SLD™

Source Level Debugger for the PowerPack® Emulator

User's Manual

MICROTEK

Microtek International, Inc.
Doc. No. I49-001015
Part No. 14913-000

Trademark Acknowledgments

PowerPack is a registered trademark and SLD is a trademark of Microtek International.

IBM, PS/2, LAN, and OS/2 are trademarks of IBM.

Microsoft is a registered trademark and MS, MS-DOS, and Windows are trademarks of Microsoft Corporation.

Intel is a registered trademark and Intel386SX is a trademark of Intel Corporation.

Motorola is a registered trademark of Motorola, Inc.

UNIX is a trademark of AT&T.

PC-NFS is a registered trademark of Sun Microsystems.

MICROTEK INTERNATIONAL INC
6, Industry East Road 3
Science-based Industry Park
Hsinchu 30077
Taiwan, ROC
Tel: +886 35 772155
Fax: +886 35 772598

Development Systems Division
3300 N.W. 211th Terrace
Hillsboro, OR 97124-7136
USA
Tel: (503) 645-7333
Fax: (503) 629-8460

Table of Contents

Getting Started

The terms "PowerPack emulator" and "emulator" refer to the PowerPack® in-circuit emulator for embedded system development. The terms "SLD", "emulator software", and "debugger software" refer to the SLD™ source-level debugger for the PowerPack® emulator and PowerScope™ hardware-assisted debugger.

SLD runs under Windows 3.1 and Windows for Workgroups 3.11.

This chapter describes the parts, features, and documentation of the emulator and tells you how to contact Microtek International for information and technical support. This chapter also briefly describes how to start and end an emulator session and considerations for various compiler toolchains.

Documentation

Up And Running In 30 Minutes	**Chapter**	**Contents**
	Getting Started	Parts; features; documentation; support
	Software Installation	Configuring your PC or workstation and installing the SLD software
	Hardware Installation	Installing the hardware and running the confidence tests
	Tutorial	Practicing basic emulator commands and tasks
	Emulator Architecture	Schematics; physical dimensions; pinouts

User's Manual	**Chapter**	**Contents**
	Getting Started	Parts; features; documentation; contacting Microtek; starting and ending an emulator session; compiling a program for emulation
	Defining the Debug Environment	Configuring memory and registers; arranging your desktop; using an initialization file
	Debugging in Source	Viewing source code, disassembly, and stack; editing variables; controlling emulation
	Debugging in Registers and Memory	Accessing CPU and peripheral signals and numeric or disassembled memory contents

Debugging with Triggers and Trace	Controlling emulation and trace collection with triggers; numeric and symbolic addresses
powerpak.ini File Reference	powerpak.ini file contents
Toolbar Reference	Toolbar menus, buttons, and dialog boxes
Shell Window Reference	Shell window contents, menus, dialog boxes, and commands
Source Window Reference	Source window contents, menus, buttons, and dialog boxes
Variable Window Reference	Variable window contents, menus, and dialog boxes
Breakpoint Window Reference	Breakpoint window contents, menus, buttons, and dialog boxes
CPU Window Reference	CPU window contents, menu, and dialog boxes
Stack Window Reference	Stack window contents, menus, and dialog boxes
Memory Window Reference	Memory window contents, menus, and dialog boxes
Peripheral Window Reference	Peripheral window contents, menus, and dialog boxes
Event Window Reference	Event window fields, menus, and dialog boxes
Trigger Window Reference	Trigger window fields, menus, and dialog boxes
Trace Window Reference	Trace window contents, menus, and dialog boxes

Related Publications

For information on	See
Windows 3.1; Windows for Workgroups 3.11	Documentation from Microsoft
Your target processor	Intel or Motorola chipset documentation
Your toolchain	Documentation that came with the compiler, assembler, and linker you are using

IEEE-695 format	*IEEE Standard 695, Trial Use for Microprocessor Universal Format for Object Modules*, Microtec Research Inc., revision 4.1, Dec. 21, 1992
S-record format	Documentation that came with the compiler, assembler, and linker you are using
OMF86 or OMF386	Documentation from Intel
C++ name mangling	*The Annotated C++ Reference Manual*, Margaret Ellis and Bjarne Stroustrup (Addison-Wesley, 1990)

How to Contact Microtek

To register for technical support and to automatically receive product update information, complete and mail the registration card enclosed with the emulator.

Contact Microtek/DSD (see the number below) to purchase an Extended System Warranty (ESW). An ESW provides firmware, software, and hardware updates and priority service, in addition to all repairs.

As a Microtek customer, you can contact Microtek technical support for help with an emulator problem during your warranty period. The email and fax contacts are available 24 hours a day, 7 days a week. The voice phone numbers are available as listed below.

Internet email	csupport@microtekintl.com
Microtek/DSD, Western USA	(503) 645-7333 voice; (503) 629-8460 fax (voice contact available Monday through Friday, 8:00 am to 5:00 pm USA Pacific Time)
Microtek, Eastern USA	(610) 783-6366 voice; (610) 783-6360 fax (voice contact available Monday through Friday, 8:00 am to 5:00 pm USA Eastern Time)
Microtek, Hsinchu, Taiwan	+886-35-77-2155 voice; +886-35-77-2598 fax (voice contact available Monday through Friday, 8:00 am to 5:00 pm Taiwan Time)
Adara International, Taipei, Taiwan	+886-2-501-6699 voice; +886-2-505-0137 fax (voice contact available Monday through Friday, 8:00 am to 5:00 pm Taiwan Time)

Before you call, please read the *PowerPack® Emulator Problem Report Form* that came with the emulator. The form is also in the problem.txt

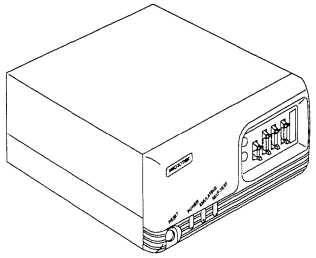
file, in your SLD installation directory (e.g. c:\powerpak) with the emulator software.

When you call, please be at your computer with SLD running and have the emulator documentation and filled-out problem report form nearby.

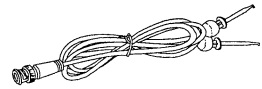
Emulator Parts

When you take the emulator out of its shipping package, check to be sure all the following are present (see the figure following this list):

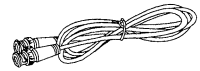
- the main chassis
- an EPOD
- a processor-specific probe for real-time emulation (yours may look different from the one in the picture)
- cables to connect the probe to the EPOD and the EPOD to the chassis
- a stand-alone self-test (SAST) or null target board (yours may look different from the one in the picture) for running emulator system diagnostics and code without your target system
- an RS-232C cable for communication between the chassis and your PC or workstation
- two BNC cables for trigger-out and trigger-in signals
- a power cord
- three SLD software program disks



Main Chassis

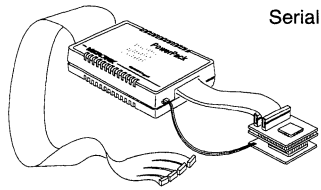


BNC-To-Hook Coaxial Cable

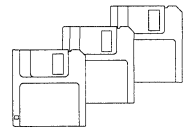


BNC-To-BNC Coaxial Cable

Serial Adapter



EPOD-Probe-Cable Assembly
with SAST or Null Target Board



Program Disks

RS-232 Cable

Power Cord

Emulator Power Requirements

CAUTION

Ensure the target is powered off before you connect or disconnect the PowerPack emulator. Otherwise, both units will be severely damaged.

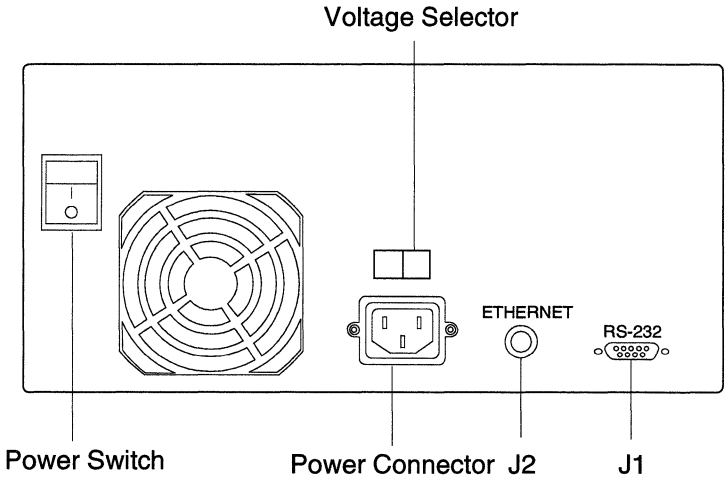
Turn off the target system before turning off the emulator. Power must be applied and removed in the correct sequence. Failure to follow this sequence will severely damage your target system and the emulator. Turn power on and off in the following sequence:

1. *Apply power to the emulator.*
2. *Apply power to the target system.*
3. *Remove power from the target system.*
4. *Remove power from the emulator.*

Ensure the line voltage selector is correctly set before applying power to the emulator.

The emulator chassis arrives from the factory configured to accept 115 VAC. For 220 VAC, be sure the emulator is powered off, then use a pen to change the line voltage switch to 220 VAC. The switch is

located above the power cord input jack on the rear of the main chassis. The following figure shows the rear of the emulator main chassis.



Emulator Features

The emulator main chassis, emulation pod, and probe module are connected by ribbon cables. A variety of adapters are available to connect the probe module to the target system. Contact Microtek for the appropriate adapter for your target processor package type. Connectors are provided for state probe clock and multiprocessor synchronization.

Communication between the main chassis and the PC host is via RS-232C (57600 bps) communications. Optionally, you can configure the emulator for an ethernet TCP/IP network for Sun Microsystems PC-NFS or for an IBM OS/2 LAN Server.

The emulator automatically configures itself for 5V or 3V operation.

You can substitute emulator-controlled overlay memory for your target RAM or ROM memory. Overlay memory allows zero wait states.

- For Intel processors, you can map 1M to 4M bytes of overlay RAM as target system memory, with up to 16 regions aligned on 4K-byte boundaries. The region sizes are multiples of 4K bytes.
- For Motorola processors, you can map 256K to 1M bytes of overlay RAM as target system memory, in two segments aligned on 64K-byte boundaries. The segment sizes are multiples of 64K bytes.

The PP SLD (PowerPack Source Language Debugger) software runs as a Microsoft Windows 3.1 or Windows for Workgroups 3.11 application with context-sensitive online help. Besides using a mouse or Windows-style keyboard entry with menus and buttons, you can enter commands via the SLD Shell window command line.

You can open several SLD windows at once. For example, you can monitor variables and view the trace while debugging at the source level. You can view two sections of source code simultaneously in the Source window. You can have up to 20 different Memory windows open simultaneously with various numeric, ASCII, and disassembly views of memory.

You can monitor the stack, the CPU registers, the peripheral registers (as appropriate for your processor), and memory contents during emulation.

A single-line assembler is available for patching loaded code.

You can debug from the vantage of your C and assembly language source:

- All symbol types are supported, including static variables, stack-based local variables, register-based variables, structures, arrays, and pointers.
- You can selectively load object code and symbolic information into target or overlay memory and into the symbol table, for load formats including OMF86 and OMF386 for Intel targets and IEEE-695 and S-record for Motorola targets.
- Source display formats include C and assembly language from your source files, disassembly from memory when the source files are unavailable, and disassembly from memory interleaved with the corresponding lines from your source files.
- Emulation control includes Go and Step operations of specifiable granularity relative to lines, statements, and function calls, with breakpoints settable on a source line, on a statement within a line, and on the address of a particular instruction.

Real-time, full-speed tracing is available:

- You can configure a single buffer to capture 256K bytes of trace; or 256 buffers to capture 1K bytes each, or various intermediate combinations of buffer size and number of buffers.
- You can collect trace before, after, or centered around a specified event or sequence of events.
- You can search the collected trace to find a specific event.

- You can display trace as instructions, bus cycles, or clock cycles.
- Trace information can include signals, addresses, and data, at each bus or clock cycle, and timestamps for each trace frame relative to a specific event or relative to the preceding trace frame.
- You can link the Trace and Source windows to scroll together.

Trace control and emulation control are independent of each other. During emulation, you can start and stop trace collection and view trace without affecting emulation.

Besides manually starting and stopping trace and emulation, you can define up to four sequential triggers to conditionally control emulation and trace collection. Each trigger is a logical combination of up to eight events, with optional counter and timer dependencies:

- An event is defined as inclusive, exclusive, and masked address and data ranges or patterns and various signal values.
- With multiple buffers specified, a triggered action can capture one buffer and start filling the next. You can break emulation when all the trace buffers are full.
- You can control triggering relative to events by programming two 10-bit counters or one 20-bit timer.

You can set breakpoints by clicking on a source line or from the menus:

- 256 software breakpoints are available.
- For Intel emulation, up to four hardware breakpoints and for Motorola emulation, two hardware breakpoints are available.
- The emulator automatically chooses whether a breakpoint is set in hardware or in software; for Intel emulation, you can access the debug registers to explicitly specify a hardware data or execution breakpoint.

Host System Requirements and Recommendations

- An Intel486 or Pentium based PC or 100% compatible system
- MS-DOS 5.0 or 6.x with Windows 3.1 or Windows for Workgroups 3.11 running in 386-enhanced mode
- At least 6M bytes of RAM
- At least 5M bytes of free memory after you have loaded Windows or Windows for Workgroups and any other applications besides SLD.

- At least 5M bytes of available disk space
- A VGA or Super VGA graphics card and color monitor (a graphics accelerator card recommended to boost performance; a monitor capable of at least 640x480 operation recommended)
- A mouse
- A serial port for connection to the emulator (16550 UART recommended for operation at 57.6K baud)
- At least 4M bytes for a swap file (permanent swap file recommended, with a disk cache such as smartdrive for improved Windows performance)
- Config.sys entries of at least Files=30 and Buffers=30

Starting an Emulator Session

CAUTION

Turn on the emulator before turning on your target system. Power must be applied and removed in the correct sequence. Failure to follow this sequence will severely damage your target system and the emulator. Turn power on and off in the following sequence:

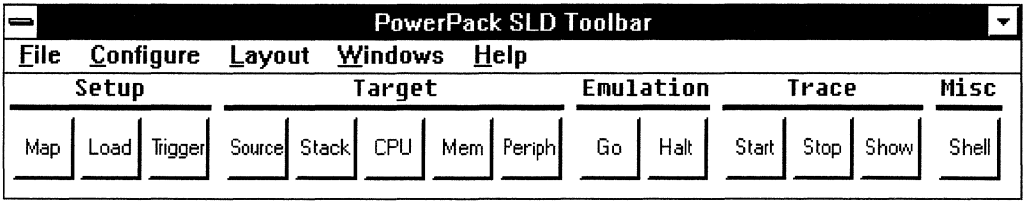
1. *Apply power to the emulator.*
2. *Apply power to the target system.*
3. *Remove power from the target system.*
4. *Remove power from the emulator.*

PowerPack SLD

Once the software is installed on your host computer, the firmware is loaded into your emulator, and your target system and the emulator are powered-on, start an emulation session from the PowerPack SLD icon (see figure at left).

The Toolbar is the first window open when you invoke SLD and must remain open throughout your emulation session. Closing the Toolbar exits SLD. Minimizing the Toolbar hides any other open (including minimized) SLD windows; restoring the Toolbar redisplays (with the same screen layout) any SLD windows that were open when you minimized the Toolbar.

The following figure shows the Toolbar. For some emulators, the buttons for unavailable operations are grayed-out; for example, the Periph button is nonfunctional on the Intel386 CX/SX emulator because no peripheral registers are available.



Buttons and menus on the Toolbar provide quick access to the most frequently used commands and other SLD windows. When you start an emulator session, use the Toolbar to map overlay memory, load code and symbols, and open the Source, Memory, and Shell windows for further work. Also, you can use the Toolbar to conveniently open the Peripheral, CPU, Trigger, and Trace windows, start and stop emulation, and start and stop tracing.

Before loading your program, map any overlay memory you need. Also, you may want to preconfigure your processor chip selects or other registers as described in the "Defining the Debug Environment" chapter.

Be sure your loadfile is in OMF86 or OMF386 for an Intel emulator or in IEEE-695 or S-record format for a Motorola emulator. Intel-compatible toolchains generally provide options for generating the appropriate OMF. Many Motorola-compatible toolchains include a converter for turning the toolchain vendor's proprietary format into IEEE-695 format. Contact your software development toolchain vendor for information on generating the appropriate loadfile format.

To debug at the source level (i.e. with source code and symbolic names for functions and variables), you must retain symbolic debugging information in your loadfile. Use compiler, assembler, and linker switches to suppress optimization and to add symbolic information. See your toolchain documentation.

CAUTION

You can load files while the emulator is running. Be sure loading is at a location other than where the program is running. Loading at a location that is in use can halt emulation in an unpredictable state.

Ending an Emulator Session

To end an emulator session, do one of:

- Choose the Exit command from the file menu on the Toolbar.
- Double-click the system box in the upper left corner of the Toolbar.
- With focus on the Toolbar, press <Alt><F4>.

CAUTION

Turn off your target system before turning off the emulator. Power must be applied and removed in the correct sequence. Failure to follow this sequence will severely damage your target system and the emulator. Turn power on and off in the following sequence:

1. *Apply power to the emulator.*
2. *Apply power to the target system.*
3. *Remove power from the target system.*
4. *Remove power from the emulator.*

Getting Online Help


PowerPack
SLD Help

Whether or not SLD is active, you can invoke the SLD online help directly from Windows Program Manager. From the PowerPack SLD group, choose the Help icon (see figure at left).

SLD online help conforms to the standard Windows help interface, as described in your Microsoft Windows documentation. From any SLD window, open the Help menu and choose a Help category; or, press <F1> at any time. In most SLD dialog and message boxes, you can choose a Help button for context-sensitive help.

If this is the first time you are using Help, you may want to choose "How to Use Help" from the Help menu. (Or, press <F1> twice.)

Compiling for Intel Processor Emulation

Because of standards developed for Intel OMF86 and OMF386 loadfile formats, there is little difference in the output formats of most Intel development toolchains.

When using the Metaware HC toolchain, compile with the switch Optimize_for_Space (-Os) OFF and the switch Align_Routines ON. This combination aligns the line number information for function entry points on the actual function execution addresses. This alignment is necessary for SLD to set source line breakpoints on the start addresses of the function entries and to successfully display local symbols for inspection.

When using the Borland C compiler, before loading your OMF386 loadfile, set the emulator's maximum bitfield size to 16 bits. On the SLD Shell command line enter:

```
maxBitFieldSize 16
```

When using PharLap LinkLoc 7.1, you can include symbolic information for register variables with LinkLoc's -regvars switch. The emulator supports register variable extensions to the Intel symbol table.

Compiling for Motorola Processor Emulation

Because of implementation-dependent variations in IEEE-695 loadfile formats, the PowerPack emulator supports different Motorola-development toolchains differently. This section describes considerations for using the supported toolchains. For a list of currently supported toolchains, see the readme file installed with SLD.

You must specify the compiler before loading your first file. Once you have specified a compiler, you need not specify it again unless you change compilers. The first time you load a file using the Toolbar Load button or the Source window File menu Load item, the emulator displays the Compiler Used dialog box. Select one of the listed compilers.

If you load the file using a Load command on the Shell command line, the Compiler Used dialog box does not appear. Before loading, enter a CompilerUsed Shell command to specify the compiler as Hiware, Intermetrics, Introl, MRI, SDS CrossCode, Sierra, or Whitesmiths. Or, in the Source window, open the Options menu, choose Compiler Used, and select the appropriate compiler. (For the most current list of supported toolchains, immediately after installing SLD look in your windows/powerpak.ini file [ToolChain] section.) If your toolchain is unsupported, specify it as Unknown.

If the code and data section names in your loadfile are not the default section names generated by your compiler, edit the [ToolChain] section to describe the section names in your loadfile. For example, if you generate a loadfile using the MRI compiler but with section names mycode and mydata, change the MRI= line in [ToolChain] as follows:

```
[ToolChain]
Compilers=Unknown,MRI,...
CompilerUsed=MRI
MRI=mycode,mydata
```

For more information on compiler support in powerpak.ini, see the "powerpak.ini File Reference" chapter.

CAUTION

The PowerPack emulator and SLD software are not guaranteed to work properly with unsupported toolchains.

MRI

Use the following switches:

-g	compiles with debug symbols.
-Gf	embeds the source path in the loadfile during compilation.
-O<*x*>	(where *x* is a letter designating an optimization) is optional. Supported optimizations include algebraic simplification, constant folding, strength reduction, redundant code elimination, unreachable code elimination, local optimizations performed globally and loop optimization (-Ol), and register coloring (-OR). Register coloring uses one register for multiple variables each of which has its own lifetime information. This includes factorization, dead code elimination, unused definition elimination, global constant propagation, global copy propagation, and branch merging. Avoid using -Oc, -Og, or -Oi.
-Wa, -f"NOPCR"	prevents the assembler from generating PC-relative jumps.

The MRI compiler truncates long variable names to 125 characters. Also, SLD recognizes only the first 125 characters of such names.

For bit fields, only some type information is preserved. The compiler uses a default unsigned type for all types of declared bit fields.

Before modifying an unused local or parameter variable, verify its storage location with a DisplaySymbols Shell command. The MRI compiler optimizes storage allocation by placing such variables into a scratch register, usually A0 for pointers and D0 for other types.

Variables in previous stack frames unused after a function call may be assigned to a scratch register which may, in turn, be used by a subsequent function. Unused parameters can also remain on the stack after function entry. If that occurs, the values displayed for such variables in the Variable window may be incorrect. To discover whether the compiler added housekeeping code to ensure such variables are popped off the stack, in the Source window open the View menu and check Mixed Source And Assembly.

If you use a tool (such as Cfront) to generate C source from C++ source, then use the MRI C compiler and linker to generate an IEEE-695 loadfile from the C source, the line number records in the loadfile will

match the C++ source lines not the C source lines. The C++ preprocessing puts #line directives in the C source corresponding to the original C++ source line numbers. Use a text editor to delete these directives before compiling the C source, to ensure the line numbers in your loadfile match the C source text in the SLD Source window. This match is necessary for tasks such as setting breakpoints interactively and selecting symbols in the Source window.

Besides the line number information, the C source contains information about the original C++ source file. To use the C source, you must delete the C++ information. For example, from a C++ source file named file.cc, the name file_cc will appear in the C source. Change all such occurrences to the name of your C source file (e.g. file_c for a C source file named file.c).

Intermetrics

Use the following compiler switches:

-d generates debug information.

-do turns off optimization

-nr is optional. This switch optimizes for algebraic simplification, constant folding, strength reduction, redundant code elimination, and unreachable code elimination.

-np is optional. This switch optimizes for register coloring, which uses one register for multiple variables each of which has its own lifetime information. This includes factorization, dead code elimination, unused definition elimination, global constant propagation, global copy propagation, and branch merging.

Avoid using -nl, -nal, or -n7<*y*>.

Use the following converter (FORM695) switches:

-d generates debug information

abs generates absolute code

Sierra

Although the Sierra compiler supports the Motorola fast-float type, the SLD Variable window does not. The value is displayed incorrectly for this type. Standard float and double types are displayed correctly.

Use the following compiler switches:

CFLAGS -q compiler flag.

CAFLAGS -6 compiler-generated-assembly-code assembler flags

AFLAGS -6 -L -S1 programmer-generated code assembler flag

LFLAGS -P linker flag

Use the -m converter (Conv68) switch to generate IEEE-695 load format.

You must specify the stack base and size in powerpak.ini or after starting SLD; the Sierra compiler does not put this stack information in the loadfile.

Introl

Use the -gg compiler switch to generate symbolic information.

For the I695.EXE converter, use -s__start (note the double underbar) to specify __start as the starting label to generate the starting PC loader record. Avoid deleting __start, which initializes the Source window display and sets up the program counter. (With no starting PC, the Source window displays memory starting at 0x0.) The compiler puts __stext in the startup code.

If you get unexplained errors on loading, turn-off On Demand symbol loading.

Whitesmiths

Use the following compiler switches:

-dxdebug turns on debug symbols.

-dmod<*m*> specifies the memory model, where *m* is one of:

- c compact
- s small
- d data
- p program
- f far

+o compiles and assembles, but does not link.

To create a stack segment at <*location*> with <*size*>, use the following linker directive last:

+bss -n stack -b <*location*> +spbss <*size*>

Use the following converter switches:

-mod<*m*> specifies the memory model, where *m* is one of:

c compact

s small

d data

p program

f far

-p6816 specifies the HC16 processor.

The stack frame for a function is invalid until the first 2 or 3 assembly instructions, generated by the compiler, have executed. Step one source statement into the function to display a valid stack.

Avoid deleting __stext (note the double underline), which initializes the Source window display and sets up the program counter (PC). When there is no starting PC, the Source window displays memory starting at 0x0. The compiler puts __stext in the startup code.

HiWare

To support bitfields properly, add the following to powerpak.ini:

```
[VariableInfo]
AutoCalcBitfieldOffsets=1
```

(For other Motorola compilers, this value must be 0.)

Defining the Debug Environment

This chapter describes:

- *Configuring SLD for your target processor and for your personal working style*
- *Running command scripts and specifying a script to run automatically when you start SLD*

Before starting emulation, initialize the emulator for the modules you are debugging and arrange the desktop for your own convenience. Such preliminary tasks can include:

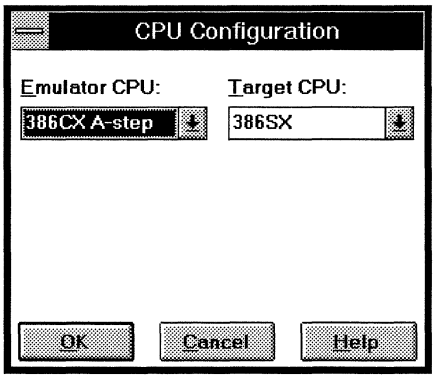
- Start a record of your Shell window activities.
- Map memory, put default values in memory, and specify some aspects of how your loadfile will be loaded.
- Enable display updates to occur during emulation.
- Enable signals and set CPU and peripheral register values specific to your processor or to your loadfile. (See the Intel and Motorola processor examples at the end of this section.)

You can do many of these tasks with the SLD menus and buttons or from the Shell window command line. Or, you can put Shell commands in a script file, then run the script with an Include command in the Shell window. For some setup, you may need to edit your powerpak.ini file (which the PowerPack installation procedure puts in your Windows directory) with a text editor.

Selecting Intel386 CX/SX and A-Step or B-Step Operation

When you are emulating an Intel386 CX or SX processor, a CPU Configuration dialog box appears the first time you start SLD. (If you first see a message box asking you to remove a jumper, ensure there is no jumper on the emulator processor's SEL3V and SELWV pins.) To configure the emulator for CX vs SX and A-step vs B-step operation, SLD uses information from powerpak.ini instead of the physical jumper used by earlier versions of the emulator.

The following figure shows the CPU Configuration dialog box.



In the Target CPU field, select the processor in your target design. In the Emulator CPU field, select the stepping of the bondout processor in the emulator probe head. To discover the processor stepping, look on the processor chip for:

Stepping	Distinguishing Mark
A-step	The number Q8543 appears on the processor.
B-step	The lot number (which starts with L) ends with B.

Leveraging Previous Emulation Sessions

After setting-up, you can shorten your setup time in subsequent emulation sessions by saving map, chip select, event, and log files.

You can save the map information to a file. In the Shell window enter a MapSave command, specifying a path and filename; or, in the Map dialog box, choose the Save button and fill-in the pathname dialog box. Later, you can restore the saved map with a Shell window MapRestore command or the Map dialog box Restore button.

You can save chip select information. In the Shell window enter the SaveCS command, specifying a path and filename; or, in the Toolbar Configure menu, choose Save Chip Selects and fill-in the pathname dialog box. Later, you can restore the saved registers with the Shell window RestoreCS command or the Toolbar Configure menu Restore Chip Selects item. See the "Shell Window Reference" chapter for a list of the registers saved for each processor.

You can save event definitions. In the Shell window enter the EventSave command, specifying a path and filename; or, in the Event window open the File menu, choose Save Events As, and fill-in the pathname dialog box. Later, you can restore the saved events with the

Shell window EventRestore command or the Event window File menu Restore Events item.

Instead of retyping command sequences, you can save the sequence to be made into a script file that you can run with a single Include command or from the initialization script. During an early emulation session, even if you usually use the SLD menus, open a log file and record lengthy or frequently repeated tasks by entering the commands in the Shell window. Edit the log file with a text editor, creating a script file of commands to be run in future emulation sessions. By logging these commands during an emulation session, you can test and record error-free procedures.

Starting a Log File

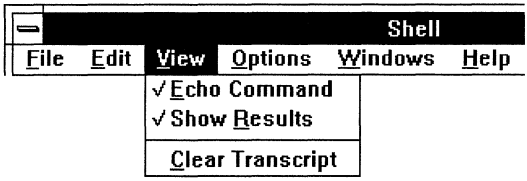
A log file records all that appears in the Transcript pane of the Shell window. The following sample sequence of commands sets up the Transcript pane and opens a log file to record any commands you enter in the Shell window and their results.

```
Echo On;                    // Commands you enter appear
                            // in the Transcript pane.

Results On;                 // Results of the commands appear
                            // in the Transcript pane.

DasmSym On;                 // Disassembly in the Transcript
                            // pane uses symbol names.

Overwrite;  // Specifying an existing filename for the log file will
            // overwrite the file's prior contents. The alternative
            // command is Append, which would add the new
            // log to the end of any existing file contents.

Log "emu1.log";             // The log filename is emu1.log.

Logging On;             // Start writing to emu1.log. The emulator
                        // puts the date and time in the log file
                        // when you start and stop logging.

Version;                // Display and log version information for
                        // the emulator, DOS, and Windows.

//...                   // Your emulation session activities....

Logging Off;            // Stop writing to emu1.log. A subsequent
                        // Logging On command will overwrite emu1.log.
```

You can do some of the above commands in the Shell window menus:

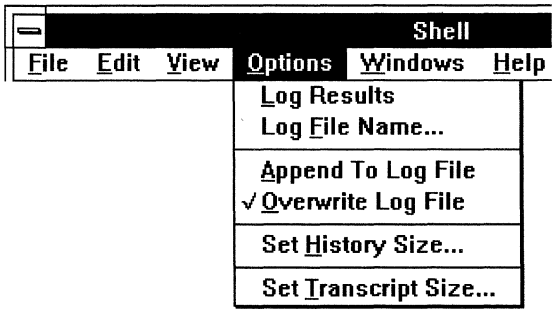
- To toggle command-echoing in the Transcript pane, open the View menu and check or uncheck Echo Command.
- To toggle the results display in the Transcript pane, open the View menu and check or uncheck Show Results.

The following figure shows a View menu with Echo Command and Show Results enabled.



- To specify whether to overwrite or append new information to an existing log file, open the Options menu and check Overwrite Log File or Append To Log File, respectively.
- To specify the log filename, open the Options menu, choose Log File Name, and fill-in the dialog box.
- To start and stop logging, open the Options menu and check or uncheck Log Results.

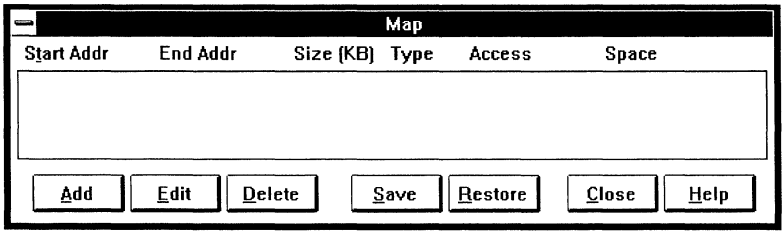
The following figure shows an Options menu with Overwrite Log File enabled. The next log file opened will be overwritten with the new log information, destroying its previous contents.



Mapping and Initializing Memory

Before loading your code or symbols, you must map memory. You can use a memory map saved from a previous emulation session or specify a new configuration.

Open the Map dialog box from the Toolbar either with the Map button or by opening the Configure menu and choosing Map. The following figure shows a Map dialog box with no memory mapped.



The Map dialog box lists any already configured sections of memory. Use the buttons along the bottom of the Map dialog box to:

Add	Configure a new section of memory.
Edit	Reconfigure the selected section. Use the mouse or arrow keys to select from the list in the dialog box.
Delete	Revert the selected section to unconfigured memory.
Save	Save to a map file the memory configuration listed in the dialog box.
Restore	Configure memory from a previously saved map file.

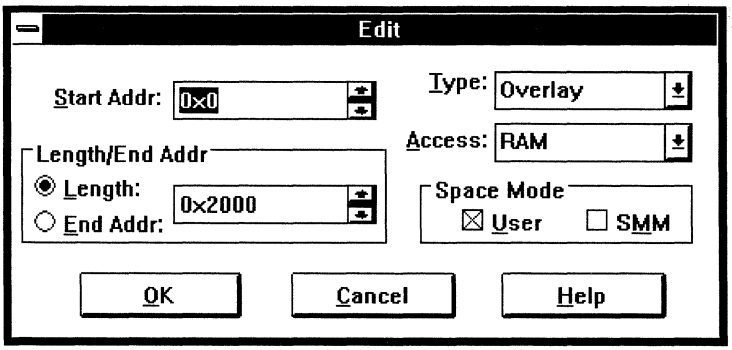
The Add and Edit buttons pop-up a dialog box. For each section configured, you can specify:

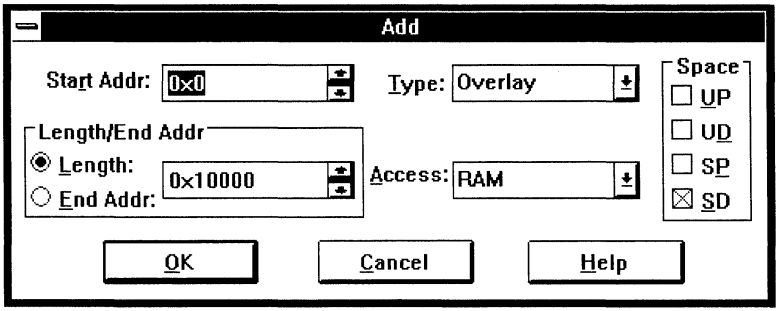
- A hexadecimal starting address, on:
  - a 4K-byte boundary for Intel processor emulators
  - a 64K or 128K -byte boundary for Motorola processor emulators with 256K bytes of overlay memory
  - a 64K, 128K, 256K, or 512K -byte boundary for Motorola processor emulators with 1M byte of overlay memory
- The size, either as a hexadecimal number of bytes (with the Length button selected) or by a hexadecimal ending address (with the End Addr button selected). For Motorola emulators the size and starting address must correspond; for example, a 128K-byte region must start on a 128K-byte boundary.
- Overlay or Target memory, as listed in the Type column of the Map dialog box.
- For Intel processors, User or SMM (system management mode) space, as shown in the Edit dialog box below, as listed in the Space column of the Map dialog box.

- For Motorola processors, UP (user program), UD (user data), SP (supervisor program), or SD (supervisor data) space, as shown in the Add dialog box below and as listed in the Space column of the Map dialog box.
- How the emulator treats memory accesses (as listed in the Access column of the Map dialog box):

RAM	allows reads and writes without breaking.
ROM break	allows reads; disallows writes; an attempted write causes a break. For Intel emulators with memory mapped to Target, writes are allowed but break emulation.
ROM nobreak	allows reads; disallows writes; does not break on any access. For Intel emulators with memory mapped to Target, writes are allowed and do not break emulation.
NONE	disallows reads and writes; breaks on any access. For Intel emulators with memory mapped to Target, accesses are allowed but break emulation.

The following figure shows an Intel map Edit dialog box followed by a Motorola map Add dialog box. For Motorola, the emulator automatically apportions the mapped regions between the two mappable segments.





You can also use the Shell window to map memory. The following sample sequence of commands prepares the emulator and memory for loading code or symbols:

```
Map Clear;                    // Maps all memory to target, removing
                              // any existing map configuration.

RestoreMap "emu1.map";        // Maps memory according to a map
                              // saved from a previous emulation session. In this case,
                              // the emu1.map file contains the line: map 0x0 0xffff ram.

Map 0x10000 RomBrk;           // Emu1.map maps only part of memory,
                              // not including the 4K-byte block starting at address 0x10000.
                              // This Map command configures memory from 0x10000 to
                              // 0x1ffff as ROM and specifies that any attempt to access
                              // this space will break emulation.
```

Loading a Loadfile

Once memory is configured, you can load the file to be debugged.

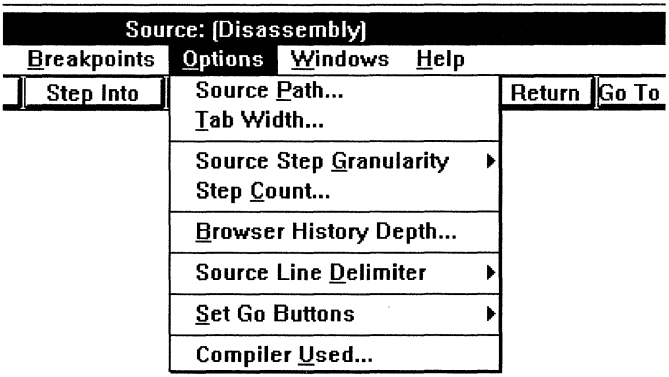
SLD supports the following loadfile formats:

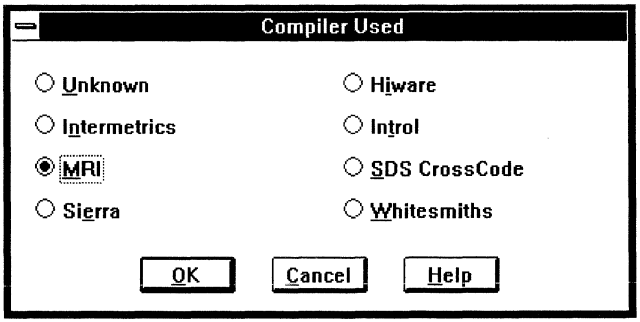
- OMF86 (Intel)
- OMF386 (Intel)
- IEEE-695 (Motorola)
- S-record (Motorola)

For Intel loadfiles generated with the Borland C compiler, before loading enter MaxBitFieldSize 16 on the Shell command line.

For Motorola loadfiles, the first time you load a file you must specify the compiler you used. On the Shell command line, enter a CompilerUsed command. Or, in the Source window, open the Options menu, choose Compiler Used, and choose one of the compilers listed in the dialog box. The following figure shows a Source window Options

menu (Compiler Used is at the end of the menu) and a Motorola emulator Compiler Used dialog box.





For Motorola loadfiles with more than 32 sections, you can shorten the load time by entering a MergeSections On Shell command before starting the load.

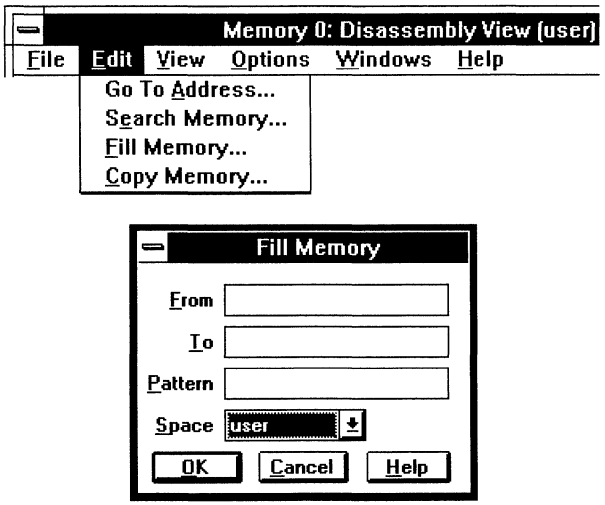
You can load a file during emulation. Be sure the file's load addresses do not overlap the memory occupied by the running program. Loading a file at a location in use stops the emulator in an unpredictable state.

The following sample sequence of commands initializes memory with 0x55aa values, then loads code and symbols:

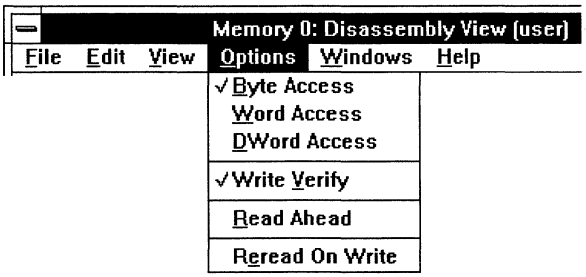
```
Fill 0x0 0xffff 0x55aa Word;  // Fills the first 64K bytes of memory
                              // with repeating 55aa values.
Loadsize Long;                // The loadfile will be written to memory in
                              // double-word accesses, which is the
                              // fastest way to load code.
Verify On;                    // Each write will be followed by a
                              // read to verify the value written.
```

```
Load "myfile.obx" code symbols nodemand nowarn status;
                    // Load code and symbols from the myfile.obx loadfile.
```

You can do the above operations using various SLD window menus. To initialize memory in the Memory window, open the Edit menu, choose Fill Memory, and fill-in the dialog box. The following figure shows an Edit menu and an Intel emulator Fill Memory dialog box.



To verify values written to memory, in the Memory window open the Options menu and check Write Verify.

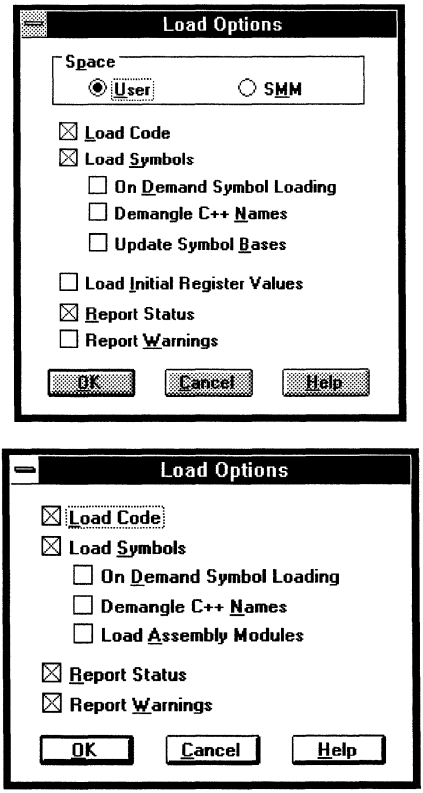


To load code and symbols, open the Load dialog box from the Toolbar with the Load button or from the Source window by opening the File menu and choosing Load File. If you are reloading one of the last four files that were previously loaded, you can open the Source window File menu and choose the loadfile pathname from the bottom of the menu.

In the Load dialog box, the name of the previous file that was loaded is automatically filled-in. Or, you can browse the directory and file lists to specify a different loadfile.

Before choosing the OK button to load the file, you can choose the Options button in the Load dialog box to open the Load Options dialog box. If you have already loaded a file, the options you specified previously are preserved.

The following figure shows two sample Load Options dialog boxes. The first is for the Intel386 EX processor; the second is for the Motorola 68332 processor. Different options are available for different processors.



For Intel loadfiles, be sure the space option (User or SMM) you select is compatible with the address space you configured in the Map dialog box.

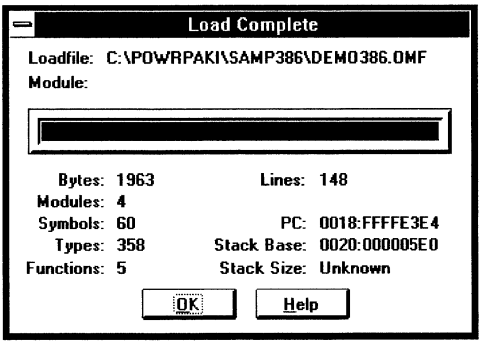
You can load code, symbols, or both from any loadfile. For example, load only code if symbols are already loaded; load only symbols for debugging ROM code. To load code, check the Load Code box. To load symbols, check the Load Symbols box and any combination of boxes under Load Symbols:

- On-demand symbol loading defers loading local symbol and line-number information for each module until it is needed; i.e. until either the module is displayed in the Source window or a breakpoint is set in the module. Advantages of on-demand symbol loading include faster initial loading, faster lookup for the symbols that are demanded, and less memory occupied by the loaded file because only the fewest required symbols are loaded.
- For overloaded C++ functions, the emulator can demangle the symbolic name of the first mangled version it encounters of each function.
- For OMF386 loadfiles, the symbol server base addresses can be updated after loading, in conjunction with initializing the Intel386 registers.

For OMF386 loadfiles, you can load the processor registers with initial values.

For Motorola loadfiles, you can load symbolic information for modules whose source files are assembly language.

You can request or suppress information about the load process and results. For a dynamic report of the loading process, check Report Status. A bar graph fills to indicate the percent loading complete; loading statistics are updated continuously during the load process.



Suppress warning messages during loading by un-checking Report Warnings.

If you are loading a Motorola loadfile with the Load dialog box, and you have not already specified a compiler, SLD displays a Compiler Used dialog box; choose one of the listed compilers.

For C++ code containing virtual functions, overloaded functions, and some other symbol types, the emulator can demangle the first instance of each such symbol. Subsequent instances remain mangled in the

emulator symbol table rather than duplicated, so you can access all symbols in your program. However, the names do not appear mangled in your source. The warning message C++ duplicate name detected alerts you to the presence of mangled names.

The emulator handles mangled names based on the Microtec Research Inc. (MRI) C++ version 1.1 name mangling algorithm. For other C++ compiler output, specify mangle with the Load command or uncheck Demangle C++ Names in the Load Options dialog box. This retains all mangled symbols.

Enabling Memory Access

You can access memory during emulation, to read or write the current values in target memory and on-chip peripheral registers (but not CPU registers). Such reads and writes take a small, additional amount of processor time.

When you invoke SLD, such memory access is disabled by default. To enable memory access, either:

- On the Shell command line, enter RunAccess On.
- Open the Toolbar Configure menu and toggle Run Access.

Run Access does not allow CPU register access. The CPU registers cannot be accessed during emulation; their display is updated only when emulation halts.

Enabling Intel386 EX Expanded Memory

You can read and write any peripheral register by editing the field values in the Peripheral window or by entering Dump, Fill, and Write commands on the Shell window command line.

To access some of the peripheral registers with the Shell commands, you must first enable expanded I/O space. Once expanded I/O space is enabled, you can use both the Peripheral window and the Shell command line to access timers, DMA, interrupt controllers, serial communications channels, and other internal peripheral registers such as chip selects, power management, and watchdog timer.

When expanded I/O space is disabled, the affected registers appear in the Peripheral window with question marks (?) in their address fields. A question mark indicates you can access the register via the Peripheral window but not from the Shell command line.

To enable expanded I/O space, **close (not minimize) the Peripheral window**, then set the ESE bit in the REMAPCFG register by three sequential writes to I/O addresses 0x22 and 0x23. (The sequence must write twice to each address.) For example, enter the following Size and Fill commands on the Shell command line:

```
Size Byte;
Fill 23p 23p 0x00 Byte IO;
Fill 22p 22p 0x80 Byte IO;
Size Word;
Fill 22p 23p 0x0080 Word IO;
```

The Size command specifies the physical size of the data access. The Byte and Word specifiers in the Fill commands inform SLD of the supplied data format.

Managing Intel386 EX Signals

RESET	Active high synchronized to CLK2. This signal can be pulled high or low during use as long as it remains stable during initialization. The signal can be disabled in the CPU window to be driven by the emulator.
RDY#	Must be synchronized to CLK2 with the proper setup time according to Intel specifications for any cycles for which the 386EX is not programmed. After the chip select unit is programmed, such signals would include any unmapped memory or I/O space, any disabled on-chip expanded I/O space, and halt or shutdown cycles. (The power-up condition for chip select and ready generation allows upper-chip-select memory accesses to the entire 64M byte address range.) RDY# should be tri-stated when the 386EX CPU is providing the ready due to LBA# cycles. RDY# should have a resistor pull-up to VCC or be pulled low with resistor of 600-820 ohm for full time zero wait states.
NA#	should be synchronized to CLK2 and driven according to the need for pipelining. Do not float the signal. NA# can be disabled in the CPU window.
BS8#	should be synchronized to CLK2 and driven according to the actual bus size. Do not float the signal.

