoint on *wait* in the *cdemon* program using the defaults for scratch RAM and exception vectors:

bi wait

1. An address comparator and status comparator are set up to monitor execution of the code in the SCRATCH memory area at 0x9fff.
2. The *wait* instruction is replaced by the trap instruction specified by TRAP 15. The original *wait* instruction is saved to be restored immediately following any breakpoint.
3. During program execution, the trap executes.
4. The trap instruction causes an interrupt through the EXVEC trap vector. Code execution then begins in the SCRATCH memory area.
5. The event system detects program execution in the SCRATCH memory area and triggers the breakpoint.
6. The break is reported to XICE, and the wait instruction is restored in code.

Working within the limits

Execution breakpoints consume event system resources and affect what is possible using other features. In general, the emulator manages these resources and warns you when it makes adjustments and presents an error when resources are exhausted or when you attempt something that creates a conflict. So you need not concern yourself with more than the following general guidelines.

- If you use them in addition to the event system, note that instruction breakpoints consume an address and a status resource in each event group. Limit event system address/status resource consumption to no more than one address and one status comparator in each group. Set up the event and trigger statements, but leave them disarmed until you are ready to use them.

- You can specify an address range. XICE handles ranges by breaking them into multiple individual single-point breakpoints. Thus, if you specify that a breakpoint should be for a range of 20 addresses, you may set only 12 additional breakpoints.
- If you use them with access breakpoints, note the limitations explained in the "Access breakpoints" section that follows.
- When XICE is invoked, it performs a read of the area designated for SCRATCH if SW_SCRATCH is specified in *xice.cfg*. If SCRATCH is set to an area of memory that does not return a DTACK at the end of the read, it will hang XICE. In such a case, comment out the default address for SCRATCH (0x9ff0) in the *xice.cfg* file. Then specify the address for SCRATCH before setting any breakpoints by using the SCRATCH command or preferably in an include file when you invoke XICE.

Access breakpoints (BA, BR, BW)

An access breakpoint sets a read, write, or read/write breakpoint at the specified memory location or range. An access breakpoint halts program execution each time the target program attempts the specified type of access at the specified location(s).

Use an ampersand (**&**) to reference symbolic addresses rather than just the symbol names. Using a symbol name alone returns its value, not the address.

You have up to 6 access breakpoints of the same type (e.g., all reads) or up to 3 each of two different types. They are set using the syntax:

[ba | br | bw] [*address* | *address_range*] [*;macro_name*]

ba for read or write access

br for read access

bw for write access

Example	Meaning
BR 0x300	Sets a read access breakpoint at address 300 (hexadecimal).
BW @sieve\\flags	Sets a write access breakpoint at the address of the variable array flags in the root named @sieve.
BA flags..flags+10	Sets read/write access breakpoint starting at the address of the array flags and ending 10 bytes after the address of flags.
BR 20h..30h;FOO()	Sets read access breakpoints from address 20h (hexadecimal) to 30h and executes the macro FOO on every breakpoint between these addresses.
BR &flags[0]	Sets a read access breakpoint at the address of array element flags[0].
BA prime	Sets a read/write access breakpoint at the address referred to by the value in variable prime. This command is correct if prime is a pointer. The breakpoint is set at the location of the variable prime. For example, if the value of prime is 0x0123, a breakpoint is set at the address 0x0123.

	This command may not be correct if prime is a scalar, since the value in prime is treated as an address and the breakpoint is set at that address rather than at the address of the variable prime.
BW &prime	Sets a write access breakpoint at the address of the variable prime regardless of its type. This command is correct if prime is a scalar; it sets a breakpoint at the address of the variable prime. If prime is a pointer, the breakpoint is set at the address of the pointer rather than at the address it is pointing to (i.e., prime).
BA &count; when (k<30)	Sets a read/write access breakpoint at the address of count and only stops when the value of k is less than 30.

Access breakpoints begin breaking on the cycle in which the access occurs but may continue or "skid" several cycles after access to the breakpoint location.

XICE performs the following functions when it encounters an access breakpoint:

1. Completes the execution of the instruction at that location.
2. Suspends program execution.
3. Executes a macro (if one was specified when the breakpoint was set). Depending on the macro, the debugger will do one of the following:

   If the macro return value is true (nonzero), the debugger resumes execution at the instruction immediately after the breakpoint.

   If the macro return value is false (zero), the debugger returns to command mode and displays breakpoint information.
4. If a macro was not specified, XICE returns to command mode and displays updated breakpoint information.

Working within the limits

Access breakpoints consume event system resources and affect what is possible using other features. In general, the emulator manages these resources and warns you when it makes adjustments and presents an error when resources are exhausted or when you attempt something that creates a conflict. So you need not concern yourself with more than the following general guidelines.

- Up to 6 access breakpoints are possible when you are not using BI execution breakpoints. You can have 6 breakpoints of one access type (read/write/either) or two groups of 3 with different access types.
- Up to 3 access breaks of the same type (**ba**, **br**, or **bw**) are possible when you do use BI execution breakpoints. The first BI uses one each of the two available address and status comparators, leaving one of each for use with access breakpoints. Each address comparator may have three address points or one range associated with it; hence the three possible access breakpoints.
- Currently, an XRAY68K core limitation causes an error when you attempt to set an access breakpoint of one access type within a previously defined range of another access type. For example, if you set a read access break over the

range 0x4000..0x47ff, you cannot set a write access break for an address within that range.

- Setting an access breakpoint disables the event system. Likewise, arming an event trigger prevents use of access breakpoints.

Breakpoint latency

Instruction breakpoints have no execution latency associated with them. The break always occurs immediately prior to the execution of the instruction in question. Since the execution break mechanism does not generate any external events, there is no time-latency either.

The execution delay associated with an access breakpoint is up to six clock cycles, plus any instructions currently in progress at the time the request is completed.

Working with the event system

The event system provides emulator, target, and execution break control, allowing you to monitor any predefined series of conditions in realtime and then perform emulator actions based on those conditions.

The event system covered in this chapter is only available in XICE, the emulator version of the debugger. It monitors target information at the bus cycle level, including every read or write cycle that the microprocessor executes. The event system "sees" every signal that can affect the target system. It can take a variety of actions based on conditions that you predefine.

You can think of events (conditions) as inputs to the event system and triggers as statements that tie inputs to outputs. Figure 4-1 shows the structure of the event system. The possible events are listed on the left, and the possible triggers are listed on the right.

Basic concepts are introduced in Figure 4-1 and outlined in "Working with the Event System." More complete coverage of each procedure can be found under "Events" and "Triggers" later in this section. The tutorial in Chapter 3 also provides a practical exercise in event system use.

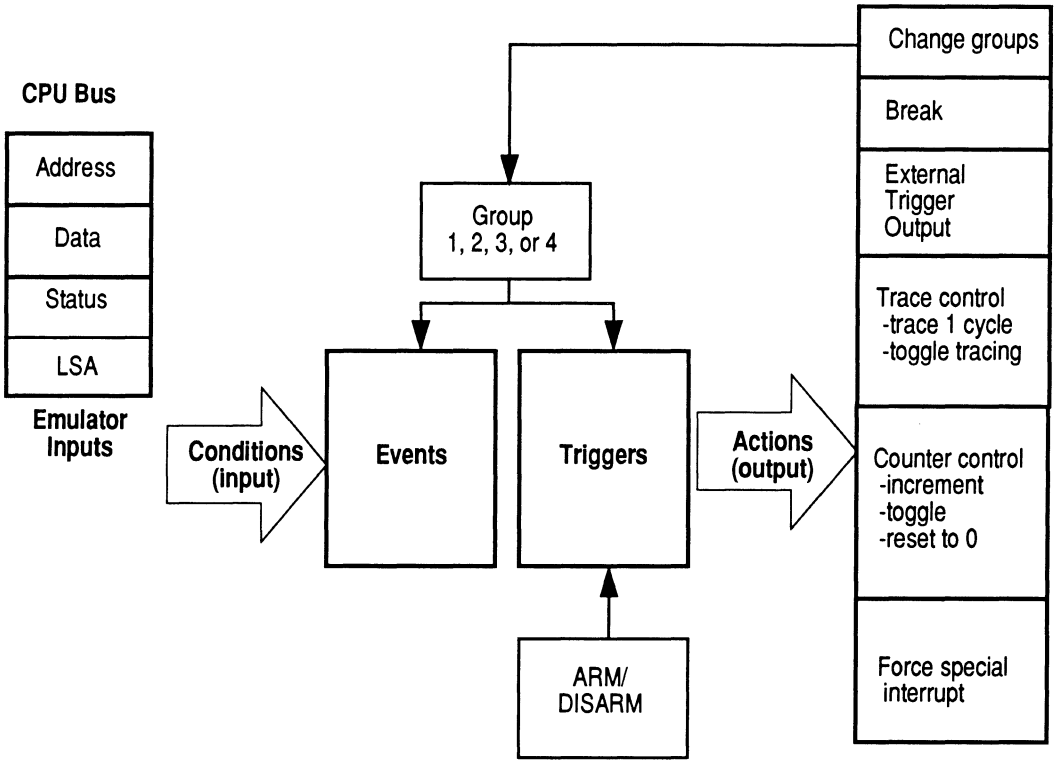

Figure 4-1 Event system structure

To use the event system, you first define conditions (event statements) using the **EV** command. Then, using the **TRIG** command, you define actions (trigger statements) that are to take place under those conditions. Up to 32 event statements and 16 trigger statements can be defined. Typically, you will modify the construction of the event system during the course of a debugging session. You can employ different events and triggers from among those defined as you go.