NMI	should be driven as needed. When NMI is floated it must be disabled in the CPU window.
SMI#	should be driven as needed. When SMI# is floated it must be disabled in the CPU window.
FLT#	must have a resistor pull-up to VCC or be floated when the emulator is attached.
HLDA	if HLDA is configured as an output port, enter Config IgnoreHLDA On in the Shell window command line, to inform the SLD software that the CPU has not granted the bus to another master.

Turning Off a Motorola Watchdog Timer

In Motorola processors, the software watchdog timer is controlled by the software watchdog enable (SWE) bit in the SYPCR register. When enabled, the watchdog timer requires that a service sequence be periodically written to the software service register (SWSR). If these writes do not occur, the watchdog timer times out and asserts RESET. This protects the system against, for example, infinitely looping code.

For the 68331/332/F333 processors, turn off the watchdog timer:

```
Write fffa21 0;
```

For the HC16 processors, turn off the watchdog timer:

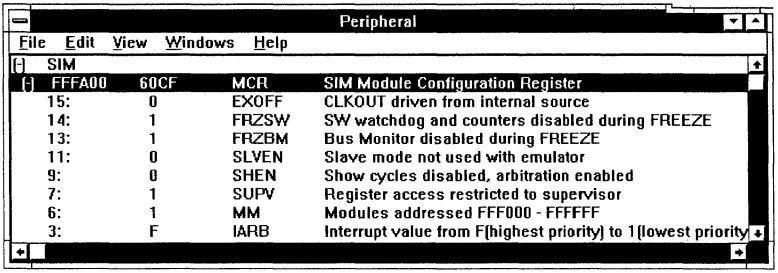
```
Write ffa21 0;
```

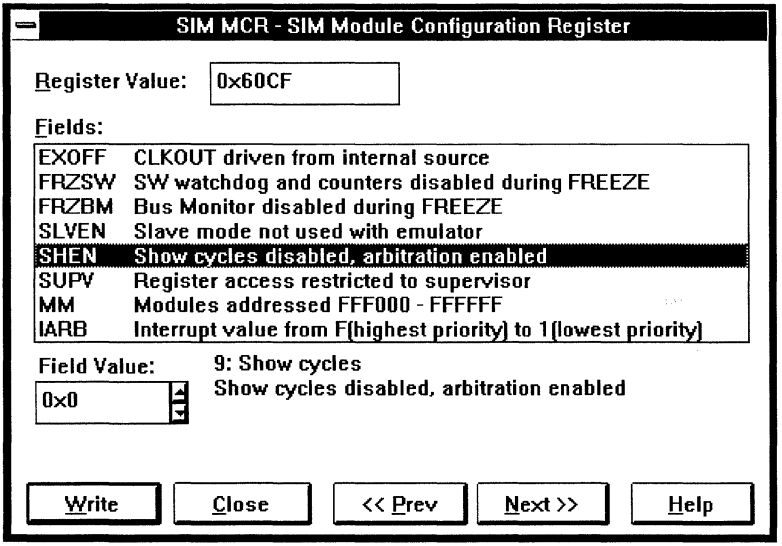
Enabling Motorola Show Cycles

In Motorola processors, you can enable or disable the show-cycle mode of the processor. The 683xx and HC16 have internal peripherals. Normally, when the CPU accesses these peripherals, the bus cycle is invisible outside the chip. With Show Cycles enabled, the internal cycles are visible in the trace buffer and can be used for triggering.

- On the Toolbar, open the Configure menu and toggle Show Cycles.

  Enabling Show Cycles sets the SHEN[0:1] bits in the SIM module control register to 1,1. Disabling Show Cycles sets these bits to 0,0 (the default). To see this in the Peripheral window (as shown in the following figure), expand the SIM peripheral and open the MCR register. Select the SHEN field. You can use the Field Value spin box to enable or disable Show Cycles. To write new values, choose Write; to close the dialog box without changing the MCR, chose Close.





Programming Motorola Chip Selects

The Motorola processors provide several independently programmable signals that you can configure as chip selects, output pins, or function codes. The number of signals and their possible configurations are different for different processors. The 68331, 68332, 68F333, and 68HC16Z1 provide 12 independently programmable chip select signals with programmable block sizes from 2K to 1M bytes. Of these 12 signals, 11 are shared with other processor signals. The 68330 and 68340 provide four independently programmable chip select signals with programmable block sizes from 256 to 4G bytes. For the 330, one is shared with another processor signals; for the 340, all four are shared.

You can configure these signals in various ways:

- Design your target hardware to configure the signals at reset.
- Design your target startup code to configure the signals. This code must be in the CS0 (for 330/340) or CSBOOT (for other processors) area of memory. Execute the initialization code.
- In the Peripheral window, use the Register Edit dialog boxes to write to the peripheral registers.
- In the Shell window, enter Write commands to the peripheral register addresses.
- Create a chip select configuration file, either from the Shell window with a SaveCS command; from the Toolbar by opening the Configure menu, choosing Save Chip Selects, and filling-in the dialog box; or with a text editor such as Windows Notepad. To program the chip selects from the file, use a Shell RestoreCS or ConfigCS command; or open the Toolbar Configure menu, choose Restore Chip Selects, and fill-in the dialog box.

For the emulator to correctly process memory mapping, execution breakpoints, triggers, and trace, the emulator's programmable hardware must be configured to match the processor's chip select configuration. Once you have configured the processor signals, either enter ConfigCS on the Shell command line or open the Toolbar Configure menu and choose Configure Chip Selects. With a chip select configuration file, you can configure the processor and emulator hardware from the Shell window with a single ConfigCS command. Entering:

```
RestoreCS config1.cs;
ConfigCS;
```

is the same as entering:

```
ConfigCS config1.cs;
```

Different signals are available in the Event and Trace windows depending on how the shared signals are configured. The following example demonstrates how various configurations of the 68332 chip selects are reflected in the Event window.

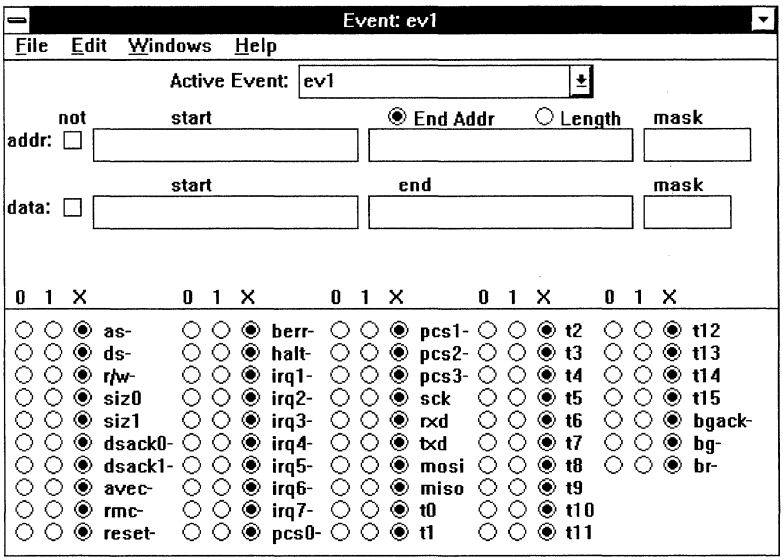
If the Event window is open when you reconfigure the registers, you must close (not minimize) and re-open it to see the changes.

In the SIM (system integration module) peripheral, CSPAR0 (Chip Select Pin Assignment Register 0) controls the use of the chip selects 0 through 5.

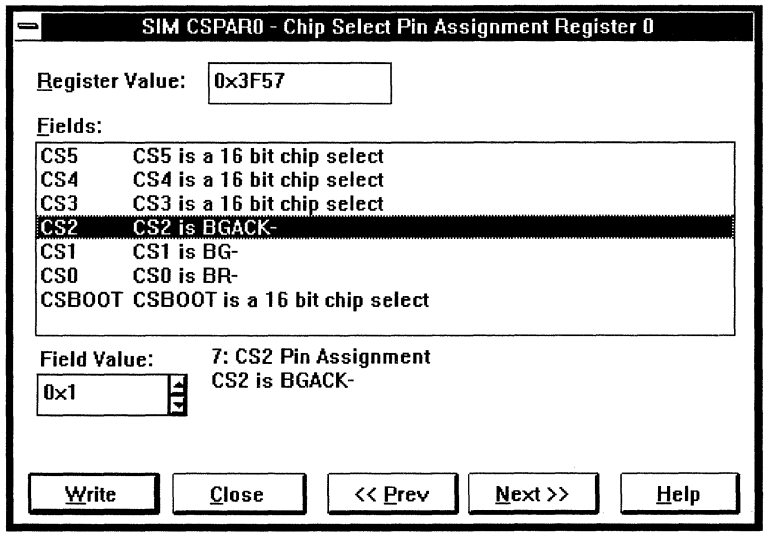
The following chart shows how the value of the bit fields CSPAR0:CS[0:5] specifies the use of these chip selects:

	3	2	1	0
CS0	8-bit chip select	16-bit chip select	BR# (Bus Request)	BR#
CS1	8-bit chip select	16-bit chip select	BG# (Bus Grant)	BG#
CS2	8-bit chip select	16-bit chip select	BGACK# (Bus Grant Acknowledge)	BGACK#
CS3	8-bit chip select	16-bit chip select	Function Code 0	Port C0
CS4	8-bit chip select	16-bit chip select	Function Code 1	Port C1
CS5	8-bit chip select	16-bit chip select	Function Code 2	Port C2

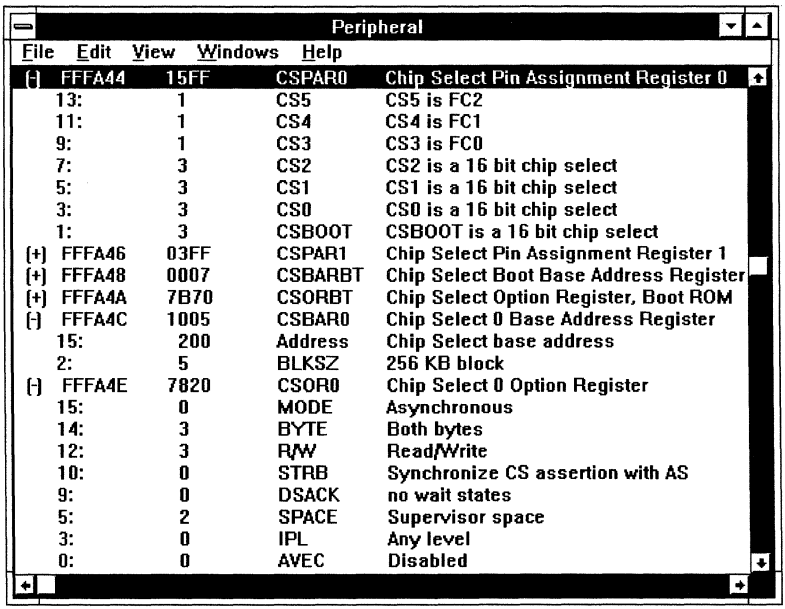
The following figure shows the BR#, BG#, and BGACK# signals in the Event window, with CS[0:2] configured for bus management and CS[3:5] configured as chip selects. The signals that appear in the Event window also appear in the Trace display.



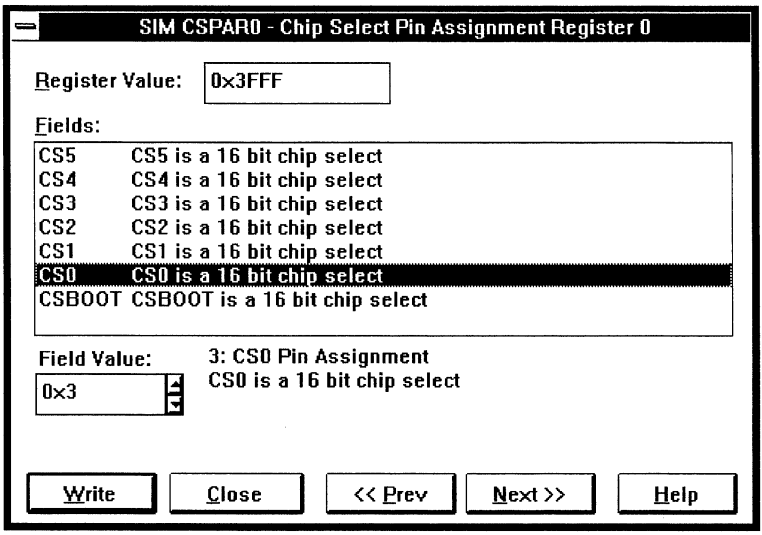
The following figure shows the CSPAR0 Register Edit dialog box for the above Event window, with CS[0:2] each set to 0x1 (BR#, BG#, and BGACK#, respectively) and CS[3:5] each 0x3 (16-bit chip selects).



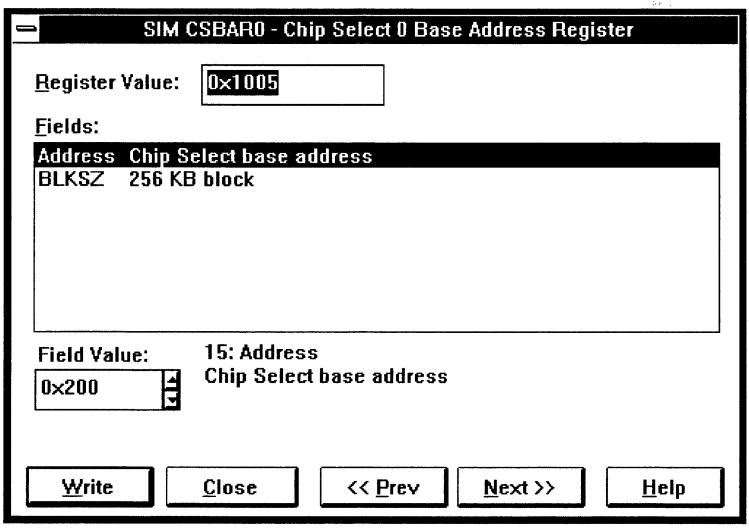
The following example configures a 68332 chip select and its memory block. The following figure shows the registers for this example (CSPAR0, CSBAR0, and CSOR0) expanded in the Peripheral window.



For this example, CSPAR0:CS0 is set to 0x3. The following figure shows the CSPAR0 Register Edit dialog box.

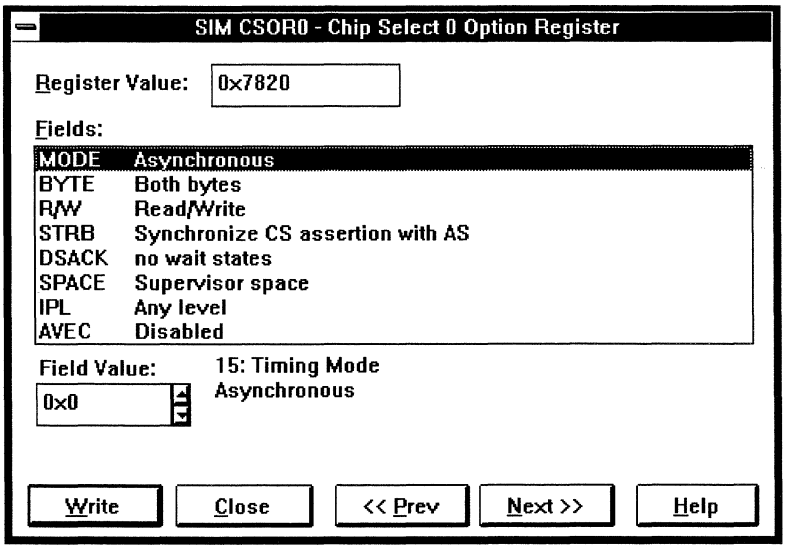


A pair of internal registers controls the memory block for each chip select. The Chip Select Base Address Register specifies the starting address and size; the Chip Select Option Register configures the access. For this example, CS0 controls a 256K byte memory block starting at 0x200. The following figure shows the CSBAR0 Register Edit dialog box with Address = 0x200 and BLKSZ = 0x5 (the Field Value for a 256K byte block).



The following figure shows the CSOR0 Register Edit dialog box for this example, with:

MODE = 0	Memory access (relative to ECLK) is asynchronous.
BYTE = 3	Both bytes of a word are accessed.
R/W = 3	Both read and write are possible.
STRB = 0	The chip select synchronizes with the address strobe.
DSACK = 0	There are no wait states.
SPACE = 2	This block is supervisor space.
IPL = 0	Any interrupt has priority.
AVEC = 0	Auto vectoring is disabled.



Using a Script

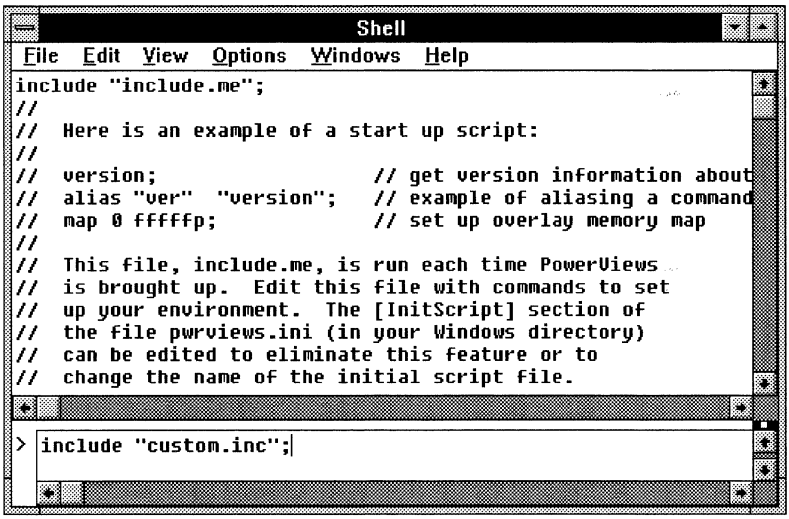
A script is a text file of Shell commands. At any time during an emulator session, you can use the Include Shell command (or, in the Shell window, open the File menu, choose Include File, and fill-in the dialog box) to execute a script.

In the powerpak.ini file [InitScript] section, you can specify a script to be executed automatically as an initialization script when you start SLD. A sample initialization script, include.me, is installed with SLD.

To create your own script for SLD initialization:

1. Use a text editor, such as Windows Notepad, to create a file of Shell commands. End each command with a semicolon.
2. Edit the line script = <pathname> in powerpak.ini:
   - <pathname> is the pathname of the script. For example: script = c:\sld\user\myscript
   - The only filename restrictions are any imposed by your DOS or Windows.
   - If you specify no pathname (for example, script = myscript), be sure your script is in the directory with the SLD files.

The following figure shows the Shell window after include.me has executed. An Include command to execute custom.inc is ready to be entered on the Shell window command line.



Keyboard Shortcuts

You can use these function keys as shortcuts instead of window commands.

Press this Key	To Do This
F1	Open a window for SLD on-line help.
F2	Halt emulation.
F3	Start trace.
F4	Stop trace.
F5	Set focus to the Toolbar window.
F6	Set focus to the next open SLD window.
F7	Step Into.
F8	Step Over.
F9	Start emulation (Go).
F10	Activate the menu bar for keyboard use.

Debugging in Source and Stack

This chapter describes how to:

- *Set, view, and clear breakpoints.*
- *Control program execution.*
- *Examine and modify variables and the stack.*

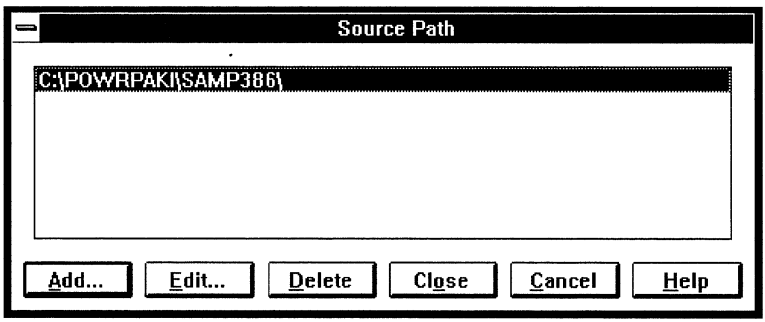
Viewing Source

After loading an executable file, you can view modules in the Source window. The Source window initially displays code starting at the current program counter (CS:EIP for Intel; PC for Motorola). The instruction pointed to by the program counter is marked by >>.

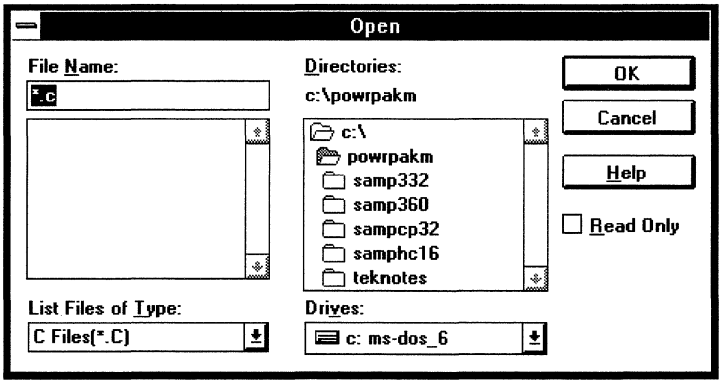
When you open the Source window after loading but before executing code, the program counter may be in the assembly startup code. In general, embedded programs start in startup code and not in main(). You or the compiler can insert initialization code to set up the processor environment. The Source window displays either the assembly source or the disassembly from memory.

To view a different module, open the File menu and choose Browse Modules. All loaded modules are listed. If a module's source has been modified more recently than the loadfile, a warning message appears and an asterisk marks the source filename in the Source window title.

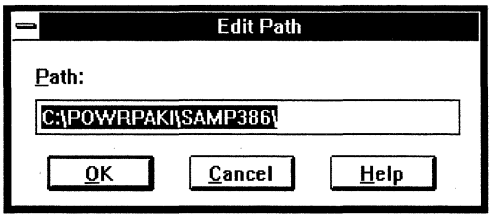
If the emulator cannot find the source file corresponding to the module you are browsing, you may need to modify the source search path list. In the Source window, open the Options menu, choose Source Path, and modify the list. The following figure shows a Source Path dialog box.



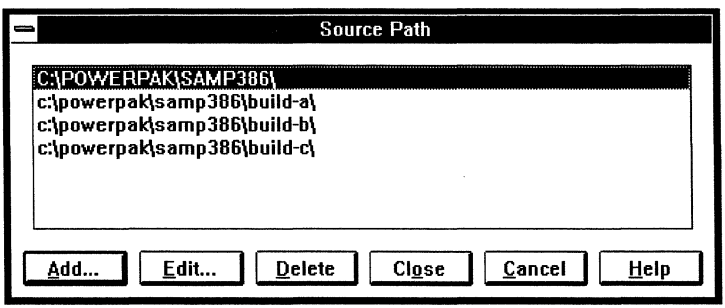
To add a path, choose the Add button and choose a source file in the dialog box. The following figure shows the Open dialog box that appears in response to the Source Path dialog box Add button.



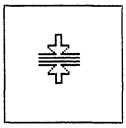
To edit a path, select a path in the Source Path dialog box; choose the Edit button; and edit the path string. To select a path from the list, move the highlight with the mouse or the <Up Arrow> and <Down Arrow> keys. The following figure shows the Edit Path dialog box.



The emulator searches the paths in the order they are listed in the Source Path dialog box, stopping at the first file that matches the source filename in the loadfile. If you have duplicate filenames in different directories, order the source path search list so the emulator finds the correct one first. For example, in the following figure, the emulator searches first samp386, then build-a, build-b, and finally build-c.



When full symbolic information (including the source file pathname) is available for a module, you can view the module as source code with or without interleaved disassembly. Use the View menu to toggle between Source Only and Mixed Source And Assembly. (Modules with no source information appear as disassembly only, regardless of the view.) To see symbols in the disassembly, on the Toolbar open the Configure menu and check Symbolic Disassembly.



You can split the Source window into two panes by clicking and dragging on the split box at the top of the vertical scroll bar. A split-box cursor appears at the right of the split bar (see figure at left). To resize the panes, point the mouse to the split box and drag the split box.

With two Source window panes, you can work in two different modules or two areas of the same module independently. To move between panes, click in the inactive pane to make it active.

Managing Breakpoints

At a breakpoint, emulation halts before executing the instruction at the breakpoint address. A temporary breakpoint is then cleared; a permanent breakpoint remains.

You can set 256 software breakpoints; in addition, for Motorola processors you can set two hardware breakpoints and for Intel processors you can set up to four hardware breakpoints. The choice of hardware or software breakpoint is automatic.

For Intel processors, you can configure the debug registers DR[0:3] to specify a hardware data or execution breakpoint. See the DR command description in the "Shell Window Reference" chapter.

If you try to set a breakpoint on a non-executable source statement, a breakpoint is set on the first subsequent executable source statement.

You can set breakpoints from:

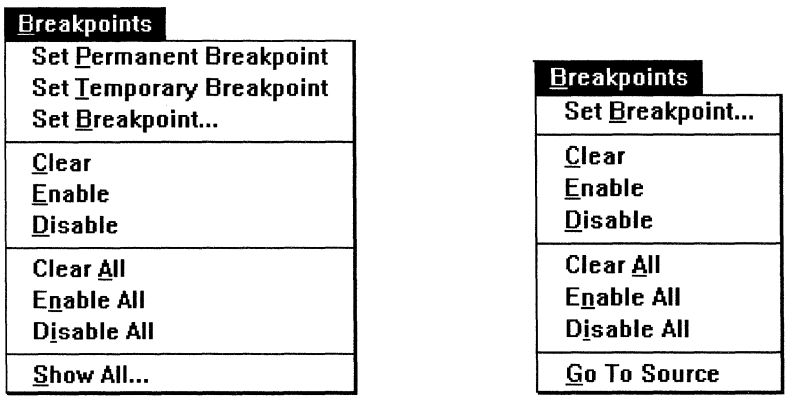
- the Source window, using the mouse in the source display or using the Breakpoints menu
- the Breakpoint window Breakpoints menu
- the Breakpoint window Set button
- the Bkpt command in the Shell window

In the Source window, using the mouse:

1. Move the mouse pointer to the left of the source line where you want to set a breakpoint.
2. When the mouse pointer changes shape to a cross-hair cursor (see figure at left), click on the primary mouse button to set a permanent breakpoint or on the secondary button to set a temporary breakpoint. (On a mouse configured for right-handed use, the primary is the left button and the secondary is the right button.) The line with the breakpoint is highlighted in red.

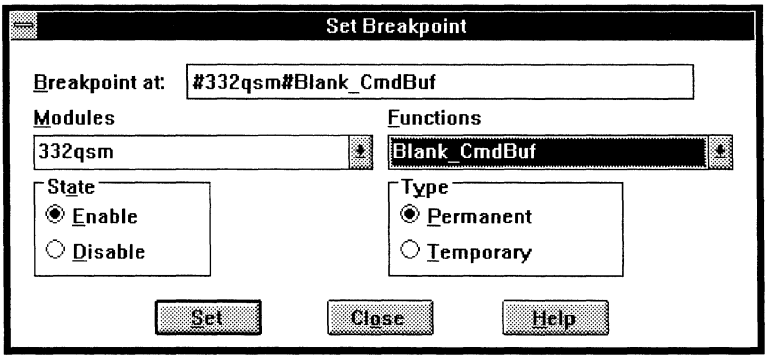
In the Source or Breakpoint window, open the Breakpoints menu. In the Source window, to set a breakpoint on the line where the Source cursor is positioned, select Set Permanent Breakpoint or Set Temporary Breakpoint. To set a breakpoint elsewhere, choose Set Breakpoint and fill-in the Set Breakpoint dialog box.

The following figure shows a Source window Breakpoints menu and a Breakpoint window Breakpoints menu. In the Source window, the Show All item opens the Breakpoint window listing all current breakpoints; in the Breakpoint window, the Go To Source item opens the Source window showing the line where the selected breakpoint is set.



In the Breakpoint window, you can also choose the Set button to pop-up the Set Breakpoint dialog box.

In the Set Breakpoint dialog box, you can enter a numeric or symbolic address in the Breakpoint At field. For a symbolic address, you can browse the Modules and Functions drop-down lists. The following figure shows a sample Set Breakpoint dialog box.



For C++ source, mangled names (which do not appear in the Source window display) are listed in the Set Breakpoint dialog box and can be listed with a DisplaySymbols Shell command. These include member functions from all classes defined in a source module and its header files, compiler-provided default constructors and destructors, and global (non-class related) functions. For information on the C++ mangling algorithm, see *The Annotated C++ Reference Manual* by Margaret Ellis and Bjarne Stroustrup.

Avoid setting breakpoints on inline functions. The Set Breakpoint dialog box does not flag inline functions. If you have set a breakpoint on a function and stepping does not advance the Source window cursor, it is an inline function. Stepping through instructions contained in your class definition will advance the program counter but not the Source window cursor. Remove the breakpoint on the function and restart emulation.

With the Source window view set to Mixed Source And Assembly, the assembly instructions for all inline functions appear after the last source line of the module.

Some toolchains allow more than one source statement per line. You can set a breakpoint on any statement in a line. For example:

Set a breakpoint: multiple statements per line

```
If (errorNumber) errorHandler(errorNumber);
```

To set a breakpoint on the errorHandler call, when errorNumber is nonzero:

1. From the Source window Options menu, set the level of step granularity by toggling Step Execution Granularity to Statement.
2. Click on errorHandler(errorNumber), open the Breakpoint menu, and choose Set Permanent Breakpoint. Or, double-click on errorHandler(errorNumber) and choose Permanent Breakpoint.
3. The entire line is highlighted as a breakpoint, with the actual

breakpoint set on the second statement. From the View menu, choose Mixed Source And Assembly to see the breakpoint on the second statement.

Tab width: effect on setting breakpoints at statement level

To set a breakpoint at the statement level, you must know how many spaces your compiler uses for a tab character. For example, when the following line containing three statements is compiled with MRI:

<tab><tab>for (j = 0; j < max_num; j++) {

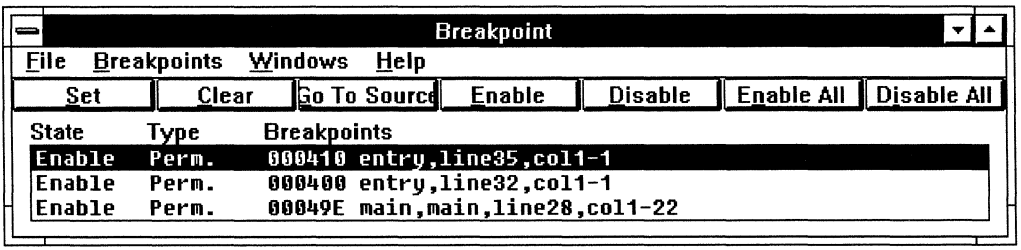
the MRI default tab width of eight characters produces the following column ranges for the three statements:

j = 0;	columns 0 through 26
j < max_num;	columns 27 through 39
j++	columns 40 through 45

Setting the Source window tab width to four instead of eight would put the first j (in j = 0;) at column 13 and the second j (in j < max_num;) at column 20. It is then difficult to set a breakpoint on the correct statement.

Symbols must be loaded before you can set breakpoints on line numbers or functions. If you chose On Demand Symbol Loading when loading your program, the symbols needed for a breakpoint are loaded either when you set the breakpoint or when you display the source for the module containing them.

To list breakpoints in a separate Breakpoint window, in the Source window open the Breakpoints menu and choose Show All; or in any SLD window open the Windows menu and choose Breakpoint. (In the CPU window, where there is no Windows menu, use the Options menu.) The Breakpoint window shows the state (enabled or disabled), type (permanent or temporary), and location in source of each currently defined breakpoint. The following figure shows a sample Breakpoint window.



The Breakpoint window button operations are duplicated in the Breakpoints menus of the Source and Breakpoint windows. In the

Breakpoint window, click on a breakpoint or use the arrow keys to select it. In the Source window, select a breakpoint by moving the Source cursor to the statement where the breakpoint is set.

List breakpoints in Shell windowl

To list breakpoints in the Shell window, enter Bkpt. For example:

```
bkpt;
// SRC bkpt: Ena Perm 470 (@0)
D:\TBIRD\M332\SAMPLES\SAMP332\main.c,main,Line21
// SRC bkpt: Ena Perm 486 (@1)
D:\TBIRD\M332\SAMPLES\SAMP332\main.c,main,Line24
// SRC bkpt: Ena Perm 492 (@2)
D:\TBIRD\M332\SAMPLES\SAMP332\main.c,main,Line26
// SRC bkpt: Ena Perm 49E (@3)
D:\TBIRD\M332\SAMPLES\SAMP332\main.c,main,Line28
```

You can enable and disable all or individual breakpoints. An enabled breakpoint is defined and active; emulation breaks when the breakpoint is reached. A disabled breakpoint is defined but inactive; emulation does not break when the breakpoint is reached.

Disabled and enabled breakpoints

For example, an interrupt handler named MyIntr (in a module named ModB) might be started at any time. To discover whether MyIntr is starting during execution of another function named Atomic (in a module named ModA), the designer does the following:

1. Set a breakpoint, enabled, at the beginning of #ModA#Atomic.
2. Set a breakpoint, enabled, at the end of #ModA#Atomic.
3. Set a temporary breakpoint, disabled, at #ModB#MyIntr.
4. Go. The MyIntr interrupt handler can execute without causing a break.
5. When the emulator halts at the first Atomic breakpoint, enable the MyIntr breakpoint. If MyIntr is called during Atomic execution, a break occurs and the MyIntr breakpoint is cleared. Otherwise, when the emulator halts at the second Atomic breakpoint, re-disable the MyIntr breakpoint.

You can change the Source window display to view the line containing any listed breakpoint. Select the breakpoint and choose Go To Source.

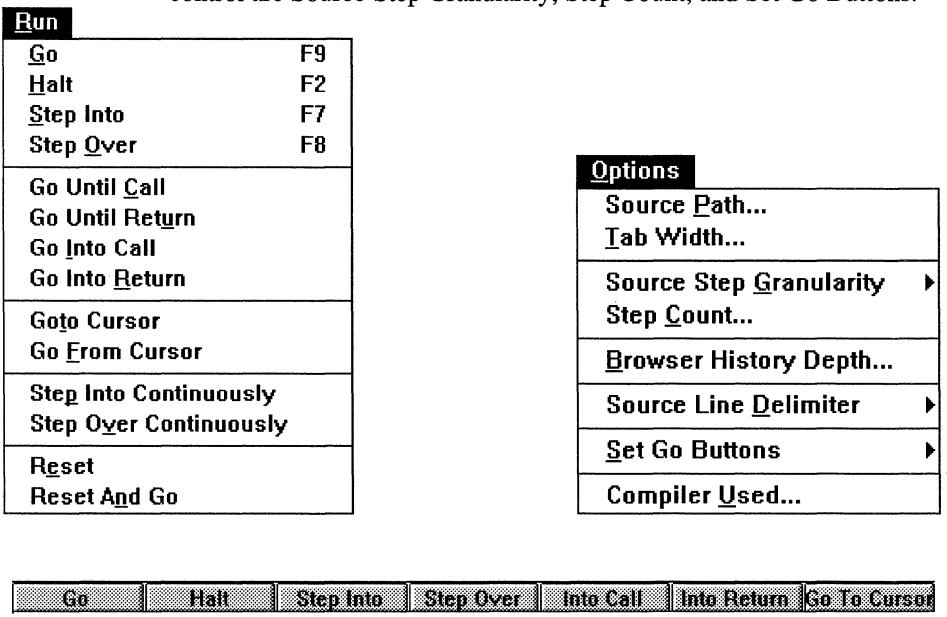
You can remove all or individual breakpoints by any of:

- In the Source or Breakpoint window, open the Breakpoints menu and select Clear All.
- In the Breakpoint window, select a breakpoint and choose Clear from either the buttons or the Breakpoints menu.

- In the Source window, click in the left margin of the red-highlighted line containing the breakpoint; or, move the cursor to the breakpoint, open the Breakpoints menu, and choose Clear.
- On the Shell command line, enter BkptClear.

Starting and Stopping Emulation

The following figure shows the Source window Run and Options menus and button bar. On the Options menu, the items involved in emulation control are Source Step Granularity, Step Count, and Set Go Buttons.



With the Source window buttons and menus, you can emulate one or more instructions at a time or as a free-running program:

Step — breaks after executing one to 100 instructions or statements, according to how you set Step Count and Source Step Granularity in the Options menu. The Shell Step and StepSrc commands can do the same.

Into — when encountering a function call instruction, executes the jump and breaks at the first instruction or statement inside the function.

	Over	when encountering a function call instruction,executes the function and breaks at the first instruction or statement after returning.
	Continuously	repeatedly Steps until you halt the emulation.
Go	executes your program to the next enabled breakpoint or until Halted. The Toolbar Go button and the Shell Go, GoInto, and GoUntil commands do the same.	
	From Cursor	moves the program counter to the instruction where the Source cursor is, then starts emulation.
	To Cursor	emulates until the program counter reaches the Source cursor.
	Into Call	breaks on the first instruction or statement inside the next called function.
	Into Return	breaks on the first instruction or statement after the next return.
	Until Call	breaks on the next call instruction.
	Until Return	breaks on the next return instruction.
	To change the Into Call and Into Return buttons to Until Call/Return buttons, open the Options menu; choose Set Go Buttons; and select Until Call/Return.	
Reset And Go	Resets your target system, then operates as Go. The Shell ResetAndGo command does the same.	
Halt	Stops emulation during a Step Continuously or a Go operation. The Toolbar Halt button and the Shell Halt command do the same.	

How fast a Step operation executes depends on the number of SLD windows open. Each window must be updated after each step. You can close any open SLD window (except the Toolbar) to improve performance. Speeding up stepping can be useful when you use long or frequent Step Continuously operations.

In C++, stepping into a declaration can call a constructor with initialization parameters, if any, and its base class constructors.

To mask interrupts during Step operations, enter a StepMask Shell command. For Motorola emulation, masking interrupts can have the following effects:

- With mask on, a single step restores the original contents of the SR (CPU32) or CCR (CPU16) register when complete. If the stepped instruction modifies this register, the modification can be lost. The following instructions can cause this problem:

CPU32	CPU16
ANDI <ea>,SR	ANDP <ea>
ORI <ea>,SR	ORP <ea>
EORI <ea>,SR	TPD
MOVE <ea>,SR	TDP
MOVE SR,<ea>	RTI
LPSTOP STOP RTE	LPSTOP

- Most instructions that access memory can generate exceptions or traps due to bus or address errors or as an expected result of the instruction. In such cases the following sequence occurs:

  1. The value of SR or CCR saved on the stack for the exception is incorrect.
  2. When the exception returns, the incorrect stack value is restored into SR or CCR.

  The following instructions can generate a trap:

CPU32			CPU16
TRAP	CHK	DIVUL	SWI
TRAPcc	DIVS	LINE A	EDIV
TRAPV	DIVSL	LINE G	EDIVS
BKPT	DIVU		

To discover whether emulating or halted, look in the Status window or icon or enter EmuStatus on the Shell command line. When emulation has halted, to discover the cause of the break, look in the Status window or enter Cause on the Shell command line.

Examining Source After Emulating

The Source window display shows the statement or instruction next to be executed:

- When emulation is halted by a breakpoint, the program counter stops at the instruction containing the breakpoint.
- When emulation is halted after a Step Into or Go Into Call, the program counter points to the first instruction in the function.
- When emulation is halted after a Step Over or Go Into Return, the program counter points to the first instruction after the return.
- When emulation is halted after a Go Until Call or Go Until Return, the program counter points to the call or return instruction.

In Source Only view, a function with no associated source is not displayed after a Step Into, although the program counter points to the first instruction in the function. To display such a function, toggle the view to Mixed Source And Assembly.

You can also view disassembled instructions in the Memory window, or by entering a Dasm command on the Shell command line.

To modify instructions, use the Memory or Shell window as described in the chapter on debugging in registers and memory. Such code patching is reflected in the disassembly shown in the Source window in Mixed Source and Assembly view. Note that the disassembly at the patched addresses no longer matches the source.

For C++, you can select the following symbols in the Source window:

- Function symbols
- Global variables (which can be edited in the Variable window)
- Global class objects (which can be edited in theVariable window as structs)
- Local variables and class objects

You cannot select class.memberFunction type objects.

The scope-resolution operator (::) is interpreted as a token separator, not recognized as part of a symbolic address.

Scrolling Trace With Source

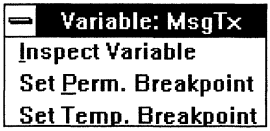
You can link the Source and Trace window displays. When the windows are linked, you can scroll through the Trace window and view the corresponding code scrolling synchronously in the Source window. To link the Source window to the Trace window:

1. In the Trace window, open the View menu and choose Instruction.
2. Re-open the View menu and choose Linked Cursor.

Examining and Editing Variables

You can examine and edit global, static, and local variables in the Variable window by either:

- In the Source window, double-click on the name of the variable you want to view. In the pop-up menu, choose Inspect Variable. The following figure shows a Variable pop-up menu.



- In any SLD window, open the Windows menu and choose Variable. (In the CPU window, where there is no Windows menu, use the Options menu.) In the Variable window, open the Variable menu, choose Add, and enter the name of the variable you want to view. Specify a fully qualified symbol name, as described in the section on symbolic addresses in the "Debugging with Triggers and Trace" chapter.

For local variables outside of the current stack context, the value unknown is displayed.

To select a variable or its value, click on it. Yellow indicates that you have selected the variable or its value. Unless currently selected (yellow), variable symbolic information appears in the following colors:

Red — indicates an editable value. Integer variables can be edited in hexadecimal or decimal, floating point variables in floating point format, and characters in their hexadecimal ASCII equivalent. To edit a value, either double-click on the value; or single-click on the value, open the Edit menu, and choose Edit. Press <Enter> to end editing.

Blue — indicates a pointer variable you can dereference by double clicking. For example, DS:000E is the address of the variable pointed to by cellPtr:

```
CELL_TYPE *printall#cellPtr = DS:000E
```

To dereference a pointer, either double click on the pointer name or open the View menu and choose Show. A new entry is added to the Variable window showing the variable that was pointed to. For example:

```
CELL_TYPE printall#*cellPtr{
    struct LINKS *next = DS:0014;
```

```
char *StringPtr = DS:0000;
short int length = 2 = 2;}
```

Magenta indicates a non-pointer variable. For enum type variables, the enumerated name follows the hexadecimal value. For example:

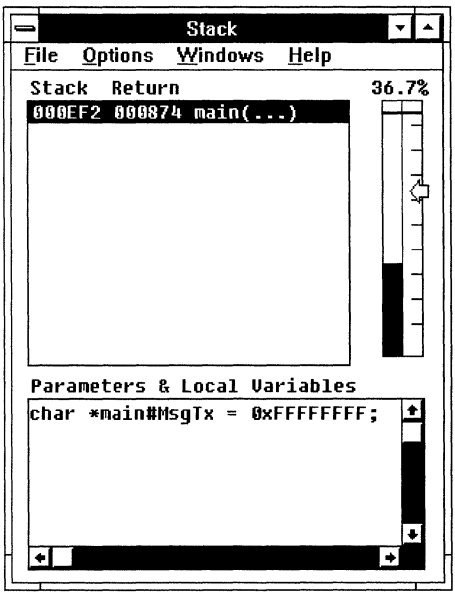
```
enum color c = 0x2 = lavender
```

To remove a variable from the display, in the Variable window click on the variable name; then either open the Variable menu and choose Delete or press the <Delete> key. (This does not delete the variable from your program, only from the current variable inspection list.) To retrieve the variable to the display, open the Variable menu and choose Undelete.

You can also examine program symbolic information using the Shell AddressOf, NameOf, ConfigSymbols, DisplaySymbols, GetBase, SetBase, and RemoveSymbols commands.

Viewing and Modifying the Stack

The Stack window contains a stack list pane, a variables list pane, and a stack meter. (You can also list the stack information in the Shell window using StackInfo and DisplayStack commands.) The following figure shows a sample Stack window.

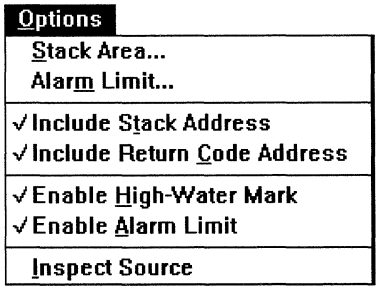


Configuring the Stack Window

Once a program has executed into one or more functions, the stack list contains frames representing the nested calls. Frame information can include the stack and return addresses of the functions, the function names, and the parameters and local variables associated with the function calls. The top frame represents the function currently in scope.

When symbolic information is available for a function, you can display the parameters and local variables in the variables list pane by selecting the frame in the stack list pane. Variables appear in the same format as in the Variable window.

Stack usage is described by the stack meter. The percent of stack area currently in use is shown in blue. Yellow indicates stack underflow. Purple indicates stack overflow. The following figure shows the Stack window Options menu.



You can configure the stack list to display stack and return addresses for each frame. Open the Options menu and toggle Include Stack Address and Include Return Code Address. The stack address is the address of the frame on the stack. The code address is the return address to the calling function in memory. Frames for functions with no symbolic information show addresses only, without function names.

To view the source of a function on the stack, select the frame; open the Options menu and choose Inspect Source. The Source window changes to show the function.

You can configure the stack meter to show the highest level the stack has reached since initialization (the high-water mark). The high-water mark is an arrow on the left side of the stack meter. Open the Options menu and toggle Enable High-Water Mark; or enter EnableHighWaterMark or DisableHighWaterMark on the Shell command line.

You can set an alarm on the stack meter to notify you when stack usage exceeds a percentage of the stack area. If the alarm limit is exceeded

when emulation halts, a warning message appears. Open the Options menu, choose Alarm Limit, and specify a percent value from 1 to 100. Then, open the Options menu again and toggle Enable Alarm Limit on. Alternatively, in the Shell window you can set an alarm limit and enable or disable the alarm message with SetStackAlarm, EnableAlarmLimit, and DisableAlarmLimit commands. The alarm limit is marked as a red line across the stack meter.

The alarm message does not appear until emulation halts. During emulation, the stack can exceed the alarm limit without displaying the warning message. To monitor the amount of memory used by the stack while emulation continues, emulate by stepping continuously. In the Source window, open the Run menu and choose Step Over Continuously or Step Into Continuously.

When emulation halts, the stack information is updated with:

- the current function and variable information
- the percentage of the stack in use
- the High-Water Mark, if enabled
- the alarm, if enabled

If, after emulation halts, the stack area is discovered to be invalid, some Stack window features are invalidated and grayed-out in the menus. For example, the alarm, high-water mark, and stack meter become unavailable.

Monitor multiple stacks

For system using multiple stacks, you can track the stack in use at any given time. Create Shell aliases to define the base and size of each stack. For example:

```
alias "s1" "SetStackArea 4000 100";
alias "s2" "SetStackArea 3000 100";
```

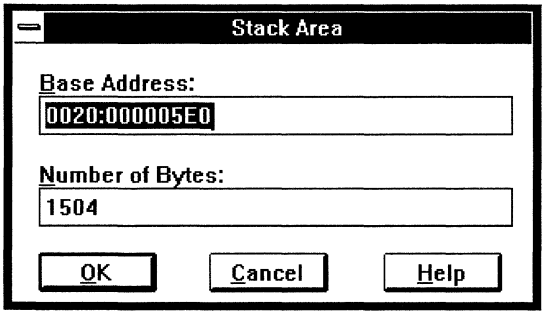
When emulation halts, switch to the current stack area by entering one of the aliases on the Shell command line.

Setting the Stack Base Address and Size

The stack base address and the stack size are typically put into the loadfile by your compiler. Otherwise, the emulator looks for a default stack base address in the powerpak.ini file. If powerpak.ini also specifies no base address, the current stack pointer value is used. If the stack size is undefined, the size defaults to 4K bytes.

To discover the current stack base and size, either enter StackInfo on the Shell command line, or in the Stack windowopen the Options menu and choose Stack Area. The values in the dialog box describe the

current stack allocation. The following figure shows a Stack Area dialog box.



If you edit these values, ensure the Base Address matches the CPU stack pointer initialized by your startup code and the Number of Bytes matches the stack size allocated for your target. Choose OK to set the stack base and size to new values, or Cancel to close the Stack Area dialog box without changing the stack area.

You can also change the stack area by a SetStackArea Shell command or by SetStackBase and SetStackSize Shell commands.

Determine how large a stack area to allocate

SLD can help you determine the minimum amount of memory to allocate for the stack. To discover the amount of memory used by the stack:

1. Open the Options menu and choose Enable High-Water Mark.
2. Execute your program for maximum code coverage.
3. Halt execution.
4. Note the high-water mark (maximum stack usage as a percentage of the allocated stack area) on the stack meter.
5. Increase or decrease the amount of memory allocated for the stack, allowing enough memory to accommodate the maximum stack usage without waste.

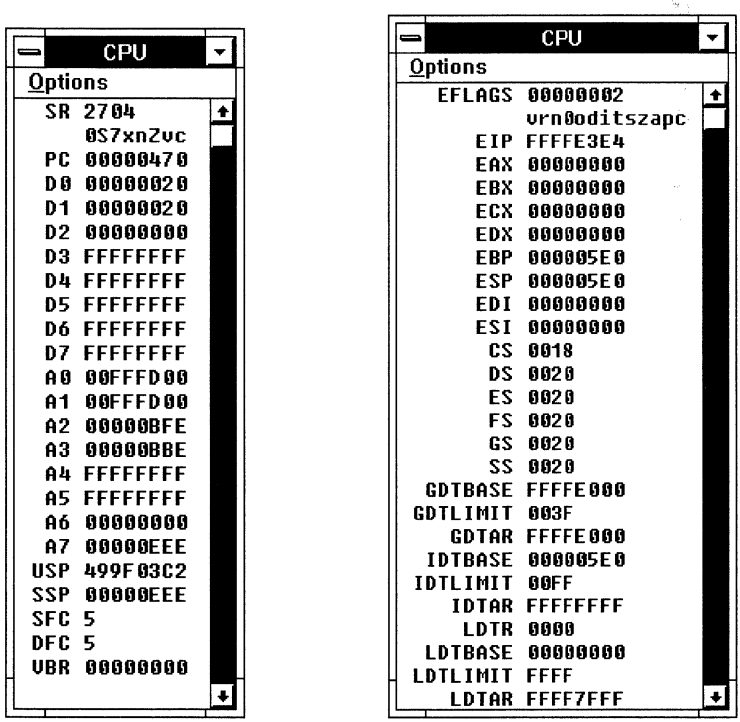
Debugging in Registers and Memory

This chapter describes how to access the CPU registers, the peripheral registers, and memory.

Viewing and Modifying the CPU Registers

You can view and change CPU registers and control signals from the CPU window, the Toolbar, the Source window, and the Shell command line.

To open the CPU window, on the Toolbar choose the CPU button, or in any SLD window open the Windows menu and choose CPU. The following figure shows CPU windows for the Motorola 68332 and Intel386EX processors:



The CPU window is updated when emulation halts. A highlight indicates a register value has changed. Selecting a register also highlights it.

Editing the CPU Registers

To edit a CPU register, you can either:

- In the CPU window, double-click on the register, or select the register and press <Enter>. Enter the new value in the dialog box.
- Enter a Register command on the Shell command line.

Resetting the CPU Registers

When you reset and reinitialize the processor:

- The processor RESET pin is asserted.
- The program counter and stack pointer are read from memory.
- All SLD windows are updated. The Stack window display is invalid because the stack is reset. The Source window displays the beginning of your startup code, at the program counter.

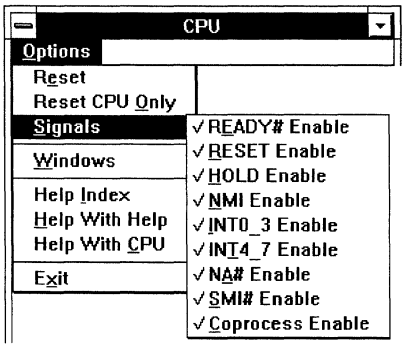
You can reset the processor from the Toolbar's Configure menu, from the Source window's Run menu, from the CPU window's Options menu, or by entering Reset on the Shell command line.

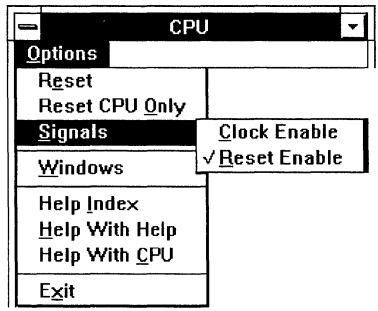
If the reset fails:

1. Open the Toolbar's Configure menu or the CPU window's Options menu and choose Reset CPU Only; or enter Reset CPUonly on the Shell command line. This resets the processor without updating the SLD windows.
2. Reset your target.
3. Reset the processor again, without specifying CPU only, to update the SLD windows.

Enabling the Target Signals

Enabling a signal uses that signal from your target system rather than from the emulator. To enable or disable the target signals, in the CPU window open the Options menu, choose Signals, and individually toggle each signal. The signals valid for your microprocessor are shown. The following figure shows the signals for an Intel386EX processor and for a Motorola 68332 processor.





For a list of the signals available for your processor, see the Signal command description in the "Shell Window Reference" chapter.

Disabling a signal disconnects it from the target and puts it under the emulator's control. For example, the emulator drives the Intel signals as:

READY#	asserted
RESET	negated
NMI	negated
INT0-INT3 (Intel386 EX processor)	negated
INT4-INT7 (Intel386 EX processor)	negated
NA#	negated
SMI# (Intel386 CX and EX processors)	negated
HOLD	negated
INTR	negated
A20M# (Intel386 CX processor)	negated
ERROR#, PEREQ, BUSY# (coprocessor)	negated

You can also enable and disable signals with the Shell Signal command.

Viewing and Modifying Memory

You can view and edit memory from the Memory window and by entering Dump, Write, Fill, Copy, and Search Shell commands.

Because reading and writing memory takes a small amount of processor time, memory access is initially disabled during emulation. Such access includes scrolling and refreshing the Memory and Peripheral windows and reading and writing memory from the Memory, Peripheral, and

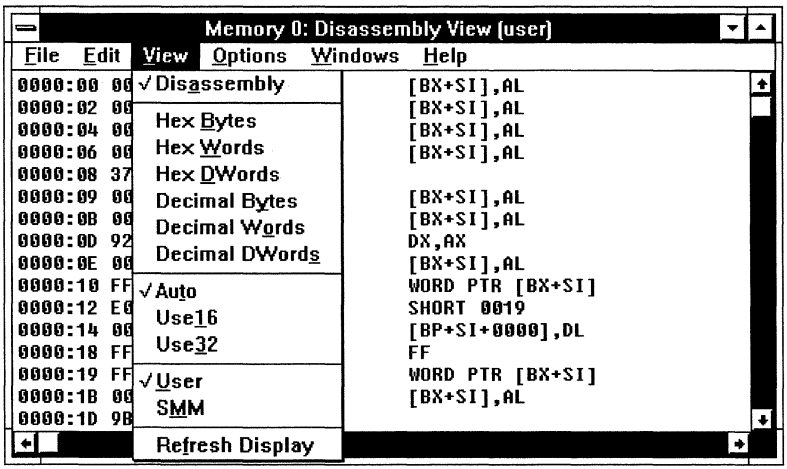
Shell windows. You can enable memory to be accessible during emulation; however, any such access can degrade your program execution. Before starting emulation, either:

- On the Toolbar open the Configure menu and check Run Access.
- Enter RunAccess ON on the Shell command line.

Changing the Memory Window Display

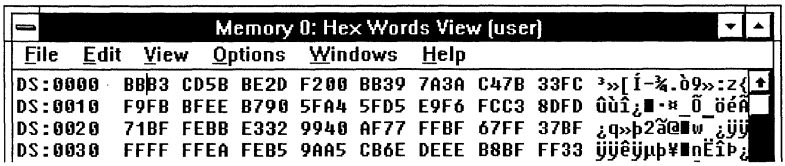
You can view memory as disassembly, hexadecimal, or decimal values. Open the View menu and choose the desired format. Up to 20 Memory windows with independent displays can be active simultaneously.

The following figure shows a sample Intel386 processor Memory window. This is the first-opened of the currently active Memory windows, as indicated by Memory 0 in the title bar. The View menu is open with disassembly format chosen.



When memory is displayed as disassembly, you can specify whether the disassembly uses your code symbols or the numeric addresses. On the Toolbar, open the Configure menu and toggle Symbolic Disassembly.

In a numeric view, memory is displayed as hexadecimal or decimal bytes, words, or double words followed by the ASCII equivalent, with periods representing non-printable characters. The following figure shows a sample Intel emulator Memory window displaying hexadecimal words. The address formats (in the left column) are different for Motorola emulators.



To view another area of memory, double-click in the address column of the Memory window; or open the Edit menu and choose Go To Address. Enter a numeric or symbolic address in the Go To Address dialog box. Any symbol you enter must have a fixed address, i.e., not a local variable or a stack-resident parameter.

If you are unsure of a symbol name or an address, you can research it from the Shell command line:

DisplaySymbols	lists module, variable, and function names with line number and address information.
AddressOf	lists the address of a specified symbol.
NameOf	lists the symbol closest to a specified address.

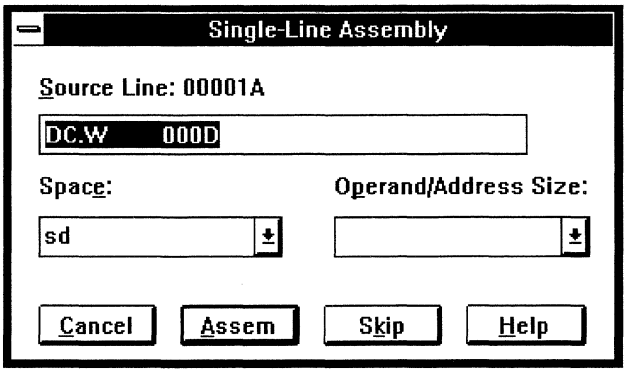
You can speed-up scrolling in the Memory window by enabling the Memory window cache. Open the Options menu and choose Read Ahead. When the Memory window cache is enabled near a non-existent memory region, the read ahead can cause a memory access failure.

Changing the Memory Contents

To change memory, you can:

- Edit the hexadecimal, decimal, or ASCII values in the Memory window. Position the cursor (a vertical bar) with the mouse, then overtype the memory display.
- Assemble code and data into memory using the Single-line Assembler dialog box in the Memory window.
- On the Shell command line, enter AsmAddr and Asm commands or Write, Fill, or Copy commands.

The following figure shows a sample Single-line Assembler dialog box for a Motorola emulator. The addresses, assembler syntax, Space, and Operand/Address Size options have different values for Intel processors.



To close the dialog box without assembling anything, choose Cancel. Once you have assembled a line, the Cancel button changes to a Close button.

To change a line in the Memory window:

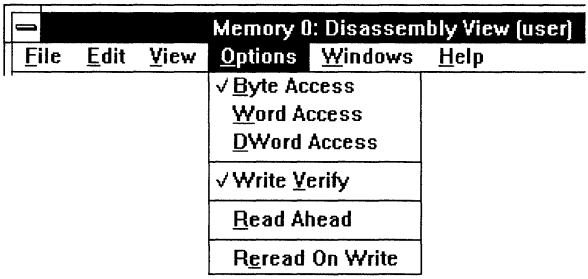
1. In the Memory window, open the View menu and choose Disassembly, displaying disassembled lines of code.
2. On the line you want to change, anywhere except in the address column, double-click. The Source Line field in the Single-line Assembler dialog box shows the address and initial value of the line to be changed.
3. Type a line of assembly code in the dialog box.
4. Select the space (user or SMM for Intel processors; SP, SD, UP, or UD for Motorola processors) and the operand/address size.
5. Choose Assem to write the code to memory and update the Memory window. The Single-line Assembler checks your assembly syntax; any error is reported and the erroneous line is not written.
6. Repeat steps 3 through 5 to assemble subsequent lines. Choose Skip to leave a line unchanged.
7. Choose Close to close the dialog box.

When the Memory window shows any view other than disassembly, you can edit the numeric and ASCII values. Position the cursor on the first value you want to change and type the new value. A value must fall within the range of the displayed radix. For example, in decimal byte radix the maximum value in a field is 255; if you try to replace 199 with 299, it is truncated to 200. An illegal entry causes a beep:

- Non-numeric values in Decimal display
- Non-hexadecimal values in Hexadecimal display

When more than one Memory window display the same area of memory, changes to that memory are reflected in all such Memory windows.

The size of values displayed in the Memory window does not affect how memory is accessed. Memory access is set by the Size command or the Options menu, not by the View menu. For example, if Size=byte, memory accesses are byte-sized even when the Memory window display is Hex Words. The following figure shows the Options menu.



Viewing and Modifying the Internal Peripheral Registers

You can view and modify the internal registers for each peripheral from the Peripheral window or from the Shell command line with a Register command. Note that your processor may require setup before some peripheral registers are accessible. See your Intel or Motorola processor documentation.

To open the Peripheral window, either open an SLD window Windows menu and choose Peripheral, or on the Toolbar choose the Periph button.

To display a specific peripheral group, register, or address in the Peripheral window, open the Edit menu and choose Go To Peripheral, Go To Register, or Go To Address, respectively.

The Intel processor registers have addresses in I/O space. In the Shell window, you can display such a register with a Dump IO command.

Because reading and writing memory takes a small amount of processor time, memory access is initially disabled during emulation. Such access includes scrolling and refreshing the Memory and Peripheral windows and reading and writing memory from the Memory, Peripheral, and Shell windows. You can enable memory to be accessible during emulation; however, any such access can degrade your program execution. Before starting emulation, either:

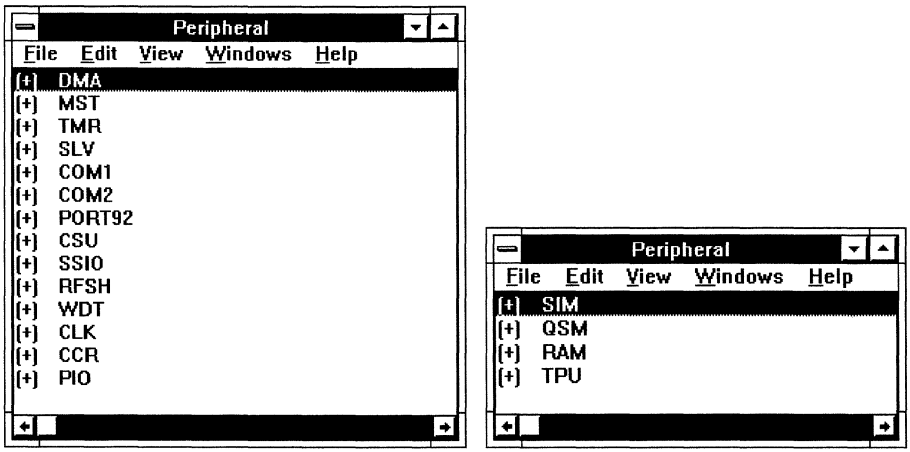
- On the Toolbar open the Configure menu and check Run Access.
- On the Shell command line, enter RunAccess ON.

Changing the Peripheral Window Display

Registers are displayed hierarchically. At the top level are the peripheral mnemonics; then the registers for each peripheral; then the bit fields for each register. You can expand or compress each level. When the Peripheral window display is fully compressed, only the peripherals appear. The columns in the Peripheral window are:

- A (+) symbol
- The peripheral mnemonic

The following figure shows the compressed display of peripherals for an Intel386EX processor and for a Motorola 68332 processor.



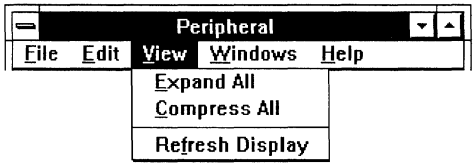
Expand a peripheral by clicking on the (+). The (+) changes to a (-) indicating the peripheral is expanded and a list of the peripheral's registers appears. Registers marked with (+) can be further expanded; to show a register's bit fields, click on the (+).

The register and bit field display columns are:

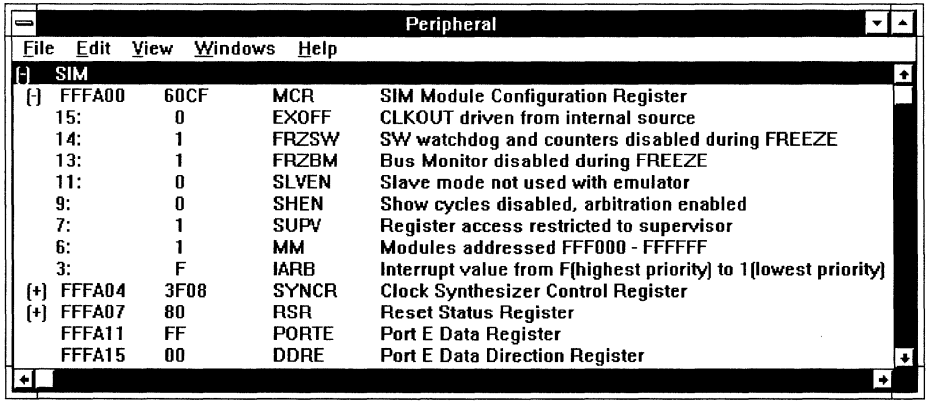
- A (+) or (-) symbol
- The register address; or, for a bit field, the bit number
- The field value
- The register or field mnemonic
- A description of the register or field

Click on the (-) to recompress the register or peripheral display.

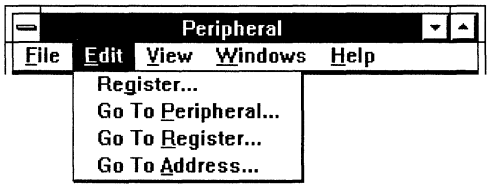
To display all peripherals and registers in expanded format, open the View menu and choose Expand All. The following figure shows a View menu.



The following figure shows part of the expanded display for the Motorola 68332 peripheral registers.



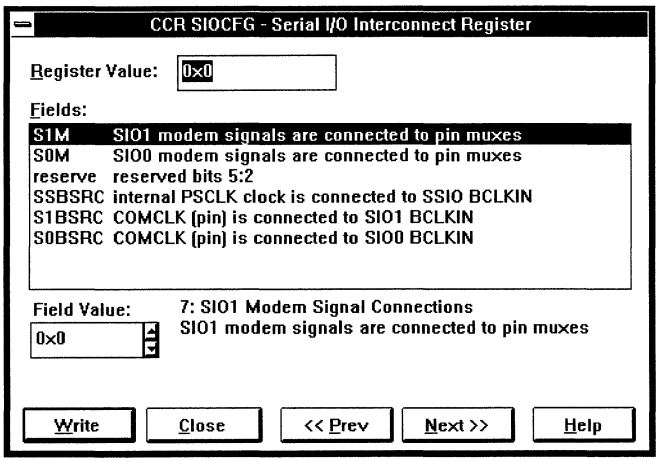
To navigate in the Peripheral window, open the Edit menu, choose one of the Go To... items, and enter the peripheral or register name or address in the dialog box. The following figure shows an Edit menu.



Changing the Peripheral Register Values

Double-click anywhere on a register line; or select the register, open the Edit menu, and choose Register. You can edit the register value or the individual register fields in the Register Edit dialog box. In the Shell window, you can use a Register command or Write, Copy, or Fill (for Intel, Write IO, Copy IO, or Fill IO) command to write to the register.

The following figure shows a sample Register Edit dialog box. This is the edit box for the Motorola 68332 CCR peripheral SIOCFG register. The register field values and descriptions are different for each register, although the layout and operation of the dialog box is consistent across registers and across processors.



Debugging With Triggers and Trace

Use events to define triggers for controlling emulation and collecting trace. Search the trace buffers for specific events to reconstruct your program activity.

An event is a combination of addresses, data, and signals occurring during emulation.

A trigger uses an event as a catalyst or condition for an action. When an event specified in a trigger occurs, the associated action is performed.

An action can control trace, emulation, and subsequent triggering.

Address Formats

This section describes the symbolic and numeric address formats you need to know for defining events and interpreting trace information.

Symbolic Addresses

Symbols, interpreted as a symbolic segment:offset, are virtual addresses. You can specify a symbolic reference in a command, dialog box, or expression. You can simplify access to program symbols by taking advantage of the way symbol names are resolved by the emulator. For example, when looking up a symbol in the current module, you need not specify the module and function.

A symbol table contains the names of all modules, functions, variables, and line numbers that were compiled into the loadfile. The loader reads information about the program symbols, including the line numbers, from the loadfile to create the symbol table.

The symbol information is hierarchical, with each symbol represented as a range of addresses:

At the top	of the hierarchy are modules, public labels, and public variables.
Modules	contain functions, static variables, and line and column numbers.
Functions	contain parameters, local variables, static variables, line numbers, and blocks.

Blocks — are handled as if they were unnamed functions. Nested blocks can also contain local and static variables defined in their scope.

Using this symbol hierarchy, you can uniquely specify a symbol. A fully qualified symbol has one, two, or three names (a name can be a number) beginning with #. If a symbol is not fully qualified, it defaults to the current module and function, that is, the scope of the current program counter.

The rules for symbol look-up are:

1. Attempt to match the symbol at the lowest level of the hierarchy.
2. If a match is not found, attempt to match the symbol at the next outer level.
3. If no match is found, attempt to match the symbol at the global level.
4. If no match is found, the symbol name does not exist and a symbol-not-found error is returned.

To find symbolic variables with one name:

- If the module and function are defined by the context, look up the name as a variable within the scope of the function.
- If the module is defined by the current context but the function is not defined by the context (e.g., you have stepped from the module into a called assembly routine), look up the name within the scope of the module.
- If no module or function is defined by the current context, look up the name as a module, or look up the name as public variable or label.
- If the name is a number, look up the number as a module name or a line number within the current module.

One-name smbols

#module1	Returns the beginning address of module1.
#function1	Is the function in the current module? If so, its address is returned. If not, the function must be in the global table (all functions are in the global table unless they are prefixed by static.)
#variable1	Is the variable in the current program? The variable can be inside a nested block, function, module, or it can be a global or public variable.
#55	Looks up the starting address of line 55 in the current module.

To find symbolic variables with two names:

- If a module is defined by the current context, look up the first name as a function contained within the module. If a module context does not exist, first look up the first name as a module, then look it up as a global function.
- If the module and function are defined by the context, look up the second name as a variable within the scope of the function.
- If the module is defined by the current context but the function is not defined by the current context, look up the second name as a variable within the scope of the module.
- If no module or function are defined by the current context, look up the second name as public variable or label.
- If the first name is a number, look up the first name as a module name or a line number within the current module. If the second name is a number, look up the second name as a line number if the first name is a module or function, otherwise as a column number.

Two-name smbols

#55#15	Look up the address in the current module on line 55, column 15.
#module1#100	Address of line 100 in module1.
#module1#func1	Address of func1 in module1.
#module1#var1	Address of static var1 in module1.
#func1#var1	Is func1 in the current module? If not, is func1 global? Then, find var1 in scope of func1.

To find symbolic variables with three names:

- The first name is always a module. The second and third can be line and column numbers. If the second and third are not line and column numbers, then the second is a function within the module and the third is a variable or line number within the function scope.
- If the third name is a variable it is first looked up within the module/function context. If not found, it is looked up as a global variable or label. This symbol's address is returned even if that symbol is not in the scope of the entered module.

Three-name smbols

#mod1#25#1	Address of start of code column 1, line 25 of module mod1.
#mod1#func1#100	Address of line 100 in module1.
#module1#func1#var1	Address of var1 in func1 in module1.

Line Numbers

To display line numbers in the Source window, open the View menu and check Line Number. In the Shell window, you can list all line-number records for the current module with displaySymbols lines.

Some line numbers are comment lines and have no compiled code.

Intel Numeric Addresses

The Intel386 processors operate in different processor modes (pmodes): real, virtual-86 (V86), protected, and (for the CX and EX) System Management Mode (SMM). Protected mode is further divided into 16-bit and 32-bit modes.

These processors have a segmented architecture, i.e. addresses consist of a segment and an offset. The segment determines the base address of an addressable region, and the offset is added to that base to arrive at the final linear address. In some modes, the linear address may be further processed by the paging unit to construct the physical address seen on the processor pins.

The segment registers consist of a 16-bit user-visible register (CS, DS, ES, FS, GS, or SS) and 3 hidden components (the segment base, limit, and access rights). The pmode affects how the processor loads the hidden portion of the segment registers.

When the 16-bit visible segment register is loaded by the user program, the processor automatically loads the hidden portion based on rules determined by the pmode. In real and V86 mode, the base is the segment multiplied by 16, the limit is always 64K bytes, and the access rights allow execution, read, and write. In protected mode, the base, limit and access rights are extracted from the segment descriptor indicated by the segment register value. The descriptor is an 8-byte data structure in one of two arrays called the global descriptor table (GDT) and local descriptor table (LDT). Bit 2 of the segment register selects which table is used. In SMM, the base and access rights are as in real mode, but the limit is always 4 gigabytes (4G bytes).

Pmode also affects whether the paging unit can be used. In real and SMM modes, the paging unit is not used, so the physical address is always the same as the linear address. In V86 and protected modes, paging is active if the PG bit in the CR0 register is set.

Finally, pmode affects the processor instruction set. The Intel386 processor has two sets of addressing modes: 16-bit and 32-bit; and two default data sizes: 16-bit and 32-bit. The default address and data sizes

are determined by the pmode and the D bit in the code segment descriptor.

In real, V86, and SMM modes, 16-bit is the default. In protected mode, the D bit determines the default address size (the difference between 16-bit and 32-bit protected modes). An address size override prefix byte can be added to any instruction to switch to the opposite (non-default) address size, so even in real mode the 32-bit addressing modes can be used. Similarly, a data size override can be used to select the opposite data size. Thus, even in real mode, a program can directly use 32-bit data quantities.

For example, the instruction 89 00 is:

addr size	data size	instruction
16	16	mov [bx+si],ax
32	16	mov [eax],ax
16	32	mov [bx+si],eax
32	32	mov [eax],eax

Specify numeric addresses as:

Format	Address Type
<offset>L	Linear Address
<offset>P	Physical Address
[(#module)]#symbol	Symbolic segment:offset interpreted as a virtual address
<ldt>::<offset>	Virtual address with specified LDT
:<offset>	Virtual address using current LDT
<offset>	Virtual address assuming current LDT and DS

To find the linear or physical equivalent of an address, use an Xlt Shell command.

The emulator checks address limits:

Type	Pmode	Processor	Limits
Virtual	SMM	all	0:0 to FFFF:FFFFFFFF
	Real	all	0:0 to FFFF:FFFF
	Virtual-86	all	0:0 to FFFF:FFFF
	Protect16, 32	all	selector ≤ table limit; offset within segment limit

Linear	all	all	0 to FFFFFFFF
Physical	all	386DX	0 to FFFFFFFF
	all	386SX	0 to FFFFFF
	all	386CX	0 to 3FFFFFF
	all	386EX	0 to 3FFFFFF

Events

An event definition is used:

- In a trigger, to control emulation and trace collection. When the event occurs, the emulator performs the specified actions.
- To find specific activity recorded in trace. In a trace buffer, search for a named event.

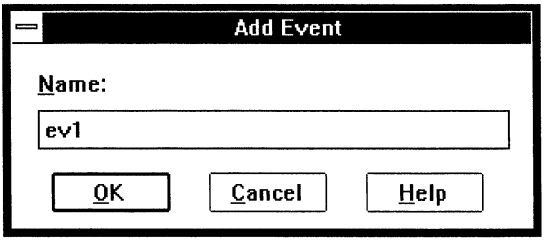
An event is a combination of:

Addresses	Reading or writing to a specific address, set of addresses, inside an address range, or "not" the described addresses. You can specify symbolic or numeric addresses.
Data	Reading or writing a specific value, set of values, range of values, or "not" the described values. You can specify symbolic or numeric data.
Signals	High or low logic levels on various processor signals. You can also specify don't-care for signals.

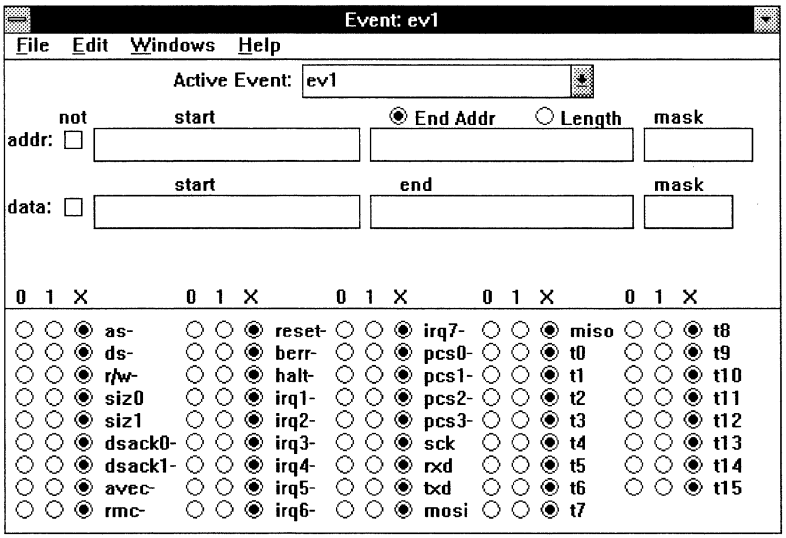
Define an event in the Event edit box, also called the Event window. Editing the Event edit box differs from editing a dialog box. The <Enter> key has no effect on the field that you are editing. To ensure a field accepts an entry, move the cursor by clicking on another field or button. Pressing the <Delete> key to delete a highlighted value has no effect; press the space-bar instead.

You can open the Event edit box from the Trigger or Trace window, by opening the Edit menu and choosing Events, or from the Windows menu of any SLD window.

If no events are defined, the Add Event dialog box appears. Otherwise, to add a new event, in the Event edit box open the Edit menu, choose Add Event, and enter the new Event name. The following figure shows an Add Event dialog box.



The following figure shows the Event edit box for a Motorola 68332 processor. The available signals differ for different processors and, for Motorola processors, can vary according to the chip select register configurations.



To define the address of an event: (If you don't care what addresses are accessed, leave all the Addr fields blank.)

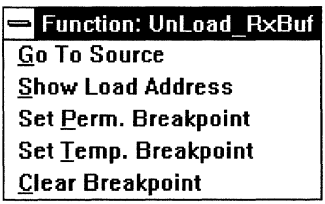
1. Enter a symbolic or hexadecimal numeric address in the Addr Start field. This is the first address in the region where the event can occur.

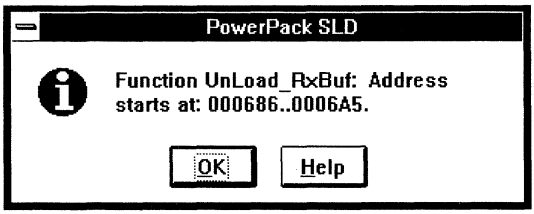
2. Select End Addr or Length. Enter either the last address in the memory region where the event can occur, or the length in bytes of the region.

   If you are unsure of an address or address range, you can use the Shell window AddressOf and NameOf commands or the Source window Function pop-up menu. For example, with the following information you can define an event relative to addresses occupied by the Load_CmdBuf function or the MsgRx variable:

```
>nameof 680              // Find what function this address is in
// #332qsm#432#1 (function Load_CmdBuf+0x30 [48])

>addressof #Load_CmdBuf
// 650..685              // Address range occupied by the function

>nameof e70             // Find the closest symbol to this address
// #main#MsgRx+0x8 [8]

>addressof #MsgRx
// E68..E87 [32]        // Address range occupied by the variable
```

Another way to find the memory region of a function is via the Function pop-up menu. In the Source window, double-click on the function name and choose Show Load Address. The following figure shows a Function pop-up menu and the Load Address information box.





3. Optinally, you can enter a binary-AND mask value. The mask dictates which bits of the address are don't-care's (0) and which must match (1).
4. To match only addresses outside of the range or set you specified, check the Not box.

To define the data of an event: (If you don't care what data is read or written, leave all the Data fields blank.)

1. Enter numeric values in the Data Start and Data End fields. The emulator interprets the numbers as decimal unless you use the 0x prefix. For example, 10 is translated to 0x000A, and 0x10 is accepted as 0x0010.
2. Enter a binary-AND mask, using all 1's to match the described data exactly.

3. To match only data outside of the range or set you specified, check the Not box.

Specify signal states for the event by toggling the low (0), high (1) or don't care (X) buttons next to each signal mnemonic. Active-low signals are shown with a hash mark (#) for Intel emulators or minus sign (-) for Motorola emulators. The signals available depend on the target processor. For some Motorola processors, the signals available can also depend on your chip select register configurations.

You can define events in one emulator session and save them for reuse in another session. To save events to a file, in the Event window open the File menu and choose Save Events As. To retrieve saved events, choose Restore Events. Or, enter EventSave and EventRestore commands on the Shell command line.

For Motorola emulation, you can specify the address space for an event as UD, UP, SD, or SP. To make the space selection available in the Event edit box, you must program the processor to output the three function codes FC0, FC1, and FC2.

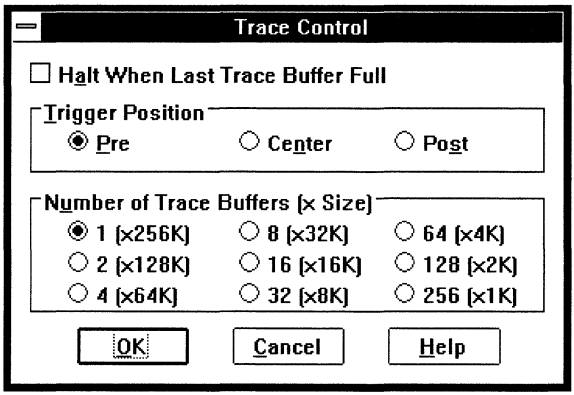
Trace

Trace is a record of the processor bus events occurring each clock cycle during emulation. With the trace information, you can find specific events and reconstruct a history of the executed instructions and the resulting data transfers to and from the processor.

Controlling Trace Collection

You can interactively control trace collection with the Toolbar Start and Stop buttons or automate trace collection with triggers based on events in your program execution. The Status window or icon message shows whether the emulator is tracing. You need not halt emulation to examine the collected trace.

To configure trace collection, in the Trace window open the Trace menu (in the Trigger window, open the Options menu); choose Trace Control. The following figure shows a Trace Control dialog box.



In the Trace Control dialog box:

- Specify the number and sizes of trace buffers to be filled. With multiple buffers, you can collect several sections of code execution.
- Locate where the triggering event occurs in the collected trace in any buffer. Unless you halt emulation, trace collection in the buffer continues after the triggering event until the buffer is full.

  Pre — collects cycles before the trigger. The triggering event appears near the end of the buffer.

  Center — collects cycles before and after the trigger. The triggering event appears in the middle of the buffer.

  Post — collects cycles after the trigger. The triggering event appears near the beginning of the buffer.
- When you are filling four or more trace buffers, you can halt emulation when all the buffers are full. This operation overwrites the first buffer with several cycles after the end of the last buffer.

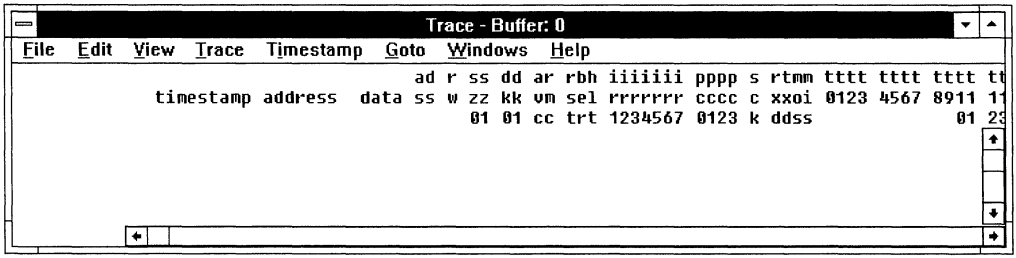
Displaying the Collected Trace

To display a trace buffer, open the Trace window. Move between multiple trace buffers by opening the Goto menu and choosing Previous Buffer, Next Buffer, or Buffer.

Each time emulation halts or you turn trace off, the Trace window is updated. The trace information includes:

- The timestamp of the clock cycle
- The values on the address and data pins during the clock cycle
- Various signal values at the time of the clock cycle

Read the abbreviated signal mnemonics vertically. The following figure shows a Trace window. The available signals differ for different processors and, for Motorola processors, can vary according to the chip select register configurations.



From the View menu, you can display trace as:

Clock mode	processor pin states at each clock
Bus mode	processor bus cycle activity
Instruction mode	disassembly of instructions executed by the processor and memory accesses associated with the executed instructions

You can link the Source and Trace window displays. When the windows are linked, you can scroll through the Trace window and view the corresponding code scrolling synchronously in the Source window. To link the Source window to the Trace window:

1. In the Trace window, open the View menu and choose Instruction.
2. Re-open the View menu and choose Linked Cursor.

With Linked Cursor, you can view the history of executed source lines in instruction mode. Linked Cursor is disabled in clock and bus modes.

Trace and Event Window Signals

The Trace and Event windows display signal name mnemonics corresponding to the Intel or Motorola mnemonics, as listed (alphabetically) in the tables in this section for each microprocessor.

You can configure some pins as secondary I/O signals. You must keep track of how your signals are configured, since the Trace and Event windows identify the signals only by their primary use.

In these tables, # (for example, ADS#) and - (for example, r/w-) indicate active-low.

Intel386EX Signals

Trace	Event	Signal
ads	ADS#	Address Status
bhe	BHE#	Byte High Enable
bs8	BS8#	Bus Size Control
bsy	BUSY#	Busy
cs6	CS6#	Chip Select 6; Muxed with REFRESH#
dc	D/C#	Data/Control Status
err	ERROR#	Error
in4	INT4	Interrupt Request 4; Muxed with TMRCLK0
in5	INT5	Interrupt Request 5: Muxed with TMRGATE0
in6	INT6	Interrupt Request 6; Muxed with TMRCLK1
in7	INT7	Interrupt Request 7; Muxed with TMRGATE1
mio	M/IO#	Memory/IO Status
na	NA#	Next Address
nmi	NMI	Non-maskable Interrupt Request
p15	P1.5	Port 1 Pin 5; Muxed with LOCK#
p16	P1.6	Port 1 Pin 6; Muxed with HOLD
p17	P1.7	Port 1 Pin 7; Muxed with HLDA
p20 - p24	P2.0 - P2.4	Port 2 Pins 0 - 4; Muxed with CS0# - CS4#
p25	P2.5	Port 2 Pin 5; Muxed with RXD0
p26	P2.6	Port 2 Pin 6; Muxed with TXD0
p27	P2.7	Port 2 Pin 7; Muxed with CTS0#
p30 - p31	P3.0 - P3.1	Port 3 Pins 0 - 1; Muxed with TMROUT0 - TMROUT1
p32 - p35	P3.2 - p3.5	Port 3 Pins 2 - 5; Muxed with INT0 - INT3
p36	P3.6	Port 3 Pin 6; Muxed with PWRDOWN
p37	P3.7	Port 3 Pin 7; Muxed with COMCLK
per	PEREQ	Processor Extension Request

rdy	READY#	Ready
rst	RESET	Reset
sma	SMIACT#	System Management Interrupt Active
smi	SMI#	System Management Interrupt
wr	W/R#	Write/Read

Intel386CX Signals

Trace	Event	Signal
a20	A20M#	Address 20 Mask
ads	ADS#	Address Status
bhe	BHE#	Byte High Enable
bsy	BUSY#	Busy
dc	D/C#	Data/Control Status
err	ERROR#	Error
hla	HLDA	Hold Acknowledge
hld	HOLD	Hold Request
int	INTR	Interrupt Request
lck	LOCK#	Bus Lock
mio	M/IO#	Memory/IO Status
na	NA#	Next Address
nmi	NMI	Non-maskable Interrupt Request
per	PEREQ	Processor Extension Request
rdy	READY#	Ready
rst	RESET	Reset
sma	SMIACT#	System Management Interrupt Active
smi	SMI#	System Management Interrupt
wr	W/R#	Write/Read

Intel386SX Signals

Trace	Event	Signal
ads	ADS#	Address Status
bhe	BHE#	Byte High Enable
bsy	BUSY#	Busy

dc	D/C#	Data/Control Status
err	ERROR#	Error
hla	HLDA	Hold Acknowledge
hld	HOLD	Hold Request
int	INTR	Interrupt Request
lck	LOCK#	Bus Lock
mio	M/IO#	Memory/IO Status
na	NA#	Next Address
nmi	NMI	Non-maskable Interrupt Request
per	PEREQ	Processor Extension Request
rdy	READY#	Ready
rst	RESET	Reset
wr	W/R#	Write/Read

MC68332/333 Signals

Trace	Event	Signal
as	as-	AS# Address Strobe
ds	ds-	DS# Data Strobe
rw	r/w-	R/W# Read/Write
sz0	siz0	SIZ0 Transfer Size
sz1	siz1	SIZ1 Transfer Size
dk0	dsack0-	DSACK0# Data and Size Acknowledge
dk1	dsack1-	DSACK1# Data and Size Acknowledge
avc	avec-	AVEC# Autovector
rmc	rmc-	RMC# Read-Modify-Write Cycle
rst	reset-	RESET# Reset
ber	berr-	BERR# Bus Error
hlt	halt-	HALT# Halt
ir1	irq1-	IRQ1# Interrupt Request Level 1
ir2	irq2-	IRQ2# Interrupt Request Level 2
ir3	irq3-	IRQ3# Interrupt Request Level 3
ir4	irq4-	IRQ4# Interrupt Request Level 4

ir5	irq5-	IRQ5# Interrupt Request Level 5
ir6	irq6-	IRQ6# Interrupt Request Level 6
ir7	irq7-	IRQ7# Interrupt Request Level 7
pc0	pcs0-	PCS0#/SS QSPI Peripheral Chip Selects/Slave Select
pc1	pcs1-	PCS1# QSPI Peripheral Chip Selects
pc2	pcs2-	PCS2# QSPI Peripheral Chip Selects
pc3	pcs3-	PCS3# QSPI Peripheral Chip Selects
sck	sck	SCK QSPI Serial Clock
rxd	rxd	RXD SCI Receive Data
txd	txd	TXD SCI Transmit Data
mos	mosi	MOSI Master-Out Slave-In
mis	miso	MISO Master-In Slave-Out
t0 to t15	t0 to t15	TP[0:15] TPU Channel Input/Output

You can program the SIM (system integration module) peripheral CSPAR0 (chip select pin assignment register 0) to make the following signals also available. For an example, see the section on programming the Motorola chip selects in the "Defining the Debug Environment" chapter.

bgack	bgack-	Bus Grant Acknowledge
bg	bg-	Bus Grant
br	br-	Bus Request
portc.2	portc2	User-configurable I/O Port 2
portc.1	portc1	User-configurable I/O Port 1
portc.0	portc0	User-configurable I/O Port 0

MC68331/MC68HC16Z1 Signals

Trace	Event	Signal
as	as-	AS# Address Strobe
ds	ds-	DS# Data Strobe
rw	r/w-	R/W# Read/Write
sz0	siz0	SIZ0 Transfer Size
sz1	siz1	SIZ1 Transfer Size

dk0	dsack0-	DSACK0# Data and Size Acknowledge
dk1	dsack1-	DSACK1# Data and Size Acknowledge
avc	avec-	AVEC# Autovector
rmc	rmc-	RMC# Read-Modify-Write Cycle (MC68331 only)
rst	reset-	RESET# Reset
ber	berr-	BERR# Bus Error
hlt	halt-	HALT# Halt
ir1	irq1-	IRQ1# Interrupt Request Level 1
ir2	irq2-	IRQ2# Interrupt Request Level 2
ir3	irq3-	IRQ3# Interrupt Request Level 3
ir4	irq4-	IRQ4# Interrupt Request Level 4
ir5	irq5-	IRQ5# Interrupt Request Level 5
ir6	irq6-	IRQ6# Interrupt Request Level 6
ir7	irq7-	IRQ7# Interrupt Request Level 7
pc0	pcs0-	PCS0#/SS QSPI Peripheral Chip Selects/Slave Select
pc1	pcs1-	PCS1# QSPI Peripheral Chip Selects
pc2	pcs2-	PCS2# QSPI Peripheral Chip Selects
pc3	pcs3-	PCS3# QSPI Peripheral Chip Selects
sck	sck	SCK QSPI Serial Clock
rxd	rxd	RXD SCI Receive Data
txd	txd	TXD SCI Transmit Data
mos	mosi	MOSI Master-Out Slave-In
mis	miso	MISO Master-In Slave-Out
ic1	ic1	IC1 GPT Input Capture 1
ic2	ic2	IC2 GPT Input Capture 2
ic3	ic3	IC3 GPT Input Capture 3
ic4	ic4	IC4/OC5 GPT Input Capture 4 / Output Cmpr 5
oc1	oc1	OC1 GPT Output Compare 1
oc2	oc2	OC2 GPT Output Compare 2
oc3	oc3	OC3 GPT Output Compare 3
oc4	oc4	OC4 GPT Output Compare 4

pai	pai	PAI Pulse Accumulator Intpu
pwa	pwma	PWMA GPT Pulse Width Modulation A
pwb	pwmb	PWMB GPT Pulse Width Modulation B

You can program the SIM (system integration module) peripheral CSPAR0 (chip select pin assignment register 0) to make the following signals also available. For an example, see the section on programming Motorola chip selects in the "Defining the Debug Environment" chapter.

bgack	bgack-	Bus Grant Acknowledge
bg	bg-	Bus Grant
br	br-	Bus Request
portc.2	portc2	User-configurable I/O Port 2
portc.1	portc1	User-configurable I/O Port 1
portc.0	portc0	User-configurable I/O Port 0

MC68330 Signals

Trace	Event	Signal
as	as-	AS# Address Strobe
ds	ds-	DS# Data Strobe
rw	r/w-	R/W# Read/Write
uwe	uwe-	UWE# Upper Write Enable
lwe	lwe-	LWE# Lower Write Enable
sz0	siz0	SIZ0 Transfer Size
sz1	siz1	SIZ1 Transfer Size
dk0	dsack0-	DSACK0# Data and Size Acknowledge
dk1	dsack1-	DSACK1# Data and Size Acknowledge
avc	avec-	AVEC# Autovector
rmc	rmc-	RMC# Read-Modify-Write Cycle
rst	reset-	RESET# Reset
ber	berr-	BERR# Bus Error
hlt	halt-	HALT# Halt
ir1	irq1-	IRQ1# Interrupt Request Level 1
ir2	irq2-	IRQ2# Interrupt Request Level 2

ir3	irq3-	IRQ3# Interrupt Request Level 3
ir4	irq4-	IRQ4# Interrupt Request Level 4
ir5	irq5-	IRQ5# Interrupt Request Level 5
ir6	irq6-	IRQ6# Interrupt Request Level 6
ir7	irq7-	IRQ7# Interrupt Request Level 7

MC68340 Signals

Trace	Event	Signal
as	as-	AS# Address Strobe
ds	ds-	DS# Data Strobe
rw	r/w-	R/W# Read/Write
sz0	siz0	SIZ0 Transfer Size
sz1	siz1	SIZ1 Transfer Size
dk0	dsack0-	DSACK0# Data and Size Acknowledge
dk1	dsack1-	DSACK1# Data and Size Acknowledge
avc	avec-	AVEC# Autovector
rmc	rmc-	RMC# Read-Modify-Write Cycle
rst	reset-	RESET# Reset
ber	berr-	BERR# Bus Error
hlt	halt-	HALT# Halt
fc3	fc3	Function Code 3
ir1	irq1-	IRQ1# Interrupt Request Level 1
ir2	irq2-	IRQ2# Interrupt Request Level 2
ir3	irq3-	IRQ3# Interrupt Request Level 3
ir4	irq4-	IRQ4# Interrupt Request Level 4
ir5	irq5-	IRQ5# Interrupt Request Level 5
ir6	irq6-	IRQ6# Interrupt Request Level 6
ir7	irq7-	IRQ7# Interrupt Request Level 7
rxa	rxda	RxDA Receive Data Channel A
txa	txda	TxDA Transmit Data Channel A
rda	rxrdya-	RxRDYA Receiver Ready
tda	txrdya-	TxRDYA Transmitter Ready

rxb	rxdb-	RXDB Receive Data Channel B
txb	txdb-	TXDB Transmit Data Channel B
ti1	tin1	TIN1 Timer Input 1
to1	tout1	TOUT1 Timer Out 1
tg1	tgate1-	TGATE1# Timer Gate 1
ti2	tin2	TIN2 Timer Input 2
to2	tout2	TOUT2 Timer Out 2
tg2	tgate2-	TGATE2# Timer Gate 2
dr1	dreq1-	DREQ1# DMA Request 1
da1	dack1-	DACK1# Data Acknowledge 1
do1	done1-	DONE1# Data Done 1
dr2	dreq2-	DREQ2# DMA Request 2
da2	dack2-	DACK2# Data Acknowledge 2
do2	done2-	DONE2# Data Done 2
br	br-	BR# Bus Request
bg	bg-	BG# Bus Grant
bga	bgack-	BGACK# Bus Grant Acknowledge

MC68360 Signals

Trace	Event	Signal
as	as-	AS# Address Strobe
ds	ds-	DS# Data Strobe
rw	r/w-	R/W# Read/Write
sz0	siz0	SIZ0 Transfer Size
sz1	siz1	SIZ1 Transfer Size
dk0	dsack0-	DSACK0# Data and Size Acknowledge
dk1	dsack1-	DSACK1# Data and Size Acknowledge
rmc	rmc-	RMC# Read-Modify-Write Cycle
rsh	reseth-	RESETH# Hard Reset
rss	resets-	RESETS# Soft Reset
ber	berr-	BERR# Bus Error
hlt	halt-	HALT# Halt

fc3	fc3	FC3 Function Code 3
ir1	irq1-	IRQ1# Interrupt Request Level 1
ir2	irq2-	IRQ2# Interrupt Request Level 2
ir3	irq3-	IRQ3# Interrupt Request Level 3
ir4	irq4-	IRQ4# Interrupt Request Level 4
ir5	irq5-	IRQ5# Interrupt Request Level 5
ir6	irq6-	IRQ6# Interrupt Request Level 6
ir7	irq7-	IRQ7# Interrupt Request Level 7
br	br-	BR# Bus Request
bg	bg-	BG# Bus Grant
bga	bgack-	BGACK# Bus Grant Acknowledge

Triggers

A trigger performs one or more actions when a condition occurs. The condition can be a combination of events, timer or counter values, and an active-low external signal. The action can be starting or stopping trace, stopping emulation, starting or stopping a counter or timer, or arming another trigger.