In order to be in effect, an event statement must:

- be defined
- be referred to in an active trigger statement

In order to be active, a trigger statement must:

- contain a reference to one or more event statements that are in effect
- be armed
- be in the group that is current

If you set EVTARM to ON, triggers are armed by default when defined. The event system hardware resources may limit the number of triggers that you can have armed at the same time. Disarming and rearming triggers with **TRIG** saves you from retyping them every time they are armed.

Example

ev{1}=data==0x1234	Event 1 is active when 1234 appears on data bus.
trig{1}=ev{1}, break	Event 1 triggers trig 1, which is a break.

Note

Break is the default action. The trigger in the example above could have been stated: `trig {1}=ev{1}`

Event and trigger statement groups

A group is a set of events and triggers that can be activated and deactivated all together. There are four groups in the event system (1, 2, 3, and 4), one of which is current at any given

time. This gives the event system a state-machine capability for debugging difficult problems. The emulator always starts in group 1 each time you enter the command GO.

When setting up events and triggers, you always associate them with a group. The group a particular event or trigger is associated with is the one current at the time you set it up.

The **GROUP** command shows the group that was current when the emulator last broke, and the **EVTGRP** command shows the group to which new event statements will be added.

Example

group	displays the number of the group current at the break.
evtgrp	displays the group new events and triggers will be added to.
evtgrp 2	Makes "group 2" the group new event or trigger statements will be added to.
group	Response not updated until next break.
evtgrp	Responds "2".

Displaying event system constructs

Event and trigger statements can be displayed using the **STATUS** command. By issuing the command **STAT EV** you can see the current event statements. By issuing the command **STAT TRIG** you can see the current trigger statements.

Clearing event system constructs

Event statements may be cleared using the **EV** or **TRIG** "clear" option. When a trigger is cleared, it is disarmed and cleared from XICE. When an event is cleared, it is removed from XICE as long as it is not currently referred to in an armed trigger. If

it is used by an armed trigger, then it is set in the emulator and cannot be cleared until all armed triggers that use it are either disarmed or cleared.

The **EVTCLR** command is provided to clear the entire event system. It clears the event and trigger statements in from XICE and resets the emulator's event hardware.

Logging event system constructs

If you wish to use a set of event and trigger statements for future debugging, you can use the **LOG** command to capture statements as you write them. You can then edit the log file with a text editor to create an include file that reproduces your event system setup.

Events

An event statement is constructed of one or more events and defines a portion of a potential bus cycle. The definition can include the state of the processor bus (the address, data, and status busses) and/or the state of other event resources (counters and LSA). You define an event to be valid when the portion of interest of the current cycle matches the value specified in the event statement. Parts of the cycle not defined in the event statement are ignored. The event system watches the busses and other event resources for valid events and matches them with actions specified in armed triggers.

You can define up to 32 event statements in XICE. A defined event can be used in any or all of the four event system groups.

Examples

```
ev{1}=addr==0x1234, status!=word
```

Event 1 is active if 0x1234 appears on the address bus and data is not a word.

Note

In an event statement, the comma (,) is the AND operator for the whole statement and the (!=) is the NOT operator for a whole event within the statement.

```
ev{1}=status==not_word|wr
```

Event 1 is active if this is not a word operation and the data bus is in write mode.

Note

The string "NOT_" is the NOT operator within the status event definition and the pipe (|) is the AND operator within the status event definition. Not all status conditions accept the "NOT_" construction. See the **SETSTATUS EV** description in Chapter 2.

The description of the **EV** command in Chapter 2 of this manual covers all the elements that can be included in an event definition. The following descriptions give detail about selected event topics that require more information.

Symbolic references

Symbol table values can be used throughout event system commands. The correct translations are performed for valid symbols to get actual symbol addresses, which can be used in defining events.

Ranges

You can specify an address range when defining a basic breakpoint (**BI**) or an event system event (**EV**). For EV commands, address ranges use a single address comparator. For BI commands, XICE handles ranges by breaking them into multiple individual single-point breakpoints. Thus, if you specify that a breakpoint should be for a range of 20 addresses, you may set only 12 additional breakpoints.

The syntax to specify a range is:

first_value..last_value

Example

```
ev{1}=addr==0x03fa..0x03ff
```

Event 1 is active when any address between 0x03FA and 0x03FF (inclusive) appears on the bus.

Don't-care masks

With the event system, you can also use don't care masks with data and LSA to further qualify a portion of a potential bus cycle for an event definition. The mask is a hexadecimal representation showing which bit positions are relevant and which are not relevant. You specify the don't care mask value immediately following the LSA or data. Bits masked with a 0 will be ignored.

The syntax is:

value &=mask_value

Example

```
ev{1}= data==0x3E13 &=0xFFFE
```

Event 1 is active when 0x3E13 or 0x3E12 is seen on the data bus.

Triggers

A trigger is a list of up to 8 ORed event statement references, followed by one or more action definitions. If no action is specified, the default action, **break**, is assumed.

Triggers are associated with one of the four event system groups, as determined by which group is current when the trigger is defined. When the event system encounters

conditions specified in an event statement, it initiates the actions called for in armed triggers for the current group that refer to that event definition.

Multiple events in a trigger are ORed; the OR operator in this construction is the pipe (|). Multiple actions are ANDed; the AND operator is the comma (,). A comma(,) is also used in the trigger definition to separate the events from the actions.

You may store up to 16 trigger statements at any given time although the number that may be armed may be smaller depending on the demand for event resources.

Example

```
trig {1}=ev{1}|ev{32},rct,grp2
```

When event 1 is valid or event 32 is valid, reset the counter, and change to group 2.

The description of the **TRIG** command in Chapter 2 of this manual covers all the elements that can be included in a trigger.

Disarming and rearming triggers

Triggers are armed (active) when they are defined if the **EVTARM** switch is ON (the default) and there are sufficient event system resources available. The purpose of the **ability to disarm triggers** is to allow you to keep triggers that you are not using or that would cause the emulator to run out of event system resources. You can then reuse them without having to retype them.

You can turn this switch on and off with the **EVTARM** command. You can also set EVTARM by including the statement SW_EVTARM:ON or SW_EVTARM:OFF in your XICE.CFG file.

At the time you define the trigger, XICE checks the **EVTARM** switch. If it is on, XICE attempts to arm the trigger by setting its events and then the trigger itself in the emulator. If arming fails—if, for example there are not enough resources

available—the trigger will be defined but disarmed. See "Event system limitations" on page 3-24 for a summary of event system resource limits.

Triggers may be explicitly disarmed and rearmed using **TRIG**.

Example

```
trig{10}=disarm
```

Using groups

Changing groups activates alternate sets of events and triggers. When setting up triggers, change the current group manually using the **EVTGRP** command if your next trigger statement is to apply to another group.

Note

The **GROUP** command shows the group that the emulator last broke in, and the **EVTGRP** command shows the group that new event statements will be added to.

As an example of the common use of groups, you may wish to trace a subroutine after it has been called by module A or module B, but not if it has been called from modules C, D, or E. In this case, you would define a set of event statements to the address ranges of modules A and B. When either of these modules is encountered, switch to group 2 and look for the subroutine. After tracing the subroutine, switch back to group 1.

Event system breaks

The event system can use either basic or complex breakpoints. However, you must choose which type to use before invoking XICE. The switch SW_EVTMODE should be set to ON or OFF in the XICE.CFG file. The default is OFF.

When the SW_EVTMODE switch is OFF in the XICE.CFG file, you can set as many triggers with BREAK as an action as can be supported by hardware. However, because the emulator cannot determine which of these caused an event system break, macros attached to event system breakpoints will not work.

Attaching macros to event system breakpoints

A BREAKCOMPLEX (BC) breakpoint is an event system breakpoint that can initiate a macro.

When the SW_EVTMODE variable is ON, only one event system breakpoint may be armed at a time. In this case you may use a complex breakpoint (BC) in order to call a macro. The **BC** command will bind the trigger specified to the sole complex breakpoint and will appear in the XICE breakpoint window. The complex breakpoint is defined using the **BC** command to tie the break to a trigger.

Note

It is illegal to specify BREAK directly in a trigger statement when SW_EVTMODE is ON.

Example

ev{1}=addr==0x0	Event 1 is valid when 0 shows up on the address bus.
trig{1}=ev{1}	Trigger 1 is set for event 1.
bc trig{1};foo()	Breaks and executes the macro "foo"

Note

SW_EVTMODE must be ON; trigger 1 must be armed.

Event system breaks and trace

The event system provides a variety of switches that affect how the system captures trace. These are listed in table 3-2 and covered in the alphabetical command reference, Chapter 2. Each switch can be set in XICE.CFG or read directly from the command prompt.