Defining a Trigger

To define a trigger, on the Toolbar select the Trigger button (in any SLD Windows menu select Trigger). The Condition pane of the Trigger window specifies the events, timer or counter values, or active-low external signal on which to trigger; the Actions pane describes the emulation actions to be taken when the conditions are met.

To specify whether the trigger occurs on a bus or clock cycle, open the Options menu and choose:

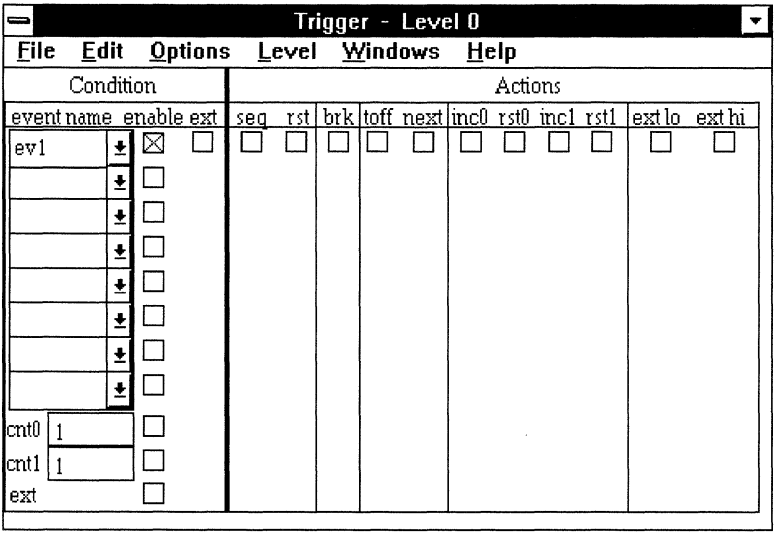
Bus automatically samples processor pins at the proper time in a bus cycle. The trigger is based on aligned samples.

Clock triggers on any cycle coming from the processor, regardless of whether it is a valid bus cycle. Use clock triggering to trigger on an I/O signal or on an interrupt input that can occur on any clock cycle.

The Trigger window provides up to four levels of triggers: Level 0, 1, 2, or 3 appears in the Trigger window title bar. Levels are processed sequentially. A sequencing (seq) action disables the set of conditions defined in the current level and enables the set of conditions in the next level.

All conditions on a level are processed in parallel. That is, if two or more conditions are true simultaneously, all associated actions occur.

The following figure shows a Trigger window at Level 0.



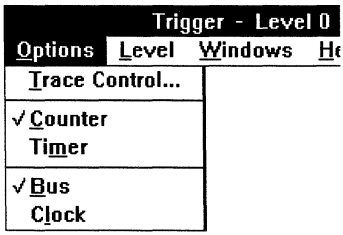
In the Condition pane, specify a previously defined event name. Click on an event name list box. In the drop-down list box, click on the event that you want to use as a trigger condition. Check the Enable box to the right of the event name. Click in the row of boxes to specify the actions to be taken if the trigger condition is met. The conditions and actions are described in detail in the "Trigger Window Reference" chapter.

The timer increments at the clock rate of the emulation processor and wraps to 0 after reaching its maximum value. To calculate the milliseconds (ms) for a complete timer cycle:

wrap time = (2^{20}) / (clock period)

For example, at 25 MHz, the timer wraps in about 42 ms; at 16 MHz, the timer wraps in about 65.5 ms.

For counter conditions and actions, open the Options menu and check Counter. For the timer, check Timer. The following figure shows an Options menu.

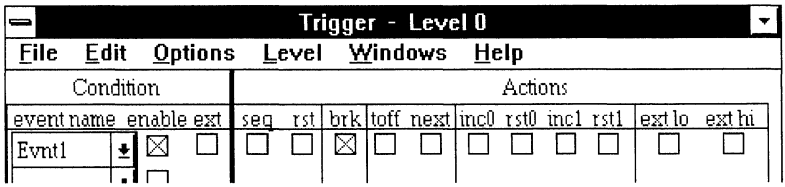


Examples of Triggering

This section demonstrates various trigger window configurations and describes their effects on emulation control.

Break Emulation

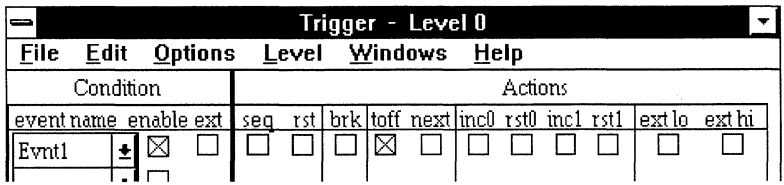
If Evnt1 occurs, emulation breaks.



1. Enable Evnt1 and choose the brk action.
2. Start emulation.
3. Tracing starts.
4. Emulation stops when the trigger occurs.

Stop Trace Without Breaking Emulation

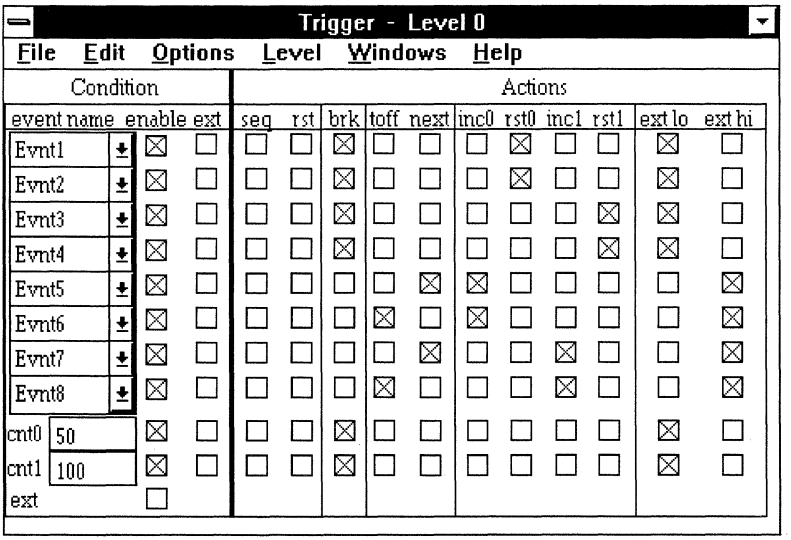
If Evnt1 occurs, trace collection stops.



1. Enable Evnt1 and choose the toff action.
2. Start emulation.
3. When the trigger occurs, the trace buffer fills according to Trace Control; tracing stops; emulation continues.

Act On Multiple Events

Enable up to eight global events. Enabled events are logically ANDed. For this example, multiple trace buffers must be defined in the Options menu Trace Control dialog box and Counters must be selected in the Options menu.

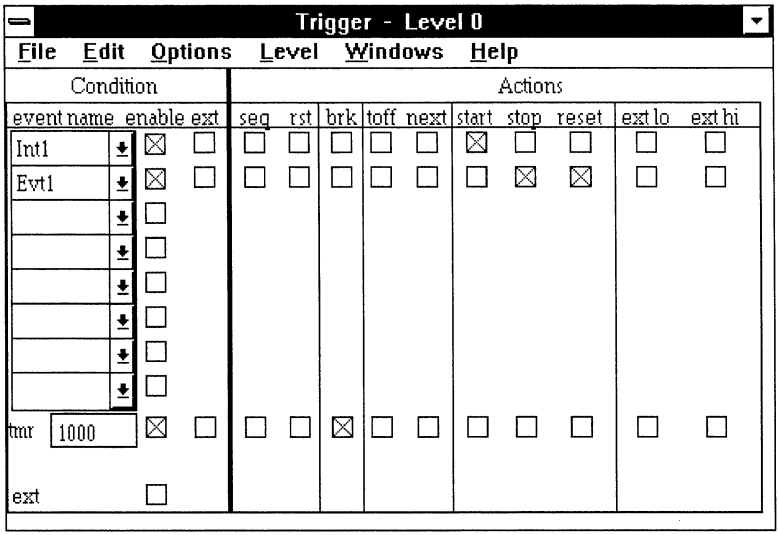


1. Enable the Event names in the eight drop-down list boxes.
2. Specify the actions to be taken when each event occurs:
3. When each event occurs, the associated actions are taken. If multiple events occur simultaneously, all associated actions are taken.

   - Evnt1, Evnt2, Evnt3, and Evnt4 break emulation, reset one of the counters, and write 0 to the external trigger-out signal.
   - Evnt5 and Evnt7 fill the current trace buffer according to Trace Control and start collecting trace into the next trace buffer; increment one of the counters; and write 1 to the external trigger-out signal.
   - Evnt6 and Evnt8 stop tracing, increment one of the counters, and write 1 to the external trigger-out signal.
   - If Evnt5 and Evnt6 together occur 50 times without Evnt1 or Evnt2 occurring, cnt0 reaches 50, breaks emulation, and writes 0 to the external trigger-out signal.
   - If Evnt7 and Evnt8 together occur 100 times without Evnt3 or Evnt4 occurring, cnt1 reaches 100, breaks emulation, and writes 0 to the external trigger-out signal.

Break On Interrupt Latency

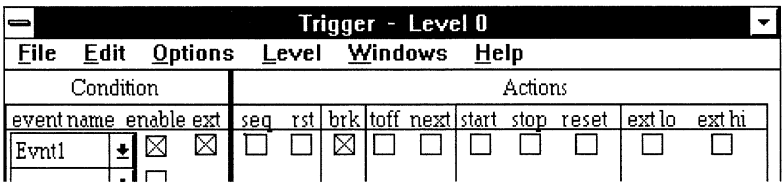
Using the number of elapsed clock cycles, you can discover whether an interrupt is serviced in a timely manner.



1. Define an interrupt event, Int1. Enable Int1 and choose start (starting the timer).
2. Define an event based on the code address of the entrance or exit from the interrupt handler, Evt1. Enable Evt1 and choose rst and stop (resetting and stopping the tmr).
3. Enable tmr and specify 1000 in the tmr edit field. Choose brk.
4. Reduce the timer value until the specified action occurs, to get the actual number of clock cycles between the two events.

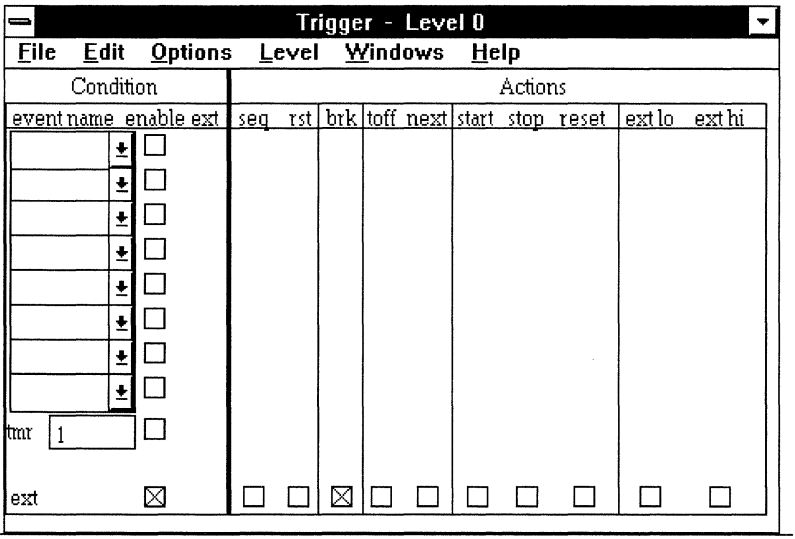
AND an Event With an External Input

Logically AND the condition with an external trigger input low signal by checking the ext box (ext is to the right of enable).



Trigger on External Input Alone

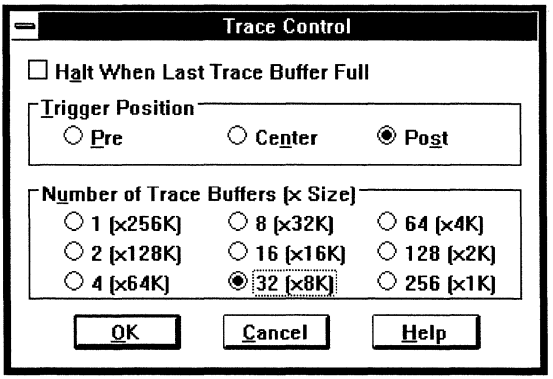
Enable ext on the last line of the Condition pane to set a trigger on an external signal alone (ext is located at the bottom of the left column).



Define Sequential Triggers For Capturing Trace

Capture trace following each of three events in three separate trace buffers. This example uses an Intel386 CX emulator running the demo386.omf sample program installed with SLD.

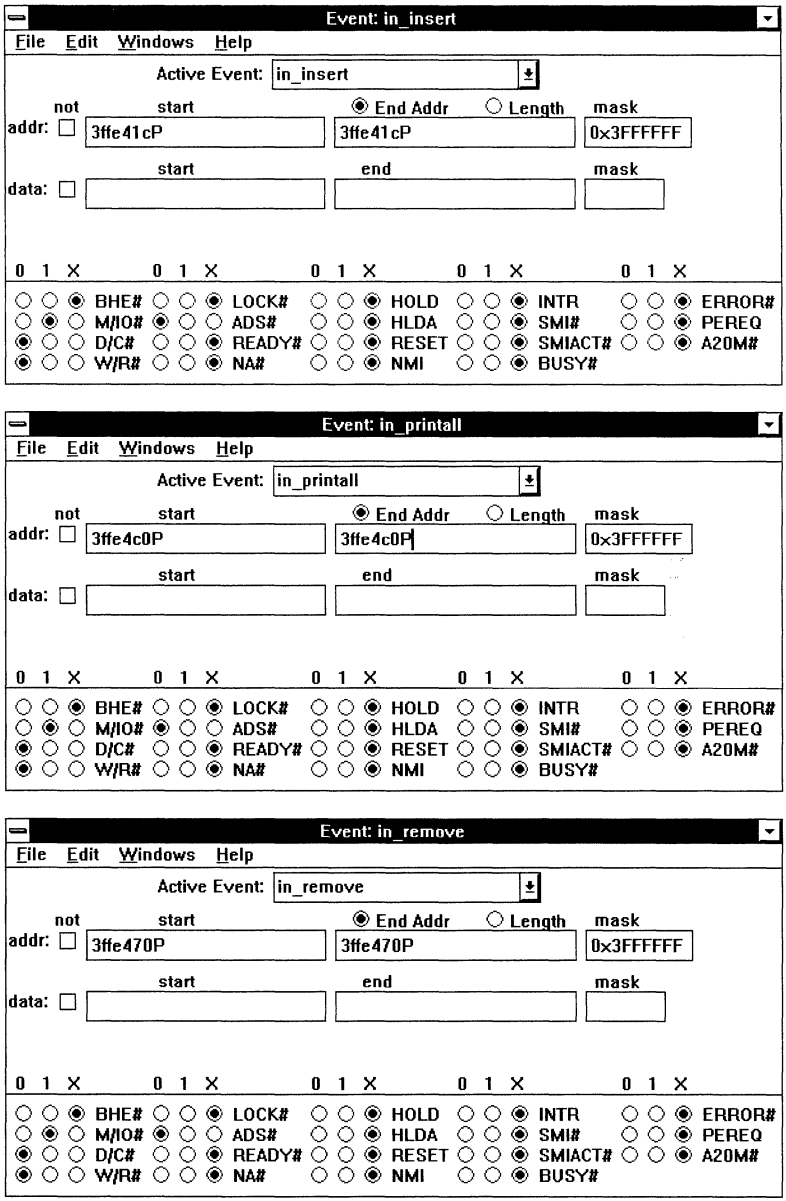
Define buffers 8K bytes long. Position the trigger so the event appears near the beginning of the buffer (Post). The following figure shows the Trace Control dialog box for this buffer configuration.



Define an event at the first code location inside each of three function calls: insert, printall, and remove. To find the addresses, use Xlt:

```
Xlt #insert;
// 0018:FFFFE41C = FFFFE41CL = 3FFE41CP
```

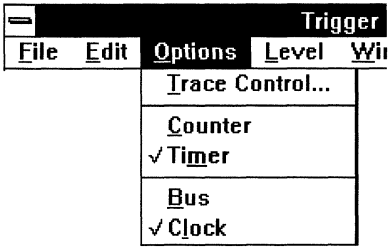
The following figure shows the three event definitions.



Enable the trigger timer and set it to count by clock cycles. The timer lets 8200 clock cycles elapse between triggers. This demo program is so small that the events defined for the triggers occur multiple times in the trace captured to post-fill an 8K-byte trace buffer. Since only one

trace-control action (toff, next) can occur in each buffer, the timer ensures that tracing moves to the next buffer before sequencing to the next trigger.

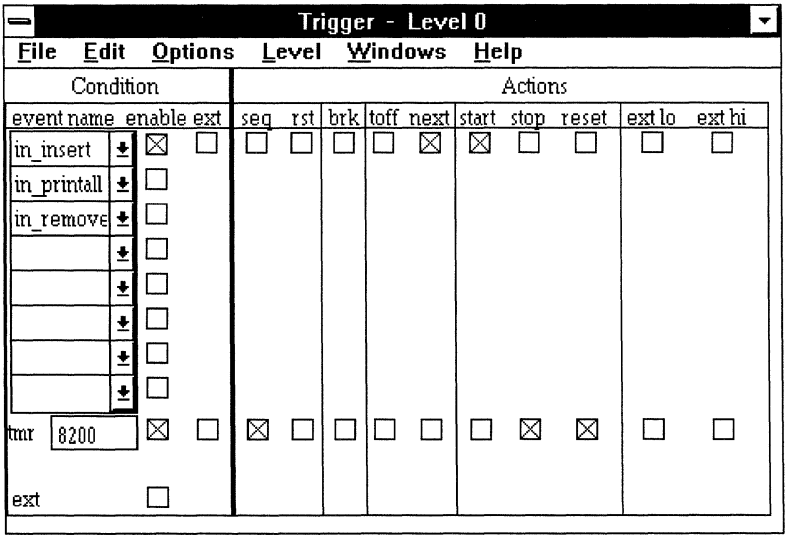
The following figure shows the Options menu with Timer and Clock.

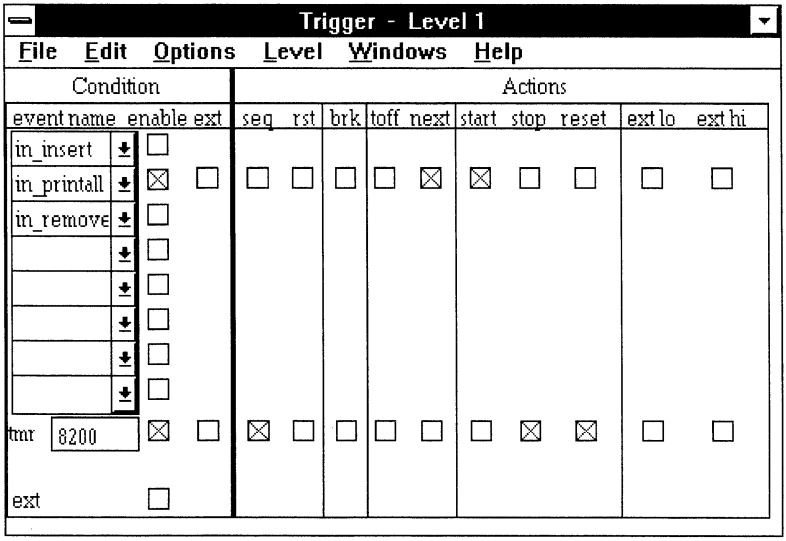


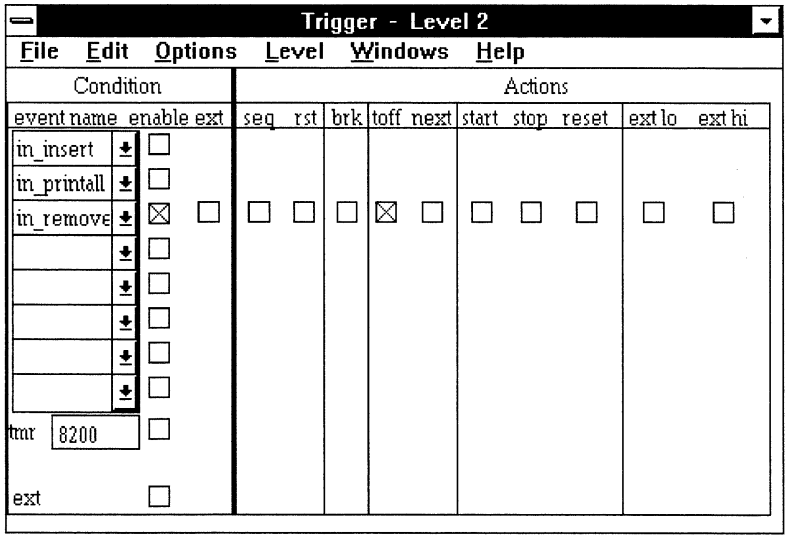
Each of the first two triggers captures trace following its event and starts a timer to run while the buffer fills. When the buffer is full, tracing begins in the next buffer. When the timer finishes, it stops, resets itself, and arms (sequences to) the next trigger.

The final trigger turns trace off, filling the current buffer. Emulation continues but trace does not.

The following figure shows the three levels of triggers.







Summary of Ways to Trigger

The following steps summarize defining a trace buffer using a trigger:

1. In the Trace window, open the Trace menu (or in the Trigger window open the Options menu) and choose Trace Control to configure trace. Set the number of trace buffers in the resulting a dialog box. Set the triggers as pre, post or center and toggle whether to break from emulation when all trace buffers are full.

2. Define the events on which to trigger using the Event window. The Event window contains different choices for different versions of the microprocessor. You can define a bus event based on an address, a data value, or a processor signal. You can include the address space. Defined events can be saved and reloaded.

3. In the Trigger window, open the Options menu and choose the Bus toggle to select bus cycle triggers. The hardware automatically samples processor pins at the proper time in a bus cycle, and triggers based on aligned samples. Or, choose the Clock toggle to enable a trigger when the trigger source is not associated with a bus cycle.

4. In the Trigger window, open the Options menu and choose the Counter toggle to select two 10-bit counters; or choose the Timer toggle to select one 20-bit timer. The counters or the timer can be used to define a trigger. For example, if you are using two counters, you can enter a value for the terminal count (cnt0, cnt1). When the counter reaches the terminal count, the actions you specified for that trigger will be executed.

5. In the Trigger window, set up the triggering hardware to capture the sequence you are interested in by doing the following steps for each event:

   a) On the left side of the Trigger window, enter the name of an event you defined in step 2. If you click on an Event selection, a drop-down list of defined events is displayed. Click on the event you want to trigger on.

   b) Select Enable to display the toggle boxes for the actions to be taken when the Event occurs.

   c) The counters or the timer can be used to define a trigger. For example, if you are using two counters, you can enter a value for the terminal count (cnt0, cnt1). When the counter reaches the terminal count, the actions you specified for that trigger will be executed.

   d) You can specify on the bottom row that the action is taken based on the external signal alone (ext).

e) You can define up to four sets of actions, each set on its own trigger level. You can specify the action of sequencing to the next trigger level. You can specify the action of resetting to trigger level 0.

powerpak.ini File Reference

This chapter describes the contents of the powerpak.ini *file.*

SLD installation creates the powerpak.ini file in your Windows directory. This file contains information used when you invoke SLD and when you open each SLD window.

CAUTION

Always back up powerpak.ini. *Once you have modified* powerpak.ini, *the only way to restore the default contents is to reinstall SLD.*

The following sections can appear in powerpak.ini:

Section	Purpose
[Comm]	Host-to-emulator communication
[CPUInfo]	Intel debug register allocation
[DefaultLayout]	Window screen locations
[InitScript]	Script file to run on invocation
[LoadOptions]	Load options
[Network]	Network information
[Serial]	Host PC COM port number
[SourceInfo]	Source window Go, Step, and View options
[StackInfo]	Stack window options
[StatusInfo]	Status window options
[SystemInfo]	Intel386 CX/SX A-step/B-step support
[ToolBarInfo]	Save settings from the Toolbar
[ToolChain]	Compiler information for Motorola loadfiles
[TraceInfo]	Trace Control and Trigger window options
[TrigInfo]	Trigger window options
[VariableInfo]	HiWare compiler support

The following pages describe the powerpak.ini entries and how to change them. Whenever possible, change entries using menus or Shell commands rather than modifying powerpak.ini in a text editor. Avoid modifying any entry not documented in this chapter.

Many entries are toggle settings with possible values of 1 or 0. For such entries, 1 is enable and 0 is disable.

[Comm]

Describes host/emulator communication

type=[serial | pcnfs | lanserver] describes how the emulator communicates with your host PC. This entry is set to serial by the SLD installation and changed by the network installation. If your network configuration changes in a way that affects communication between the host PC running SLD and the emulator, you must edit powerpak.ini to switch networks or return to serial communication.

serial	specifies serial communication.
pcnfs	defines the emulator as a node on a PC-NFS network.
lanserver	defines the emulator as a node on an OS/2 LAN server.

For example:

```
[Comm]
type=serial
```

[CPUInfo]

Allocates debug register use

dr [<num>]=[user | system] specifies whether the <num> debug register is reserved for use by your program or by the emulator for breakpoints.

<num>	specifies the debug register as 0, 1, 2, or 3.
user	enables access to the debug register for your program.
system	reserves the debug register for use by the emulator, blocking your program's access to the register.

For example:

```
[CPUInfo]
dr 0=system
dr 1=user
dr 2=system
dr 3=system
```

[DefaultLayout]

Specifies Window screen locations

The<PVWindow>Presenter=[<Dimensions>] defines whether each SLD window is displayed when you invoke SLD and the screen locations and sizes for the initially displayed windows.

Move and resize the SLD windows using the Windows mouse or cursor.

Then, to save the layout without exiting SLD, on the Toolbar open the Layout menu and choose Save Layout Now. If you are likely to change the layout again before exiting SLD but want the same initial layout the next time you invoke SLD, be sure Save Layout On Exit (also in the Layout menu) is unchecked.

[InitScript]

Defines which Shell script file executes when you invoke SLD

script=[<scriptFile>] sets <scriptFile> as the filename or pathname of the initialization script (the file of Shell commands run each time you start SLD. Unless you specify a full pathname, SLD looks only in the SLD directory (e.g., c:/powerpak). When no <scriptFile> is specified, none is read.

To change this entry, edit powerpak.ini.

For example, when you install SLD, the initialization script file is include.me:

```
[InitScript]
script=include.me
```

[LoadOptions]

Specifies load options

[LoadOptions] entries can be changed in the Load Options dialog box. To open the Load Options dialog box, from the Toolbar choose Load; or in the Source window, open the File menu and choose Load Code. In the Load dialog box, after browsing the filename to be loaded, choose the Options button. Shell Load command arguments override the [LoadOptions] entries.

AddressSpace=[user | smm] specifies Intel SMM or User address space when the file is loaded. In the Load Options dialog box, choose the User or SMM button.

LoadCode=[1 | 0] specifies whether to load code. For example, when debugging in ROM, turn off code loading and load only symbols. In the Load Options dialog box, toggle Load Code.

LoadSymbol=[1 | 0] specifies whether symbols are loaded. For example, when symbols are already loaded, turn off symbol loading and load only code. In the Load Options dialog box, toggle Load Symbols.

LoadOnDemand=[1 | 0] specifies whether symbolic information is loaded for all modules immediately or not until needed. Symbolic information includes local symbol and line-number information for a

module. Such information is needed when either the module is displayed in the Source window or a breakpoint is set in the module. Advantages of on-demand symbol loading include faster initial loading, faster lookup for the symbols that are demanded, and less memory occupied by the loaded file since only the required symbols are loaded. In the Load Options dialog box, toggle On Demand Symbol Loading.

LoadDemangle=[1 | 0] specifies whether symbols are demangled for the first instance of each overloaded function in a C++ program. In the Load Options dialog box, toggle Demangle C++ Names.

LoadUpdateBase=[1 | 0] specifies whether Intel386 symbol base addresses are updated. For example, if your descriptor table bases are nonzero, you can save time by having the load process update your symbol base addresses from the descriptor table information. In the Load Options dialog box, toggle Update Symbol Bases. This option must be used in conjunction with LoadRegister (in the Load Options dialog box, the Load Initial Registers option).

LoadRegister=[1 | 0] specifies whether Intel386 initial register values are loaded. For example, if your initialization code does nothing but initialize the registers, you can save time by having the load process extract the register information from your initialization code. Then, you need not execute the initialization code. In the Load Options dialog box, toggle Load Initial Register Values.

LoadReportStatus=[1 | 0] specifies whether the load progress indicator appears during loading. In the Load Options dialog box, toggle Report Status.

LoadReportWarnings=[1 | 0] specifies whether warning messages can appear during loading. In the Load Options dialog box, toggle Report Warnings.

For example:

```
[LoadOptions]
// 1=enable, 0 = disable
LoadSymbol=1
LoadCode=1
LoadReportStatus=1
LoadReportWarning=0
LoadOnDemand=0
LoadDemangle=0
LoadAsmModules=0
LoadUpdateBase=0
LoadRegister=0
```

[Network]

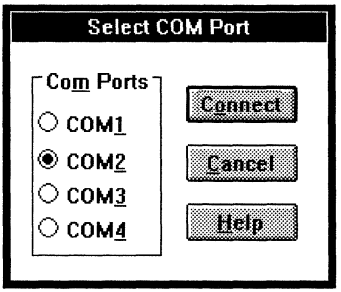
Lists available emulators

emulators=<name>[,<name>...] specifies one or more emulators that SLD can communicate with on the network. When more than one <name> appears in the list, SLD displays a dialog box for you to choose one. Change this entry by editing powerpak.ini directly.

[Serial]

Defines the COM port attached to the PowerPack hardware

comport=com[1 | 2 | 3 | 4] sets the COM port. The first time you start SLD, you must set the COM port number. To use a different COM port, you must edit powerpak.ini. The following figure shows the Select COM Port dialog box that appears when you first start SLD.



For example:

```
[Serial]
comport=com2
```

[SourceInfo]

Controls the Source window display and options

DisplayLineNum=[0 | 1] specifies whether source line numbers are displayed in the Source window. In the Source window, open the View menu; toggle Line Number.

StepCount=<num> specifies how many steps (1 to 0x7FFFFFFF) are executed per Step command. In the Source window, open the Options menu; choose Step Count; fill-in the dialog box. Or, enter a Step or StepSrc Shell command.

ViewSource=[1 | 0] specifies the Source window display either as source from the source file (1) or as a combination of source and disassembly (0). In the Source window, open the View menu and choose Source Only or Mixed Source And Assembly.

UseGoInto=[1 | 0] specifies whether the Call and Return buttons in the Source window perform Go Into (1) or Go Until (0) emulation. In the Source window, open the Options menu, choose Set Go Buttons, and choose Until Call/Return or Into Call/Return.

UseLineExecGranularity=[1 | 0] specifies whether a step executes an entire source line (1) or a single source statement (0). In the Source window, open the Options menu; choose Set Step Granularity; choose Source Line or Source Statement. Or, enter a StepSrc Line or StepSrc Statement Shell command.

HistoryDepth=<num> specifies how many source browsing locations (5 to 100) are saved. In the Source window, open the Options menu, choose Browser History Depth, and fill-in the dialog box.

TabWidth=<num> specifies the number of spaces (1 to 32) that replace a tab character in the Source display. When SLD is installed, powerpak.ini contains TabWidth=8. In the Source window, open the Options menu; choose Tab Width; fill-in the dialog box.

SourceDelimiterUseCRLF=[1 | 0] specifies the source delimiter (the ASCII character string used by the debugger to delimit a source line) as carriage return/linefeed (1), the DOS newline string or as linefeed only (0), the UNIX newline string. When SLD is installed, the delimiter is carriage return/linefeed. In the Source window, open the Options menu; choose Source Line Delimiter; choose Carriage Return/Linefeed or Linefeed Only.

OperandAddressSize=[0 | 1 | 2] specifies the Intel address mode for viewing disassembly in the Source window as:

0 derives the address mode based on the pmode.

1 uses 16-bit address mode.

2 uses 32-bit address mode.

In the Source window, open the View menu; choose Operand/Address Size; choose Auto, Use16, or Use32.

DefaultModuleExtensions=[C, ASM, CPP, CXX, S] specifies the default source file extensions. To change this entry, edit powerpak.ini. When the source filename is stripped of its extension, the emulator searches for the filename with the default module extension.

LoadFile0-3=<pathname> specifies the pathnames of the last four source files you have loaded. This entry is updated automatically when you load a module with associated source.

NumAliasPath=<number> specifies how many directories are listed as source paths. This entry is updated automatically when you add or

delete a source path.

SourcePathAlias<num>=<path> specifies a source path. There are as many of these entries as are counted in NumAliasPath. A SourcePathAlias<num>=<path> entry is added, changed, or deleted each time you add, change, or delete a source path. In the Source window, open the Options menu; choose Source Path. In the Source Path dialog box, to add a new path, choose Add and fill-in the dialog box; to change a path, select the path, choose Edit, and fill-in the dialog box; to delete an existing path, select the path and choose Delete.

For example:

```
[SourceInfo]
DisplayLineNum=1
StepCount=1
ViewSource=1
UseGoInto=0
UseLineExecGranularity=1
HistoryDepth=50
TabWidth=8
SourceDelimiterUseCRLF=1
// 0=auto, 1 = use16, 2 = use32
OperandAddressSize=0
// default source module extensions
DefaultModuleExtensions=C,ASM,CPP,CXX,S
LoadFile0=C:\POWERPAK\SAMP386\DEMO.OMF,9,13
LoadFile1=C:\POWERPAK\SAMP386\DEMO386.OMF,9,13
LoadFile2=
LoadFile3=
NumAliasPath=1
SourcePathAlias0=C:\PV241\SAMP386\
```

[StackInfo]

Controls the display and other options in the Stack window.

StackSize=<num> specifies the stack size and must match the target's allocated stack size. Unless specified in the load file, the stack size defaults to 4K bytes. In the Stack window open the Options menu, choose Stack Area, and fill-in the dialog box; or in the Shell window enter a SetStackArea or SetStackSize command.

StackBaseAddr=<hex_addr> specifies the stack base address, as defined in the load file. In the Stack window open the Options menu, choose Stack Area, and fill-in the dialog box; or in the Shell window enter a SetStackArea or SetStackBase command.

PercentAlarmLimit=<num> specifies the alarm limit as a percentage of the stack size, from 1 to 100. In the Stack window open the Options menu, choose Alarm Limit, and fill-in the dialog box; or in the Shell window enter a SetStackAlarm command.

EnableAlarmLimit=[1 | 0] specifies whether the emulator displays a warning message when stack usage reaches the percentage of the stack area specified by PercentAlarmLimit. In the Stack window open the Options menu and toggle Enable Alarm Limit; or in the Shell window enter EnableAlarmLimit or DisableAlarmLimit.

EnableHWM=[1 | 0] enables or disables the high water mark. In the Stack window open the Options menu and toggle Enable High-Water Mark; or in the Shell window enter EnableHighWaterMark or DisableHighWaterMark.

ViewStackAddr=[1 | 0] enables or disables displaying the Stack window stack address (the location of the frame on the stack). In the Stack window, open the Options menu; toggle Include Stack Address.

ViewCodeAddr=[1 | 0] enables or disables displaying the Stack window code address (the called function's return destination). In the Stack window, open the Options menu; toggle Include Code Address.

For example:

```
[StackInfo]
StackSize=100
StackBaseAddr=0x000D82
PercentAlarmLimit=95
EnableAlarmLimit=1
EnableHWM=1
ViewStackAddr=1
ViewCodeAddr=1
```

[StatusInfo]

Specifies whether the Status window appears on top of other windows

Topmost=[1 | 0] specifies whether the Status window (or icon, when minimized) appears on top of other SLD windows. With Topmost = 1, the Status window or icon cannot be hidden behind any other overlapping SLD window, regardless of which window is in focus. In the Status window, open the Control menu and toggle Always on Top.

For example:

```
[StatusInfo]
Topmost=0
```

[SystemInfo]

Supports Intel386 CX/SX and A-step/B-step emulation

386EmulatorCPU=[386CX A-step | 386CX B-step | none] describes the Intel386 CX/SX bondout processor in the emulator probe head.

386TargetCPU=[386SX | 386CXSA | 386CXSB] describes the Intel386 CX/SX processor in your target design.

The first time you start SLD for Intel386 CX/SX emulation, a dialog box appears wherein you can set 386EmulatorCPU and 386TargetCPU. If you ever need to change these settings, you must either edit powerpak.ini directly or reinstall SLD to see the dialog box again.

386EmulatorCPUs=386CX A-step,386CX B-step lists the Intel386 CX/SX bondout processors recognized by SLD as emulator processors.

386TargetCPUs=386SX,386CXSA,386CXSB lists the Intel386 CX/SX processors recognized by SLD as target processors.

[ToolBarInfo]

Saves the window layout and masks interrupts during single stepping.

SaveLayoutOnExit=[1 | 0] specifies whether the SLD window layout (the SLD windows as you have opened, positioned, and sized them) is saved when you exit SLD. If the layout is not saved, the next SLD invocation reverts to the previously saved or default layout. On the Toolbar, open the Layout menu and toggle Save Layout On Exit.

stepMask=[1 | 0] masks interrupts during single stepping. To toggle interrupt masking, in the Shell window enter a StepMask command.

For example:

```
[ToolBarInfo]
SaveLayoutOnExit=1
stepMask=0
```

[ToolChain]

Specifies which software tools were used to generate the loadfile. (Motorolla processors only)

Compilers=Unknown,Hiware,Intermetrics,Introl,MRI,SDS CrossCode,Sierra,Whitesmiths[,<others>] lists the compilers recognized by SLD. This list can change when you install a new version of SLD. Or, if you are using an unsupported toolchain, you can, instead of specifying an Unknown compiler, edit powerpak.ini to add your compiler and its section names. However, the recommended procedure

for unsupported toolchains is to specify Unknown. (SLD is not guaranteed to work correctly with unsupported toolchains. Adding your compiler name to powerpak.ini does not add support for that compiler.)

<compiler>=<code_section>,<data_section> specifies the default names of the code and data sections in your loadfile. If your loadfile contains section names other than the default sections generated by your compiler, edit powerpak.ini to change this entry. If you add an unsupported compiler to the Compilers entry, add a corresponding section name entry. (SLD is not guaranteed to work correctly with unsupported toolchains. Adding an unsupported compiler's section names to powerpak.ini does not add support for that compiler.)

OMFBaseTypeNames=CODE,DATA specifies the names of the code and data sections in your OMF86 loadfile. If your loadfile contains section names other than the default sections generated by your compiler, edit powerpak.ini to change this entry.

CompilerUsed=[Unknown | Hiware | Intermetrics | Introl | MRI | SDS CrossCode | Sierra | Whitesmiths | <others>] describes the compiler used to generate the loadfile. In the Source window, open the Options menu, choose Compiler Used, and select the appropriate compiler; or enter a CompilerUsed command on the Shell command line. The compiler you specify must be named in the Compilers entry. If you are using an unsupported toolchain, specify Unknown. (SLD is not guaranteed to work correctly with unsupported toolchains.)

If you have not specified the compiler you are using, a dialog box appears the first time you load a file using a button or a menu item. Choose a supported complier in this dialog box.

MergeSections=[1 | 0] specifies whether to merge all your loadfile's code and data sections into two default sections. This can save memory for loadfiles with more than 32 sections. On the Shell command line, enter a MergeSections command.

varIndexCpu16Reg=[none | xk:ix | yk:iy | zk:iz] specifies which Motorola CPU16 register to use for loadfiles with 20-bit addressing.

maxBitFieldSize=[16 | 32] specifies the bitfield size in your OMF386 loadfile. Set this entry to 16 for loadfiles generated with the Borland C compiler and to 32 for other toolchains.

For example:

```
[ToolChain]
MergeSections=0
Compilers=Unknown,Hiware,Intermetrics,Introl,MRI
CompilerUsed=MRI
```

[TraceInfo]

Sets the Trace window options

linkedCursor=[on | off] turns on or off the code address link between the Trace and Source windows. The link is valid only when the Trace window displays instructions (see viewType in this section) and the Source window displays mixed source and disassembly (see viewSource in the [SourceInfo] section).

When cursors are linked, the Source window scrolls automatically to match the Trace display.

To turn linkedCursor on:

1. In the Source window open the View menu; check Mixed Source And Assembly.
2. In the Trace window open the View menu; check Instruction Cycles.
3. In the Trace window re-open the View menu; check Linked Cursor.

To turn linkedCursor off, in the Trace window open the View menu; uncheck Linked Cursor.

viewType=[bus | clock | instruction] sets the trace view as:

clock displays the processor signals at each clock cycle.

bus displays the processor signals at each bus cycle.

instruction displays the instructions executed by the processor (and some prefetched instructions) and the resulting data cycles.

In the Trace window open the View menu; choose Clock, Bus, or Instruction Cycles.

timestamp=[on | off] turns on or off the trace timestamp display. In the Trace window open the View menu; toggle Timestamp.

systemFrequency=<frequency> specifies the target system clock frequency; 0.01 Hz ≤ <frequency> ≤ 40 MHz. In the Trace window open the Timestamp menu; choose Setup, and fill-in the dialog box.

tsmode=[relative | delta] specifies the timestamp mode as:

relative calculates timestamps relative to a specified base frame.

delta calculates each timestamp relative to the previous frame.

In the Trace window, open the Timestamp menu; choose Relative To Frame or Delta.

btmCycles=[enabled | disabled] specifies whether BTM (branch-taken message) cycles are collected and shown. A BTM cycle indicates

a change in execution flow, such as a jump. The emulator must collect BTM cycles to display trace as instructions. In the Trace window, open the View menu; toggle BTM Cycles.

For example:

```
[TraceInfo]
linkedCursor=on
viewType=instruction
timestamp=on
systemFrequency=25MHz
tsmode=relative
btmCycles=enabled
```

[TrigInfo]

Sets the Trace Control and Trigger window options

numTraceBuffers=[1 | 2 | 4 | 16 | 32 | 64 | 128 | 256] specifies the number of trace buffers. Specifying the number of trace buffers also specifies the size of each trace buffer, from one 256K-byte buffer to 256 1K-byte buffers.

In the Trace window open the Trace menu, or in the Trigger window open the Options menu; choose Trace Control; fill-in the Number Of Trace Buffers (X Size) frame of the dialog box.

traceAlignment=[center | pre | post] specifies where relative to the trigger the trace buffers fill: event

center	Trace buffers fill before and after the trigger. The trigger appears in the center of the trace display.
pre	Trace buffers fill up to the trigger. The trigger appears near the end of the display.
post	Trace buffers fill up after the trigger. The trigger appears near the beginning of the display.

In the Trace window open the Trace menu, or in the Trigger window open the Options menu; choose Trace Control; fill-in the Trigger Position frame of the dialog box.

breakOnFull=[on | off] specifies whether the emulator breaks when all trace buffers become full. In the Trace window open the Trace menu, or in the Trigger window open the Options menu; choose Trace Control; in the dialog box toggle the Halt When Last Trace Buffer Full check box..

counterTimer=[counter | timer] specifies whether the two 10-bit counters or the 20-bit timer can be used to specify triggers. In the

Trigger window open the Options menu; choose Counter or Timer.

trigMode=[bus | clock] specifies the type of cycle used for triggering:

bus	automatically samples processor pins at the proper time in a bus cycle. The trigger is based on aligned samples.
clock	triggers on any cycle coming from the processor, regardless of whether it is a valid bus cycle. Use clock triggering to trigger on an I/O signal or on an interrupt input that can occur on any clock cycle.

In the Trigger window, open the Options menu; choose Bus or Clock.

For example:

```
[TrigInfo]
numTraceBuffers=1
traceAlignment=pre
breakOnFull=off
counterTimer=counter
trigMode=bus
```

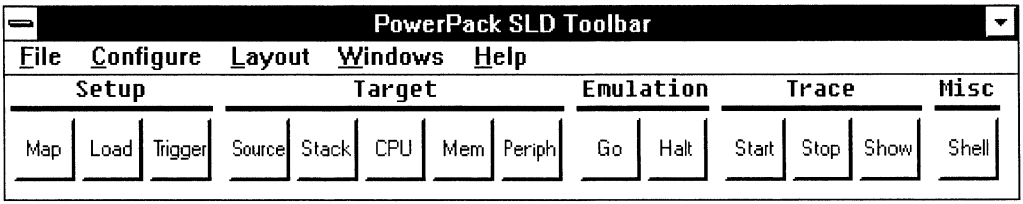
[VariableInfo]

Supports HiWare bitfield types

AutoCalcBitfieldOffsets=[1 | 0] specifies whether to calculate bitfield offsets as generated by the HiWare compiler. Set this entry to **1** for loadfiles compiled with HiWare and to **0** for other toolchains.

Toolbar Reference

The following figure shows the Toolbar.



This chapter describes the toolbar menus, buttons, and dialog boxes.

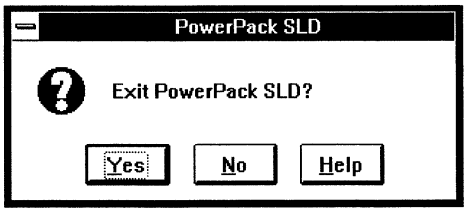
The Toolbar is the first window opened when you start SLD and is always available during your debugging session. Closing the Toolbar exits SLD, ending your emulator session. Minimizing the Toolbar hides all other SLD windows and icons.

Toolbar Menus

Menu	Use To:
File	Exit SLD.
Configure	Configure and initialize the debugging environment.
Layout	Save your screen layout of SLD windows.
Windows	Select a closed or iconized SLD window to open.
Help	Open a window for help with SLD.

File Menu

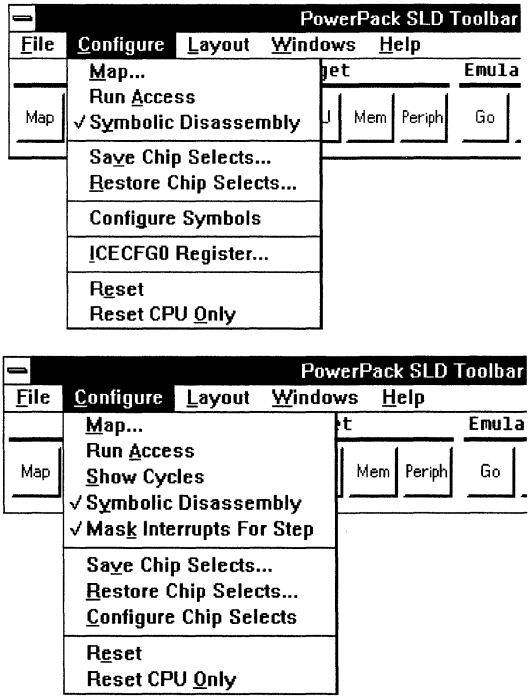
You can exit SLD as you would exit any Windows application; or you can open the File menu and choose Exit. The emulator asks you to confirm exiting. The following figure shows an Exit dialog box.



In any SLD window other than the Toolbar, choosing Exit closes only that window. Exit is on every SLD window File menu except in the CPU window, where Exit is on the Options menu.

Configure Menu

The following figure shows two sample Configure menus. The first is for the Intel386 EX processor; the second is for the Motorola 68332 processor. Different menu items are available for different processors.



Map... opens the Map dialog box for examining and modifying your memory map. Choosing this menu item has the same effect as choosing the Map button. The Map dialog box is described in the "Map Dialog Boxes" section later in this chapter. You can also configure memory with Map and RestoreMap Shell commands.

Run Access, when checked, enables memory access during emulation. Memory access is used to scroll and refresh the Peripheral and Memory windows and to read or write peripheral registers and memory. Because such memory accesses take a small amount of processor time, doing these operations during emulation can degrade your program performance.

When you start SLD, run access is disabled (unchecked) and memory access is available only when emulation is halted.

Run access does not affect the access of CPU registers. The CPU registers are inaccessible during emulation.

You can also enable and disable run access with the RunAccess Shell command.

Show Cycles, when checked, makes the Motorola processor internal cycles visible for tracing.

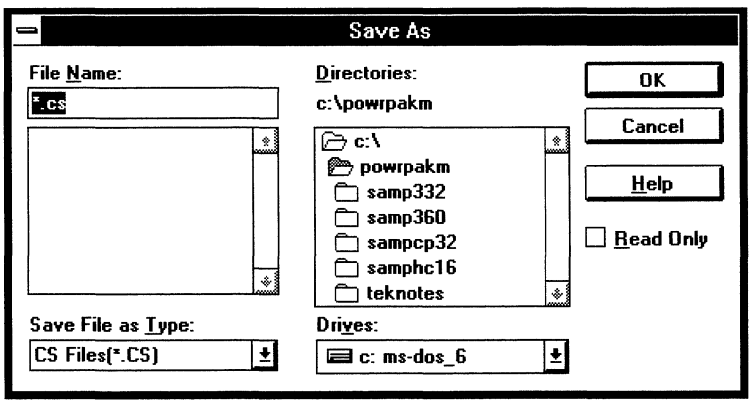
Symbolic Disassembly, when checked, uses symbolic addresses in the disassembly displayed in the Source and Memory windows.

Mask Interrupts For Step, when checked, prevents interrupts from pre-empting a Step operation in a Motorola emulator. You can also enable and disable interrupt masking with the StepMask Shell command.

Save Chip Selects... records the chip-select registers in an ASCII file. The registers can be restored from the file using the Restore Chip Selects command.

You can also save the chip select registers with the SaveCS Shell command. For a list of which registers are saved for each processor, see the SaveCS description in the "Shell Window Reference" chapter.

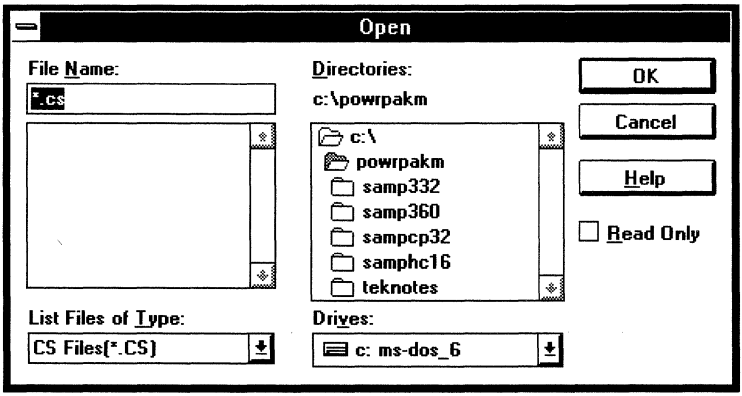
The following figure shows a sample Save As dialog box for saving chip select information to a chip select (*.cs) file.



Restore Chip Selects... restores the chip-select registers to the values specified in an ASCII file. You can create this file with the Save Chip Selects item, with a SaveCS Shell command, or with a text editor such as Windows Notepad.

You can also restore the chip select registers with the RestoreCS Shell command (or, for Motorola targets, restore the target chip selects and configure the emulator chip selects at the same time with a single ConfigCS command). For a list of which registers are saved for each processor, see the SaveCS description in the "Shell Window Reference" chapter.

The following figure shows a sample Open dialog box for restoring chip select values from a saved chip select (*.cs) file.

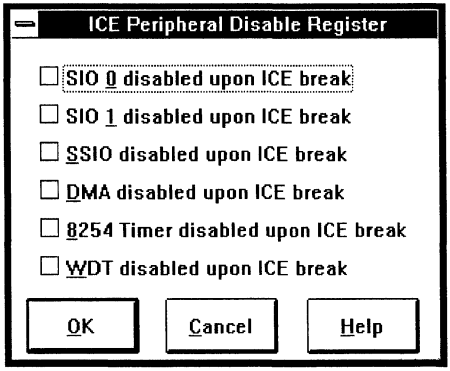


Configure Chip Selects configures the emulator hardware to match the chip select values in the Motorola target processor.

You can also configure the emulator chip selects with the ConfigCS Shell command.

Configure Symbols updates the loaded symbols with the base address from the Intel processor descriptor table (GDT or LDT). Your program must provide the GDTR and LDTR values and GDT and LDT contents.

ICECFG0 Register... opens the ICE Peripheral Disable Register dialog box for setting bits in the Intel386 EX processor ICECFG0 register. To enable or disable specific peripherals on ICE break, check or uncheck each option. The following figure shows the ICE Peripheral Disable Register dialog box with all peripherals disabled on ICE break.



Reset resets and reinitializes the target processor:

- The processor reset pin is asserted.
- The program counter is read from memory; the Source window is scrolled to the beginning of code.
- The stack pointer is read from memory, resetting the stack; the Stack window display becomes invalid.
- All SLD windows are updated.

You can also reset the processor with the Source window Run menu Reset item, the CPU window Options menu Reset item, or the Reset Shell command.

Reset CPU Only resets only the processor and does not update the windows. Use Reset CPU Only if Reset fails to reset the processor.

You can also reset only the the processor with the CPU window Options menu Reset CPU Only item or the Reset Shell command.

Layout Menu

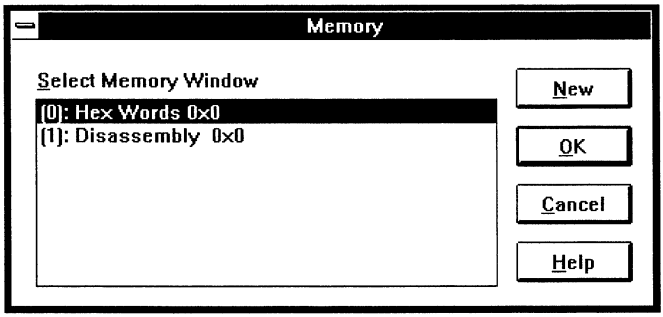
Save Settings Now saves the current coordinates of the SLD windows and icons.

Save Settings On Exit saves the coordinates of the SLD windows and icons when you exit from SLD.

Toolbar Buttons

Button	Use To:
Map	Open the Map dialog box (described later in this chapter) to examine or change the memory configuration. This button has the same effect as the Configure menu Map item. You can also configure memory with the Map and RestoreMap Shell commands.
Load	Open the Load dialog box (described later in this chapter) to load code and/or symbols. You can also load code and symbols with the Load Shell command or the Source window File menu Load Code item.
Trigger	Open the Trigger window to define triggers and events for controlling emulation and trace collection. This button has the same effect as the Windows menu Trigger item.
Source	Open the Source window to examine source and disassembly, control emulation with breakpoints and stepping, and find source corresponding to trace displayed in the Trace window. This button has the same effect as the Windows menu Source item.
Stack	Open the Stack window to view the current nested calls, associated parameters and variables, and stack usage statistics. This button has the same effect as the Windows menu Stack item. You can also examine the stack with the StackInfo and StackArea Shell commands, or modify the stack with the StackArea, StackBase, and StackSize Shell commands.
CPU	Open the CPU window to view and change processor registers. This button has the same effect as the Windows menu CPU item. You can also display and edit the CPU registers with the Register Shell command.

Mem — Open or change focus to one of up to 20 Memory windows to view and change memory. This button has the same effect as the Windows menu Memory item. If more than one Memory window (including minimized windows) is open, a dialog box appears in which you can choose an existing Memory window or open a new one. The following figure shows a sample Memory dialog box.



You can also view and change memory with the Dump, Write, Fill, Search, and Copy Shell commands.

Periph — Open the Peripheral window to view and change peripheral register values. This button has the same effect as the Windows menu Peripheral item.

Go — Start emulation from the current program counter, subject to control by previously defined breakpoints and triggers. This button has the same effect as pressing the <F9> key. You can also start emulation with the Source window buttons and Run menu items and with various Shell commands.

Halt — Stop emulation. This button has the same effect as pressing the <F2> key. You can also stop emulation with the Source window buttons and Run menu Halt item and with various Shell commands.

Start — Begin collecting trace. Tracing starts automatically when emulation starts. You can start and stop trace collection during emulation without affecting emulation. You can also start trace with the Trace window Trace menu Start item.

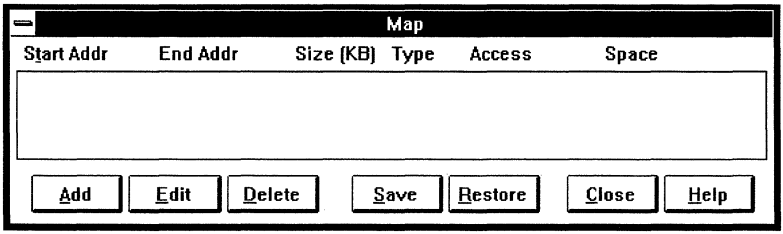
Stop — Stop collecting trace. You can also stop trace with the Trace window Trace menu Stop item.

Show — Open the Trace window to display collected trace. You can examine trace during emulation. This button has the same effect as the Windows menu Trace item.

Shell — Open the Shell window for command-line entry. This button has the same effect as the Windows menu Shell item.

Map Dialog Boxes

The following figure shows a Map dialog box with no memory mapped. When memory has been mapped, the configuration of each mapped region is listed in the central panel. To select a listed region, click on it or use the <Up Arrow> and <Down Arrow> keys to move the highlight.

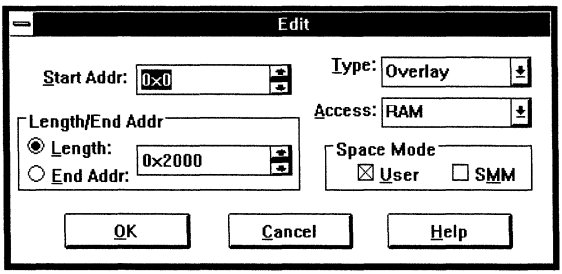


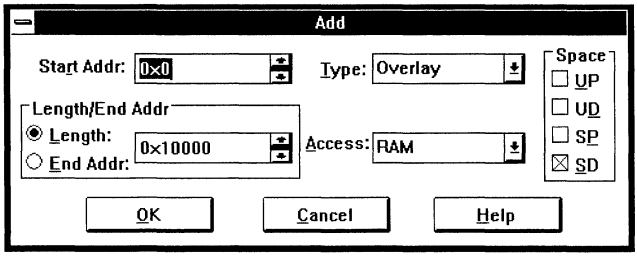
Map Dialog Box Buttons

Button **Use To:**

Add — Open a dialog box to configure unmapped memory.

The following figure shows two sample Map Add/Edit dialog boxes. The first is an Edit box for the Intel386 EX processor; the second is an Add box for the Motorola 68332 processor. The Space choices depend on whether you have an Intel or a Motorola processor. Valid Start Addr and Length/End Addr values also depend on which processor and on how much memory you have configured.



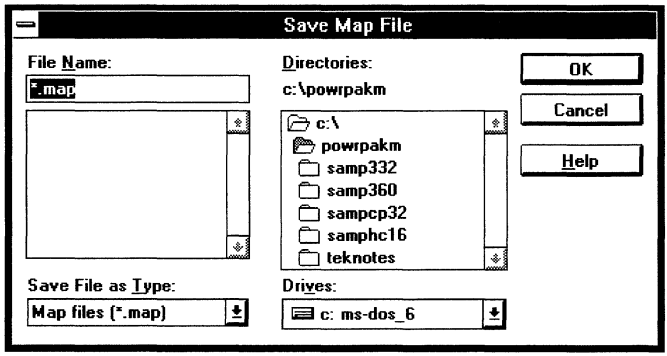


For more information on the Start Addr, Length/End Addr, and Access field values, see the list of Map dialog box field contents below.

Edit — Open a dialog box (see the Add button description above) to reconfigure a mapped region. This button is available when a listed region is selected.

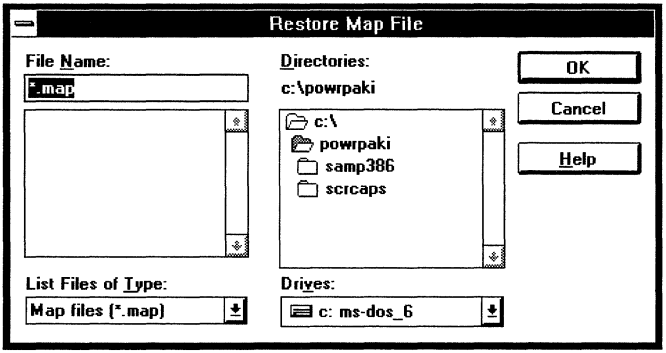
Delete — Revert a mapped region to unmapped memory. This button is available when a listed region is selected.

Save — Open a dialog box to save the listed configuration to a map (*.map) file. The following figure shows a sample Save Map File dialog box.



You can also use the SaveMap Shell command to save the map configuration.

Restore — Open a dialog box (see the Save button description above) to configure regions from a previously saved map (*.map) file. The following figure shows a sample Restore Map File dialog box.



You can also use the RestoreMap Shell command to restore a previously saved map configuration.

Close — Close the Map dialog box.

Help — Open a window for help on mapping.

You can also use the Map Shell command to examine your memory map and for the same effect as the Add, Edit, and Delete buttons.

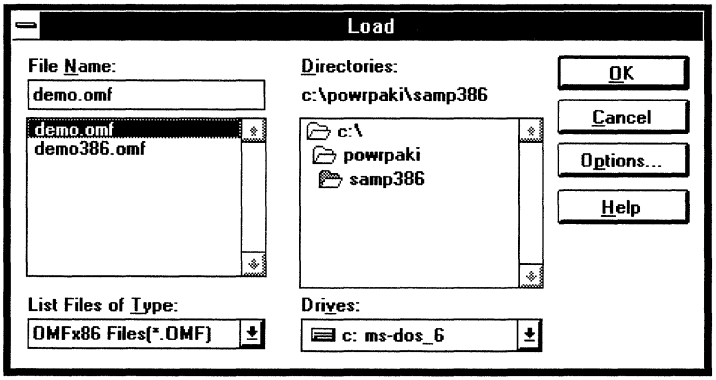
Map Dialog Box Field Values

Field	Contents
Start Addr	Where the region begins: For Intel emulators, the region must start on a 4K boundary. For Motorola, the starting address must match the region size. The emulator automatically configures memory into two regions, depending on whether you have 256K or 1M bytes of overlay memory. For 256K bytes: • 64K-byte region must start on 64K boundary. • 128K-byte region must start on 128K boundary. For Motorola with 1M bytes of overlay memory: • 64K-byte region must start on 64K boundary. • 128K-byte region must start on 128K boundary. • 256K-byte region must start on 256K boundary. • 512K-byte region must start on 512K boundary.
End Addr	Where the region ends.

Size	For Intel, 4K, 8K, 12K, 16K, etc. bytes. For Motorola with 256K bytes of overlay memory, 64K or 128K bytes. For Motorola with 1M bytes of overlay memory, 64K, 128K, 256K, or 512K bytes of memory. Specify a region size instead of an end address by choosing the Length rather than the End Addr button in the Map Add/Edit dialog box, then filling-in an appropriate value in the Length/End Addr field.
Type	Overlay or Target.
Access Rights	RAM allows read and write access. ROM BREAK allows read access; prevents write access; breaks on attempted write access. (For Intel emulators, with Target memory, write access is allowed but causes emulation to break.) ROM NOBREAK allows read access; prevents write access; does not break on attempted write access. (For Intel emulators, with Target memory, write access is allowed.) NONE prevents any access; breaks on attempted access. (For Intel emulators, with Target memory, read and write accesses are allowed but cause emulation to break.)
Space	For Intel, User or SMM (system management mode) For Motorola, UP (user program), UD (user data), SP (supervisor program), or SD (supervisor data)

Load Dialog Boxes

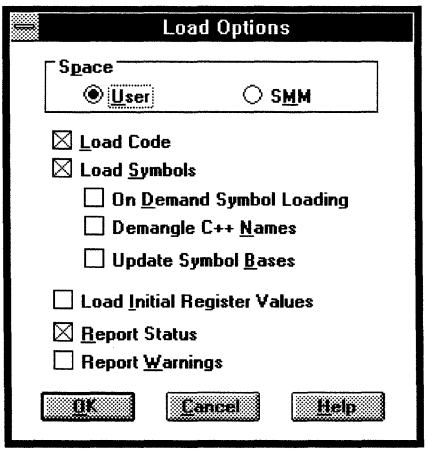
To open a dialog box for loading code and symbols, choose the Toolbar Load button. The following figure shows a sample Load dialog box.

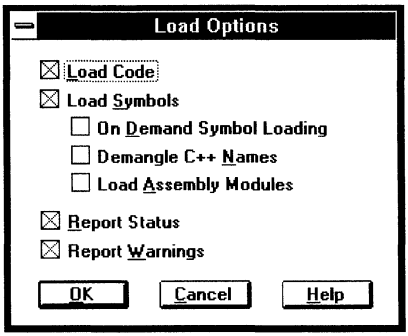


When you select a loadfile, the Options button in the Load dialog box becomes available. Choosing this button opens the Load Options dialog box for specifying how to load code and/or symbols from the loadfile.

When you are ready to load, choose the OK button. To exit the Load dialog box without loading, choose the Cancel button. To open a window with help on loading, choose the Help button.

The following figure shows two sample Load Options dialog boxes. The first is for the Intel386 EX processor; the second is for the Motorola 68332 processor. Different options are available for different processors.





For Intel loadfiles, be sure the space option (User or SMM) you select is compatible with the address space configured in the Map dialog box.

To enable an option, check the box beside the option. To disable an option, uncheck the corresponding box. The options are:

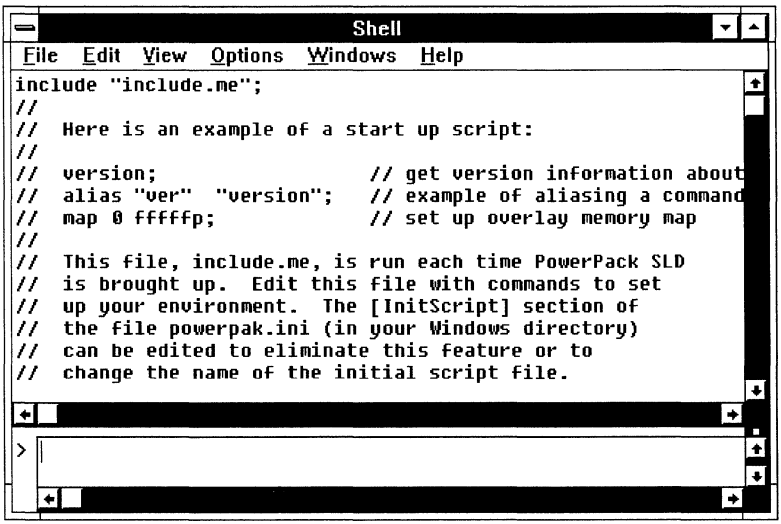
Option	Effect
Load Code	loads executable code sections from your loadfile.
Load Symbols	loads data sections and relevant symbolic information from your loadfile. When this option is enabled, several sub-options are available.
On Demand Symbol Loading	waits to load symbolic information for each module until it is needed, for example when you display the module in the Source window.
Demangle C++ Names	uses an MRI algorithm to demangle some C++ symbols, for example overloaded function names.
Update Symbol Bases	reads base addresses for symbol tables, once the Intel386 registers are initialized.
Load Assembly Modules	loads symbolic information for modules whose source files are assembly language.
Load Initial Register Values	initializes Intel386 EX processor registers from loadfile information.
Report Status	displays an information box showing the load operation progress.
Report Warnings	displays information boxes with non-fatal anomolies encountered during loading.

You can load a file during emulation. Be sure the file's load addresses do not overlap the memory occupied by the running program. Loading a file at a location in use stops the emulator in an unpredictable state.

You can specify equivalent load options with the Load Shell command.

Shell Window Reference

The following figure shows a sample Shell window.

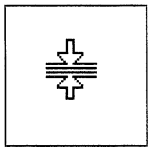


This chapter describes the the Shell window contents, menus, dialog boxes, and commands; and how to execute commands in the Shell window.

The Shell window contains two panes:

Transcript — in the top part of the window, echoes commands and command output.

Command Entry — in the bottom part of the window, is where you enter commands.



You can change the relative sizes of the Shell window panes. A split box between the vertical scroll bars defines the edge between the Transcript and Command Entry panes. When the mouse is pointing to the split box, SLD displays a split-box cursor (see figure at left). Then you can drag the split box to resize the panes as you wish.

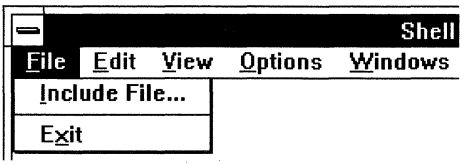
To change focus from one pane to the other, click in the inactive pane or press the <Tab> key.

Shell Window Menus

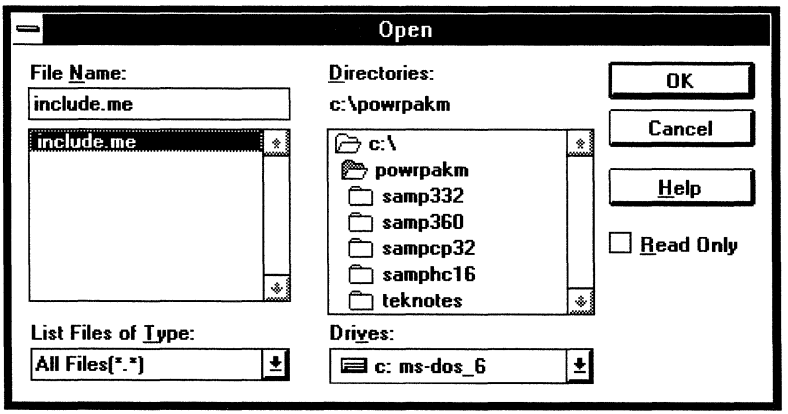
Menu	Use To:
File	Run a script; close the Shell window.
Edit	Cut and paste text in the Command Entry pane and copy text from the Transcript pane, using Windows Clipboard.
View	Display commands and/or output in the Transcript pane.
Options	Manage a log file and the command history buffer.

File Menu

The following figure shows a File menu.



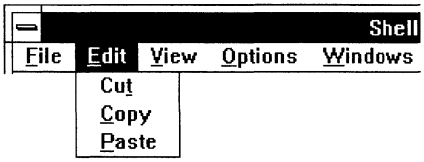
Include File... opens a dialog box wherein you can select a script (a text file containing Shell commands) to be run immediately. The following figure shows the Include dialog box with the include.me sample script (provided with SLD) selected.



Exit closes the Shell window without exiting SLD.

Edit Menu

The following figure shows an Edit menu.



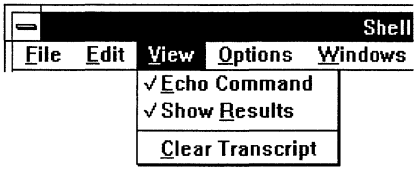
Cut moves highlighted strings from the Command Entry pane to the Windows Clipboard, deleting the strings from the Command Entry pane.

Copy copies highlighted strings from the Command Entry or Transcript pane to the Windows Clipboard, leaving the original strings unaffected.

Paste copies strings from the Clipboard to the Command Entry pane.

View Menu

The following figure shows a View menu.



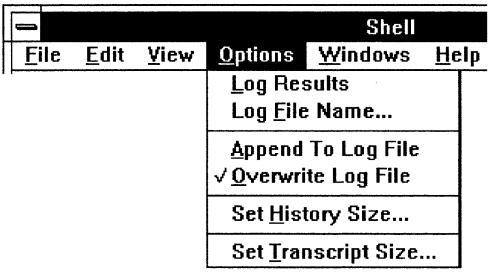
Echo Command, when checked, displays in the Transcript pane all text you enter in the Command Entry pane.

Show Results, when checked, displays in the Transcript pane the results of any text you enter in the Command Entry pane.

Clear Transcript blanks the Transcript pane.

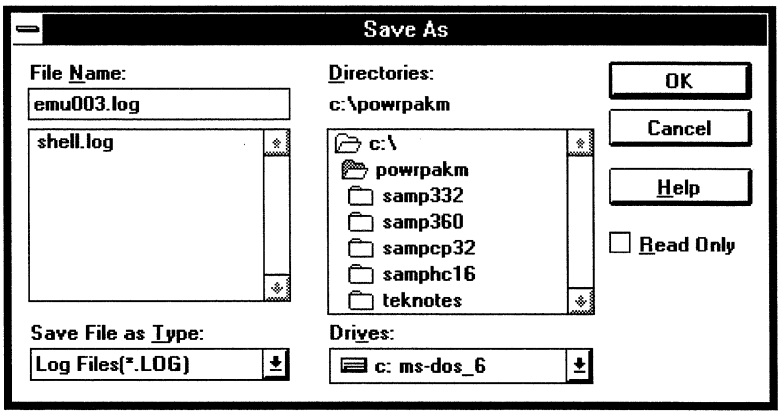
Options Menu

The following figure shows an Options menu.



Log Results starts recording into a text file all that appears in the Transcript pane. If you have not previously specified a log filename, the emulator uses shell.log in your SLD directory (e.g. c:\powerpak).

Log File Name... opens a dialog box for specifying the log file path and name. The following figure shows a sample Log Filename dialog box, creating a file named emu003.log.

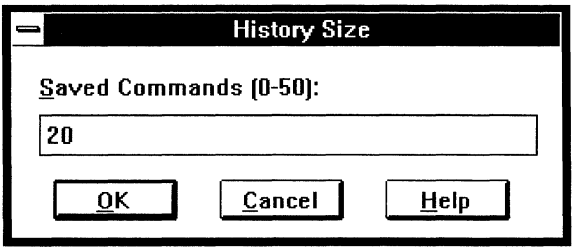


Append To Log File, when checked, ensures that text recorded into an existing file is added to the end of the file and does not destroy any prior contents of the file.

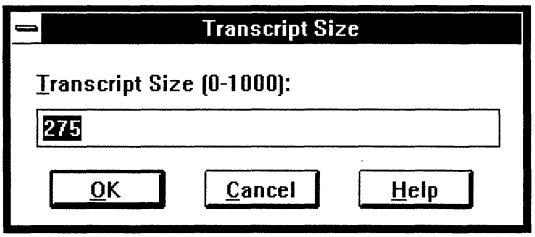
Overwrite Log File, when checked, ensures that text recorded into an existing file is written starting at the beginning of the file, destroying any prior contents of the file.

Set History Size... opens a dialog box to specify the maximum number of commands to be retained in the history buffer. Use the <Up Arrow> and <Down Arrow> keys to recall previously entered text from the

history buffer into the Command Entry pane. The following figure shows a sample History Size dialog box.



Set Transcript Size... opens a dialog box to specify the maximum number of lines to be retained in the scrollable Transcript pane. The following figure shows a sample Transcript Size dialog box.



Entering Commands in the Shell Window

Enter commands in the Shell window by one of:

- Type one command. Press <Enter> to execute it.
- Type a sequence of commands. Follow each command with a semicolon (;). Press <Ctrl><Enter> to start a new line without executing the already typed commands. Press <Enter> to execute the sequence of commands.
- Execute a script, that is, a file containing multiple commands separated by semicolons. For example, you can create a script by logging a series of commands and editing the log file with a text editor. To execute a script at any time during an emulator session, use the Include command (described later in this chapter). In the powerpak.ini file, you can specify a script to be executed automatically when you invoke SLD. The default script specified in powerpak.ini is include.me.
- Recall a previously entered command from the history buffer by entering <Ctrl><Up Arrow> or <Ctrl><Down Arrow> to scroll

through saved commands, edit the command as needed, then press <Enter> to execute the command. To specify the number of commands to be saved, open the Options menu, choose Set History Size, and fill-in the dialog box.

To cancel a command line without executing it, press <Esc> instead of <Enter>. To interrupt command execution, press <Esc>.

Enter addresses as hexadecimal values. Enter data values in either decimal or hexadecimal radix, with the 0x prefix to indicate any hexadecimal value. For example:

```
Reg PC 55;                    // Set register PC to 55 decimal.
Dump 400;          // Dump memory at address 400 hexadecimal.
Write 10:50 0x33;          // Write 33 hexadecimal to segment 10
                           // hexadecimal, offset 50 hexadecimal.
```

Shell Window Commands

Notational Conventions

The following notational conventions are used in the following pages:

Notation	Meaning
COMMANDNAME commandname CommandName	Case is not significant in command names and aliases.
<placeholder>	Indicates an argument. Substitute a value or a symbol for the place holder.
[option]	Brackets delimit an item that can be repeated no more than once. The brackets are not to be entered as part of the command, unless otherwise noted.
{<many_values>}	Braces delimit an item that can be repeated zero or more times. The braces are not to be entered as part of the command, unless otherwise noted.
<series>...	Ellipsis indicate a series of repeating items.
option_1 | option_2	A vertical line separates options, one of which can be selected.
(option_1 | option_2)	Parentheses around an options list indicates that one of the options must be selected. Do not enter the parentheses.
"<string_constant>"	String constants must be surrounded by double quotation marks.
/* comment */	Comments are delimited C-style.
//command output	Command output is preceded by forward slashes.
<address>	A linear, physical, virtual, or symbolic address, as described in the Address Formats chapter.

Commands and System Variables Grouped by Functionality

The following table groups the commands and system variables by functionality:

To Do	For Processor	Use
Address translation	Intel	Xlt
Assembly/disassembly	Any	Asm
	Any	AsmAddr
	Any	Dasm
	Any	DasmSym
Breakpoints	Any	Bkpt
	Any	BkptClear
	Any	DR
Bus	Any	BusRetry
Compiler setup	Motorola	CompilerUsed
	Intel386	MaxBitFieldSize
	Motorola CPU16	VarIndexCPU16Reg
Chip Select setup	Motorola	ConfigCS
	Intel386 EX; Motorola	RestoreCS
	Intel386 EX; Motorola	SaveCS
Emulation	Any	Go
	Any	GoInto
	Any	GoUntil
	Any	Halt
	Any	ResetAndGo
	Any	Step
	Motorola	StepMask
	Any	StepSrc
Events	Any	EventRestore
	Any	EventSave

Help	Any	Help
Load Code	Motorola	BDMspeed
	Any	Load
	Any	LoadSize
	Motorola	MergeSections
	Any	ResetLoaders
Map memory	Any	Map
	Motorola 68360	MapRanges
	Any	RestoreMap
	Any	SaveMap
Memory	Any	Copy
	Any	Dump
	Any	Fill
	Any	RunAccess
	Any	Search
	Any	Size
	Any	Verify
	Any	Write
Register	Intel386 EX	Config
	Any	Register
Reset	Any	Reset
	Any	ResetAndGo
Shell	Any	Alias
	Any	Append
	Any	Clear
	Any	Delete
	Any	Echo
	Any	Exit
	Any	History
	Any	If
	Any	Include
	Any	Integer

	Any	List
	Any	Log
	Any	Logging
	Any	Overwrite
	Any	Print
	Any	String
	Any	Results
	Any	Transcript
	Any	While
Stack	Any	DisableAlarmLimit
	Any	DisableHighWaterMark
	Any	DisplayStack
	Any	EnableAlarmLimit
	Any	EnableHighWaterMark
	Any	FillStackPattern
	Any	SetStackAlarm
	Any	SetStackArea
	Any	SetStackBase
	Any	SetStackSize
	Any	StackInfo
Status	Any	$BREAKCAUSE
	Any	$EMULATING
	Any	$SHELL_STATUS
	Any	Cause
	Any	EmuStatus
	Any	IsEmuHalted
	Any	Signal
	Any	Time
	Any	Version
Symbols	Any	AddressOf
	Any	ConfigSymbols
	Any	DisplaySymbols

	Intel	DT
	Intel	GDT
	Any	GetBase
	Intel	IDT
	Intel	LDT
	Any	NameOf
	Intel	PMode
	Any	RemoveSymbols
	Any	SetBase
	Any	SymbolCloseFile
	Any	SymbolOpenFile
	Intel	TSS
Test Hardware	Any	RAMtst
	Any	Test
Timing	Any	LapTimer
	Any	StartTimer
	Any	StopTimer
Trace	Motorola 68360	AuxTrace

$BREAKCAUSE

System Variable: Discovers what caused emulation to break.

Related topics: $EMULATING, Cause, Go, GoInto, GoUntil, Halt, ResetAndGo, Step, StepSrc

$BREAKCAUSE

Case is significant. Enter this variable in upper case.

Knowing what caused emulation to break can be useful, for example, to abort script execution because of a certain reason for the break.

$BREAKCAUSE is updated when emulation breaks. Its value indicates the cause of the break:

0	No cause (for example, emulation not yet started)
1	Target processor was reset
2	Emulator was halted
4	Processor single step
5	Execution breakpoint reached
7	Processor received a double bus fault
8	External break request
9	Unknown cause

```
/* Following is part of an include file that aborts execution only
when an execution breakpoint occurs. $Z is an undeclared Shell
variable that will halt the script. */

go;
while ($EMULATING) {;};     /* loop until emulator halts */
if ($BREAKCAUSE==5) {$Z;}; /* test for execution breakpoint */
```

$EMULATING

System Variable: Discovers whether the emulator is running.

Related topics: $BREAKCAUSE, Cause, Go, GoInto, GoUntil, Halt, ResetAndGo, Step, StepSrc

$EMULATING

Case is significant. Enter this variable in upper case.

Knowing whether the emulator is running can be useful, for example, to control script execution flow based on emulation status. $EMULATING has the value:

1	The emulator is running.
0	The emulator is halted.

```
bkpt #main;                          /* stop after registers initialized */
ResetAndGo;                          /* start from the power-on level */
while ($EMULATING) {;};                  /* loop until emulator halts */
```

$SHELL_STATUS

System Variable: Discovers whether the last shell command completed successfully.

$SHELL_STATUS

Case is significant. Enter this variable in upper case.

Knowing whether a Shell command completed successfully can be useful, for example, if you want to control the execution flow of a script based on whether earlier commands executed as expected. $SHELL_STATUS has the value:

0	The command completed normally.
nonzero	An error occurred.

```
bkpt #main;                              /* stop after registers initialized */
Reset;          /* try to reset processor and update SLD windows */
If ($SHELL_STATUS) {
  Print "Didn't Reset";
  Reset CPUonly};        /* Reset without updating SLD windows */
```

AddressOf

Returns the numeric address of a module, function, line, or variable.

Related topics: DisplaySymbols, GetBase, NameOf, RemoveSymbols, SetBase

AddressOf <address>

<address> is a partly or fully qualified symbol name.

The associated numeric address is returned.

You cannot use AddressOf to obtain the address of a local variable, because the local variable has no fixed location. Instead, use DisplaySymbols to find the stack offset of a local variable.

```
addressof #Blank_TxBuf;                  // address range of a function
// 6A6..6BF

addressof #MsgRx;                  // address range of an array variable
// E68..E87 [32]
```

For function names, you can obtain the same information in the Source window by double-clicking on the function name to display the Function pop-up menu, then choosing Show Load Address.

Alias

Define or list an alias.

Alias ["<name>" ["<value>"]]

<name> is the alias. The quotation marks are required.

<value> assigns a value to the specified name. The quotation marks are required. Inside <value>, replace double quotation marks with single quotation marks.

Entering alias with no parameters lists all currently defined aliases. Entering alias "<name>" displays the value of <name>.

Use alias to shorten or change commonly used command strings.

```
alias "s1" "include 's1.inc'";

Alias "increment" "$a = $a + 1; $a;"
$a = 0;
increment;
// 0x1 1
increment;
// 0x2 2
```

Append

Appends to log file.

Related topics: Log, Logging, Overwrite, Echo, Results

Append

When Append has been specified, opening a log file adds text to the end of the file, preserving the file's prior contents.

You can also configure logging to append to a file by opening the Shell window Options menu and choosing Append To Log File.

Asm

Write assembly to memory.

Related topics: AsmAddr, Dasm, DasmSym

Asm <string>

<string> is an assembly language statement.

Check the syntax of <string> and write the instruction bytes to memory at the current assembly address. (Determine the current assembly address with AsmAddr.)

Symbolic assembly is not supported.

```
Asm nop;
// 000000 4E71          nop
```

```
// Number of bytes: 2
```

You can also assemble new instructions and data into memory with the single-line assembler. In the Memory window, display memory as instructions. Double-click on a line to open the single-line assembler dialog box.

AsmAddr

Set the address where the Asm command will write.

Related topics: Asm, Dasm, DasmSym

AsmAddr [<mode>] [<address>] [<space>]

<mode>	Specifies the Intel addressing mode:	
	Auto	derives the addressing mode based on the pmode.
	Use16	uses 16-bit operands and addresses.
	Use32	uses 32-bit operands and addresses.
<address>	is a numeric or symbolic address of the location where the next Asm command will write.	
<space>	Specifies the Intel address space as user, smm, or io.	

With no <address>, AsmAddr displays the current assembly address.

```
AsmAddr 2000;
// Asm address offset: 2000
```

AuxTrace

Control Motorola 68360 port A and C multiplexing.

AuxTrace [portA | portC]

portA	Puts the Port A signals onto the most significant word of the auxiliary trace connector (ATC).
portC	Puts the Port C signals onto the most significant word of the ATC.

With no parameters, AuxTrace displays the current port.

AuxTrace is saved and restored when SLD is exited and restarted.

The least significant word of the ATC always provides Port B [0:15].

BDMspeed

Examine or set the BDM speed.

BDMspeed [slow | fast]

slow specifies a system clock slower than 1 MHz.

fast (default) specifies a clock equal to or faster than 1 MHz.

With no parameters, BDMspeed displays its current setting.

Use this command for processors with system clocks slower than 1 MHz. The downloading speed with BDMspeed fast is about five times the downloading speed with BDMspeed slow.

Bkpt

Display, set, or modify breakpoints.

Related topics: BkptClear, DR

Bkpt [enable | disable] [temporary | permanent] [<address>] [@<ID>] [<space>]

enable	with @<ID> specified, enables the breakpoint; otherwise enables all breakpoints.
disable	with @<ID> specified, disables the breakpoint; otherwise disables all breakpoints.
temporary	removes the breakpoint when it halts emulation.
permanent	retains the breakpoint when it halts emulation. To remove the breakpoint, explicitly delete it.
<address>	a numeric or symbolic address. When this address is accessed, the breakpoint (if enabled) halts execution.
<ID>	an integer from 0 to 65534. When you do not specify an ID for a breakpoint entry, the system assigns one. When the specified ID matches an existing breakpoint, the existing breakpoint is modified. The at (@) is required.
<space>	For an Intel emulator, smm or user. smm sets a breakpoint in SMM address space. user sets a breakpoint in user address space (the default). For a Motorola emulator, sp, sd, up, or ud.

With no parameters, Bkpt displays all permanent and temporary breakpoints. Source information is also displayed whenever a match exists with the symbol table.

```
bkpt disable temporary @12
                    /* disable the temporary breakpoint with ID 12 */
```

You can also set breakpoints using the Source window mouse or Breakpoints menu, or the Breakpoint window Set button or Breakpoints menu.

BkptClear

Remove breakpoints.

Related topics:
Bkpt

BkptClear [@<ID> | <address> [<space>] | all]

<ID>	removes the breakpoint with the specified ID number. The at (@) is required.
<address>	removes the breakpoint at the specified code address.
<space>	used with <address>, optionally specifies the Intel address space (user or smm) of the breakpoint.
all	removes all temporary and permanent breakpoints.

Use BkptClear to remove a specified breakpoint or all temporary and permanent breakpoints.

```
BkptClear @1;                    /* remove breakpoint with id 1 */
BkptClear all;                   /* remove all breakpoints */
```

You can also clear breakpoints using the Source window mouse or Breakpoints menu, or the Breakpoint window Clear button or Breakpoints menu.

BusRetry

Asserts bus error after timeout.

BusRetry [on | off]

on	turn retry on.
off	turn retry off.

With no parameters, BusRetry displays its current setting.

Disable retry when contention exists with another driver or when a slow device takes longer than the time out.

Cause

Display the cause of the last break in emulation.

Related topics:
$BREAKCAUSE

Cause

Use this command when emulation is halted to discover the reason for the most recent halt. Possible Cause responses are:

- No cause is recorded.
- The target processor was reset.
- You entered a Halt command.
- The emulator completed a Step.
- Emulation encountered an execution breakpoint.
- The emulator detected a double bus fault.
- The emulator received an external break request.
- The cause is unknown.

The break cause also appears in the Status window.

Clear

Clear the Shell window Transcript pane.

Clear

Use Clear to remove all text from the Shell window Transcript pane. The Shell window View menu Clear Transcript item does the same.

CompilerUsed

Specify the toolchain used for a Motorola loadfile.

Related topics:
MergeSections

CompilerUsed [<compiler>]

<compiler> is a supported compiler for Motorola processors. Look in the powerpak.ini file, in your windows directory, for a list of the supported compilers.

This command specifies your toolchain (compiler, linker, translator, and loader). Specify the toolchain before the first time you load code or symbols; and thereafter only when you change compilers.

```
CompilerUsed MRI;                    // Using the MRI toolchain.
```

You can also specify a compiler with the Source window Options menu Compiler Used item.

Config

Defines Intel386 EX HLDA pin function.

Config ignoreHLDA [on | off]

on — causes the emulator to ignore the HLDA pin state. Set config ignoreHlda on when HLDA is programmed as an I/O bit.

off — (initial default) causes the emulator to examine the HLDA pin state before generating overlay RAM or trace/trigger strobe.

With no parameters, Config displays its current setting.

On the 386EX, you can program the HLDA pin to function either as HLDA function or as an I/O bit. The emulator hardware must know when the bus has been granted to an external master so that overlay RAM cycles are disabled to prevent corruption. If the HLDA pin is visible, the emulator disables overlay RAM cycles. Otherwise, the emulator assumes that no external masters exist.

When using the Intel Evaluation Board, which programs the HLDA pin to be an I/O bit, set config ignoreHlda on.

ConfigCS

Sets up the emulator hardware to match the target Motorola processor chip selects.

Related topics: RestoreCS, SaveCS

ConfigCS ["<filename>"]

<filename> — is a file containing chip select register value specifications. The quotation marks are required.

This command uses the CS registers in the system integration module (SIM) to program the emulator trace, trigger, and overlay hardware. The emulator reads the chip select signal mapping and matches the hardware to these programmed pins and operation modes of the target.

Entering ConfigCS with a filename is the same as entering RestoreCS with the filename followed by ConfigCS with no filename.

You can also configure chip selects, after programming the processor SIM peripheral, with the Toolbar Configure menu Configure Chip Selects item.

ConfigSymbols

Updates symbol base address from the Intel descriptor table.

ConfigSymbols [<basename>]

<basename> — is the base name for a specific group of symbols.

Updates the symbols with the base address obtained from the descriptor

table (either GDT or LDT). To get the correct symbol base, the target program must set up the correct values of GDTR and LDTR and the contents of those tables.

With no parameters, all symbols are reconfigured. To update a specific group of symbols, specify the base name for the symbols.

Copy

Copies one region of target or overlay memory to another.

Related topics: Dump, Fill, RunAccess, Search, Size, Verify, Write

Copy <start> [<end> | Length <len>] [<space>] [Target] to [<dest>] [<space>] [Target]

<start>	specifies the starting address of the region to be copied.
<end>	specifies the ending address of the region to be copied.
length <len>	specifies the number of bytes to be copied.
<space>	for Intel emulators specifies smm or user (the default) address space. for Motorola CPU16 emulators specifies data (the default) or program address space. for Motorola CPU32 emulators specifies sp, sd (the default), up, ud, cpu, s0, s3, or s4 address space.
Target	Use this parameter to override the mapping of the region. If specified, target memory is used as the source or destination.
to [<dest>]	Specifies the starting address that will be copied into.