Switch	Description	Manual Ref.
PPT	Enable/Disable tracing of emulator peeks and pokes	S2-114
TED	Controls whether external DMA is traced	S2-139
TID	Controls whether internal DMA is traced	S2-141
TRCBRK	Controls the tracing of breakpoints	S2-138
TRCFRAME	Establish baseline traceframe number	S2-143
TRCINT	Display timestamps as INTERVAL/OFFSET	S2-144
TRCMODE	Display disassembled trace as ASM, SRC or BOTH	S2-145
TSTAMP	Controls whether timestamp or LSA is traced.	S2-153
TUNITS	Sets the timestamp units	S2-154

Event system limitations

XICE can keep track of up to 32 event statements and up to 16 trigger statements at the same time. Depending on the events and actions specified, different numbers of hardware resources are required by each trigger statement. In general, the emulator manages these resources and warns you when it makes adjustments and presents an error when resources are exhausted or when you attempt something that creates a conflict. So you need not concern yourself with more than the following general guidelines.

Statement limits

You can mix events (e.g. status, data, count...) in a single event statement. However, only one status, address, data, or LSA definition can be given in a single event statement. When you write the event statement, you can improve the flexibility by compounding resource references within the status, address, or data definition. Multiple status states can be ANDed within a status definition. Addresses and data can be given as ranges or qualified with don't care masks, but not both at the same time.

Example

```
ev{1}=addr==0x1234,data==0x0 &=0x0ff,status==byte|rd
```

Event 1 is valid when the address is 0x1234, and the least significant 8 bits of the data bus are 0x00, and read is asserted and access is by byte.

Trigger statement limits

Up to 8 event statements can be referred to in a trigger statement. (Multiple event statement references are ORed together.)

For a given XICE session, event system breaks (caused by a break action in a trigger definition) must be exclusively basic or exclusively complex. This is specified by the SW_EVTMODE switch in the XICE.CFG file. Basic breaks cannot be attached to macros, but you can specify a larger number of them. A complex break initiates a macro, but you are restricted to only one being armed at a time.

Hardware resource limits

You can free up event resources by disarming the triggers that refer to the event statements that employ those resources or by switching groups.

Events that are used by armed triggers use comparator hardware in the emulator.

The system keeps track of available resources and prompts you when you exceed the maximum. So you need not remember the specific number of event resources you have used. When you see "Limitation Error" or "Comparator Unavailable," you have reached the maximum. You may choose to eliminate unneeded events or use the **LOG** command to create and save separate event system setups for use in separate sessions.

Note

For BI, XICE handles ranges by breaking them into multiple individual single-point breakpoints. Thus, if you specify that an instruction breakpoint should be for a range of 20 addresses, you may set only 12 additional execution breakpoints.

Single-stepping emulation

The event system is not active during single-stepping in assembly-level mode. Trace will accumulate as you single step.

Chapter 5
Using Performance Analysis

The performance analysis features of XICE for the EL1600 Motorola 68000 and 68302 development system allow you to:

- determine which areas of a program use the most CPU time
- identify bottlenecks in time critical applications
- monitor the effects of programming changes made to improve throughput.

The performance analysis features are implemented using statistical performance analysis, which involves: sampling instruction fetch bus cycles at regular intervals using the event system; determining what function was active when a sample was taken; keeping a tally of the number of samples falling within each function; and reporting the sample information. The report is in a user-specifiable format. The default format is:

- Function name
- Percentage of samples falling within that function
- A horizontal histogram showing this percentage in relation to other functions.

A sample report is shown below.

```
Hits used: 8186  (8186 total, 0 excluded)
FUNCTION        PERCENT
-----------------------------------------------
func9:            30.8  ***************
func8:            17.1  ********
main:             16.6  ********
func7:            12.4  ******
func6:            11.1  *****
func5:             6.2  ***
func4:             3.1  *
func3:             1.6  *
func2:             0.8  *
func1:             0.3  *
```

Commands relating to performance analysis are listed in the table below. These commands are explained in more detail in Chapter 2 of this supplement.

PERFACT [STATISTICAL | OFF]	Off: no performance analysis data is gathered. Statistical: trace is uploaded periodically from the emulator and processed. This flag is tested while in run.
PERFCLR	Removes all accumulated performance analysis data.
PERFDATA [*symbol* | *string*]	Displays address range and number of hits associated with a symbol in the performance analysis display.
PERFDEPTH [0...]	Specifies maximum number of lines of performance analysis display output.
PERFDISP	Displays performance analysis information.
PERFEX [*address* | *address range* | *symbol*]	Allows convenient exclusion of address ranges and functions from performance analysis data (e.g. delay loops or functions).
PERFEXCLR [*address* | *address range* | *symbol*]	Clears exclusion(s) set with PERFEX.
PERFFORMAT [St*andard | Pe*rcent | Hi*ts | Ba*r | PH | PB | HB | PHB | All]	Determines display format of performance analysis information from the command line.
PERFINT [1 - 120]	Time interval in seconds between trace uploads from the emulator.

PERFMODE [A*lways | D*emand]	Always: Information is displayed whenever it is uploaded from the emulator. Demand: Information is only displayed on command (PERFDISP).
PERFTOL [*distance*]	Specifies maximum distance when searching for symbols.
Sw_perffmt_stat: [Standard | Percent | Hits | Bar | PH | PB | HB | PHB | All]	Determines display format of performance analysis information from XICE.CFG.

Event system setup

To use performance analysis, you must set up the event system to wait a number of bus cycles, then capture the next instruction fetch cycle. An include file to set up the event system is supplied; it is also shown below. To use it, follow these steps:

1. Load the program file.
2. Include the file for event system setup (perf.inc in your bin directory).
3. Give the following commands:

```
>go
```

Approximately every three seconds, captured trace will be uploaded from the emulator, processed and displayed.

Sample include file

Below is an include file suitable to set up the event system for performance analysis:

```
;; This file can be used to set up the EL-1600 68000
;; or 68302 event
;; system for statistical performance analysis.

;remove any residual event system setup
evtclr

evtgrp 1

ev{1}=stat==0xffffffff&=0xffffffff
trig{1}=ev{1},trc

ev{2}=addr==0..0xffffffff
trig{2}=ev{2},cnt

;; This timer value is the number of bus cycles that
;; are allowed to elapse
;; between grabs by the trace system. You will probably
;; want to change it
;; a few times during a P.A. run to assure that the
;; number is not in sync
;; with your program. If the period of this timer
;; matches a periodicity in
;; your program, very inaccurate results might occur.

ev{3}=count==1000
trig{3}=ev{3},rct,grp2

evtgrp 2


ev{4}=stat==prog|rd
trig{4}=ev{4},trc,grp1


; Gather P.A. info during run.
perfact statistical

; data collection interval
perfint 3

; display P.A. data every time trace is uploaded.
perfmode always
```

Special considerations

XICE must halt the processor when uploading trace data. It does not take very long, but it could interfere with the operation of your program if you have timing constraints.

Performance analysis relies on all instruction fetches appearing on the bus. If you are using the processor's cache, instructions may be fetched from the cache and performance analysis reports will be unreliable at best. For example, a tight loop which consumes 90% of all CPU cycles might appear on the bus only when it is entered (if it fits entirely in the cache), which would produce a very misleading performance analysis report.

Prefetch information is included as bus cycles are captured. This can add inaccuracies, since prefetch data often goes past the boundaries of a function.

Performance analysis uses the majority of event system resources, which means that you cannot do performance analysis and use a complicated event system setup at the same time. You might be able to share resources with the performance analysis setup, but you should check carefully to ensure that your additions do not interfere with the needs of the performance analysis setup.

Performance data and address range exclusions are automatically cleared when a **load** command is executed.

Limitations of statistical performance analysis

Statistical performance analysis can be inaccurate if the interval between samples matches a period within the program. For example, suppose you have the event system set up to sample every 1000 bus cycles (as in the include file above). Further suppose that there is a small routine in your program which is executed from a loop, and the length of the loop is roughly 1000 bus cycles. It could easily occur that almost

every sample would come from that same small routine, and hence that routine would appear to consume most of the CPU cycles even though it actually consumes much less than that.

The uploading of trace offers another opportunity for inaccuracy. For example, suppose you have XICE set up to upload trace data every three seconds (as in the include file above). Further suppose that you have a routine which is executed roughly every three seconds. That routine might never be sampled, if it always is executed in the period between when the trace buffer fills and when trace is uploaded. While a routine that is executed only every several seconds is unlikely to be a performance problem, the picture presented in the performance analysis reports will still be inaccurate.

For maximum accuracy, you should run performance analysis on your program several times, with different intervals between samples and uploads. Note that if you greatly increase the interval between samples, you may wish to increase the interval between uploads as well so you get a full trace buffer each time, although it will not cause problems if you do not. If you notice a drop in the number of samples collected between reports, this is what is happening.

Exclusion of address ranges

The ability to exclude address ranges allows you to discard data from functions and areas which are not of interest. For example, if you have a delay function, it may consume the majority of CPU cycles. This information might be interesting if your program should not be spending that much time in the delay function, but if you are interested in the performance of the rest of the program, it simply gets in the way.

The performance analysis system allows you to exclude address ranges in two different ways. The exclusion may be performed on the host or on the emulator. Each has advantages and disadvantages, and both may be used simultaneously to solve problems for which each is best suited.

The first way is via the **perfex** command. Its primary advantage is that any number of exclusions may be specified, covering arbitrary address ranges. Its primary disadvantage is that the exclusion is done on the host. For example, if you use **perfex** to exclude a delay function which consumes 90% of the CPU cycles, then roughly 90% of uploaded trace data will be discarded. This is highly wasteful. If additional exclusions are in force, all trace data in a given upload might be discarded.