Because reading and writing memory takes a small amount of processor time, memory access (such as copying) is initially disabled during emulation. Use RunAccess to enable memory copying during emulation; however, such copying can degrade your program execution.

```
/* Copy 64 KB from address 0x0 to overlay at the same address: */
map 0 10000;
copy 0 length 1000 target to 0;

/* To copy from overlay to target, the commands are */
copy 0 length 1000 to 0 target;

/* To copy from overlay to overlay*/
copy 1000 length 1000 to 4000;

/* Using symbolic addresses*/
```

copy #func1 #func2 to #ram_area target;

You can also copy memory with the Memory window Edit menu Copy Memory item.

Dasm

Disassemble memory.

Dasm [<mode>] [<start> [<end>] [<space.]]

<mode> Specifies the Intel addressing mode:

- Auto derives the addressing mode based on the pmode.
- Use16 uses 16-bit operands and addresses.
- Use32 uses 32-bit operands and addresses.

<start> is the first address of the region to disassemble.

<end> is the last address of the region to disassemble.

<space> for Intel emulators specifies smm or user (the default) address space.

When no addresses are specified, 10 instructions are disassembled beginning at the previous last address. When only <start> is specified, 10 instructions starting at that address are disassembled.

You can also view disassembled memory with the Memory window View menu Disassembly item., or interleaved in your source text with the Source window View menu Mixed Source And Asm item.

DasmSym

Control symbolic disassembly in the Shell window.

Related topics: Asm, AsmAddr, Dasm

DasmSym [on | off]

on (default) turns on symbolic disassembly.

off turns off symbolic disassembly.

With no parameters, DasmSym displays the current status of symbolic disassembly.

Symbolic disassembly displays symbols in the disassembly shown in the Memory window in Disassembly view, the Source window Mixed Source And Asm view, and the Trace window Instruction view.

You can also toggle symbolic disassembly with the Toolbar Configure menu Symbolic Disassembly item.

Delete

Delete a Shell variable or alias

Delete (Alias "<name>" | <variable>)

<name> is the alias to be deleted. The Alias keyword and the quotation marks are required.

<variable> is the Shell variable to be deleted.

```
$a = $b = 0;
list;
// $a = 0
// $b = 0
Delete $a
list
// $b = 0
Alias "a" "$a;" ;
Alias;
// a: "$a;"
Delete Alias "a";
Alias;
```

DisableAlarmLimit

Disable the warning message for excessive stack usage.

Related topics:
DisableHighWaterMark,
DisplayStack,
EnableAlarmLimit,
EnableHighWaterMark,
FillStackPattern,
SetStackAlarm,
SetStackBase,
SetStackSize,
StackInfo,
SetStackArea

DisableAlarmLimit

You can set an alarm (using EnableAlarmLimit) to notify you when stack usage exceeds a specified percentage of the stack. DisableAlarmLimit turns off this alarm.

You can also disable the alarm by un-checking the Stack window Options menu Enable Alarm Limit item.

DisableHighWaterMark

Disable keeping track of the stack maximum usage.

Related topics:
DisableAlarmLimit,
DisplayStack,
EnableAlarmLimit,
EnableHighWater-Mark,
FillStackPattern,
SetStackAlarm,
SetStackArea,
SetStackBase,
SetStackSize,
StackInfo

DisableHighWaterMark

You can set an indicator in the Stack window to keep track of the stack high-water mark, that is, the maximum stack usage. DisableHighWaterMark turns off this indicator.

You can also disable the high-water mark by un-checking the Stack window Options menu Enable High-Water Mark item.

DisplayStack

Display the stack frames.

Related topics:
DisableAlarmLimit,
DisableHighWater-Mark,
EnableAlarmLimit,
EnableHighWater-Mark,
FillStackPattern,
SetStackAlarm,
SetStackBase,
SetStackSize,
StackInfo,
SetStackArea

DisplayStack [locals | hex]

locals includes symbols for automatic variables.

hex displays the stack in the hexadecimal radix of 16 bytes per line.

When you specify no parameters, the display defaults to:

- Addresses only if no symbolic information is available
- Addresses and function names if symbolic information is available

You can also view the stack frames, with stack and return addresses, parameters, and local variables, in the Stack window.

DisplaySymbols

Display all symbols or display one of the following: modules, functions, public symbols, or lines.

Related topics:
AddressOf,
GetBase, NameOf,
RemoveSymbols,

DisplaySymbols [modules | functions | publics | lines | sorted | #<module name>]

modules	displays modules only.
functions	displays modules, global variables, functions, and blocks.
publics	displays all printable symbols including publics (those code labels and variables defined publicly for

the purpose of linking modules together). For example, libraries normally do not contain local symbols but any accessible global variables are displayed as public symbols.

lines	displays each module followed by the line numbers loaded for that module. The information for each line includes the line number, its ending column, and its start address.
sorted	displays the modules sorted alphanumerically.
#<module name>	displays all symbols for the specified module.

With no parameters, DisplaySymbols displays modules, global variables, functions, and local variables, but not publics nor individual line numbers.

If you have previously issued a SymbolOpenFile command, the DisplaySymbols output is directed to the symbol file.

The output is displayed in four columns:

- The first column contains the symbol type (MODULE, VARIABLE, FUNCTION, BLOCK, PUBLIC VAR, PUBLIC LABEL). Each line is indented to show the level or scope of the symbol in the symbol hierarchy. Modules and publics are at the root level. Functions defined in a module are indented one level. Variables local to a function are indented under that function. Blocks are treated as unnamed functions and indented for each nesting level.
- The second column contains the symbol name.
- The third column contains the symbol type for a variable, the return type for a function, and the range of source line numbers for a module. For local variables and parameters allocated as registers, the register name and type are displayed in the third column.
- The fourth column shows the symbol's address information. For static (fixed) address symbols, the fourth column shows the address range followed by the size of the range in decimal in square brackets ([<size>]). The end address points to the last byte of the range. For local variables allocated on the stack, the address is a signed offset from the stack frame pointer.

DR

Control Intel386 debug register use.

DR <num> [bkpt | user | [data <mode> <address> <size> [exact]]]

<num> identifies the debug register as 0, 1, 2, or 3.

bkpt makes the register available for execution breakpoints.

user reserves the register for use by your program. The emulator avoids using this register for execution breakpoints and modifies DR7, allowing user access to any debug register.

data configures the register as a data read/write breakpoint.

<mode> is one of:

- x sets the register to instruction execution mode. Emulation breaks on execution of the instruction whose first byte is at <address>.
- w sets the register to data write mode. Emulation breaks on a write to <address> in user, SMM, or I/O space.
- rw sets the register to data read/write mode. Emulation breaks on a read or write to <address> in user, SMM, or I/O space.

<address> specifies the virtual or linear base address of the breakpoint.

<size> specifies the 1, 2, or 4 bytes starting with <address> as the address range of the data breakpoint. Emulation breaks on any data access completely or partly overlapping this range.

exact ensures the processor waits after each instruction for all data cycles to complete. Any data breakpoint thus occurs immediately after the instruction that caused the breakpoint data cycle. (Execution breakpoints always occur exactly.) With exact not specified, several instructions can execute beyond the one that caused the breakpoint data cycle. Using exact can degrade your program's performance.

With no parameters, DR lists all four debug register allocations.

Use DR to allocate the four Intel386 debug registers for use by the emulator as execution or data breakpoints or for use by your target system. When you install SLD, all four debug registers are configured for execution (hardware) breakpoints. Changing this configuration reduces the number of execution breakpoints available.

Reserving a debug register for use by your program also allows undetected program access to system registers and to DR7. Your program can thus make changes to DR7 that can cause the emulator to behave unpredictably.

```
dr 0 user;                    /* Reserve dr0 for the target system. */
dr 1bkpt;    /* Allow dr1 to be used as an execution breakpoint. */
```

```
dr 2;                          /* Show the current configuration of dr2. */
dr;   /* Show the current configuration of dr1, dr2, dr3, and dr4. */
dr 3 w 1000p dword;            /* Define a dword sized, data write */
                               /* breakpoint at physical address 1000. */
```

DT

Displays the descriptor table.

Related topics: gdt, idt, ldt, tss

DT (<selector_range> | <register> | base <address> [limit <bytes>]) [all]

<selector_range>	specifies either a single value (e.g., 0 or 0x08) or two values to specify a range (e.g., 4 14 or 0x10 0x400).
<register>	specifies a register.
base <address>	specifies the descriptor table base address.
[limit <bytes>]	If a base address is specified, you must also specify either <selector_range> or limit <bytes> to define the range to be displayed.
all	displays all entries, including invalid or reserved.

Use DT to display the descriptor table entries for a single selector or range of selectors. The selector displayed is determined by <selector range>, <register>, or base <address>, one of which must be specified. Specifying <register> uses the register value for the selector.

You need not specify which descriptor table. The table is determined by bit 2 (TI) of the selector.

```
dt 0x08 0x48 all;              /* displays all descriptor entries */
                               /* from selector 0x08 to 0x48 */
dt ds;              /*displays just the current ds descriptor entry */
```

Dump

Dumps memory contents to the screen, formatted.

Related topics: Copy, Fill, RunAccess, Search, Size, Verify, Write

Dump [loop] <address1> [<address2>] [byte | word | long | dword] [<space>]

<address1>	specifies the frist address to be displayed.
<address2>	specifies the last address to be displayed. If <address2> is not specified, 16 bytes (one line) is displayed. An address can be symbolic or numeric.

byte	displays the values as bytes.
word	displays the values as words.
long, dword	displays the values as double words.
<space>	for Intel emulators specifies smm, user (the default), or io address space.
	for Motorola CPU16 emulators specifies data (the default) or program.
	for Motorola CPU32 emulators specifies sp, sd (the default), up, ud, or cpu address space.
loop	repeatedly preforms the operation but prints no output to the screen, even if errors occur.

The physical read of memory uses the Size command settings rather than the format size set by Dump. For example, if size=byte, Dump reads byte-sized memory accesses regardless of how the data is to be displayed.

Because reading and writing memory takes a small amount of processor time, memory access (such as dumping to the screen) is initially disabled during emulation. Use RunAccess to enable Dump during emulation; however, such access can degrade your program execution.

You can also view memory contents in up to 20 simultaneously active Memory windows as hexadecimal or decimal bytes, words, or dwords with equivalent ASCII characters; or as disassembled instructions.

Echo

Display or toggle command echo.

Related topics: Append, Echo, Log, Logging, Overwrite, Results

Echo [on | off]

on	enable command echo.
off	disable command echo.

When no options are entered, Echo displays the current setting.

With command echo enabled, commands entered in the Command Entry pane are echoed to the Transcript pane before command execution. This Echo command is the same as the Echo in the Options menu of the Shell window.

You can also toggle the Shell command echo with the View menu Echo item.

EmuStatus

Report the current emulation status.

Related topics:
IsEmuHalted,
$EMULATING

EmuStatus

Use EmuStatus to discover whether the processor is halted after IsEmuHalted returns no result.

```
isemuhalted;

emustatus;
// Processor is running.

halt;
// 961C60  0000 0000      ORI.B     #00,D0

isemuhalted;
// The emulator is halted.
```

The emulation status (halted or running) also appears in the Status window or icon title. You can also use $EMULATING to discover the emulation status.

EnableAlarmLimit

Enable a stack alarm limit.

Related topics:
DisableAlarmLimit,
DisableHighWater-Mark,
DisplayStack,
EnableHighWater-Mark,
FillStackPattern,
SetStackAlarm,
SetStackArea,
SetStackBase,
SetStackSize,
StackInfo

EnableAlarmLimit

If, when emulation halts, the stack usage is exceeding the alarm limit set by SetStackAlarm, you are notified.

You can also enable the alarm limit by checking the Stack window Options menu Enable Alarm Limit item.

EnableHighWaterMark

Track maximum stack usage.

Related topics:
DisableAlarmLimit,
DisableHighWater-Mark,

EnableHighWaterMark

This command turns on a graphical indicator (an arrow on the stack meter) in the Stack window that keeps track of the maximum amount of memory used by the stack. The indicator is the stack high-water mark.

DisplayStack, EnableAlarmLimit, FillStackPattern, SetStackAlarm, SetStackArea, SetStackBase, SetStackSize, StackInfo

You can also enable the high-water mark by checking the Stack window Options menu Enable High-Water Mark item.

EventRestore

Restore saved Events.

EventRestore "<filename>"

"<filename>" specifies the file where the event definitions are to be stored. The quotation marks are required.

You can also restore events from a file with the Event window File menu Restore Events item.

EventSave

Save Events to a file.

EventSave "<filename>"

"<filename>" specifies the file in which to store the event definitions. The quotation marks are required.

You can also save events to a file with the Event window File menu Save Events item.

Exit

Exit the Shell window.

exit

This command closes the Shell window. To exit from SLD, on the Toolbar open the File menu and choose Exit.

You can also close the Shell window with the Shell window File menu Exit item.

Fill

Fill memory with data.

Fill <address1> <address2> <data> [byte | word | long | dword] [<space>]

Related topics: Copy, Dump,

<address1> is the first address in the region to be filled. Addresses can be symbolic or numeric.

RunAccess, Search, Size, Verify, Write

<address2>	is the last address in the region to be filled.
data	is the data top be written.
byte	specifies the data is a byte value.
word	specifies the data is a word value.
long, dword	specifies the data is a double word value.
<space>	for Intel emulators specifies smm or user (the default) address space. for Motorola CPU16 emulators specifies data (the default) or program. for Motorola CPU32 emulators specifies sp, sd (the default), up, ud, or cpu address space.

Fill fills memory from <address1> to <address2> with one or more repetitions of <data>. When the number of data bytes is less than the address range, the data is repeated enough times to fill the address range. Up to 256 bytes of data can be specified.

The physical write to memory uses the Size command settings rather than the format size set by the fill command. For example, if size=byte, any fill command fills memory by byte-sized memory accesses.

Because reading and writing memory takes a small amount of processor time, memory access (such as filling memory) is initially disabled during emulation. Use RunAccess to enable Fill during emulation; however, such access can degrade your program execution.

```
Fill 0 1234 0x0 dword;     /* Fills memory from 0 to 64K with 0x0 */
// Fill successful.
```

You can also fill memory with the Memory window Edit menu Fill Memory item.

FillStackPattern

Initialize the stack.

Related topics: DisableAlarmLimit, DisableHighWater-Mark, DisplayStack, EnableAlarmLimit, EnableHighWater-Mark,

FillStackPattern

With FillStackPattern, you can initialize the stack with a special pattern to enable the tracking of the stack usage.

Other commands can also initialize the stack:

- If you specify the stack base and size with FillStackArea, you can also initialize the stack in the single FillStackArea command.

SetStackAlarm, SetStackArea, SetStackBase, SetStackSize, StackInfo

- Enabling the high-water mark automatically fills the stack with a pattern.

GDT

Displays the global descriptor table.

Related topics: dt, idt, ldt, tss

GDT [<selector_range> | <register> | base <address> [limit <bytes>]] [all]

Use GDT to display the global descriptor table entries for a single selector or a range of selectors. Which selectors are displayed is determined by <selector range>, <register>, base <address>, or the current gdt_base and gdt_limit.

With no parameters, GDT shows all valid entries in the range gdt_base to gdt_base+gdt_limit.

<selector_range>	specifies a single value (0 or 0x08) or a range between two values (4 14 or 0x10 0x400).
<register>	specifies a register to be used for the selector.
base <address>	specifies the descriptor table base address.
[limit <bytes>]	If a base address is specified, you must also specify either <selector_range> or limit <bytes> to define the range to be displayed.
all	displays all entries, including invalid or reserved.

```
gdt 0x00 0x18 base 501010L;       /* Displays global descriptor */
                                  /* table entries. The table base is 501010L. */
                                  /* This command displays global descriptor */
                                  /* table entries from 501018L (selector 0x08) */
                                  /* to 501028L (selector 0x18). */
```

GetBase

Get one or all base names and their address offsets.

Related topics: AddressOf, DisplaySymbols, NameOf, RemoveSymbols, SetBase

GetBase [<basename>]

<basename> displays only the specified base.

With no parameters, all bases loaded into the symbol table are displayed along with their offset values.

Compilers and linkers place symbols into groups called bases, assigning names to the groups. GetBase displays these symbol bases.

Go

Start emulation.

Related topics: $BREAKCAUSE System Variable, $EMULATING System Variable, Cause, GoInto, GoUntil, Halt, ResetAndGo, Step, StepSrc

Go

Emulation does not start unless a function can be found in the symbol table that includes the current program counter address.

Other ways to start emulation include:

- On the Toolbar, choose the Go button.
- In the Source window, choose the Go button.
- In the Source window, open the Run menu and choose Go.
- Press the <F9> key.

GoInto

Emulate to a stepped-into or returned-into function.

Related topics: $BREAKCAUSE System Variable, $EMULATING System Variable, Cause, Go, GoUntil, Halt, ResetAndGo, Step, StepSrc

GoInto [call | return] [line | statement]

GoInto emulates until a call or return is executed, steps into the call or return, and stops at a line or statement of the entered function.

With no parameters specified, the first GoInto you use defaults to GoInto call statement. If you have previously used GoInto with parameters, any GoInto without parameters defaults to the parameters you used before.

call	If a call is executed within the current function, emulation continues through the call and into the called function. Emulation halts on the beginning of a line or statement of that function. This line or statement may be the first instruction of the function or later, depending on how the compiler generates code and line-number start addresses.
return	If a return instruction is executed within the current function, emulation continues through the return and stops on the beginning of the next line or statement of the function that was returned into.
line	The break is on a source line.
statement	The break is on a C statement.

call and return are mutually exclusive; statement and line are mutually exclusive.

You can also do these variations of "Go Into" with the Source window buttons (as configured by the Source window Options menu Set Go Buttons item) and the Source window Run menu.

GoUntil

Emulate until a call or return.

Related topics: $BREAKCAUSE System Variable, $EMULATING System Variable, Cause, Go, GoInto, Halt, ResetAndGo, Step, StepSrc

GoUntil [call | return] [line | statement]

call	within the current function, emulates until a call or return is executed.
return	within the current function, emulates until a return instruction is executed.
line	breaks on a source line.
statement	breaks on a C statement.

call and return are mutually exclusive; statement and line are mutually exclusive.

With no parameters specified, the first GoUntil you use defaults to GoUntil call statement. If you have previously used GoUntil with parameters, any GoUntil without parameters defaults to the parameters you used before.

GoUntil emulates until a call or return is executed, then stops.

Because of how call and return work, some assembly instructions prior to the call or return are not necessarily executed.

You can also do these variations of "Go Until" with the Source window buttons (as configured by the Source window Options menu Set Go Buttons item) and the Source window Run menu.

Halt

Halt emulation.

Halt

Use Halt to stop emulation with no dependence on breakpoints or triggers.

Other ways to manually stop emulation include:

- On the Toolbar, choose the Halt button.
- In the Source window, choose the Halt button.
- In the Source window, open the Run menu and choose Halt.
- Press the <F2> key.

Help

Invoke on-line help.

Help [<command>]

Use Help to display the command syntax for one or more Shell window commands. When you specify no <command>, Help displays an alphabetical list of all commands.

You can also get on-line help from any SLD window Help menu (or the CPU window Options menu) or by pressing the <F1> key.

History

Control number of saved commands.

History [<size>]

<size> specifies the number of commands to save in the Shell command history buffer.

With no parameters, History reports the number of previously entered commands that are saved in the history buffer. To change the size of the list, specify a <size>.

Press <Ctrl><Up Arrow> or <Ctrl><Down Arrow> to recall commands from the history buffer to the Command Entry pane.

You can also set the history size with the Shell window Options menu History Size item.

IDT

Display the interrupt descriptor table.

IDT [<index_range> | <register> | base <address> [limit <bytes>]] [all]

Related topics: dt, gdt, ldt, tss

<index_range>	specifies either a single value (e.g., 0 or 0x08) or two values to specify a range (e.g., 4 14 or 0x10 0x400).
<register>	specifies a register; the selector for the specified register is used.
base <address>	specifies the descriptor table base address.
[limit <bytes>]	If a base address is specified, you must also specify either <selector_range> or limit <bytes> to define the range to be displayed.
all	displays all entries, including invalid or reserved.

With no parameters, IDT shows all valid entries in the range idt_base

to idt_base+idt_limit.

Use IDT to display the interrupt descriptor table entries for a single index or a range of indices. Which selectors are displayed is determined by <index range>, <register>, base <address>, or the current idt_base and idt_limit.

```
idt 0x00 0x18 base 501010L      /* Displays interrupt descriptor */
                                /* tables. The table base is 501010L. */
                                /* This command displays interrupt descriptor */
                                /* tables from 501018L (selector 0x08) */
                                /* to 501028L (selector 0x18). */
```

If..Else

Conditionally execute Shell window commands.

If (<condition>) { <block> } [else { <block> }]

<condition>	evaluates to nonzero (true) or zero (false). The parentheses are required.
<block>	is a list of Shell commands delimited with semicolons. The braces are required.

If <condition> is true, the first block of statements executes. Otherwise, if the else block is present, the second block of statements executes.

```
$a = 0;
If ($a) {
  "true";
}
else {
  "false";
};
// false
$a = 1;
If ($a) {
  "true";
}
else {
  "false";
};
// true
```

Include

Read commands from a file.

include "<filename>"

<filename> is the name of a file containing Shell commands (a script). The quotation marks are required.

The commands are executed as if entered in the Command Entry pane.

```
include "d:\shell.cmd";  /* executes d:\shell.cmd */
```

You can also run a script with the Shell window File menu Include item.

Integer

Identifies an integer.

Integer (<variable>)

Related topics: String

<variable> is the name of a Shell variable.

Use Integer to discover whether a variable value is an integer. Integer returns true (1) if <variable> is an integer and false (0) otherwise.

```
$a = 0;
Integer($a);
// 1 1

If (integer($a)) { "it is an integer"; }
// it is an integer
```

IsEmuHalted

Discover whether emulator is halted.

IsEmuHalted

Use IsEmuHalted to discover whether the emulator is halted. No response indicates the emulator is not halted. If you get no response, also use EmuStatus or $EMULATING.

Related topics: EmuStatus, $EMULATING

```
isemuhalted;

halt;
// 961C60  0000 0000      ORI.B     #00,D0

isemuhalted;
// The emulator is halted.
```

The emulation status (halted or running) also appears in the Status window or icon title. You can also use $EMULATING to discover the emulation status.

LapTimer

Takes a snapshot of the timer.

Related topics: StartTimer, StopTimer

LapTimer

Returns the number of milliseconds elapsed since the timer was started, but does not stop the timer.

```
LapTimer;
while (laptimer < 5000) {};
```

LDT

Displays the local descriptor table.

Related topics: dt, gdt, idt, tss

LDT [<selector_range> | <register> | base <address> [limit <bytes>]] [all]

<selector_range>	specifies either a single value (e.g., 0 or 0x08) or two values to specify a range (e.g., 4 14 or 0x10 0x400). when a single value is specified, it is used as the selector from the GDT to specify the LDT base and limit.
<register>	specifies a register; the selector for the specified register is used.
base <address>	specifies the descriptor table base address.
[limit <bytes>]	If a base address is specified, you must also specify either <selector_range> or limit <bytes> to define the range to be displayed.
all	displays all entries, including invalid or reserved.

With no parameters, LDT shows all valid entries in the range ldt_base to ldt_base+ldt_limit.

Use LDT to display the interrupt descriptor table entries for a single index or a range of indices. Which selectors are displayed is determined by <selector_range>, <register>, base <address>, or the current ldt_base and ldt_limit.

```
ldt 0x00 0x18 base 501010L;  /* Displays local descriptor tables. */
                             /* The table base is 501010L. This command displays */
                             /* local descriptor tables from 501018L (selector 0x08) */
                                                          /* to 501028L (selector 0x18). */
```

List

List Shell variables.

List [<variable>]

With no parameters, List displays all the Shell variables and their values. To list the value of a single variable, specify the variable name.

```
List;
// (system) $SHELL_STATUS = 262158
```

Load

Load code and symbols to mapped or target memory.

Load "<filename>" [user | smm] [[no]code] [[no]symbols] [[no]asm] [[no]demand] [[no]demangle] [[no]updatebase] [module <name>] [reload] [[no]loadregister] [[no]warn] [[no]status]

Related topics: LoadSize

"<filename>"	is the pathname of the file to be loaded. The quotes are required.
user	For Intel emulators, loads code into user memory (default).
smm	For Intel emulators, loads code into system management mode memory.
[no]code	loads (default) or does not load code.
[no]symbols	loads (default) or does not load symbols.
[No]asm	loads or does not load (default) Motorola assembly module names.
[no]demand	loads symbolic information only on demand (default) or loads all symbols (globals, locals, line numbers) for all modules in the program initially. On-demand loading initially loads just global symbols (variables, module names, global function names, type definitions). Local variables and line numbers are not loaded until needed.
[no]demangle	demangles or does not demangle (default) C++ names.
[no]updatebase	updates symbol bases or does not update symbol bases, for Intel emulators. This parameter is valid for OMF386 loadfiles only. Use updatebase in conjunction with loadregister.
module <name>	After an initial on-demand load, load symbols for

	the specified module Use in a script if you know you will be debugging a specified module or modules. If you load symbols with this option, there is no delay when you view one of these modules.
reload	To purge old symbols and load new ones with one command, use the reload option.
[no]loadregister	loads or does not load (default) initial register values from OMF386 loadfiles.
[no]warn	displays or does not display (default) warnings from the loader.
[no]status	displays (default) or does not display load statistics.

You can load code and symbols during emulation. Avoid loading into an area of memory occupied by the executing code. Loading into memory that is being executed can stop the emulator in an unpredictable state.

This command is the same as the Load button on the Toolbar.

```
/* on-demand symbol loading*/
Load demo.abs;
// 1986 bytes code loaded.
// 2 module(s) loaded.
// Load complete.

/* load module */
Load demo.abs module dm_main;

/* load symbols only, on demand (no code) */
load demo.abs nocode;

/* load code only (don't load symbols) */
Load demo.abs nosym;

/* code and all symbols are loaded */
load demo.abs nodemand;

/* load a new file, do not display warnings*/
load sample.abs reload nowarn;
```

You can also load files with the Toolbar Load button or from the Source window File menu.

LoadSize

Set the memory write-access size for the load command.

Related topics: Load, Size

LoadSize [byte | word | long | dword]

byte writes memory by bytes.

word writes memory by words.

long (default) writes memory by longs. Writing in long is the fastest way to load code.

dword is the same as long.

Log

Display or set the name of the log file.

Related topics: Logging, Append, Overwrite, Echo, Results

Log ["<filename>"]

<filename> is the name of the log file to be opened or created. The quotation marks are required.

With no parameters, Log displays the name of the current log file.

To start recording into the logfile, use Logging.

```
Logfile "c:\shell.log";
Log;
// log file name: c:\shell.log
```

You can also open a log file with the Options menu Log File Name item.

Logging

Display or toggle the logging setting.

Related topics: Log, Append, Overwrite, Echo, Results

Logging [on | off]

With no parameters, Logging reports whether logging is on.

on turns logging on.

off turns logging off.

When logging is on, the lines that are written to the transcript window are also written to the log file.

When you turn logging on, if overwrite mode is in effect, previously logged information is destroyed. To preserve information recorded earlier to the same file, enter Append before Logging on.

You can also toggle logging with the Options menu Log Results item.

Map

Replaces all or part of the target system memory with emulator memory.

Related topics: SaveMap, RestoreMap, MapRanges

map [clear | <base> [<end>] [target] [<access>]] [<space>]

clear — clears all map blocks.

<base> — is the address to start an overlay memory range. The address is rounded down to the nearest boundary block equal to the amount of memory mapped. In an Intel emulator, you can start a region on any 4K boundary. In a Motorola emulator, you must start a region on a boundary corresponding to the size of the region. (For example, 64K-byte regions must start on a 64K boundary; 128K-byte regions must start on a 128K boundary.)

<end> — is the last address of the range. If no <end> is specified:

Intel emulators map a 4K-byte region. The end address is rounded up to the top of the 4K-byte region containing the end address. With options for 1M bytes or 4M bytes of overlay, you can map up to 16 regions.

Motorola emulators map a 64K-byte region. The end address is rounded up to the top of the 64K-byte region containing the end address. With 256M bytes of overlay, you can map 64K-byte and 128K-byte regions. With 1M bytes of overlay, you can also map 256K-byte and 512K-byte regions.

target — map memory range to the target.

<access> — specifies access permissions:

- ram — allows read and write access (the default).
- rom — allows read access; prevents write access; does not break on attempted write access. (For Intel386 emulation in overlay memory, writes are allowed.)
- rombrk — allows read access; prevents write access; breaks on attempted write access. (For Intel386 emulation in overlay memory, writes are allowed but break emulation.)
- none — prevents any access; breaks on attempted access. (For Intel386 emulation in overlay memory, access is allowed but breaks emulation.)

<space> — for Intel emulators specifies smm, user (the default), or io address space.

for Motorola CPU16 emulators specifies data (the default) or program.

for Motorola CPU32 emulators specifies sp, sd (the default), up, ud, or cpu address space.

With no parameters, map displays the current map settings.

```
map 0 ram;
// Mapped block starting at address 00000000 to 0000FFFF RAM
```

You can also map memory with the Toolbar Map button.

MapRanges

Configure overlay memory for a Motorola 68360 emulator.

Related topics: SaveMap, Map, RestoreMap

MapRanges [0 | 2 | 4]

0 No map ranges; four hardware breakpoints are available.

2 Two map ranges; two hardware breakpoints are availble.

4 Four map ranges; no hardware breakpoints are available.

With MapRanges you can configure zero, two, or four blocks of overlay memory and a corresponding (four, two, or zero) number of hardware breakpoints.

When you use MapRanges, the map is reset to target RAM. Use the Toolbar Map button or the Map command to reconfigure memory.

MaxBitFieldSize

Set the maximum bit field size for OMF386 loadfiles.

MaxBitFieldSize [16 | 32]

16 Sets the maximum bit field size to 16 bits.

32 Sets the maximum bit field size to 32 bits (default).

If you use the Borland C compiler in generating your OMF386 loadfile, set the maximum bit field size to 16 bits.

MergeSections

Merge setions from a Motorola loadfile.

Related topics: CompilerUsed

MergeSections [on | off]

on merges the loadfile into two default sections.

off loads the sections as they appear in the loadfile (default).

For Motorola loadfiles containing more than 32 sections, merging sections can save memory.

NameOf

Find the symbol representing an address.

Related topics: AddressOf, DisplaySymbols, GetBase, RemoveSymbols, SetBase

NameOf <address>

<address> is a numeric address.

Use NameOf to look up a specified address and display the symbol that most closely matches the address.

```
NameOf 0x0900;
// #main#14#1 (function main)
```

Overwrite

Overwrites the log file.

Related topics: Append, Log, Logging, Echo, Results

Overwrite

When Overwrite has been specified, opening a log file (Log) or starting to log (Logging On) destroys the file's prior contents.

You can also configure logging to overwrite a file by opening the Shell window Options menu and choosing Overwrite Log File.

Pmode

Returns the processor mode.

Pmode

The 386 processors operate in various pmodes. These are real, virtual-86 (V86), protected, and System Management Mode (SMM).

Protected mode is further divided into 16-bit protected mode and 32-bit protected mode. The Intel386 DX and Intel386 SX processors do not have System Management Mode. The Intel386 CX and Intel386 EX have SMM.

```
pmode;
// Processor mode = Prot32
```

The pmode also appears at the bottom of the Status window icon.

Print

Print a value.

Print (<variable> | "<string>")

<variable>	is the name of a Shell variable.
<string>	is a string constant. The quotation marks are required.

Use Print to display the value of variables and strings.

```
Print("abc");
// abc

$a = 5;
Print($a);
// 0x5 5
```

RamTst

Run the memory hardware confidence tests.

Related topics:
Test

RamTst [loop] <address1> <address2> [<space>]

loop	repeats the low-level operations in the specified test so the operation can be observed on an oscilloscope. Press <Esc> to stop looping. An error does not halt the test loop.
<address1>	starting address to test.
<address2>	last address to test.
<space>	for Intel emulators specifies smm, user (the default), or io address space. for Motorola CPU16 emulators specifies data (the default) or program. for Motorola CPU32 emulators specifies sp, sd (the default), up, ud, or cpu address space.

```
ramtst 0x0000 0xFFFF;      /* Test memory from 0x0 to 0xffff. */
```

Register

Display or set register values.

Register [<name> [value]] [...]

<name>	is an Intel or Motorola register mnemonic.
<value>	is the value to be put into the register.

With no parameters, Register displays all the registers. A <name> without a <value> displays the value of the specified register; with a <value> sets the register to <value>.

You can also view and edit the registers in the CPU window.

RemoveSymbols

Remove symbols and clear symbol tables.

Related topics: AddressOf, DisplaySymbols, GetBase, Load, NameOf, SetBase

RemoveSymbols

Use this command to remove all loaded symbols and clear all allocated symbol tables.

Reset

Reset the processor.

Related topics: ResetAndGo

Reset [cpuonly]

Reset sends a RESET signal to the processor. All CPU register contents are lost on reset:

- The processor RESET pin is asserted.
- The program counter and stack pointer are read from memory.
- All SLD windows are updated. The Stack window display is invalid because the stack is reset. The Source window displays the beginning of your startup code, at the program counter.

With cpuonly specified, Reset resets only the processor and does not update the SLD windows. Use this parameter only if Reset without cpuonly fails to reset the processor:

1. Enter Reset CPUonly, resetting the processor without updating the SLD windows.
2. Reset your target.
3. Enter Reset again, without CPUonly, to update the SLD windows.

You can also reset the processor and optionally update the SLD windows from the Toolbar Configure menu, the Source window Run menu, or the CPU window Options menu.

ResetAndGo

Assert and release the target reset line.

Related topics: Reset

ResetAndGo

This operation is required to start some target systems. For example, targets that use an external watchdog timer or power-saver hardware may require that you use ResetAndGo..

You can also reset the processor and start emulation with the Source window Run menu Reset And Go item.

ResetLoaders

Reinitialize the loaders.

ResetLoaders "<pathname>"

<pathname> is the path to the directory containing the loaders.ini file. The quotation marks are required.

If you do not specify the pathname, the emulator looks for loaders.ini in the current SLD directory (e.g. c:\powerpak).

ResetLoaders causes SLD to reinitialize loaders. Use this command when you get an error message telling you to do so.

RestoreCS

Restores the chip-select register values.

Related topics: SaveCS, ConfigCS

RestoreCS "<filename>"

<filename> is an ASCII file containing chip select values. The quotation marks are required.

This command restores the chip-select registers to the values specified in the ASCII file saved with SaveCS. This file contains a line for each of up to 30 chip select registers. Each line can be up to 80 characters long, containing the following sequential fields:

<CHIP SELECT REGISTER NAME>
<space(s) (20)>
<hex value>
<new line or optional white space>
<anything other than 0A and 0>
<new line>

The register name must be in upper case and must match a valid chip register name. Only values different from the default values need be entered. The <anything other than...> field is for a short comment.

For Motorola emulators, use RestoreCS "<filename>" to restore chip selects if you don't want to configure the emulator hardware to match; otherwise, use ConfigCS "<filename>" to restore chip selects and configure the emulator hardware.

You can also restore the chip selects from a file with the Toolbar Configure menu Restore Chip Selects item.

RestoreMap

Restores a saved map configuration.

Related topics: SaveMap, Map, MapRanges

RestoreMap "<filename>"

<filename> contains the map configuration to restore. The quotation marks are required.

You can also restore the map from a file with Map dialog box Restore button, accessible via the Toolbar Map button.

Results

Set the Transcript window results display.

Related topics: Log, Logging, Append, Overwrite, Echo, Results

Results [on | off]

on enable command results echo.

off disable command results echo.

Without parameters, Results displays the current setting.

Use this command to toggle whether the transcript window displays the Shell command results.

You can also toggle the echo with the View menu Show Results item.

RunAccess

Set the target processor access mode during emulation.

Related topics: Copy, Dump, Fill, Search, Size, Verify, Write

RunAccess [on | off]

off (default) disables reading and writing memory during emulation.

on enables reading and writing memory during emulation.

Without parameters, RunAccess shows whether run access is on or off.

Because reading and writing memory takes a small amount of processor time, memory access is initially disabled during emulation. Such access includes scrolling and refreshing the Memory and

Peripheral windows and reading and writing memory from the Memory, Peripheral, and Shell windows. Use RunAccess to make memory accessible during emulation; however, such access can degrade your program execution.

You can also toggle run access with the Toolbar Configure menu Run Access item.

SaveCS

Saves the chip-select registers.

Related topics: RestoreCS, ConfigCS

SaveCS "<filename>"

<filename> creates or overwrites a file with an ASCII description of the chip select register values. The quotation marks are required.

Use SaveCS to record the chip select values. The values can be restored from the file using RestoreCS.

Different chip select registers are saved for different processors. The following lists the registers saved for each processor.

Motorola 68330 and 68340:

MBAR	CS2MASK	MCR
CS0MASK	CS2BASE	PPARB
CS0BASE	CS3MASK	PPARA1
CS1MASK	CS3BASE	PPARA2
CS1BASE		

Motorola 68331, 68332, 68333, and 68HC16:

CSPAR0	CSOR2	CSBAR7
CSPAR1	CSBAR3	CSOR7
CSBARBT	CSOR3	CSBAR8
CSORBT	CSBAR4	CSOR8
CSBAR0	CSOR4	CSBAR9
CSOR0	CSBAR5	CSOR9
CSBAR1	CSOR5	CSBAR10
CSOR1	CSBAR6	CSOR10
CSBAR2	CSOR6	

Motorola 68360:

MBAR	BR2	BR5
GMR	OR2	OR5
MSTAT	BR3	BR6
BR0	OR3	OR6
OR0	BR4	BR7

BR1	OR4	OR7
OR1		

Intel386 EX:

P1CFG	DMACFG	P1LTC
P2CFG	INTCFG	P1DIR
P3CFG	TMRCFG	P2LTC
PINCFG	SIOCFG	P2DIR
CS0ADL	CS3ADL	P3LTC
CS0ADH	CS3ADH	P3DIR
CS0MSKL	CS3MSKL	CS6ADL
CS0MSKH	CS3MSKH	CS6ADH
CS1ADL	CS4ADL	CS6MSKL
CS1ADH	CS4ADH	CS6MSKH
CS1MSKL	CS4MSKL	UCSADL
CS1MSKH	CS4MSKH	UCSADH
CS2ADL	CS5ADL	UCSMSKL
CS2ADH	CS5ADH	UCSMSKH
CS2MSKL	CS5MSKL	
CS2MSKH	CS5MSKH	

Since no peripheral registers are available in the Intel386 CX/SX, none are saved.

You can also save the chip selects with the Toolbar Configure menu Save Chip Selects item.

SaveMap

Saves a map configuration.

Related topics: RestoreMap

SaveMap "<filename>"

<filename> specifies the drive, directory, and name of the file where the map configuration is saved. The quotation marks are required.

You can later restore the map configuration with RestoreMap.

You can also save the map from the Map dialog box, accessible from the Toolbar Map button.

Search

Find the address of a pattern.

Related topics:

Search <start> <end> [not] <data> [byte | word | long | dword] [<space>]

<start>	is the first address in the range of addresses to search. Addresses can be symbolic or numeric.
<end>	is the last address in the range to search.
not	searches for the first pattern mismatch rather than the first pattern match.
<data>	specifies a pattern for which to search, up to 256 bytes long.
byte	specifies the data is a byte value.
word	specifies the data is a word value.
long, dword	specifies the data is a double word value.
<space>	for Intel emulators specifies smm, user (the default), or io address space. for Motorola CPU16 emulators specifies data (the default) or program. for Motorola CPU32 emulators specifies sp, sd (the default), up, ud, or cpu address space.

Search searches the specified address range for the described data pattern and returns the address of the match.

The physical read of memory uses the Size command settings rather than the format size set by the Search command. For example, if size=byte, Search reads memory in byte-sized memory accesses.

Because reading and writing memory takes a small amount of processor time, memory access (such as searching memory) is initially disabled during emulation. Use RunAccess to enable Search during emulation; however, such access can degrade your program execution.

```
Fill 0 ffff 0x0 user;
Write 400 0x1234 user;

Search 0 ffff 0x1234 user;
// pattern found at 400
```

You can also search for a pattern in memory with the Memory window Edit menu Search Memory item.

SetBase

Relocate symbols.

SetBase <base name> <address>

Related topics:

<base name>	is the base name for the symbols to be relocated.

AddressOf, DisplaySymbols, GetBase, NameOf, RemoveSymbols

Case is significant in specifying this parameter.

<address> numeric or symbolic address. This is an offset that is added to the address of each symbol contained in the base.

SetBase relocates the symbols in the specified <base name> to their offset address plus the specified <address>.

Each base has a base address; each symbol in a base is assigned an offset from the base address. Adding an amount to the base address increases the symbol addresses by that amount. Use SetBase to change the base address. The default base address is 0.

You can use SetBase to quickly relocate all symbols in a base. For example, if code is loaded by the target program into memory other than where it was linked, you can set the base address to the new load address using SetBase, thus matching the code symbol addresses to the memory where the code is loaded.

To discover the base names and their address offsets, use GetBase.

SetStackAlarm

Set the stack alarm limit.

Related topics: DisableAlarmLimit, DisableHighWaterMark, DisplayStack, EnableAlarmLimit, EnableHighWaterMark, FillStackPattern, SetStackArea, SetStackBase, SetStackSize, StackInfo

SetStackAlarm <percent>

<percent> is a percentage of the stack area, from 1 to 99.

Use SetStackAlarm to set the stack alarm limit as a percentage of the stack. The alarm appears as a red line on the stack meter in the Stack window.

With the stack alarm enabled, SLD notifies you when the stack usage is exceeding the stack alarm limit at the time the emulator halts.

You can also set the stack alarm with the Stack window Options menu Alarm Limit item.

SetStackArea

Redefine the stack location and size.

Related topics: DisableAlarmLimit, DisableHighWaterMark,

SetStackArea <address> <stack size> [fillArea]

<address> is the numeric or symbolic address for the base of the stack.

<stack size> is the stack size.

DisplayStack, EnableAlarmLimit, EnableHighWaterMark, FillStackPattern, SetStackAlarm, SetStackBase, SetStackSize, StackInfo

fillArea Initializes the stack area.

There are separate Shell commands to set the stack base and size. Since there is a delay between command executions, invoking the first command to change the value of the stack base or size can inadvertently define an invalid stack area. To avoid this problem, use SetStackArea to set both the stack base and the stack size with one command.

To show the current stack settings, use StackInfo. To fill the stack area with a pattern without changing the stack base and size, use FillStackPattern.

```
setstackarea 0x1000 0x500 fillarea;
```

You can also set the stack base and size with the Stack window Options menu Stack Area item.

SetStackBase

Set the stack base address.

Related topics: DisableAlarmLimit, DisableHighWaterMark, DisplayStack, EnableAlarmLimit, EnableHighWaterMark, FillStackPattern, SetStackAlarm, SetStackArea, SetStackSize, StackInfo

SetStackBase <address>

<address> is the numeric or symbolic address for the base of the stack.

You can set the stack base address separately from setting the stack size with SetStackBase.

There are separate Shell commands to set the stack base and size. Since there is a delay between command executions, invoking the first command to change the value of the stack base or size can inadvertently define an invalid stack area. To avoid this problem, use SetStackArea to set both the stack base and the stack size with one command.

To show the current stack settings, use StackInfo.

```
SetStackBase F000;
```

You can also set the stack base with the Stack window Options menu Stack Area item.

SetStackSize

Set the stack size.

Related topics: DisableAlarmLimit,

SetStackSize <stack size>

<stack size> is the stack size.

DisableHighWater-Mark, DisplayStack, EnableAlarmLimit, EnableHighWater-Mark, FillStackPattern, SetStackAlarm, SetStackArea, SetStackBase, StackInfo

You can set the amount of memory used by the stack separately from setting the stack base address with SetStackSize.

There are separate Shell commands to set the stack base and size. Since there is a delay between command executions, invoking the first command to change the value of the stack base or size can inadvertently define an invalid stack area. To avoid this problem, use SetStackArea to set both the stack base and the stack size with one command.

To show the current stack settings, use StackInfo.

```
SetStackSize 200;
```

You can also set the stack size with the Stack window Options menu Stack Area item.

Signal

Display or set the signal-enabled status.

Signal [[<signal name> [enable | disable]] | [all enable | all disable]]

Enabling or disabling a signal connects or disconnects, respectively, the signal between the CPU and the rest of the system. With no parameters are specified, Signal displays the status of all signals. To display the status of a particular signal, specify only <signal name>.

enable	connects the specified signal.
disable	disconnects the specified signal.
all enable	connects all signals.
all disable	disconnects all signals.
signal name	The signal name from the following list:

386DX, SX	RESET, READY#, NMI, INTR, HOLD, NA#, coprocessor signals
386CX	RESET, READY#, NMI, INTR, HOLD, NA#, SMI#, A20M#, coprocessor signals
386EX	RESET, READY#, NMI, INT0_3, INT4_7, HOLD, NA#, SMI#, coprocessor signals
Motorola 68360	RESET
Other Motorola	RESET, CLK

```
signal;
// CLK  DISABLE
// RESET  DISABLE
signal reset enable;
// RESET  ENABLE
```

You can also toggle the signal connections with the CPU window Options menu Signals item.

Size

Selects memory access size.

Related topics: Copy, Dump, Fill, RunAccess, Search, Verify, Write

Size [byte | word | long | dword]

Byte, word, long, and dword specify the size of subsequent memory accesses. The memory access size is independent of the display size.

You can also specify the memory access size from the Memory window Options menu.

StackInfo

Display the stack information.

Related topics: DisableAlarmLimit, DisableHighWaterMark, DisplayStack, EnableAlarmLimit, EnableHighWaterMark, FillStackPattern, SetStackAlarm, SetStackArea, SetStackBase, SetStackSize

StackInfo

This command displays the current calling stack information. The number of frames shows the call nesting level.

```
StackInfo;
// stack base = 12345678
// size = 0
// current stack pointer = 87654321
// frames = 0
// alarm limit = 0%, DISABLED
// high water mark = 00000000
// stack type = high to low
```

The same information appears in the Stack window.

StartTimer

Start the timer.

Related topics: LapTimer, StopTimer

StartTimer

This command resets the elapsed time to zero and starts the timer.

Step

Step emulation.

Related topics: $BREAKCAUSE System Variable, $EMULATING System Variable, Cause, Go, GoInto, GoUntil, Halt, ResetAndGo, StepSrc

Step [into | over] [<count>]

Step emulates one or more instructions in the target.

into	if a function call is encountered, steps into the function.
over	if a function call is encountered, the step executes the entire function (and any functions it calls) and stops on the instruction after the call.
<count>	specifies how many instructions to step. A large <count> can cause stepping to go for a long time. Press <ESC> to break out of stepping before the step count is finished.

The Source window Options menu Source Step Granularity item affects the Step operation. The <count> overrides the Source window Options menu Step Count specification.

You can also do these variations of "Step" with the Toolbar Step button, the Source window buttons, and the Source window Run menu.

StepMask

Mask interrupts during single stepping in a Motorola emulator.

StepMask [on | off]

on	masks interrupts.
off	allows interrupts.

Use StepMask in a Motorola emulator to prevent interrupts from interfering when you single-step through your code.

You can also mask interrupts with the Toolbar Configure menu Mask Interrupts For Step item.

StepSrc

Step emulation by source lines or statements.

Related topics: $BREAKCAUSE System Variable, $EMULATING System Variable, Cause, Go, GoInto, GoUntil, Halt,

StepSrc [into | over] [line | statement] [<count>]

into	if a function call is encountered, steps into the function.
over	if a function call is encountered, executes the entire function (and any functions it calls) and stops on the instruction after the call.
line	the step granularity is one source line. There can be more than one statement per source line. Lines can be out-of-

order relative to the sequence of instructions the compiler generates. For example, an execution sequence can be lines 33, 34, 31, 35.

statement the step granularity is one statement.

<count> specifies how many steps to go. A large <count> can cause stepping to go for a long time. Press <Esc> to stop stepping before the step count is finished.

Line or statement overrides the Source window Options menu Source Step Granularity specification.. The <count> overrides the Source window Options menu Step Count specification.

You can also do these variations of "Step" with the Toolbar Step button, the Source window buttons, and the Source window Run menu.

StopTimer

Stop and report on the timer.

Related topics: LapTimer, StartTimer

StopTimer

Stop the timer and return the number of milliseconds elapsed since the timer was started.

String

Discover whether a variable is a string.

Related topics: Integer

String (<variable>)

<variable> is the name of a Shell variable. The parentheses are required.

String returns true (1) if the variable is a string and false (0) otherwise.

```
$a = "qrs";
String($a);
// 0x1 1
if (string($a)) { "it is a string"; }
// it is a string
```

SymbolCloseFile

Close the symbol text file.

SymbolCloseFile

Closes the previously opened text file created by SymbolOpenFile.

SymbolOpenFile

Open a text file.

SymbolOpenFile <filename>

<filename> is the name of a file.

Opens a text file with the specified filename. Subsequent output from DisplaySymbols is directed to the specified file. The file can be viewed with an editor or file browser.

Test

Run the hardware confidence tests.

test [loop] [repeat | continue] [brief | verbose] [<test name> | <test number>]

Related topics: Ramtst

loop	repeats the low-level operations in the specified test so the operation can be observed on an oscilloscope. Press <Esc> to stop looping.
repeat	repeats the specified test until you press <Esc>.
continue	continues through all tests, even if one fails.
brief	displays only the final test result.
verbose	displays every test result and progress report.
<test name>	runs the test specified by name.
<test number>	runs the test specified by number.

With no parameters, Test runs all tests and displays the results.

The confidence tests are designed to run with the Stand-Alone Self-Test (SAST) board as the target.

Time

Display the date and time.

time

This command displays the date and time.

Transcript

Set the number of lines saved in the transcript pane.

Transcript [<size>]

<size> is the number of transcript lines to be saved, from 0 to 1000.

Related topics:

You can scroll the transcript pane of the Shell window.

History

You can also set the transcript size with the Options menu Set Transcript Size item.

TSS

Displays task state segments.

Related topics: dt, gdt, idt, ldt

TSS [<selector> | <register> | base <address> [limit <bytes>] [tss286 | tss386]] [all]

TSS displays the task state segments for any selector or base address. If you specify <register>, the selector for that register is used.

<selector>	specifies a single value (e.g., 0 or 0x08) used as the selector, referenced from the GDT. When no selector is specified, the tss_base and tss_limit are used.
<register>	specifies a register; the selector for the specified register is used.
base <address>	specifies the descriptor table base address.
[limit <bytes>]	If a base address is specified, you must also specify either <selector_range> or limit <bytes> to define the range to be displayed.
all	displays all task state segments plus the I/O bit map. Displays all entries, including invalid or reserved entries.
tss286	specifies Intel286 processor segmentation.
tss386	specifies Intel386 processor segmentation.

VarIndexCPU16Reg

Specify the registers used for index variables in Motorola CPU16 loadfiles.

Related topics: CompilerUsed

VarIndexCPU16Reg [none | xk:ix | yk:iy | zk:iz].

none	uses no register.
xk:ix	uses the xk:ix register.
yk:iy	uses the yk:iy register.
zk:iz	uses the zk:iz register.

The maximum address size for CPU16 is 16 bits. Some toolchains support 20-bit addressing for large memory model programs. For such programs, the additional four bits are assigned to a special register. Use VarIndexCPU16Reg before loading to inform the emulator which

register is used for 20-bit addressing in your loadfile.

Verify

Toggles on and off a read-after-write.

Related topics: Copy, Dump, Fill, RunAccess, Search, Size, Write

Verify [on | off]

on turns verify on (default).

off turns verify off.

With Verify on, write integrity is checked. If the byte read back does not match the byte written, an error is returned. Verification can happen after a Write, Fill, or Load. Verification does not affect the target processor during emulation.

You can also toggle write verification with the Memory window Options menu Write Verify item.

Version

Report the version of the emulator.

version

Use version when logging an emulator session to record which version of the emulator hardware, software, and firmware is in use. The information from this command is also needed when you contact Microtek for technical support.

You can also view some version information from any SLD window Help menu About item.

While

Repeatedly execute statements while the condition is true.

While (<condition>) { <statements> }

<condition> evaluates to true (non-zero) or false (zero). The parentheses are required.

<statements> is one or more Shell commands. The braces are required. Delimit commands with semicolons.

While <condition> is true, the <statement list> executes.

```
$a = 0;  While ($a < 500) {$a = $a + 1;}
```

Write

Write to an address.

Related topics: Copy, Dump, Fill, RunAccess, Search, Size, Verify

Write [loop] <address> <data> [byte | word | long | dword] [<space>]

loop	repeatedly preforms the operation but prints no output to the screen, even if errors occur.
<address>	specifies a numeric or symbolic address.
<data>	specifies up to 256 data values to write to memory starting at <address>.
byte	specifies the data is a byte value.
word	specifies the data is a word value.
long, dword	specifies the data is a double word value.
<space>	for Intel emulators specifies smm, user (the default), or io address space. for Motorola CPU16 emulators specifies data (the default) or program. for Motorola CPU32 emulators specifies sp, sd (the default), up, ud, or cpu address space.

The physical write to memory uses the Size command settings rather than the format size specified in the Write command. For example, if size=byte, Write commands write by byte-sized memory accesses.

Because reading and writing memory takes a small amount of processor time, memory access is initially disabled during emulation. Use RunAccess to enable Write during emulation; however, such access can degrade your program execution.

You can also edit memory in the Memory windows.

Xlt

Translates an Intel numeric address.

Related topics: AddressOf, NameOf

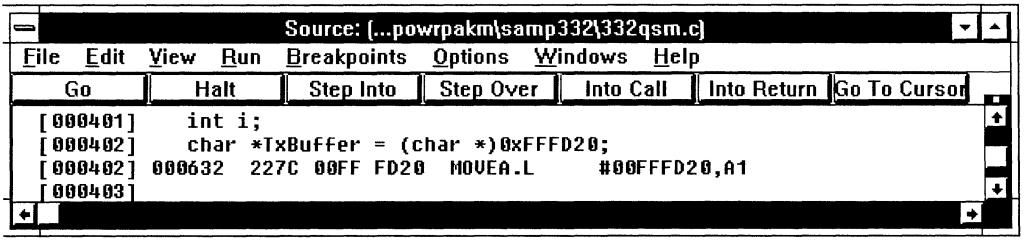
Xlt <address>

<address> is a numeric or symbolic address.

Xlt translates any numeric or symbolic address to its equivalent linear or physical form, according to Intel numeric addressing rules. For a virtual <address>, Xlt displays the linear and physical equivalents. For a linear or physical <address>, Xlt displays the physical equivalent.

Source Window Reference

The following figure shows a sample Source window.



This chapter describes the Source window contents, menus, buttons, and dialog boxes.

The Source window displays:

- When enabled, the source line numbers
- When available, the source (e.g. C or Assembly) from the source file
- When enabled, the disassembly corresponding to each source line, including the load address, hexadecimal code, and instruction

You can display two independently scrolling Source window panes. To reveal the second pane, drag the split box above the top arrow of the vertical scroll bar. When the mouse points to the split box, a split-box cursor (see figure at left) appears.

To change focus to a pane, click in the inactive pane or press <Tab>.

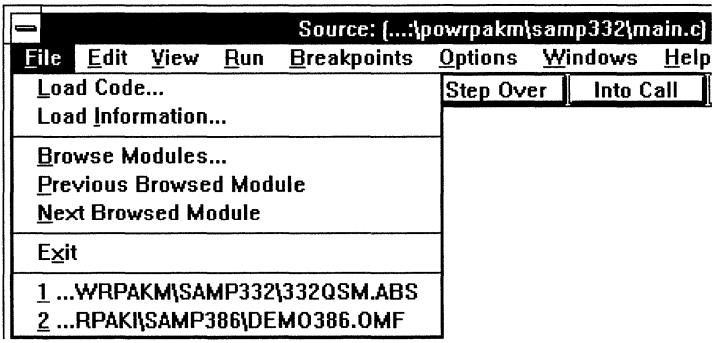
Source Window Menus

Menu	Use To:
File	Load; view loadfile information; display another module; close the Source window.
Edit	Navigate through source.
View	Configure the source and disassembly display.
Run	Start or stop emulation; step; reset.
Breakpoint	Define and manage breakpoints.
Options	Manage source display options and emulation controls.

Windows	Open another SLD window.
Help	Open a window for help on SLD commands.

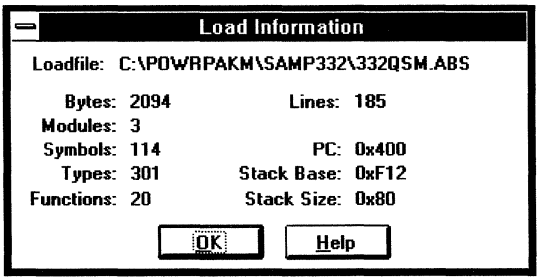
File Menu

The following figure shows a sample Source window File menu.

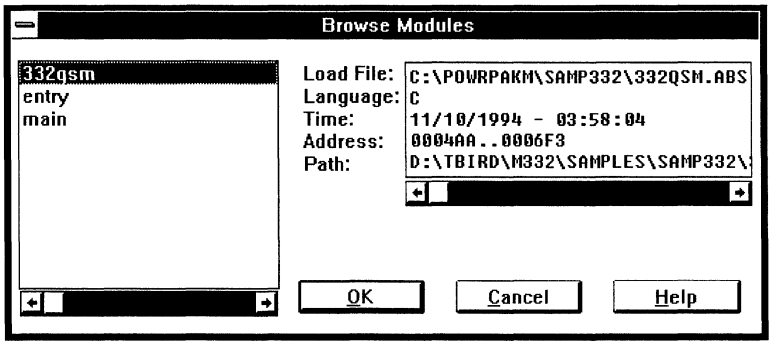


Load Code... opens the Load dialog box to load code or symbols from a loadfile. This has the same effect as choosing the Toolbar Load button, as described in the "Toolbar Reference" chapter. To reload a file, choose from the (up to four) files listed at the bottom of the Source window File menu.loading:Source window

Load Information... opens an information box describing the loadfile and what has been loaded into the emulator. The following figure shows a sample Load Information box for the Motorola 68332 emulator.



Browse Modules... opens a dialog box to change the module (source, disassembly, and symbols) displayed in the Source window. The following figure shows a sample Browse Modules dialog box.



To select a module, click on the module name or use the <Up Arrow> and <Down Arrow> keys to scroll the cursor. For the selected module, the dialog box displays:

Load File:	The loadfile path and filename
Language:	The language (e.g. C or Assembly) of the source file
Time:	The date and time the loadfile was created
Address:	Where in memory the module is loaded
Path:	The source file path and filename

Choose OK to browse to the selected module or Cancel to exit the dialog box without changing the Source window display.

Previous Browsed Module changes the Source window display back to the module you last viewed. SLD maintains a history list of which modules you have browsed and in what order you browsed them.

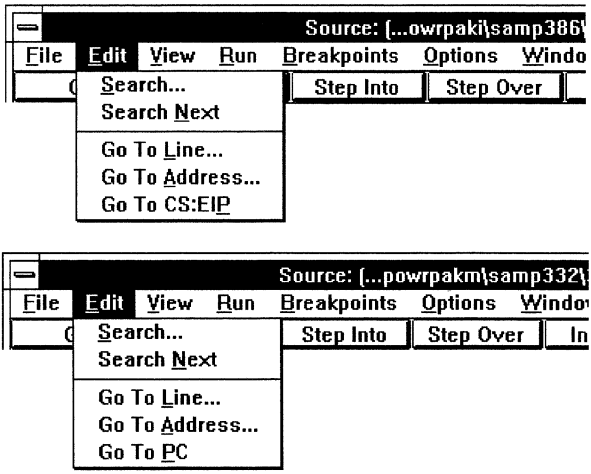
Next Browsed Module changes the Source window display to the next module in the browse history list.

Exit closes the Source window. To exit SLD, use Exit from the Toolbar File menu.

1, **2**, **3**, **4** lists the last four files you loaded. Reload a file by choosing it from this list. This method of reloading a file bypasses the Load and Load Options dialog boxes.

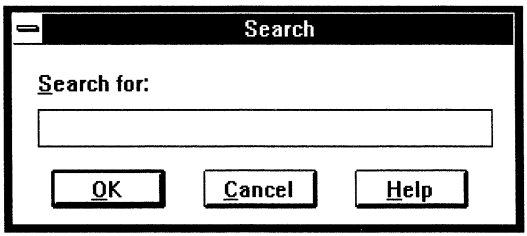
Edit Menu

The following figure shows two sample Edit menus. The first is for the Intel386 EX processor; the second is for the Motorola 68332 processor. Different menu items are available for different processors.



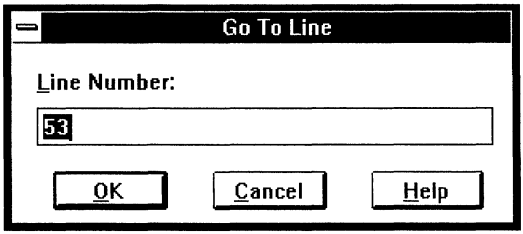
Search opens a dialog box for searching the Source window text for a specific string. Case is significant in the search string. The search starts from the Source cursor and stops at the first instance of the string found. If the string is not found, the search stops at the end of the module. To search the entire module, position the Source cursor at the beginning of the module before starting the search.

The following figure shows a Search dialog box.



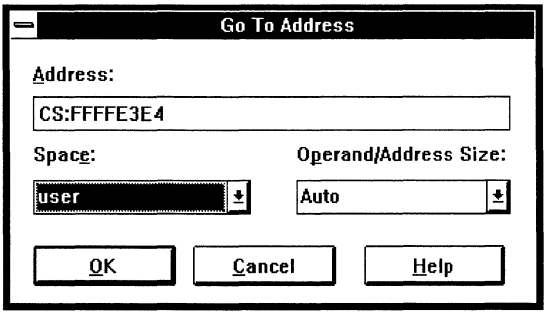
Search Next searches again for the last string you entered in the Search dialog box. The search starts from the cursor and stops at the first match or the end of the module.

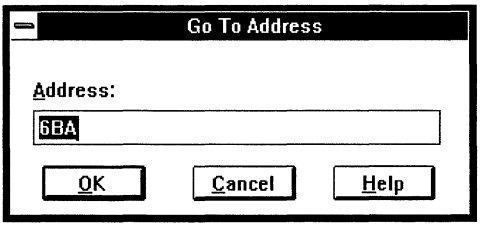
Go To Line... opens a dialog box to move the Source cursor to a specific line. If you specify a line number beyond the last line in the current module, the Source cursor moves to the end of the module. The following figure shows a Go To Line dialog box.



Go To Address... opens a dialog box to move the Source cursor to a specific address. If no source is available for the address you specify, the Source window shows disassembled code beginning at that address.

The following figure shows two sample Go To Address dialog boxes. The first is for the Intel386 EX processor; the second is for the Motorola 68332 processor. Different fields are available for different processors.





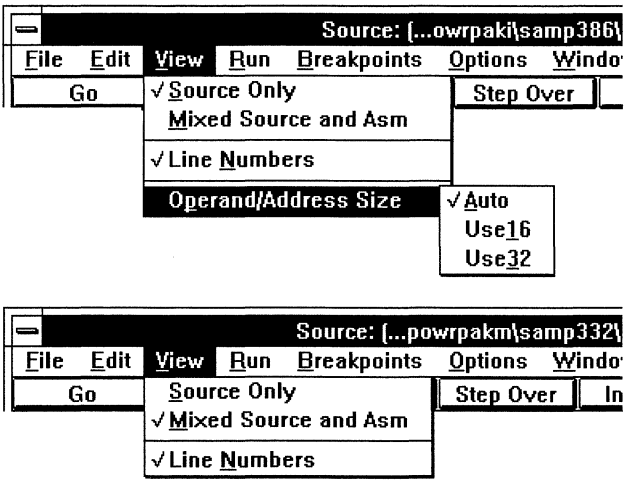
For Intel processors, you can specify:

Space:	as User or SMM (system management mode)
Operand/Address Size:	as Use16 (16-bit addressing mode), Use32 (32-bit addressing mode), or Auto (addressing mode derived from the pmode).

Go To CS:EIP (for Intel processors) or **Go To PC** (for Motorola processors) moves the Source cursor to the current program counter.

View Menu

The following figure shows two sample View menus. The first is for the Intel386 EX processor; the second is for the Motorola 68332 processor. Different menu items are available for different processors.



Source Only, when checked, displays only your source code.

Mixed Source and Asm, when checked, displays lines of disassembly from memory interleaved with the corresponding source code lines.

Line Numbers, when checked, displays your source file line numbers

Operand/Address Size, for Intel processors, opens a sub-menu with the following choices to display disassembly text:

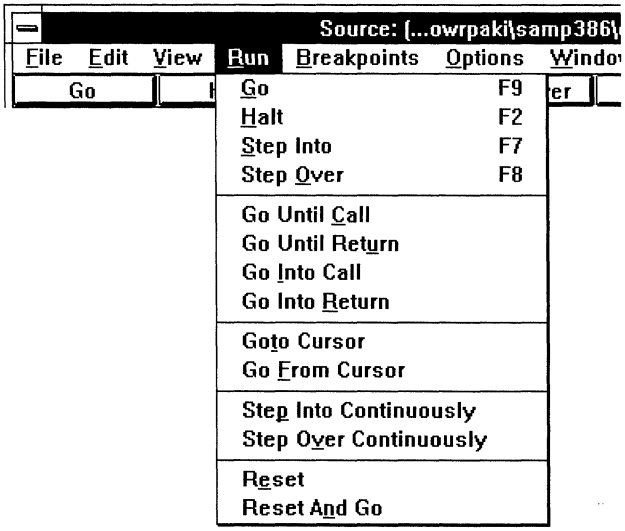
Auto Operand/address size is 16-bit or 32-bit, depending on the pmode.

Use16 Operand/address size is 16-bit.

Use32 Operand/address size is 32-bit.

Run Menu

The following figure shows a sample Run menu.



Go or pressing <F9> starts emulation.

Halt or pressing <F2> stops emulation.

Step Into or pressing <F7>, when the program counter is on a function call, executes the call to the function and stops before the first instruction in the function. The Source window displays the beginning of the function.

To step into a function with no associated source, before stepping open the View menu and check Mixed Source and Asm. Otherwise, Step Into operates the same as Step Over for that function.

Step Into and Step Over are indistinguishable from each other when the program counter is not on a function call.

Step Over or pressing <F8>, when the program counter is on a function call, executes the call as a single step. This step executes the function, returns, and stops before the first instruction following the return. (However, encountering a breakpoint in the stepped-over function stops emulation at the breakpoint.) The Source window continues to display the calling function.

Go Until Call executes from the program counter to the beginning of a statement or line (depending on the granularity) containing a function call.

Go Until Return executes from the program counter to the beginning of a statement or line (depending on the granularity) containing a return.

Go Into Call executes from the program counter and stops before the first instruction in the next called function.

Go Into Return execute from the program counter through the first return instruction, and stops before the first instruction after the return.

Go To Cursor executes from the program counter and stops before the selected (highlighted) line or statement in the Source window.

Go From Cursor moves the program counter to the selected (highlighted) line or statement in the Source window, then starts emulation.

Step Into Continuously does Step Into operations until you halt it.

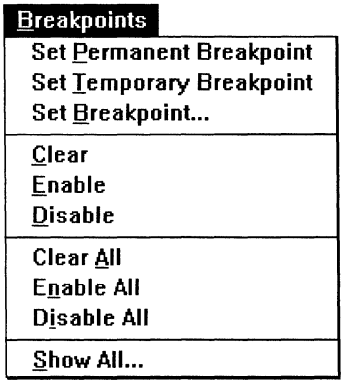
Step Over Continuously does Step Over operations until you halt it.

Reset asserts the RESET pin of the target processor, causing the CPU to reset its internal registers and to load the program counter and stack pointer from the reset vector locations. The RESET pin is then released. All SLD windows are updated; the Source window displays the beginning of code (where the program counter points) and the Stack window display is invalid.

Reset And Go does a Reset, as above, and starts emulation from the power-up reset vectors. To use Reset And Go, you must have the reset vectors set.

Breakpoints Menu

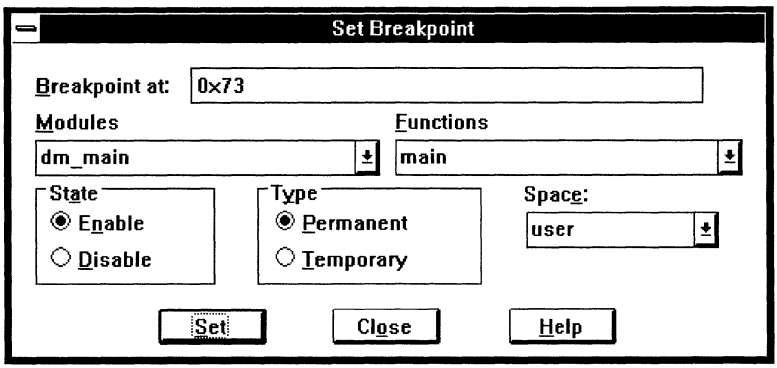
The following figure shows a sample Breakpoints menu. Set Permanent Breakpoint, Set Temporary Breakpoint, Set Breakpoint..., and Show All... are always available; Clear, Enable, and Disable are available when you have selected a breakpoint from those listed in the Breakpoint window; Clear All, Enable All, and Disable All are available when one or more breakpoints are listed. To select a breakpoint, click on it or use the <Up Arrow> and <Down Arrow> keys to move the highlight.

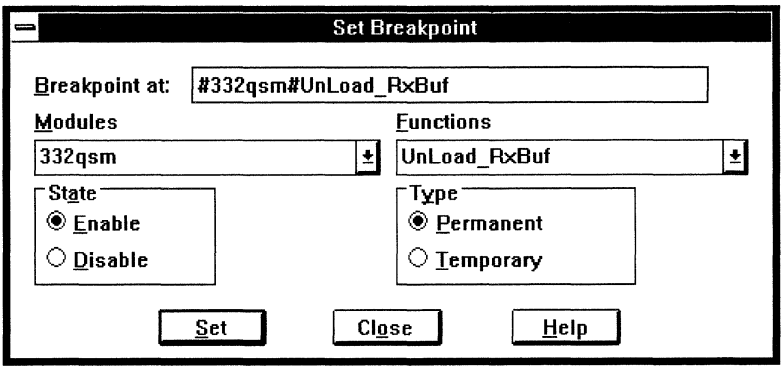


Set Permanent Breakpoint sets a permanent breakpoint at the Source cursor.

Set Temporary Breakpoint sets a temporary breakpoint at the Source cursor.

Set Breakpoint... opens a dialog box to set a breakpoint at a specific address. The following figure shows two sample Set Breakpoint dialog boxes. The first is for the Intel386 EX processor; the second is for the Motorola 68332 processor. Different fields are available for different processors.





Fill-in the dialog box as follows:

Breakpoint at:	can be a numeric or symbolic address. For symbolic addresses, choose a module and a function from the drop-down list boxes.
State	can be toggled to Enable or Disable. The emulator ignores a disabled breakpoint.
Type	can be permanent or temporary. A temporary breakpoint is removed after it causes the break.
Space:	for Intel processors, can be User or SMM.

Choose the Set button to define the breakpoint or the Close button to close the dialog box without defining a new breakpoint.

Clear removes a breakpoint at the Source cursor.

Disable marks the breakpoint at the Source cursor to be ignored when emulation executes through the code where the breakpoint is located. A disabled breakpoint highlight in the Source window is grey.

Enable marks the breakpoint at the Source cursor to cause a break when emulation executes through the code where the breakpoint is located. An enabled breakpoint highlight in the Source window is red.

Disable All disables all currently defined breakpoints. The breakpoints remain defined.

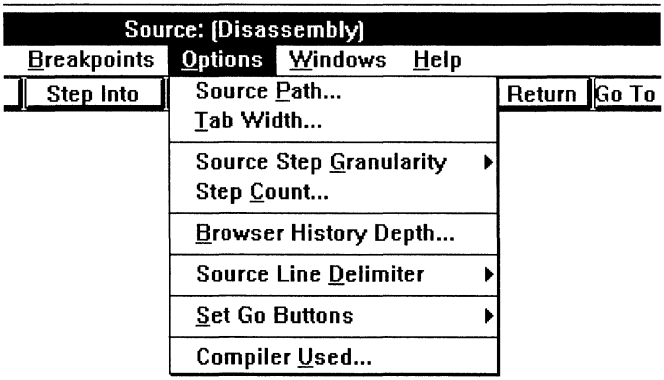
Enable All enables all currently defined breakpoints.

Clear All removes all breakpoints. No breakpoints remain defined.

Show All... opens the Breakpoint window, described in the Breakpoint Window Reference chapter.

Options Menu

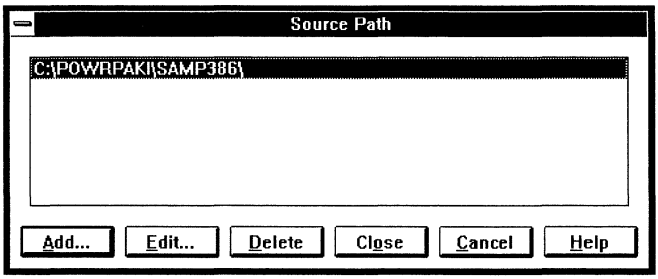
The following figure shows a sample Options menu for the Motorola 68332 processor. Different menu items are available for different processors.



Source Path opens a dialog box to add, delete, or change the paths to the source files used in generating your loadfile. You can define up to 50 source paths. The paths are saved in powerpak.ini for the next time you run SLD.

When you browse a module in the Source window, the emulator searches the source paths for the corresponding source file in the order they appear in the dialog box, from top to bottom.

The following figure shows a sample Source Path dialog box.

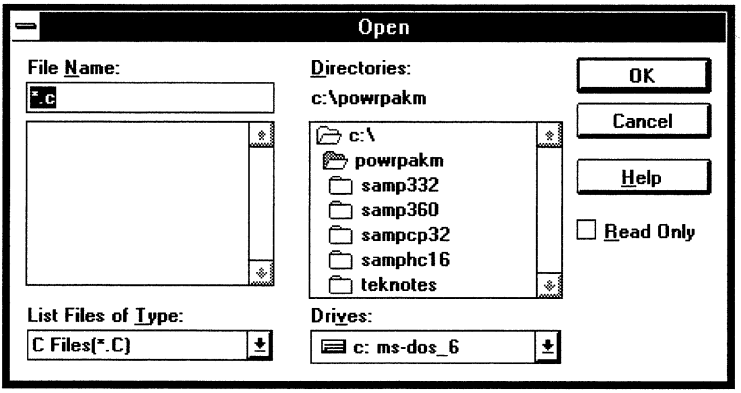


To select a source path for editing or deleting, click on it or use the <Up Arrow> and <Down Arrow> keys to move the highlight.

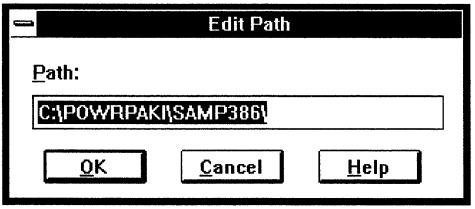
The Source Path dialog box buttons are:

Add... opens a dialog box for adding a new source path to the emulator's list of source paths. The following figure shows a sample Open dialog box. Select a source file; choose OK to

add the directory to the source path list or Cancel to close the dialog box without adding the path.



Edit... opens a dialog box for editing the selected source path. The following figure shows a sample Edit Path dialog box.

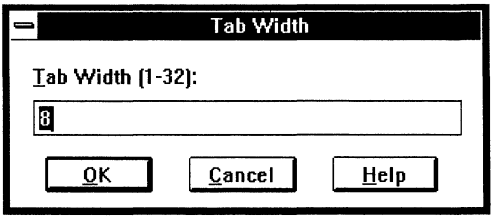


Delete removes the selected path from the emulator's list of source paths.

Close closes the Source Path dialog box, automatically keeping all Add, Edit, and Delete changes you have made.

Cancel closes the Source Path dialog box, first asking you to confirm whether to keep or abandon the Add, Edit, and Delete changes you have made.

Tab Width... opens a dialog box to specify the number of spaces the Source window uses to replace a tab character in your source file. The default tab width is eight spaces. The following figure shows a sample Tab Width dialog box.



Tab Width And Statement-Level Breakpoints

To set a breakpoint at the statement level, you must know how many spaces your compiler uses for a tab character. For example:

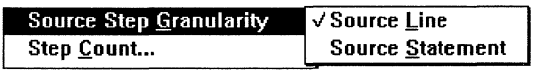
<tab><tab>for(i = 0; i < MAX_NUM; I++){ /*source line*/

The compiler generates column range information for the three statements in this line, using a tab width of 8:

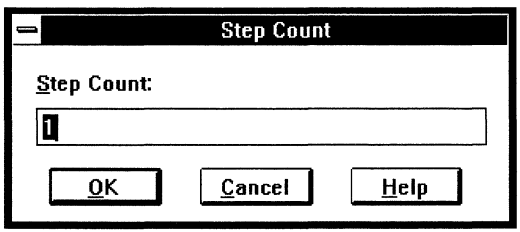
i = 0	columns 0 to 26
i < MAX_NUM	columns 27 to 39
i++	columns 40 to 45

If you set the Source window Tab Width to 4, then use the Source cursor to set a breakpoint on the first i (column 13) or the second i (column 20), the breakpoint is within the first statement's column range. The third i is within the second statement's range.

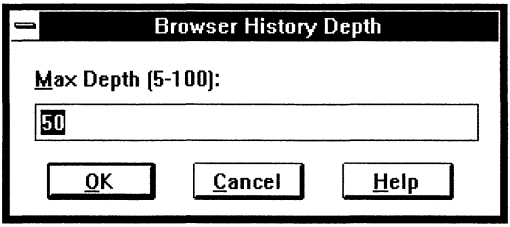
Source Step Granularity opens a sub-menu to specify whether a Step command steps by source lines (the default) or by source statements. Some C compilers allow more than one statement per line, separated by semicolons. You can step through such a source line by statements. The following figure shows a sample Source Step Granularity sub-menu, with stepping by line specified.



Step Count opens a dialog box to set how many steps (1 to 100) are executed per Step command. The following figure shows a sample Step Count dialog box.



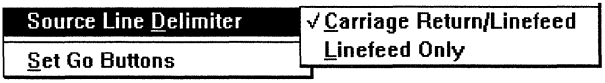
Browser History Depth opens a dialog box to set the maximum number of modules that can be recalled. SLD remembers the sequence of modules and functions you have browsed. The following figure shows a sample Browser History Depth dialog box.



Previous Browsed Module displays the next earlier module in your browse history.

Next Browsed Module displays the next later module in your browse history.

Source Line Delimiter opens a sub-menu to set the ASCII string used by the compiler to delimit a source line. The following figure shows a sample Source Line Delimiter sub-menu toggled for displaying a DOS source file.

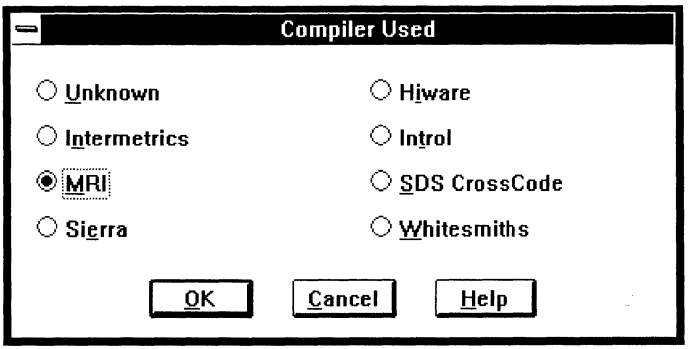


Carriage Return/Linefeed (the default) recognizes a carriage return followed by a linefeed as the string indicating the end of a line. This is the DOS standard line delimiter. If you display a UNIX file with Carriage Return/Linefeed, the entire source file appears as a single line in the Source window.

Linefeed Only recognizes a linefeed as the end-of-line indicator. This is the UNIX standard line delimiter. If you display a DOS source file with Linefeed Only, a black dot appears at the end of each line.

Set Go Buttons opens a sub-menu to toggle the operation of the Call and Return buttons (described later in this chapter) between Go Until and Go Into. The following figure shows a sample Set Go Buttons sub-menu, followed by the two possible button combinations. The check on Into Call/Return in the sub-menu corresponds to the Into Call and Into Return buttons shown in the first button bar configuration.

Compiler Used... opens a dialog box to identify the toolchain you used in generating your loadfile. The following figure shows a Compiler Used dialog box.

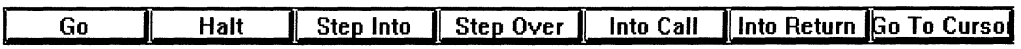


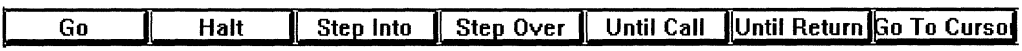
If your compiler is not listed in the dialog box, choose Unknown. The emulator is not guaranteed to work with unsupported toolchains.

Source Window Buttons

These buttons provide quick access to commonly used Run menu items, described earlier in this chapter.

The Source window button bar has two possible configurations. To toggle between them, open the Options menu, choose Set Go Buttons, and choose Until Call/Return or Into Call/Return. The following figure shows the two possible button bar configurations.



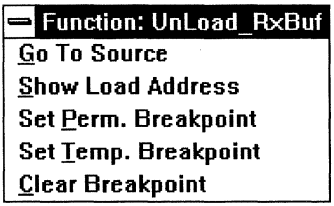


Button	Use To:
Go	Start emulation from the program counter, the same as the Run menu Go.
Halt	Stop emulation, the same as the Run menu Halt.
Step Into	Step into a function call at the program counter, the

	same as the Run menu Step Into.
Step Over	Step over a function at the program counter, the same as the Run menu Step Over.
Until Call	Go from the program counter and break before the next function call, the same as the Run menu Go Until Call.
Into Call	Go from the program counter and break after the next function call, before executing the function, the same as the Run menu Go Into Call.
Until Return	Go from the program counter and break before the next return instruction, the same as the Run menu Go Until Return.
Into Return	Go from the program counter and break after the next return instruction, the same as the Run menu Go Into Return.

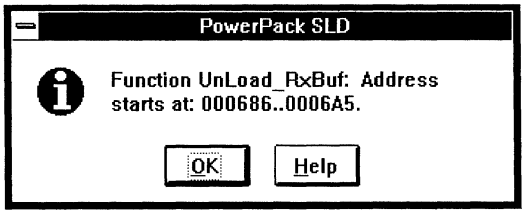
Function Popup Menu

To pop-up the Function menu, select (double-click on) a function name in the source. The selected function name is highlighted. The following figure shows a sample Function menu.



Go To Source puts the Source cursor at the beginning of the function source code. If no source is available, the Source window can display the function in disassembly. To enable the disassembly display, open the View menu and choose Mixed Source and Asm.

Show Load Address opens an information box listing the memory address range occupied by the function. The following figure shows a sample load address information box.



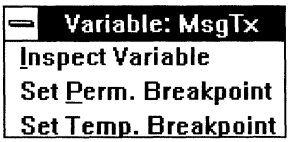
Set Perm. Breakpoint sets a permanent breakpoint at the highlight.

Set Temp. Breakpoint sets a temporary breakpoint at the highlight.

Clear Breakpoint clears the breakpoint at the highlight.

Variable Popup Menu

To pop-up the Variable menu, select (double-click on) a variable name in the source. The selected variable name is highlighted. The following figure shows a sample Variable menu.



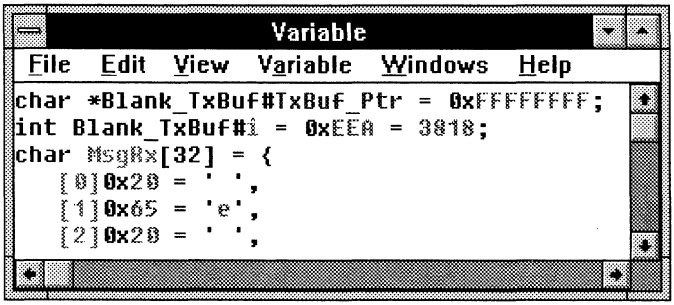
Inspect Variable adds the variable to the Variable window, described in the Variable Window Reference chapter. If the Variable window is not already open, this opens it.

Set Perm. Breakpoint sets a permanent breakpoint on the highlight.

Set Temp. Breakpoint sets a temporary breakpoint on the highlight.

Variable Window Reference

The following figure shows a sample Variable window.



This chapter describes the Variable window contents, menus, and dialog boxes.

Variable Window Contents

The Variable window displays the types, symbolic names, and values of global and local variables. Variable symbolic information appears in the following colors:

Red — indicates an editable value. Integer variables can be edited in hexadecimal or decimal, floating point variables in floating point format, and characters in their hexadecimal ASCII equivalent. To edit a value, either double-click on the value; or single-click on the value, open the Edit menu, and choose Edit. Press <Enter> to end editing.

Blue — indicates a pointer variable you can dereference by double clicking. To dereference a pointer, either double click on the pointer name or open the View menu and choose Show. A new entry is added to the Variable window, showing the variable that was pointed to.

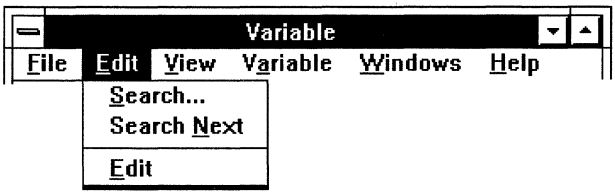
Magenta — indicates a non-pointer variable. For enum type variables, the enumerated name follows the hexadecimal value.

Variable Window Menus

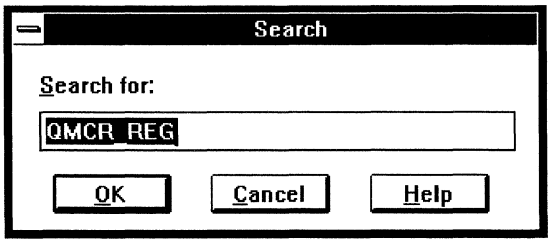
Menu	Use To:
File	Close the Variable window.
Edit	Find and edit a listed variable.
View	Reorganize or refresh the display.
Variable	Add or remove variables from the display.
Windows	Open another SLD window.
Help	Open a window for help with SLD.

Edit Menu

The following shows the Edit menu.

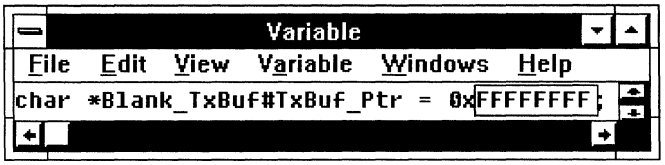


Search... opens a dialog box to find any variable listed in the Variable window. The search is case sensitive and stops at the first occurrence or at the end of the Variable window. The following figure shows a sample Search dialog box.



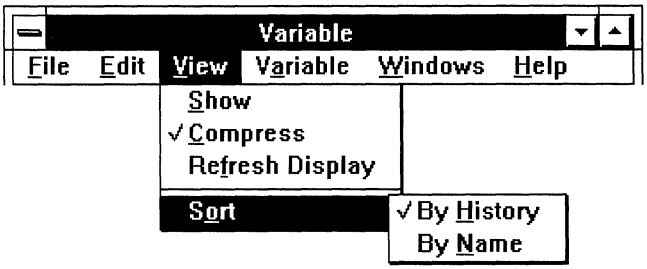
Search Next finds the next occurrence of the last variable searched for.

Edit positions an edit field on the selected value. This item is available when you put the Variable cursor on an editable (red) value. Type the new value in the edit field and press <Enter>. Floating-point numbers use floating-point format. Characters use hexadecimal or ASCII format. Integers use decimal or hexadecimal. The following shows an edit field.

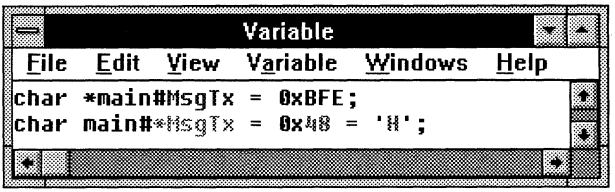


View Menu

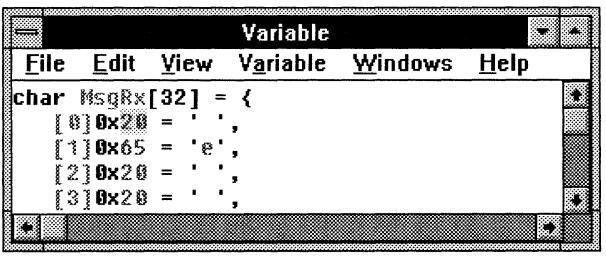
The following shows the View menu.

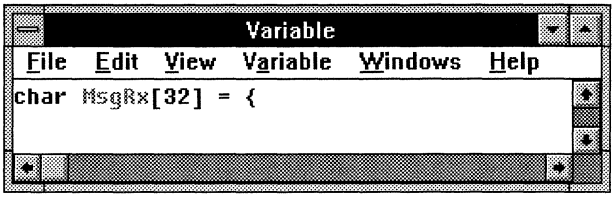


Show adds a line to the Variable window dereferencing the selected variable. This item is available when you have put the Variable cursor on a dereferenceable (blue) symbol, such as a pointer. The following figure shows a pointer and its dereferenced equivalent.



Compress collapses multi-line variables, such as an array or structure, to show only the first line of the variable. The following shows an array, first in expanded (only the first four of the 32 array elements appear in this picture) then in compressed display.





Refresh Display updates the displayed symbols and values.

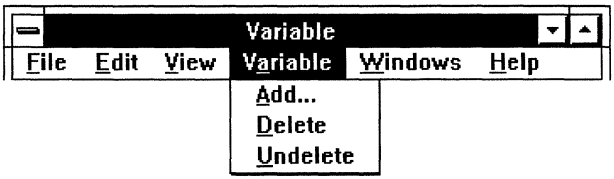
Sort opens a sub-menu to arrange the variables:

By History in the order they were added to the display.

By Variable Name alphabetically.

Variable Menu

The following shows a Variable menu:



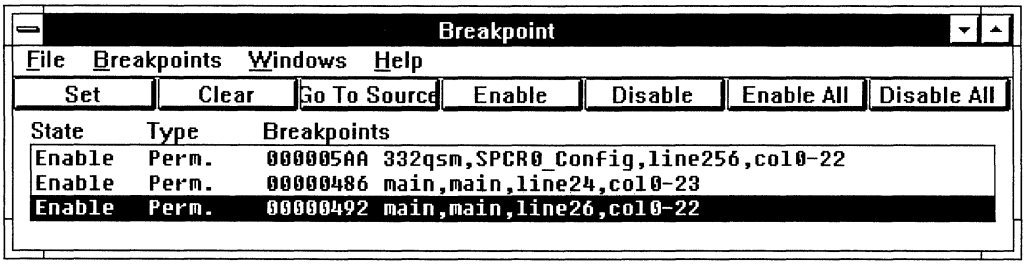
Add... opens a dialog box to add a variable to the window. You can specify a partly or fully qualified variable name.

Delete removes the selected variable from the display.

Undelete restores to the display the last variable removed.

Breakpoint Window Reference

The following figure shows a sample Breakpoint window for a Motorola emulator. The address format is different for Intel emulators; however, the window layout is consistent.



This chapter describes the Breakpoint window contents, menus, buttons, and dialog boxes.

The Breakpoint window displays the following information about each breakpoint:

State	Whether the breakpoint will cause a break (Enable) or not (Disable) when emulation executes through the code where the breakpoint is located.
Type	Whether the breakpoint will remain defined (Perm.) or be removed (Temp.) after causing a break.
Breakpoints	The load address, module name, function name, source line number, and source column number where the breakpoint is located. (The column number can be affected by the number of spaces your compiler uses to replace a tab character.)tab width

Breakpoint Window Menus

Menu	**Use To:**
File	Exit the Breakpoint window.
Breakpoints	Define, remove, enable, and disable breakpoints.
Windows	Open another SLD window.
Help	Open a window for help with SLD.

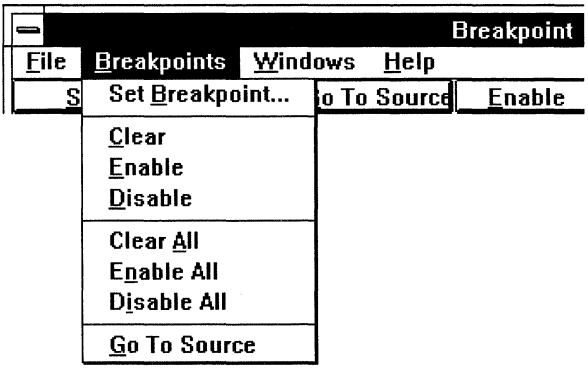
File Menu

Exit closes the Breakpoint window.

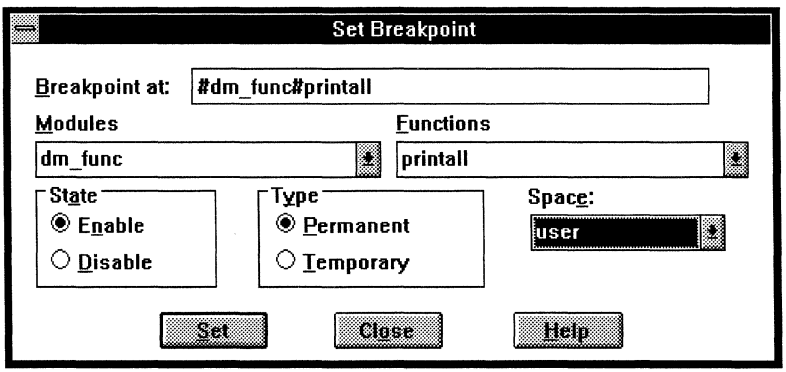
Breakpoints Menu

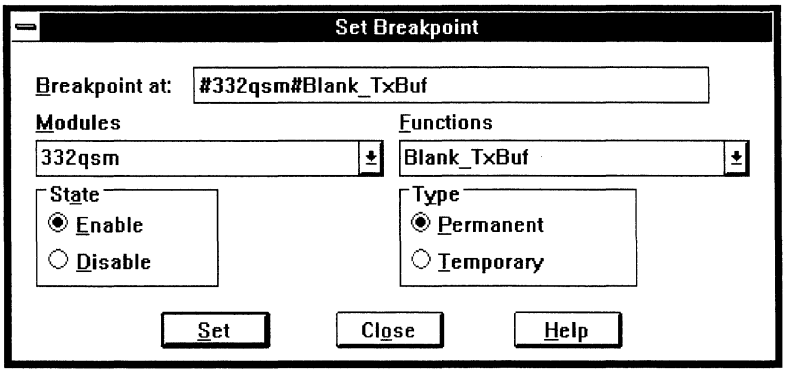
The items available in the Breakpoints menu depend on whether breakpoints are defined and selected. Set Breakpoint... and Go To Source are always available; Clear, Enable, and Disable are available when you have selected a breakpoint from those listed in the Breakpoint window; Clear All, Enable All, and Disable All are available when one or more breakpoints are listed. To select a breakpoint, click on it or use the <Up Arrow> and <Down Arrow> keys to move the highlight.