The second way is by appending an event negating the range to ev{10} in the supplied event system setup include file (see above). The primary advantage of this method is that the exclusion is done on the emulator and resources are not wasted uploading trace data which falls within the excluded range. The primary disadvantage is that only one address range can be excluded in this manner. Note that if you use this method, you may wish to increase the interval between trace data uploads, since if (for example) you are excluding 90% of the potentially-sampled cycles, then it will take ten times as long to fill up the trace buffer.

You may use both methods at once. Suppose you have a delay function which consumes most of the CPU cycles, but wish to exclude other functions or ranges at the same time. You can use the event system to exclude the delay function and **perfex** to exclude the other functions. This setup means that resources are not wasted uploading thousands of samples which would be excluded, but at the same time you can exclude any number of functions or address ranges.

Chapter 6
Using the Time Stamp Module

This chapter describes the Time Stamp module and how to install and use it with the XICE debugger and the EL1600 emulator. Complete examples are provided for using the module to do each possible type of measurement. The sample raw trace display screens are for a Motorola 68302 processor; however, they are directly applicable to Motorola 68000 and Intel 80C18X family processors.

Overview

The Time Stamp feature is used for measuring time and for counting occurrences of events.

Commands Used to Set Up Time Stamp	
Command	**Description**
tstamp **on/off**	Choose timestamp or LSA
trcint **offset/interval**	Display the offset or interval time value
tunits **0x0-0xf**	Select time base (use same value as Time Stamp module switch)
trcframe ***n***	Center timestamp display around trace frame number n
ev{***n***}	Advanced Event System event statements
trig{***n***}	Advanced Event System trigger statements

Possible measurements

There are eight distinct measurements that can be made using the Time Stamp module. They are categorized into the two general groups shown below:

Elapsed time measurements

- Measure time spent in a module
- Measure time spent between modules
- Measure duration of time when memory is accessed (opcode or data)

- Measure duration of time when code is accessed (opcode only)
- Measure interrupt response time directly

Count occurrences

- Count number of times address or range of memory is accessed (opcode or data)
- Count number of times code is accessed (opcode only)
- Count module linkage activity (the number of times one module calls another)

Each time measurement is based on one of five scales: .1uS, 1uS, .01mS, .1mS or 1mS, which you specify. This allows you to collect your data using the appropriate time scale. The maximum number of counts for any time base is 65,535 so you have a maximum period of 65 seconds without overflow.

Time can be measured on an offset time frame, or on an interval time frame. When you use the offset time frame, the measurement is from the time the counter is reset and is centered around the raw trace frame selected by the XICE trace cycle number variable, **trcframe**.

When you use the interval time frame, the measurement is from one traced cycle to the next traced cycle. For example, if you were measuring the elapsed time for entering and exiting a module, the time displays would show as follows:

	Offset	Interval
enter	0ns	700ns
exit	-700ns	700ns
enter	0ns	700ns
exit	-700ns	700ns
enter	0ns	700ns
exit	-700ns	700ns

Installation

Hardware installation

The Time Stamp module consists of the module itself and the cable to connect it to the emulator.

There are three steps to hardware installation:

1. Turn the emulator off.
2. Remove the front cover of the emulator.
3. Connect the module to the 40-pin connector on the trace/ break board, as shown in the following illustration. Note that you cannot use the Logic State Analysis probe and the Time Stamp module at the same time.

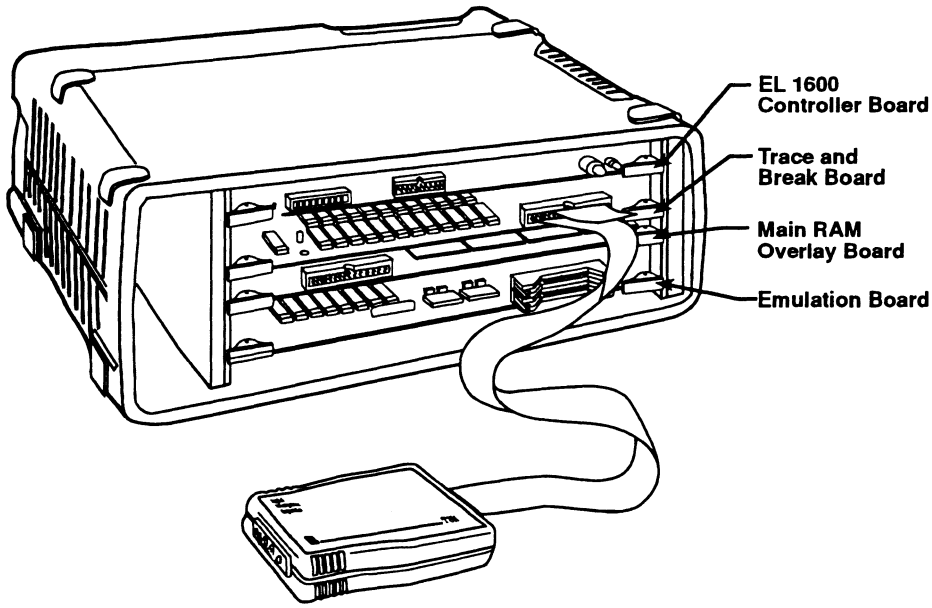

Figure 6-1 Connecting the Time Stamp module to the EL 1600

Caution

Before plugging in the cable, turn off power to the EL1600 emulator to prevent damage to the cable and module. Do not plug in or unplug the Time Stamp module with emulator power turned on.

Using the Time Stamp module

This section defines the labels, buttons, switches and LEDs on the Time Stamp module, and provides information on how the unit works.

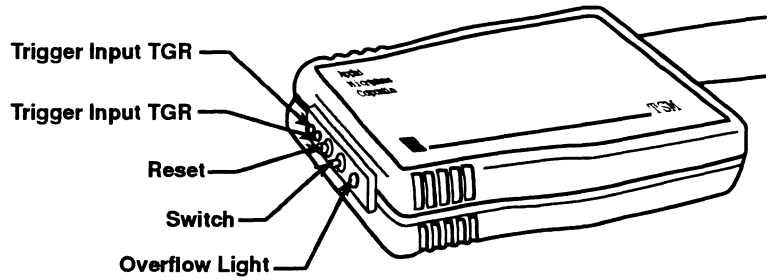

Figure 6-2 Time Stamp module

Getting started

Figure 6-3 shows the end of your Time Stamp module including the locations of the trigger inputs, reset button, switch and overflow indicator LED.

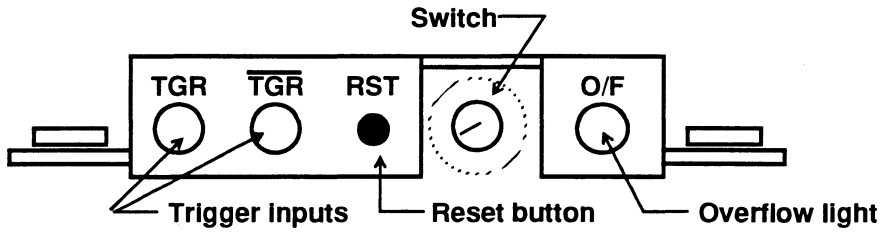

Figure 6-3 End view of Time Stamp module

TGR — The TGR input measures interrupt latency directly. The TGR input connects directly to the interrupt line in your target circuit to avoid any logic delays due to use of the Advanced Event System. It is designed for processors that pull lines low for interrupts. (Motorola) (see page 22)

TGR — The $\overline{\text{TGR}}$ input measures interrupt latency directly. The TGR input connects directly to the interrupt line in your target circuit to avoid any logic delays due to use of the Advanced Event System. It is designed for processors that pull lines high for interrupts. (Intel processors) (see page 22)

RST — The reset button resets the time stamp counter to 0.

Switch — The switch determines the time base and counting type. (see page 10)

O/F — The overflow LED indicates when the counter overflows the 65,535 limit.

The examples that follow give information on when to use the manual reset button, TGR and $\overline{\text{TGR}}$, and how to use the switch to choose the time stamp mode and time base.

Warning

Do not plug in or unplug the Time Stamp module when power is turned on to the emulator.

Making a measurement

The basic steps to make a measurement are as follows:

1. Set up the XICE variables for time stamp. Set the XICE variable **tstamp on** to display timestamp information in raw trace. Set the XICE **trcint** and **trcframe** variables to the appropriate values for the measurement you want to make.
2. Choose a switch setting on the Time Stamp module and set the XICE variable **tunits** to the same value as the switch.
3. Set up the trigger inputs.
4. Set up the Advanced Event System to trigger the Time Stamp module at the appropriate program states.
5. Run your program.
6. View the time stamp information.
7. Interpret the time stamp information.

Each step is described in detail below.

Step 1: Set up XICE timestamp variables

The XICE variables **tstamp**, **trcint**, **tunits**, and **trcframe** control the raw trace display of information coming in on the LSA/time stamp port. Settings **tstamp on, trcint offset** and **interval, tunits**, and **trcframe** are used with the Time Stamp option. Setting **tstamp off** is used with the LSA pod.

tstamp off	Default: LSA value shown as 16 bits
tstamp on, **trcint offset**	Display the offset time value
tstamp on, **trcint interval**	Display the interval time value
tunits 0x0-0xf	Select timestamp time base (use same value as switch)
trcframe **n**	Center timestamp display around trace frame number *n*

Offset time values are used when you want to measure the total amount of time spent or the number of occurrences. Interval time values are used when you are interested in the time spent between points A and B in your code, but are not interested in how long it takes to get to point A.

You may select values by editing the xice.cfg file or by using the following commands within XICE:

To select timestamp display in raw trace, enter:

```
>tstamp on.
```

To select display mode offset or interval, enter:

`>trcint x` (where *x* is **offset** or **interval**)

To select raw trace frame number for centering timestamp display in interval mode:

`> trcframe n` (where *n* is the raw trace frame number)

Step 2. Set up Time Stamp module switch

The time base for the timestamp is controlled by setting a switch on the Time Stamp module and setting the same value for the XICE timestamp units variable, **tunits**.

Choose a switch setting on your Time Stamp module based on your measurement type and preferred time base. We recommend starting with the slowest time frame: 1 mS. The table below shows the maximum measurable time period for each switch setting.

Time Base	Maximum Measurable Time Period
0.1 uS	6.5 milliseconds
1.0 uS	65 milliseconds
.01 mS	.65 second
0.1 mS	6.5 seconds
1.0 mS	65 seconds

Note

If the counter overflows, the yellow overflow LED will be lit. Check to see if you are using the correct time base for the duration of your measurements. When the counter overflows the 65,355 limit, it starts again at 0.

When the emulator is paused, no TGR is generated by the Advanced Event System in positions 0-4, so the counter is not reset and is likely to overflow. This is not a problem.

For example, the **drt** display might be as follows.

FRAME	ADDRESS	DATA	IPL	FCn	MEM	TIMESTAMP
#20	001124	4EF8	000	SP	RWO	700ns
#19	001100	4EF8	000	SP	RWO	700ns
#18	001124	754B	000	SP	RWO	700ns
#17	001100	4E71	000	SP	RWO	700ns

The following table summarizes the switch positions.

The trigger to start and stop the counter in the Time Stamp module is either the TGR signal from the Advanced Event System (Step 4), or the TGR or $\overline{\text{TGR}}$ direct input from your target interrupt line (Step 3).

Position	Time Base	Effect of TGR on Time Stamp Counter	Useful Measurements
0	.1 uS	Any TGR high causes the time stamp	Elapsed time
1	1 uS	counter to be reset to 0. No manual	
2	.01 mS	reset is required in this mode for either	
3	.1 mS	Offset or interval time stamping.	
4	1 mS		
5	.1 uS	While the TGR is held high by the	Elapsed time
6	1 uS	Advanced Event System, the time stamp	
7	.01 mS	counter counts. Manual reset is required	

8	.1 mS	in this mode for offset time stamping,	
9	1 mS	but not for interval time stamping.	
A	.1 uS	In this mode, a long TGR signal[1] from	Elapsed time
B	1 uS	the Advanced Event System resets the	
C	.01 mS	counter. After that, successive short TGR	
D	.1 mS	signals turn the counter on and off. Manual	
E	1 mS	reset stops the counter and sets it to zero.	
F	n.a.	This setting is used to count occurrences.	Count occurrences
		Each time the TGR signal goes high, the time stamp counter is incremented. Manual reset is required.	

[1] A long TGR is defined as being longer than 1.6 uS. This is the only mode where the length of the TGR matters. The following diagram shows what happens to the counter depending on the TGR signal.

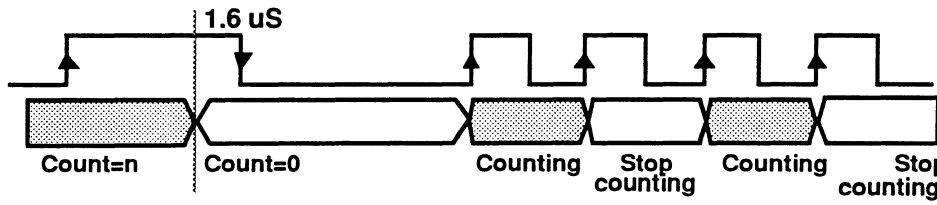

Figure 6-4 Positions A-E: Effects of multiple TGR signals

Step 3. Set up TGR input

The counter in the Time Stamp module can be controlled in one of three ways:

1. The Advanced Event System TGR action.
2. The TGR input.
3. The $\overline{\text{TGR}}$ input.

The default is the Advanced Event System trigger input. No additional wires are necessary.

To use the TGR and $\overline{\text{TGR}}$ lines to measure interrupt latency, you must connect one of these lines to an interrupt line on your target. Use of the TGR and $\overline{\text{TGR}}$ external inputs is described fully in the example on page 22.

Step 4. Set up the Advanced Event System

Set up the Advanced Event System to selectively trace the memory, program activity, or modules you are interested in time stamping. You can set up the Advanced Event System through the XICE command line or by using an XICE "include" command file.

The three steps to set up the Advanced Event System follow:

1. Decide what condition you want to look at, and what actions to take when that condition is reached.
2. Set up the event statements to isolate that condition.
3. Set up trigger statements using the appropriate events and actions.

For more information on using the Advanced Event System, please see Chapter 4 of this manual. The examples beginning on page 15 provide examples of using the Advanced Event System to specify conditions appropriate for time stamping.

Step 5. Run your program

Run XICE using the **go** command.

Step 6. View Time Stamp information

Displaying the time stamp information follows:

First enter XICE command line mode by either:

- stopping emulation with the **<Control-c>** command
- using the Advanced Event System to break emulation

Then view the trace, using the **drt** command.

The last column of the raw trace display shows either offset or interval time stamp, depending on the setting you specified with the **trcint** command.

In offset mode, use the **trcframe** ***n*** variable to select a raw trace frame number to center the timestamp display around.

Step 7. Interpret Time Stamp information

The time stamp information is displayed in either offset mode or interval mode depending on the setting of the XICE **trcint** variable.

In offset mode, timestamps are the time relative to the raw trace frame number specified by the **trcframe** command.

In the interval mode, timestamps are displayed as the time interval between successive bus cycles.

Collecting time stamp information in a file

After setting up your Advanced Event System and Time Stamp module to provide just the information you need, you can use the XICE **journal** command to save the specific drt displays to an ASCII file. You can use a spreadsheet or data base management program to analyze the data stored in the file.

Below is an example demonstrating saving 42 frames of raw trace to a file named *trace.raw*.

1. jou on="trace.raw"
2. drt 0..42
3. jou off

Examples

The two basic measurement modes are "elapsed time" and "counting occurrences". The examples are organized as follows:

Measuring elapsed time

- measure the time it takes to go from event A to event B
- measure the time the program is in the specified range
- measure the time between an interrupt and interrupt servicing

Counting occurrences

- count the number of times the program transitions from event A to event B
- count the number of accesses to a memory location or range

Measuring elapsed time

The elapsed time measurement is used to measure in-module time, out-of-module time, inter-module time, and memory and program access time. These measurements use switch positions 0 to E. You must set the XICE variable **tunits** to be the same value as selected with the Time Stamp module switch.

Conceptually, there are three types of elapsed time measurements:

1. Measuring the time from event A to event B

- used for measuring program time, out-of-module execution time, and inter-module execution time

2. Measuring the time spent in an address range

- used for measuring memory time and program time (excluding calls to other modules)

3. Measuring the time between an interrupt and interrupt servicing

- used for measuring interrupt latency

A to B mode

To measure the time it takes a program to get from event A to event B, set up the Advanced Event System so only event B appears in the trace display.

Step 1. Set LSA display type

`>trcint offset` (Set display format to offset time stamp)

Step 2. Select Time Stamp module switch setting

Use positions 0-4, depending on your preferred time base. In positions 0-4, the TGR from the Advanced Event System resets the time stamp counter to 0.

If you're not sure which time base to use, use position 4 for the slowest. If the counter overflows, the yellow overflow LED will light. See "Step 2. Set up Time Stamp module switch" on page 10 for a chart of maximum time periods per setting.

`>tunits 0xN` (Set N to same value as the switch)

Step 3. Set up the trigger input

To measure elapsed time, use the Event System trigger input.

Step 4. Set up the Advanced Event System

ev{1}=addr=="A"	(Specify event 1 in group 1 to be event A)
ev{2}=addr=="B"	(Specify event 2 in group 1 to be event B)
trig{1}=ev{1},tgr,trc	(Reset the time stamp counter to 0 and trace one cycle at event A)
trig{2}=ev{2},trc	(Trace event B)

Step 5. Run your program

>go	Run program

Step 6. Stop emulation

<Control-c>	Stop emulation

Step 7. View time stamp data

drt	Display the trace

Step 8. Interpret time stamp information

The last column of the trace display shows the offset time stamp information. Note that if event A and B are called more than once, you will get the time between events for each occurrence.

The following screen shows the raw trace display. The Time Stamp module switch was set to position #0 (.1 uSec). The time to go from A to B is shown to be 700 nSec.

FRAME	ADDRESS	DATA	IPL	FCn	MEM	DMA IAC	FLAGS	TIMESTAMP
41	001100	4E71	000	SP	RWO			700ns
40	001124	4EF8	000	SP	RWO			0ns
39	001100	4E71	000	SP	RWO			700ns
38	001124	4EF8	000	SP	RWO			0ns
37	001100	4E71	000	SP	RWO			700ns
36	001124	4EF8	000	SP	RWO			0ns
35	001100	4E71	000	SP	RWO			700ns
34	001124	4EF8	000	SP	RWO			0ns
33	001100	4E71	000	SP	RWO			700ns
32	001124	4EF8	000	SP	RWO			0ns
31	001100	4E71	000	SP	RWO			700ns
30	001124	4EF8	000	SP	RWO			0ns
29	001100	4E71	000	SP	RWO			700ns
28	001124	4EF8	000	SP	RWO			0ns
27	001100	4E71	000	SP	RWO			700ns
26	001124	4EF8	000	SP	RWO			0ns
25	001100	4E71	000	SP	RWO			700ns
24	001124	4EF8	000	SP	RWO			0ns
23	001100	4E71	000	SP	RWO			700ns
22	001124	4EF8	000	SP	RWO			0ns
21	001100	4E71	000	SP	RWO			700ns
20	001124	4EF8	000	SP	RWO			0ns
19	001100	4E71	000	SP	RWO			700ns
18	001124	4EF8	000	SP	RWO			0ns
17	001100	4E71	000	SP	RWO			700ns
16	001124	4EF8	000	SP	RWO			0ns
15	001100	4E71	000	SP	RWO			700ns
14	001124	4EF8	000	SP	RWO			0ns
13	001100	4E71	000	SP	RWO			700ns
12	001124	4EF8	000	SP	RWO			0ns
11	001100	4E71	000	SP	RWO			700ns
10	001124	4EF8	000	SP	RWO			0ns
9	001100	4E71	000	SP	RWO			700ns
8	001124	4EF8	000	SP	RWO			0ns
7	001100	4E71	000	SP	RWO			700ns
6	001124	4EF8	000	SP	RWO			0ns
5	001100	4E71	000	SP	RWO			700ns
4	001124	4EF8	000	SP	RWO			0ns
3	001100	4E71	000	SP	RWO			700ns
2	001124	4EF8	000	SP	RWO			0ns
1						BREAK		

Figure 6-5 Sample DRT screen for measuring time from A to B

Range Mode

In range mode, the trace display shows the amount of time the program is in the specified range.

Press the module's manual reset button prior to performing this measurement.

Step 1. Set LSA display type

trcint offset	Set display format to offset time stamp

Step 2. Select Time Stamp module switch setting

Use positions 5-9, depending on your preferred time base. In these positions, the Advanced Event System TGR enables the counter.

If you're not sure which time base to use, use position 9 for the slowest. If the counter overflows, the yellow overflow LED will light. See page 10 for a chart of maximum time periods per setting.

tunits 0xN	Set N to same value as the switch

Step 3. Set up the trigger input

To measure elapsed time, use the Event System Trigger input.

Step 4. Set up the Advanced Event System

ev{1}=addr=="range"	Set event 1 to be the specified address range
ev{2}=addr!="range"	Set event 2 to be outside the specified address range
trig{1}=ev{1},tgr,grp2	When range is accessed, enable counter and go to group 2

evtgrp 2	Begin arming triggers for group 2
trig{2}=ev{1}|ev{2},tgr	Keep counter enabled while in group 2
trig{3}=ev{2},grp1	Disable counter when not accessing range

Step 5. Run your program

go	Run program

Step 6. Stop emulation

<Control-c>	Stop emulation

Step 7. View time stamp data

drt	Display the trace

Step 8. Interpret time stamp information

The last column of the trace display shows the amount of time accumulated while the program was in the specified range.

The following screen shows the raw trace display, for the above example using a range of 0x1100 to 0x1124. The Time Stamp module switch was set to position #5 (0.1 uSec). The time spent in this range was 5.4 uSec.

FRAME	ADDRESS	DATA	IPL	FCn	MEM	DMA IAC	FLAGS	TIMESTAMP
34						BEGINNING OF TRACE		
33						BREAK		
32	00100A	4E71	000	SP	RWO			-5.400us
31	00100C	4E71	000	SP	RWO			-5.400us
30	00100E	4EF8	000	SP	RWO			-5.400us
29	001010	1000	000	SP	RWO			-5.400us
28	001000	4E71	000	SP	RWO			-5.400us
27	001002	4E71	000	SP	RWO			-5.400us
26	001004	4E71	000	SP	RWO			-5.400us
25	001006	4EF8	000	SP	RWO			-5.400us
24	001008	1100	000	SP	RWO			-5.400us
23	001100	4E71	000	SP	RWO			-5.400us
22	001102	4E71	000	SP	RWO			-5.300us
21	001104	4E71	000	SP	RWO			-5.000us
20	001106	4E71	000	SP	RWO			-4.700us
19	001108	4E71	000	SP	RWO			-4.400us
18	00110A	4E71	000	SP	RWO			-4.100us
17	00110C	4E71	000	SP	RWO			-3.800us
16	00110E	4E71	000	SP	RWO			-3.500us
15	001110	4E71	000	SP	RWO			-3.100us
14	001112	4E71	000	SP	RWO			-2.800us
13	001114	4E71	000	SP	RWO			-2.500us
12	001116	4E71	000	SP	RWO			-2.200us
11	001118	4E71	000	SP	RWO			-1.900us
10	00111A	4E71	000	SP	RWO			-1.600us
9	00111C	4E71	000	SP	RWO			-1.300us
8	00111E	4E71	000	SP	RWO			-1.000us
7	001120	4E71	000	SP	RWO			-600ns
6	001122	4E71	000	SP	RWO			-300ns
5	001124	4EF8	000	SP	RWO			0ns
4	001126	100A	000	SP	RWO		X	300ns
3	00100A	4E71	000	SP	RWO		X	700ns
2	00100C	4E71	000	SP	RWO		X	1.000us
1						BREAK		

Figure 6-6 Sample DRT screen for measuring time in range

Interrupt latency

To measure the amount of time between the detection of an interrupt and when it is serviced, connect your target interrupt line directly to the TGR or $\overline{\text{TGR}}$ lines on the Time Stamp module. As shown in Figure 6-7, these lines perform exactly

the same function as the Advanced Event System TGR signal, but the direct trigger bypasses the delays inherent in going through the additional Advanced Event System logic.

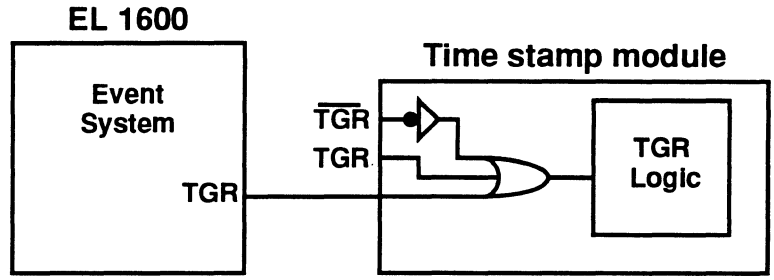

Figure 6-7 Trigger input logic

Figure 6-8 shows the trigger pattern for the TGR and $\overline{\text{TGR}}$ inputs.

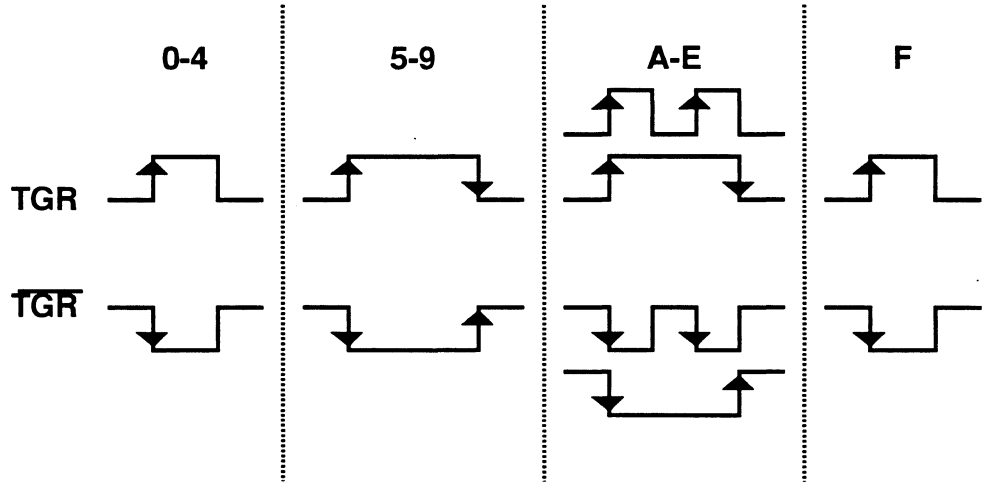

Figure 6-8 Trigger pattern for TGR and $\overline{TGR}$

Step 1. Set LSA display type

trcint offset	Set display format to offset time stamp

Step 2. Select Time Stamp module switch setting

Use positions 0-4, depending on your preferred time base. In positions 0-4, the TGR from the external TGR, external $\overline{TGR}$ or Advanced Event System TGR resets the time stamp counter to 0.

If you're not sure which time base to use, use position 4 for the slowest. If the counter overflows, the yellow overflow LED will light. See page 10 for a chart of maximum time periods per setting.

tunits 0xN	Set N to same value as the switch

Step 3. Set up the trigger input

Connect either the TGR or $\overline{\text{TGR}}$ input on the Time Stamp module to the interrupt line on your target that you want to check. For example, to check the interrupt latency for interrupt INT0 on the 80C186, use the setup shown in Figure 6-9.

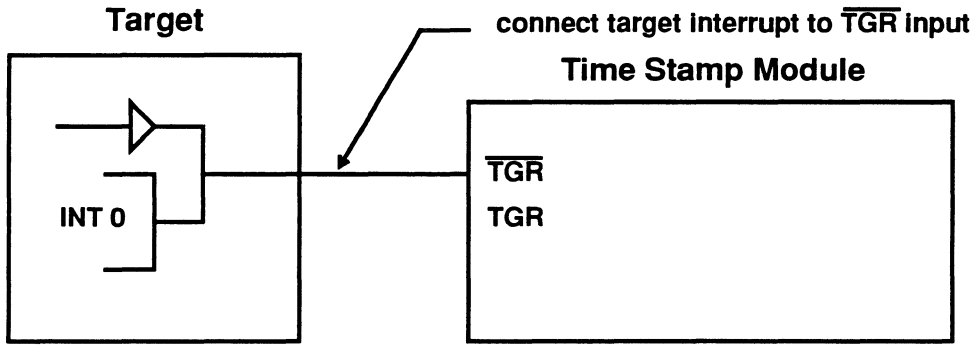

Figure 6-9 Target setup for measuring interrupt latency

Step 4. Set up the Advanced Event System

ev{1}=addr== **intservice_start**	Specify event 1 in group 1 to be the start of the interrupt service routine
trig{1}=ev{1},trc	Start tracing at the beginning of the interrupt service routine

Step 5. Run your program

go	Run program

Step 6. Stop emulation

<Control-c>	Stop emulation

Step 7. View time stamp data

drt	Display the trace

Step 8. Interpret time stamp information

The Advanced Event System traces the first cycle of the interrupt service routine. The last column of the trace display shows the amount of time elapsed between the start of the interrupt service routine and the actual interrupt processing.

Counting occurrences

The number of occurrences measurement is used to measure memory and program activity, module linkage activity and program flow activity. Use switch position F (count TGR pulses) for all counting measurements.

Two types of counting occurrences measurements follow:

1. Counting the number of times the program transitions from event A to event B

- used for measuring module linkage activity

2. Counting the number of accesses to some memory location(s).

- used for measuring memory program activity

A to B Mode

This mode records the number of times the transition from event A to event B occurs. Trace is only recorded on exit from module B. Press the module's manual reset button prior to performing this measurement.

Step 1. Set LSA display type

trcint offset	Set display format to offset timestamp

Step 2. Select Time Stamp module switch setting

Use timestamp module switch position F. For counting occurrences, the time base is irrelevant. In position F, when the TGR from the Advanced Event System goes high, the time stamp counter increments.

tunits 0xf	Set timestamp to count occurrences

Step 3. Set up the trigger input

To count occurrences, use the Event System Trigger input.

Step 4. Set up the Advanced Event System

ev{1}=addr==start_a	Specify event 1 in group 1 to be the start of module A
ev{2}=addr== start_b	Specify event 1 in group 2 to be the start of module B
ev{3}=addr== end_b	Specify event 2 in group 2 to be the end of module B
trig{1}=ev{1},grp2	Go to group 2 when module A is accessed
evtgrp 2	Begin arming triggers for group 2
trig{2}=ev{2},tgr	Increment counter when entering module B from module A
trig{3}=ev{3},trc,grp1	Exit module B, record count in trace memory

Step 5. Run your program

go	Run program

Step 6. Stop emulation

<Control-c>	Stop emulation

Step 7. View time stamp data

trcframe 2048	Set trace frame variable to end of trace
drt	Display the trace

Step 8. Interpret time stamp information

The last column shows the number of times module B was entered from module A. Note that only the locations **start_a** (1000) and **end_b** (1124) are traced. In the following screen we see that module B is called once each time from module A. The total number of calls is 0x3A31.

FRAME	ADDRESS	DATA	IPL	FCn	MEM	DMA IAC	FLAGS	TIMESTAMP
41	001124	4EF8	000	SP	RWO			3A1E
40	001000	4E71	000	SP	RWO			3A1E
39	001124	4EF8	000	SP	RWO			3A1F
38	001000	4E71	000	SP	RWO			3A1F
37	001124	4EF8	000	SP	RWO			3A20
36	001000	4E71	000	SP	RWO			3A20
35	001124	4EF8	000	SP	RWO			3A21
34	001000	4E71	000	SP	RWO			3A21
33	001124	4EF8	000	SP	RWO			3A22
32	001000	4E71	000	SP	RWO			3A22
31	001124	4EF8	000	SP	RWO			3A23
30	001000	4E71	000	SP	RWO			3A23
29	001124	4EF8	000	SP	RWO			3A24
28	001000	4E71	000	SP	RWO			3A24
27	001124	4EF8	000	SP	RWO			3A25
26	001000	4E71	000	SP	RWO			3A25
25	001124	4EF8	000	SP	RWO			3A26
24	001000	4E71	000	SP	RWO			3A26
23	001124	4EF8	000	SP	RWO			3A27
22	001000	4E71	000	SP	RWO			3A27
21	001124	4EF8	000	SP	RWO			3A28
20	001000	4E71	000	SP	RWO			3A28
19	001124	4EF8	000	SP	RWO			3A29
18	001000	4E71	000	SP	RWO			3A29
17	001124	4EF8	000	SP	RWO			3A2A
16	001000	4E71	000	SP	RWO			3A2A
15	001124	4EF8	000	SP	RWO			3A2B
14	001000	4E71	000	SP	RWO			3A2B
13	001124	4EF8	000	SP	RWO			3A2C
12	001000	4E71	000	SP	RWO			3A2C
11	001124	4EF8	000	SP	RWO			3A2D
10	001000	4E71	000	SP	RWO			3A2D
9	001124	4EF8	000	SP	RWO			3A2E
8	001000	4E71	000	SP	RWO			3A2E
7	001124	4EF8	000	SP	RWO			3A2F
6	001000	4E71	000	SP	RWO			3A2F
5	001124	4EF8	000	SP	RWO			3A30
4	001000	4E71	000	SP	RWO			3A30
3	001124	4EF8	000	SP	RWO			3A31
2	001000	4E71	000	SP	RWO			3A31
1						BREAK		

Figure 6-10 Sample DRT screen for counting occurrences

Range Mode

This mode records the number of accesses to some memory location(s). Trace is always recorded. The last trace cycles recorded show the accumulated access counts. Press the module's manual reset button prior to performing this measurement.

Step 1. Set LSA display type

trcint offset	Set display format to offset timestamp

Step 2. Select Time Stamp module switch setting

Use timestamp module switch position F. For counting occurrences, the time base is irrelevant. In this position, when the TGR from the Advanced Event System goes high, the time stamp counter increments.

tunits 0xf	Set timestamp to count occurrences

Step 3. Set up the trigger input

To count accesses, use the Event System Trigger input.

Step 4. Set up the Advanced Event System

ev{1}=addr=="range"	Specify the range to be monitored
trig{1}=ev{1},tgr	Increment counter whenever range is accessed

Step 5. Run your program

go	Run program

Step 6. Stop emulation

<Control-c>	Stop emulation

Step 7. View time stamp data

trcframe 2048	Set trace frame variable to end of trace
drt	Display the trace

Step 8. Interpret time stamp information

The last column of the last line of the trace display gives you the number of times the range was accessed. In the following sample screen, the range is set from 0x1100 to 0x1124. The range was accessed 4 times.

FRAME	ADDRESS	DATA	IPL	FCn	MEM	DMA IAC	FLAGS	TIMESTAMP
41	001122	4E71	000	SP	RWO			0002
40	001124	4EF8	000	SP	RWO			0002
39	001100	4E71	000	SP	RWO			0003
38	001102	4E71	000	SP	RWO			0003
37	001104	4E71	000	SP	RWO			0003
36	001106	4E71	000	SP	RWO			0003
35	001108	4E71	000	SP	RWO			0003
34	00110A	4E71	000	SP	RWO			0003
33	00110C	4E71	000	SP	RWO			0003
32	00110E	4E71	000	SP	RWO			0003
31	001110	4E71	000	SP	RWO			0003
30	001112	4E71	000	SP	RWO			0003
29	001114	4E71	000	SP	RWO			0003
28	001116	4E71	000	SP	RWO			0003
27	001118	4E71	000	SP	RWO			0003
26	00111A	4E71	000	SP	RWO			0003
25	00111C	4E71	000	SP	RWO			0003
24	00111E	4E71	000	SP	RWO			0003
23	001120	4E71	000	SP	RWO			0003
22	001122	4E71	000	SP	RWO			0003
21	001124	4EF8	000	SP	RWO			0003
20	001100	4E71	000	SP	RWO			0004
19	001102	4E71	000	SP	RWO			0004
18	001104	4E71	000	SP	RWO			0004
17	001106	4E71	000	SP	RWO			0004
16	001108	4E71	000	SP	RWO			0004
15	00110A	4E71	000	SP	RWO			0004
14	00110C	4E71	000	SP	RWO			0004
13	00110E	4E71	000	SP	RWO			0004
12	001110	4E71	000	SP	RWO			0004
11	001112	4E71	000	SP	RWO			0004
10	001114	4E71	000	SP	RWO			0004
9	001116	4E71	000	SP	RWO			0004
8	001118	4E71	000	SP	RWO			0004
7	00111A	4E71	000	SP	RWO			0004
6	00111C	4E71	000	SP	RWO			0004
5	00111E	4E71	000	SP	RWO			0004
4	001120	4E71	000	SP	RWO			0004
3	001122	4E71	000	SP	RWO			0004
2	001124	4EF8	000	SP	RWO			0004
1						BREAK		

Figure 6-11 Sample DRT screen counting occurrences in a range

Chapter 7
Simulated I/O

During the development of a system, it is often desirable to simulate input functions with canned input from a file or interactive input from the keyboard. Likewise, it is useful to be able to capture output functions and put the results in a file or display them on a screen.

This chapter explains how to use simulated I/O with XICE for character I/O. It also covers using simulated I/O with XRAY.

See your XRAY manual for documentation of the INPORT and OUTPORT commands.

Using simulated character input with XICE

Simulated character input is available for the EL 1600. The three steps required to do simulated character input are described below.

1. Start the XICE program. Activate an input trap by setting a breakpoint:

   In low-level mode, set a breakpoint immediately before the read of the location **_simulated_input** in the **INCHRW** routine inside the **inchrw.s** or **inchrw.src** file. The command to do so is as follows:

   ```
   bi INCHRW\inchrw;inport(&_simulated_input,1)
   ```

2. Define the stream to be used for input.

   You may use the keyboard or set up a file as the input stream for the simulated input.

   The command to set up input from the keyboard is as follows:

   ```
   INPORT &_simulated_input,std
   ```

   The command to set up input from a file is as follows:

   ```
   INPORT &_simulated_input,f=file_name
   ```

   You may need to enter other **INPORT** commands if you wish additional input streams.

3. Type **go** to begin processing your file. XICE performs the following tasks as it processes your file:

- XICE instructs the emulator to select a suitable hardware or software breakpoint type.
- The code emulation begins.
- When the breakinstruction point is reached, the emulator stops.
- XICE then waits for one byte to be available from the input channel you specified using the **INPORT** command.
- Next, XICE internally transfers the byte read to the location **_simulated_input**.
- The emulator single-steps over the original program code word (restored for a software breakpoint, if necessary).
- The code emulation continues.

Using simulated character output with XICE

Simulated character output is available for EL 1600. There are three steps required to do simulated character output which are described below.

1. Start the XICE program. Activate an output trap by setting a breakpoint:

   In low-level mode, set a breakpoint immediately after the write to the location **_simulated_output** in the **OUTCHR** routine inside the **outchr.c** file. The command to do so is as follows:

   ```
   bi OUTCHR\_sim_out_brk;outport(&_simulated_output,
   1,_simulated_output)
   ```

2. Define the stream to be used for output.

   You may use a viewport or set up a file as the destination for the simulated output.

   The command to set up output to the standard I/O window is as follows:

   ```
   OUTPORT &_simulated_output,std
   ```

   The command to set up output to a file is as follows:

   ```
   OUTPORT &_simulated_output,f=file_name
   ```

   You may need to enter other **OUTPORT** commands if you wish additional destinations.

3. Type **go** to begin processing your file. XICE performs the following tasks as it processes your file:

- XICE instructs the emulator to select a suitable hardware or software breakpoint type.
- The code emulation begins.
- When the breakinstruction point is reached (**_sim_out_brk**), the emulator stops.

- The emulator automatically restores the original program code word if you used a software breakpoint.
- Next, XICE internally transfers the byte that was written to **_simulated_output** to the port assigned via the command **OUTPORT.**
- The emulator then single-steps over the original program code word (restored for a software breakpoint, if necessary).
- The code emulation continues.

Using simulated character input with XRAY

The two steps required to do simulated character input using XRAY are described below.

1. Start the XRAY program. Define the stream to be used for input. You may use either the keyboard or set up a file as the input stream for the simulated input.

   The command to set up input from the keyboard is as follows:

   ```
   INPORT &_simulated_input, std
   ```

   The command to set up input from a file is as follows:

   ```
   INPORT &_simulated_input, f=file_name
   ```

   You may need to enter other **INPORT** commands if you wish additional streams for the input.

2. Type **go** to begin processing your file. When the access break occurs due to a read from the **_simulated_input** address, XRAY reads the simulated input from either the keyboard or from the file you specified, depending upon what you specified in the **INPORT** command.

XRAY can interrupt an instruction execution and redirect accesses in mid-instruction. For example, if the instruction

```
MOVE.B _simulated_input,D0
```

appears in the instruction flow, XRAY detects an access to **_simulated_input** and replaces it with the data from the keyboard or file depending on which you specified in the **INPORT** command. XRAY then substitutes the byte from the simulated input into the register **D0**, and continues to run.

Using simulated character output with XRAY

The two steps required to do simulated character output using XRAY are described below.

1. Start the XRAY program. Define the viewport or file to be used for output.

   You may use any legal XRAY viewport for simulated output. You may also set up a file as the destination for the simulated output.

   The command to set up a viewport for simulated output is as follows:

   ```
   OUTPORT &_simulated_output, std
   ```

   The command to set up output to a file is as follows:

   ```
   OUTPORT &_simulated_output, f=file_name
   ```

   You may need to enter other outport commands if you wish to direct the simulated output to other places as well.

2. Type **go** to begin processing your file. When the access break occurs due to a read from the **_simulated_output** address, XRAY detects this special instance and writes instead the simulated output to either the viewport or the file you specified.

XRAY can interrupt an instruction execution and redirect accesses in mid-instruction. For example, if the instruction

```
MOVE.B D0, _simulated_output
```

appears in the instruction flow, XRAY detects an access to **_simulated_output** and writes the data to the viewport or file depending on which you specified in the **OUTPORT** command. XRAY then continues to run.

The Master Index contains the indexes for the full documentation set related to the Applied toolchain and page numbering has been set up for this purpose. Numbers beginning with an R refer to reference manuals. Numbers beginning with a U refer to user guides. Numbers beginning with an I refer to installation guides. Lettered sections refer to appendices within manuals. For example, RA- refers to appendix A of a reference manaul. The document set to which the manual belongs appears in parenthesis following the page number reference. The following abbreviations are used:

ASM - ASM Documentation Set
MCC - MCC Documentation Set
CCC - CCC Documentation Set
XRAY - XRAY Documentation Set
SUP - XICE Supplement for the 68020, 68030, 68EC030 and the EL 3200
FLEX - Flexible License Manager Documentation Set

68000, 68EC000, 68HC000, 68302 Documentation Set Master Index

Applied Microsystems Corporation

Applied Microsystems Corporation maintains a worldwide network of direct sales offices committed to quality service and support. For information on products, pricing, or delivery, please call the nearest office listed below. If you are unsure which office to contact, call 1-800-426-3925 for assistance.

CORPORATE OFFICE
Applied Microsystems Corporation
5020 148th Avenue Northeast
P.O. Box 97002
Redmond, WA 98073-9702
(206) 882-2000
1-800-426-3925
Customer Support
1-800-ASK-4AMC
TRT TELEX 185196
Fax (206) 883-3049

EUROPE
Applied Microsystems Corporation Ltd
AMC House
South Street
Wendover
Aylesbury, Bucks
HP22 6EF England
44 (0) 296-625462
Telex 265871 REF WOT 004
Fax 44 (0) 296-623460

GERMANY
Applied Microsystems GmbH
Dammstrasse 6
W-6453 Seligenstadt
Germany
06182/9203-0
Fax 06182/9203-15

JAPAN
Applied Microsystems Japan, Ltd.
Nihon Seimei
Nishi-Gotanda Building
7-24-5 Nishi-Gotanda
Shinagawa-Ku
Tokyo T141, Japan
3-3493-0770
Fax 3-3493-7270

U.S. SALES OFFICES
Applied Microsystems
Corporation of Washington
3333 Bowers Avenue
Suite #220
Santa Clara, CA 95054
(408) 727-5433
Fax (408) 727-9011

Applied Microsystems
Corporation of Washington
25909 Pala Place
Suite #280
Mission Viejo, CA 92691
(714) 588-0585
Fax (714) 588-1476

Applied Microsystems Corporation
14643 Dallas Parkway
Suite 230, LB-76
Dallas, Texas 75240
(214) 991-6344
Fax (214) 991-4581

Applied Microsystems
919E North Plum Grove Road
Schaumburg, IL 60173
(708) 240-2000
Fax (708) 240-1309

Applied Microsystems
Corporation of Washington
6 Cabot Place
Stoughton, MA 02072
(617) 341-3121
Fax (617) 341-0245

Part No.	Revision History	Minimum SW Ver.	Date
922-17320-00	First release of XICE for 68000 and 68302 on an EL1600 emulator	XICE -PC 5.04 XICE-SUN 5.02	5/92
922-17320-01	Added support for X-windows, C++, operations during run	XICE 6.10	10/92
922-17320-02	Add 68HC/EC000	XICE 6.40	5/93

P/N 922-17320-02
May 1993