The following shows a breakpoint menu.



Set Breakpoint opens a dialog box to define a new breakpoint. The following figure shows two sample Set Breakpoint dialog boxes. The first is for an Intel emulator; the second is for a Motorola emulator. Different fields are available for different processors.





Fill-in the dialog box as follows:

Breakpoint at: can be a numeric or symbolic address. For symbolic addresses, you can choose a module and a function from the drop-down list boxes.

State can be toggled to Enable or Disable. The emulator ignores a disabled breakpoint.

Type can be permanent or temporary. A temporary breakpoint is removed after it causes the break.

Space: for Intel processors, can be User or SMM.

Choose the Set button to define the breakpoint or the Close button to close the dialog box without defining a new breakpoint.

Clear removes the selected breakpoint.

Disable marks the selected breakpoint to be ignored when emulation executes through the code where the breakpoint is located.

Enable marks the selected breakpoint to cause a break when emulation executes through the code where the breakpoint is located.

Disable All disables all currently defined breakpoints. The breakpoints remain defined.

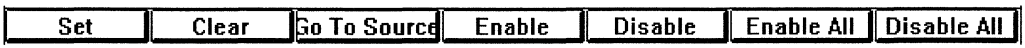
Enable All enables all currently defined breakpoints.

Clear All removes all breakpoints. No breakpoints remain defined.

Go to Source opens the Source window, described in the "Source Window Reference" chapter, and positions the source cursor at the specified breakpoint.

Breakpoint Window Buttons

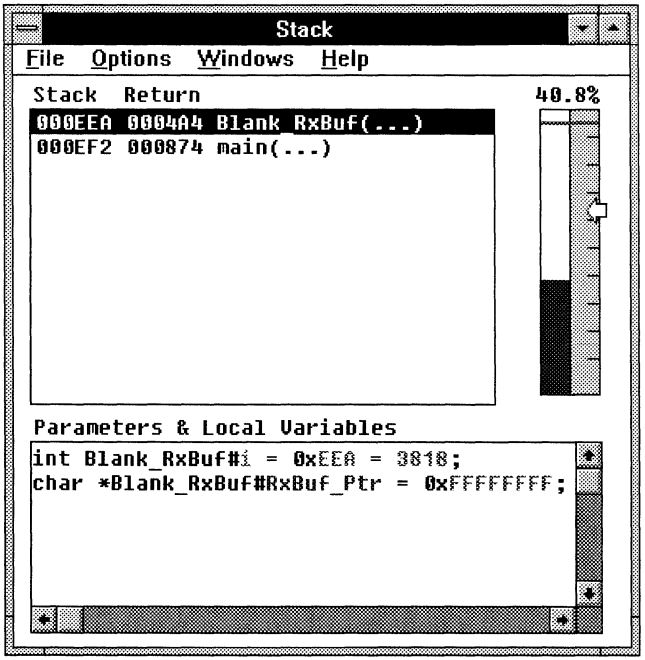
These buttons provide quick access to commonly used Breakpoints menu items, described earlier in this chapter.



Button	Use To:
Set	Open a dialog box to set a breakpoint, the same as the Breakpoints menu Set Breakpoint...
Clear	Remove a selected breakpoint, the same as the Breakpoints menu Clear.
Go To Source	Open the Source window to show the specified breakpoint in source or disassembly, the same as the Breakpoints menu Go To Source.
Enable	Define that the specified breakpoint will cause a break next time it is encountered in emulation, the same as the Breakpoints menu Enable.
Disable	Define that the specified breakpoint will cause no break next time it is encountered in emulation, the same as the Breakpoints menu Disable.
Enable All	Enable all breakpoints, the same as the Breakpoints menu Enable All.
Disable All	Disable all breakpoints, the same as the Breakpoints menu Disable All.

Stack Window Reference

The following figure shows a sample Stack window for a Motorola emulator. The address formats are different for Intel emulators; however, the window layout is consistent..



This chapter describes the Stack window contents, menus, and dialog boxes.

The Stack window has three panes:

The top pane (Frame List)	lists the stack address, the return address, and the name of each function on the current call stack. Each such item is a stack frame.
Parameters and Local Variables	lists the type, name, and value of each parameter and local variable in the selected stack frame. The format and colors are the same as in the Variable window.
Stack Meter	shows the stack usage statistics, including the percent of the stack area currently in use, an alarm marker at a specified usage level, and a mark at the highest percent usage for the current emulation

session. Yellow indicates stack underflow. Purple indicates stack overflow.

Stack Window Menus

Menu	Use To:
File	Close the Stack window; refresh the stack display.
Options	Configure the stack area; toggle the Frame List address display; manage stack usage statistics; inspect the source.
Windows	Open another SLD window.
Help	Open a window for help on SLD.

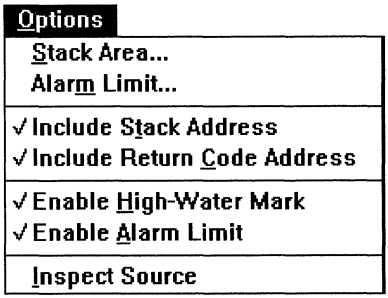
File Menu

Refresh Display reads memory and updates the displayed information.

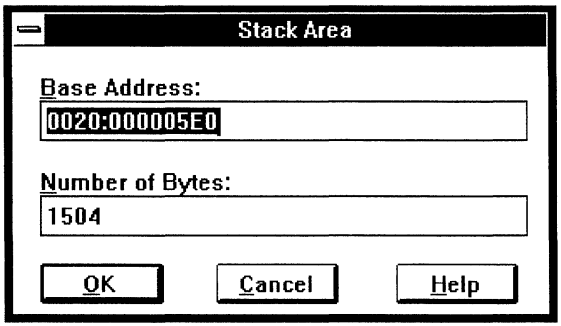
Exit closes the Stack window.

Options Menu

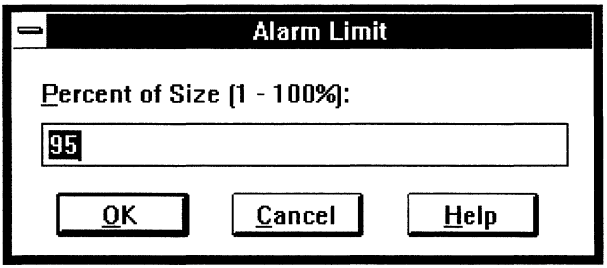
The following shows a sample Options menu.



Stack Area... opens a dialog box to set the stack base address and size. The following shows a sample Stack Area dialog box.



Alarm Limit... opens a dialog box to define the alarm limit as a percentage (1 to 100) of the Stack Meter. The following shows a sample Alarm Limit dialog box.



Include Stack Address, when checked, displays stack addresses in the Frame List, in a column labeled Stack. The stack address is the address of the frame in the stack area.

Include Return Code Address, when checked, displays code addresses in the Frame List, in a column labeled Return. The code address is the return address to the calling function.

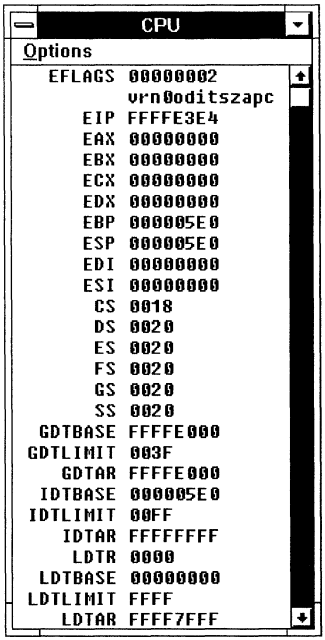
Enable High Water Mark, when checked, displays the high-water mark on the Stack Meter. The high-water mark indicates the highest percentage that has been used of the stack area.

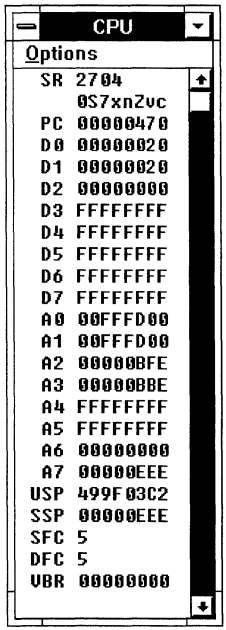
Enable Alarm Limit displays a warning message each time emulation stops while the alarm limit is exceeded.

Inspect Source opens the Source window, described in the "Source Window Reference" chapter, and positions the Source cursor to show the selected function's source. To select a function, in the Frame List click on the frame or use the <Up Arrow> and <Down Arrow> keys to move the highlight.

CPU Window Reference

The following figure shows two sample CPU windows. The first is for an Intel386 EX processor; the second is for a Motorola 68332 processor. Different registers are shown for different processors.

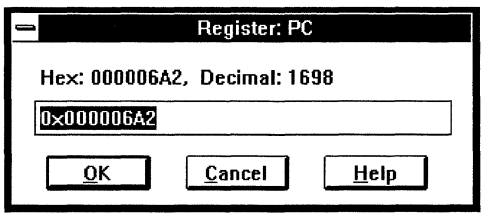




This chapter describes the CPU window contents, menu, and Register Edit dialog box.

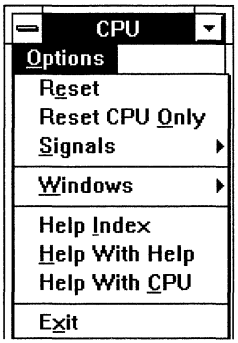
The CPU window lists the processor registers. The register mnemonics conform to the Intel or Motorola mnemonics. The register values are updated and the changed values highlighted each time emulation halts.

To edit the register values, double-click on a register value; or use the <Up Arrow> and <Down Arrow> to move the highlight then press <Enter>. The following is a sample Register Edit dialog box.



Options Menu

The following is a sample Options menu.

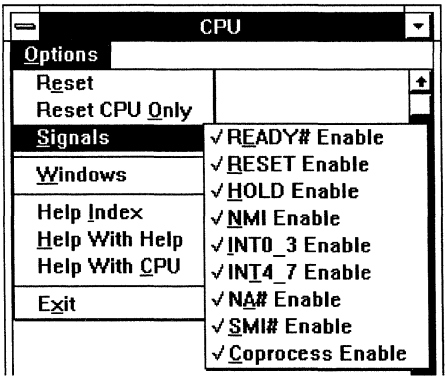


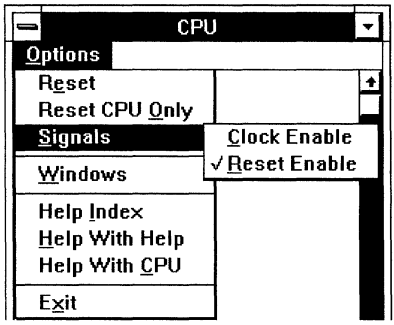
Reset resets and reinitializes the target processor:

- The processor RESET pin is asserted.
- The program counter is read from memory; the Source window is scrolled to the beginning of code.
- The stack pointer is read from memory, resetting the stack; the Stack window display becomes invalid.
- All SLD windows are updated.

Reset CPU Only resets only the processor and does not update the windows. Use Reset CPU Only if Reset fails to reset the processor.

Signals opens a sub-menu to specify whether certain signals are controlled by the target (unchecked) or by the emulator (checked). The following figure shows two Signals sub-menus. The first is for an Intel386 EX processor; the second is for a Motorola 68332 processor. Different signals can be enabled for different processors.





Windows opens a sub-menu to open another SLD window. This item is equivalent to the Windows menu in other SLD windows.

Help Index opens a window with the table of contents for SLD help.

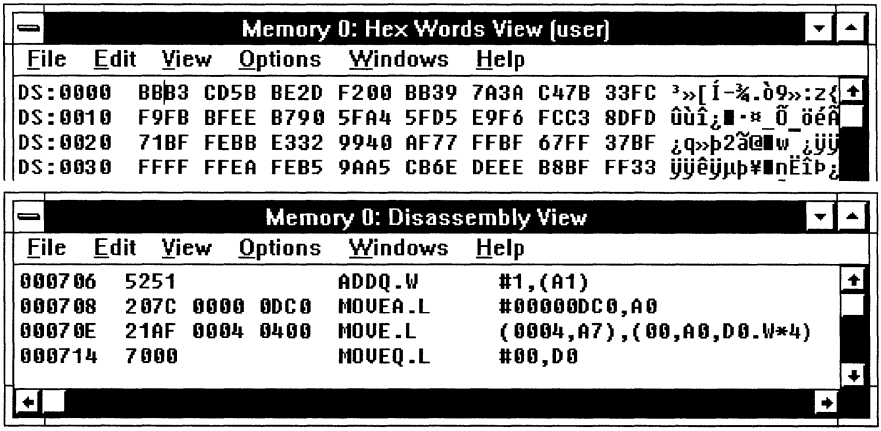
Help With Help opens a window on using a Windows help facility.

Help With CPU opens a window with SLD CPU window help.

Exit closes the CPU window.

Memory Window Reference

The following figure shows two sample Memory windows. The first is for an Intel386 EX processor; the second is for a Motorola 68332 processor. Different addresses and disassembly mnemonics are shown for different processors.



This chapter describes the Memory window contents, menus, and dialog boxes.

The Memory window shows the contents of memory:

- The window title lists which of up to 20 Memory windows you are viewing; the format of the display; and (for Intel processors) whether the display is of User or SMM space. Different Memory windows can display different areas or formats of memory.
- The leftmost column is the address. Address formats differ for different processors. To view another area of memory, double-click in the address column of the Memory window. Enter a numeric or symbolic address in the Go To Address dialog box. Any symbol you enter must have a fixed address, i.e., not a local variable or a stack-resident parameter.
- The memory contents can be in disassembly or numeric format. Numeric format shows the hexadecimal or decimal values and, in the rightmost column, the equivalent ASCII values. You can edit memory contents directly in the numeric and ASCII formats by positioning the cursor (a vertical bar) with the mouse, then overtyping the memory display. Disassembly format can include

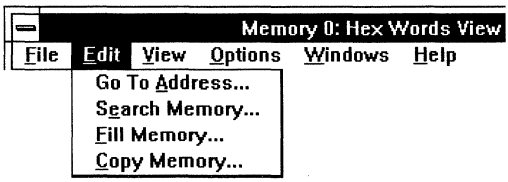
symbols; on the Toolbar open the Configure menu and check or uncheck Symbolic Disassembly.

Memory Window Menus

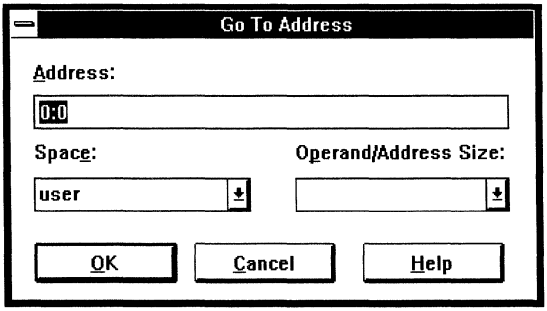
Menu	Use To:
File	Exit the Memory window.
Edit	Edit memory; navigate the memory display.
View	Choose numeric or disassembly display formats.
Options	Manage memory access options.
Windows	Open another SLD window.
Help	Open a window for help with SLD.

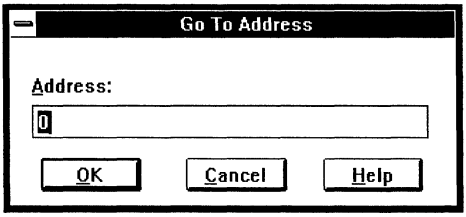
Edit Menu

The following is a sample Edit menu.



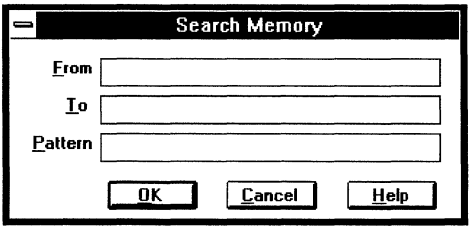
Go To Address... opens a dialog box to change the Memory window display to a specified numeric or symbolic address. The following figure shows two sample Go To Address dialog boxes. The first is for an Intel386 EX processor; the second is for a Motorola 68332 processor. Different fields are available for different processors.



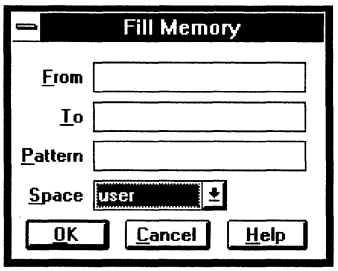


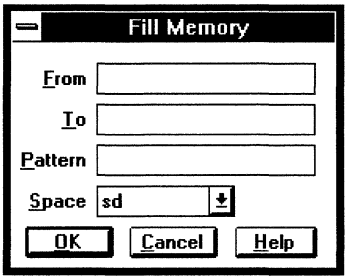
You can fill-in a numeric or symbolic address. For Intel processors, you can also specify User or SMM space and what addressing mode to use.

Search Memory... opens a dialog to search a specified address range for a specified pattern. The search stops at the first occurrence of the pattern in the range. If the pattern is not found, the Memory cursor does not move. The following figure shows a sample Search Memory dialog box.

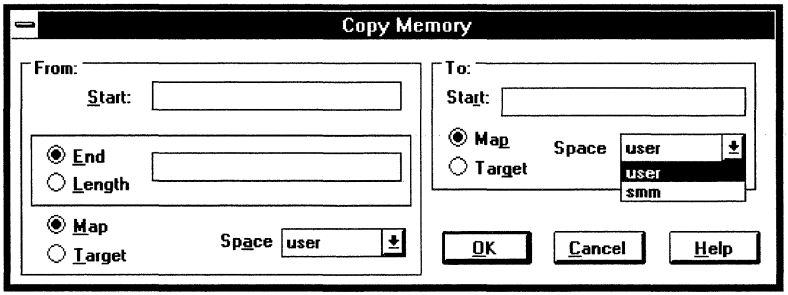


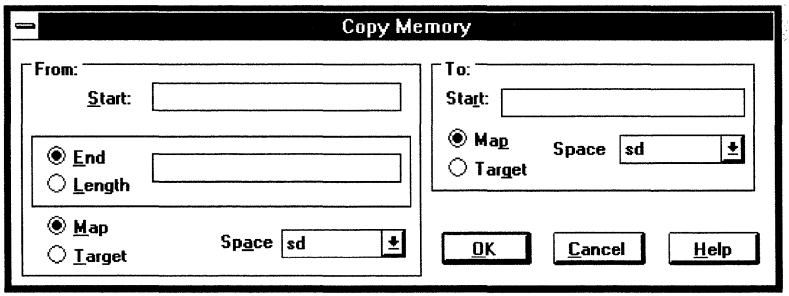
Fill Memory... opens a dialog box to fill an address range with a specified pattern. The following figure shows two sample Fill Memory dialog boxes. The first is for an Intel386 EX processor; the second for a Motorola 68332 processor. The Space field values vary.





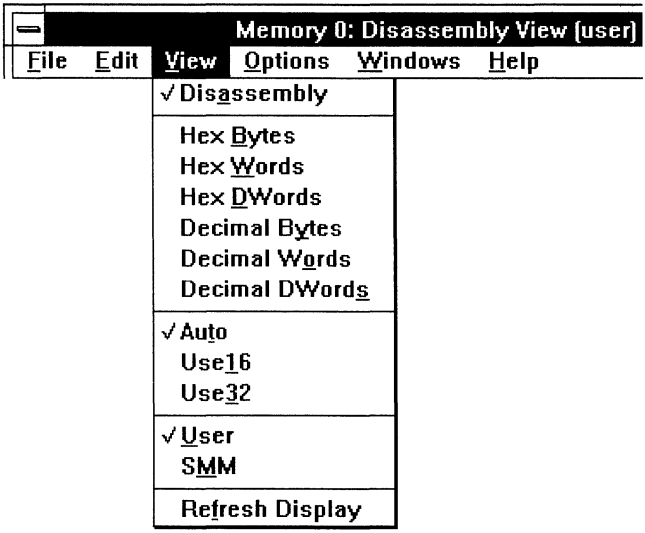
Copy Memory... opens a dialog box to copy one address range to another or to copy target memory to overlay memory. The following figure shows two sample Copy Memory dialog boxes. The first is for an Intel386 EX processor; the second is for a Motorola 68332 processor. Different space field values are available for different processors.

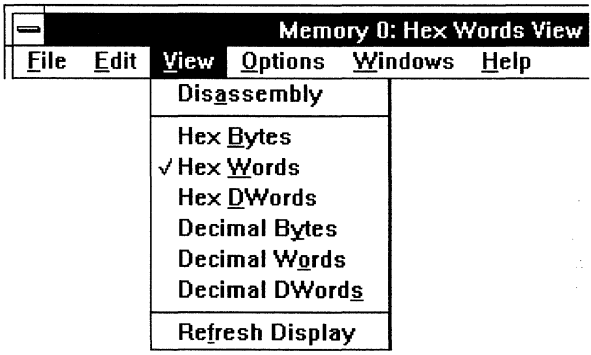




View Menu

The following figure shows two sample View menus. The first is for an Intel386 EX processor; the second is for a Motorola 68332 processor. Different items are available for different processors.





Disassembly displays memory disassembled. In Disassembly view, you can double-click on a disassembled line to open the Single Line Assembler dialog box (described later in this chapter).

Hex Bytes displays memory as hexadecimal 8-bit integers with values from 0 to FF.

Hex Words displays memory as hexadecimal 16-bit integers with values from 0 to FFFF.

Hex Dwords displays memory as hexadecimal 32-bit integers with values from 0 to FFFFFFFF.

Decimal Bytes displays memory as decimal 8-bit integers with values from 0 to 255.

Decimal Words displays memory as decimal 16-bit integers with values from 0 to 65,535.

Decimal DWords displays memory as decimal 32-bit integers with values from 0 to 4,294,967,295.

Auto uses the Intel386 processor pmode to determine whether operands and addresses are interpreted as 16-bit or 32-bit values. For a description of the pmodes, see the section on Intel numeric addresses in the "Debugging with Triggers and Trace" chapter.

Use16 interprets Intel386 operands and addresses as 16-bit values.

Use32 interprets Intel386 operands and addresses as 32-bit values.

User displays Intel processor user memory.

SMM displays Intel processor system management mode memory.

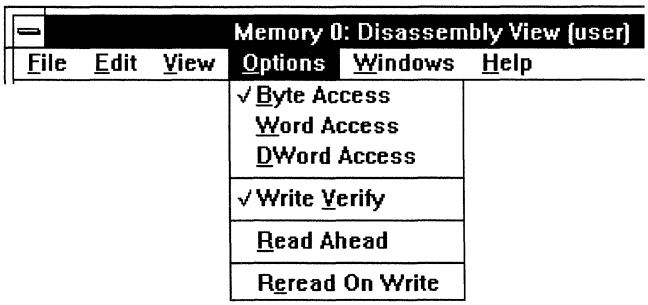
Refresh Display re-reads memory and refreshes the screen. This happens automatically when emulation halts.

To update or scroll the Memory window during emulation, you must enable Run Access before starting emulation. On the Toolbar, open the configure menu and check Enable Run Access; or enter a RunAccess Shell command.

Any memory access, such as that used to update the Memory window, takes a small amount of time from the processor and thus can degrade your program performance.

Options Menu

The following shows a sample Options menu.



Byte Access specifies 8-bit cycles for memory access.

Word Access specifies 16-bit cycles for memory access. For writing a byte, the word containing the byte is read, the appropriate byte replaced, and the word re-written. Words at even addresses are read and written

as words. Words at odd addresses are read and written as two words. For example, for writing a word of data at an odd address:

1. The word containing the first byte (odd address minus 1) is read.
2. The lower byte of the data is put into the upper byte of the word.
3. The word is re-written at odd address minus 1.
4. The word containing the second byte (odd address plus 1) is read.
5. The upper byte of the data is put into the lower byte of the word.
6. The word is re-written at odd address plus 1.

DWord Access specifies two 16-bit cycles for memory access. Long-word memory writes act as follows:

1. Long-word writes on long-word boundaries use long accesses.
2. Word writes and byte writes read long words, replace the byte or word, and write back as long words.

For Motorola, memory reads and writes always use supervisor data (SD) space. To access any other space, use Shell commands.

Set the memory access size to long (dword) for faster loading.

Write Verify, when checked, compares any value written with write or fill with the expected value and reports discrepancies.

Toggling write verify does not affect load verification. Use the verify Shell command to toggle load verification. With verify=on, a byte read back that does not match the byte written returns an error.

Read Ahead, when checked, reads ahead and caches more data than is displayed in the Memory window screen, for faster scrolling.

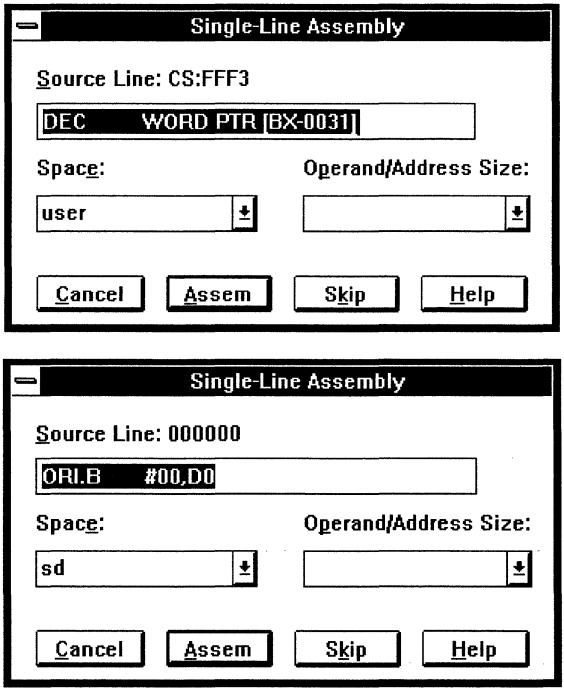
With read-ahead enabled, scrolling through peripheral registers or near invalid memory regions can cause Unterminated Memory Access errors.

Reread On Write, when checked, refreshes the memory display when you edit the numeric or ASCII fields in the display. Toggling Reread On Write does not affect Memory window refreshing for memory changes done outside of the memory display. For example, load, fill, and copy operations always refresh the memory display.

Single-Line Assembler Dialog Box

You can patch code into memory an assembly-line at a time with the single-line assembler. With the Memory window in Disassembly view, double-click on the line you want to replace.

The following figure shows two sample Single-Line Assembler dialog boxes. The first is for an Intel386 EX processor; the second is for a Motorola 68332 processor. Different space field values are available for different processors.

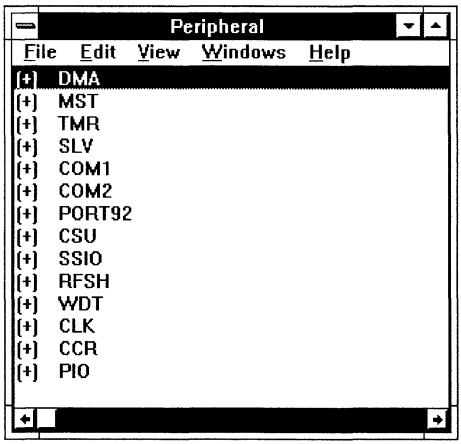


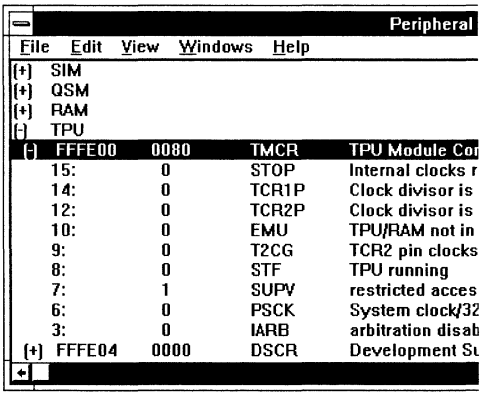
Type a line of assembly language in the text box.

Source Line:	shows the address where the line will be assembled.
Space:	for Intel, can be User or SMM; for Motorola, can be SP, SD, UP, or UD.
Operand/ Address Size:	is unavailable.
Cancel	closes the single-line assembler dialog box without assembling. Once you have assembled a line, this button changes to Done. Choosing Done closes the dialog box; your assembled changes remain in memory.
Assem	assembles the line into memory; advances the address.
Skip	advances the address without assembling the line.
Help	opens a window for help on the single-line assembler.

Peripheral Window Reference

The following figure shows two sample Peripheral windows. The first is for an Intel386 EX processor; the second is for a Motorola 68332 processor. Different peripherals are available for different registers. The Peripheral window is unavailable in Intel386 CX/SX emulators.





This chapter describes the Peripheral window contents, menus, and dialog boxes.

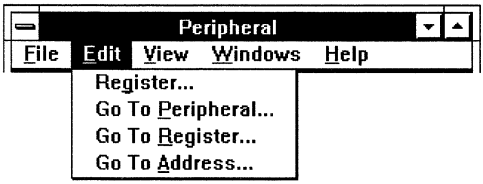
The Peripheral window shows the peripheral register information heirarchically. Click on the (+) or (-) at the left of a line to expand or collapse the hierarchy. At the top level (the only level visible when the heirarchy is fully collapsed) are the peripherals. Expanding a peripheral shows its registers. Expanding a register shows its bit fields. Full expansion lists, the register address, bit field bit position, value, name, and description. The peripheral, register, and bit field names conform to the Intel and Motorola mnemonics.

Peripheral Window Menus

Menu	Use To:
File	Exit from the Peripheral window.
Edit	Edit a register; navigate the Peripheral display.
View	Refresh, expand, or compress the display.
Windows	Open another SLD window.
Help	Open a window for help with SLD.

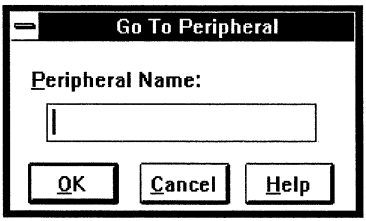
Edit Menu

The following shows an Edit menu.

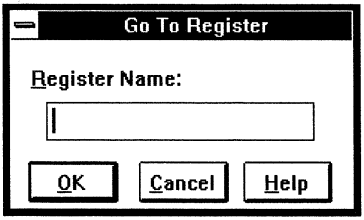


Register... opens a Register Edit dialog box (described later in this chapter) to edit the selected register. To select a register or bit field, use the mouse or <Up Arrow> and <Down Arrow> keys to move the highlight. Selecting a peripheral selects its the first register.

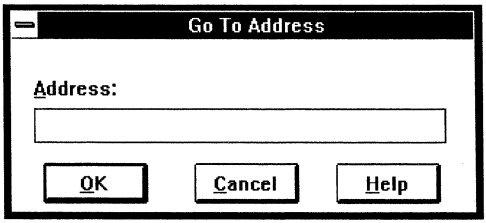
Go To Peripheral... opens a dialog box to scroll to the peripheral specified by name. The following is a Go To Peripheral dialog box.



Go To Register... opens a dialog box to scroll to the register specified by name. The following is a Go To Register dialog box.

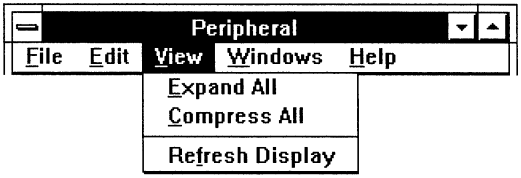


Go To Address... opens a dialog box to scroll to the register specifed by address. The following is a Go To Address dialog box.



View Menu

Following is a sample View menu.



Expand All expands the hierarchy completely, showing all peripheral, register, and bit field mnemonics, with the addresses or bit positions, values, and descriptions of the registers and bit fields.

Compress All collapses the hierarchy completely, showing only the peripheral mnemonics.

Refresh Display re-reads the register contents (except write-only registers) and refreshes the screen. This occurs automatically when emulation halts.

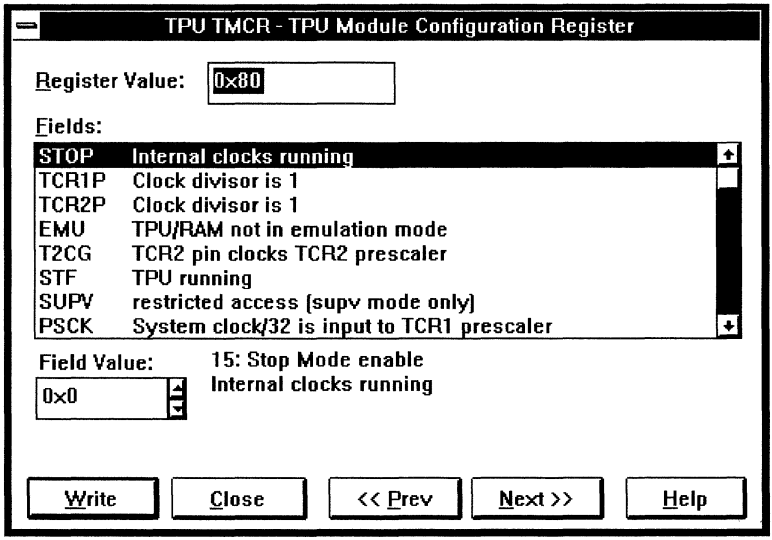
To update or scroll the Peripheral window during emulation, you must enable Run Access before starting emulation. On the Toolbar, open the configure menu and check Enable Run Access; or enter a RunAccess Shell command.

Any memory access, such as that used to update the Peripheral window, takes a small amount of time from the processor and thus can degrade your program performance.

For write-only registers, SLD reports the most recent value you entered using the Peripheral or Shell window interface. Values written by the execution of your program are not captured in SLD.

Register Edit Dialog Boxes

The following shows a sample Register Edit dialog box. This example is for a register in the Motorola 68332 processor. Different registers have different fields and values; however, the layout of the Register Edit dialog box is consistent.



Register Value — shows the register contents in hexadecimal. You can edit this field.

Fields — lists each bit field mnemonic in the register and its effect on the processor. To select a bit field, click or use the <Up Arrow> and <Down Arrow> keys to move the highlight.

Field Value — is a spin box showing the value of the bit field selected in the Fields box. You can edit this field. To ensure you enter an acceptable value for the bit field, click on the spin arrows or use the <Up Arrow> and <Down Arrow> keys to change the value. Editing the Field Value changes the Register Value.

The selected bit field position and a description of the bit field according to its current value are listed under the Fields box, to the right of the Field Value spin box. This description changes when you change the bit field value.

Write — writes the value shown in Register Value:.

Close — closes the Register Edit dialog box.

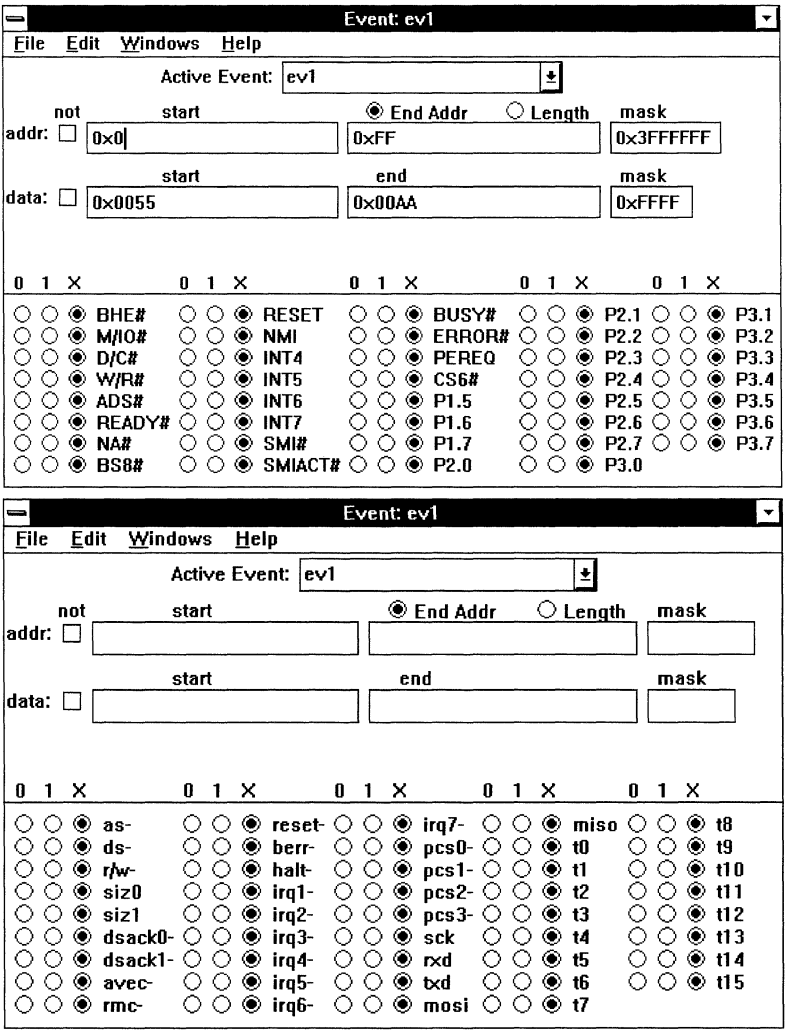
<<Prev — displays the Register Edit dialog box for the previous register in the Peripheral window list.

Next>> — displays the Register Edit dialog box for the next register in the Peripheral window list.

Help — opens a help window on the Register Edit dialog box.

Event Window Reference

The following figure shows two sample Event windows (also called Event edit boxes). The first is for an Intel386 EX processor; the second is for a Motorola 68332 processor. Different signals and address formats are available for different processors. For some Motorola processors, the signals available also depend on the chip selects.



This chapter describes the Event window fields, menus, and dialog boxes.

Event Window Contents

The Event window defines an event to be used as a condition for triggering. The fields are:

Active Event	is the name of the event described in the Event window. (This name also identifies the event in the Trigger and Trace windows.)
addr:	describes a single address or range of addresses. Select End Addr to specify the last location in a range or Length to specify the number of bytes in the range.
data:	describes a data value or range of data values.
mask	is a hexadecimal value to be bitwise-ANDed with the described addresses or data. Use all F's to include all contiguous values in the described range. Vary the mask to describe a discontinuous pattern of values.
not	when checked, defines the event as any memory access that does not match the described range or pattern.
0 1 X	specifies each signal value as low (0), high (1), or don't-care (X). Active-low signals are shown with a hash mark (#) for Intel emulators or minus sign (-) for Motorola emulators. The signals available depend on the target processor. For some Motorola processors, the signals available can also depend on your chip select register configurations.

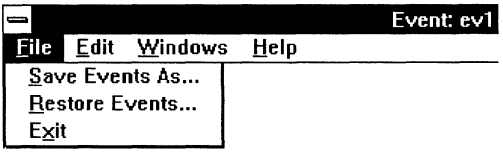
For Motorola emulation, you can specify the address space for an event as UD, UP, SD, or SP. To make the space selection available in the Event edit box, you must program the processor to output the three function codes FC0, FC1, and FC2.

Event Window Menus

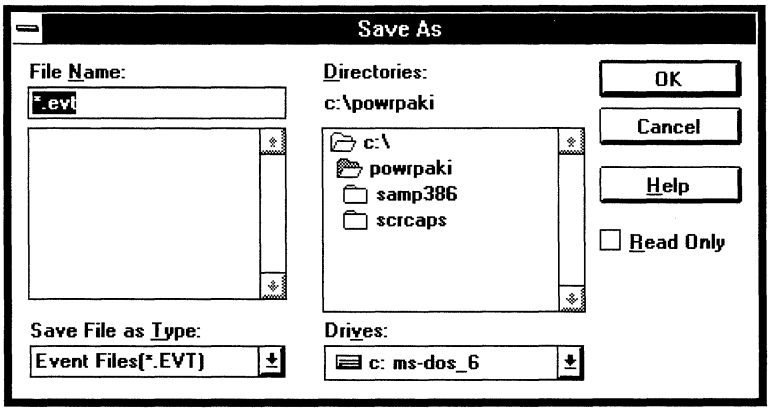
Menu	Use To:
File	Save and restore events in files; close the Event window.
Edit	Add, delete, and redefine events.
Windows	Open another SLD window.
Help	Open a window for help with SLD.

File Menu

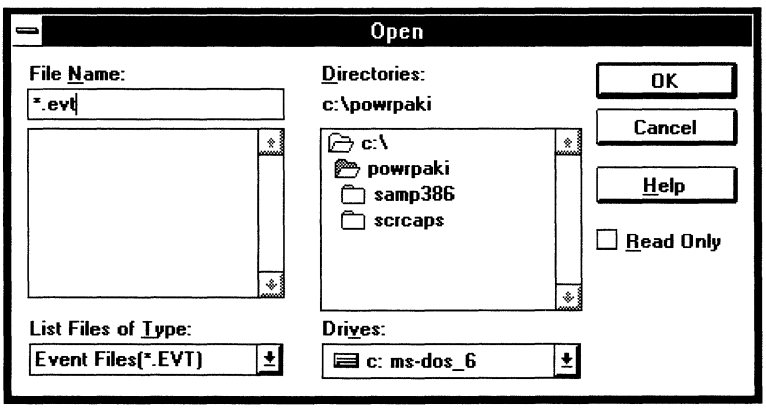
The following is a sample File menu.



Save Events As... opens a dialog box to save the events to a file. The following figure shows an event Save As dialog box. Select a path and filename, then choose OK to save.



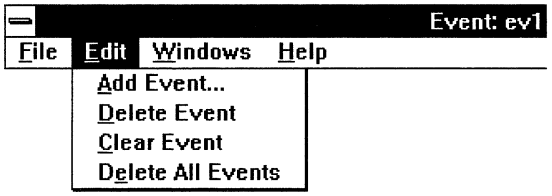
Restore Events... opens a dialog box to add events from a previously saved file. Currently defined events are not deleted; but events with duplicate names are overwritten from the file. The following figure shows an event Open dialog box.



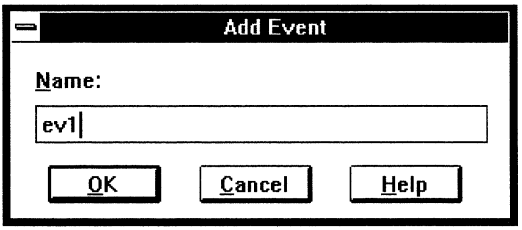
Exit closes the Event window.

Edit Menu

The following is a sample Edit menu.



Add Event... opens a dialog box to create a new event. Enter the name of a new event in the box and choose OK. The new event then appears as the Active Event, with all fields cleared, in the Event window. The following figure shows an Add Event dialog box.



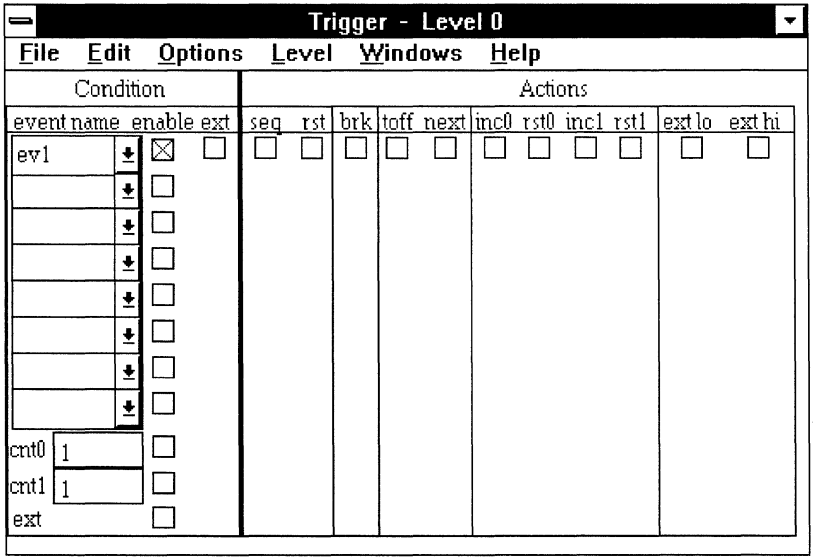
Delete Event deletes the currently displayed event.

Clear clears the event definition fields without deleting the event name.

Delete All Events deletes all currently defined events.

Trigger Window Reference

The following shows a sample Trigger window.



This chapter describes the Trigger window fields, menus, and dialog boxes.

The Trigger window has two panes:

Condition — describes one or more conditions, including events, an external trigger-low signal, and either two counter values or a timer value.

Actions — specifies one or more actions to be taken for each condition met during emulation. When multiple conditions are met simultaneously, all associated actions are taken.

The title bar displays a level number from 0 to 3. The level 0 trigger is enabled when you start emulation. Each trigger can, as one of its actions, disable itself and enable the next level trigger. Thus you can define up to four sequential triggers.

Trigger Condition Fields

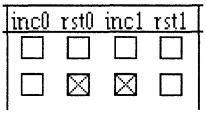
At the bottom of the Condition pane is either a pair of counters (cnt0 and cnt1) or a timer (tmr). To choose the counters or the timer, open the Options menu (described later in this chapter) and check Counter or Timer. This toggle also configures the Actions pane for resetting and incrementing the counter or for starting, stopping, and resetting the timer. The following figure shows sample counter and timer configurations.

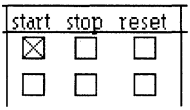
Field	Use To
event name	Select an event by the name defined in the Event window. You can use up to 8 events per trigger. If no event is defined when you click on an event name condition, the Event Name dialog box appears for defining a new event.
enable	Activate a condition. You can define several conditions and actions, then vary your triggering scheme by enabling them in different combinations.
ext	(This is the ext that appears when a condition is enabled.) Specify that the condition must occur at the same time as an active-low external trigger signal.
cnt0/1	Count from 1 to 1023. Type a target value in a counter field and enable the counter. Trigger actions can reset (to 1) or increment (by 1) the counter. When the count caused by the trigger actions matches the target count you specified, the counter condition is met and the associated actions occur.
tmr	Time from 1 to 1048575 clock cycles. Type a target value in the timer field and enable the timer. Trigger actions can start counting clock cycles from the current number; stop counting without resetting the timer; or reset the timer to 1. You can pair resetting with either starting or stopping the timer. When the timer count caused by the trigger actions matches the target time you specified, the counter condition is met and the associated actions occur. The timer increments at the clock rate of the emulation

	processor and wraps to 0 after reaching its maximum value. To calculate how much time is represented by a complete cycle of the timer, use: wrap time = (2^{20}) / (clock period) For example, at 25 MHz, the timer wraps in about 42 ms; at 16 MHz, in about 65.5 ms.
ext	(This is the ext in the lower left corner of the Trigger window.) Detect an active-low external trigger signal.

Trigger Action Fields

The fourth column of the Actions pane contains actions to reset or increment the counters (inc0, rst0, inc1, rst1) or to start, stop, or reset the timer (start, stop, reset). To choose the counter or timer actions, open the Options menu and check Counter or Timer. This toggle also configures the Condition pane with a pair of counters or a timer. The following figure shows sample counter and timer configurations.





Field	Use To
seq	Disable the current trigger and enable the next level trigger.
rst	Disable the current trigger and enable the level 0 trigger.
brk	Halt emulation.
toff	Turn trace off.
next	Fills the current buffer according to the Trace Control dialog box settings, then starts collecting trace in the next buffer. Available when multiple trace buffers are defined.
inc0/1	Increment the specified counter (ctr0 or ctr1) by 1.
rst0/1	Reset ctr0 or ctr1 to 1.
start	Start the timer (tmr) from its current value.
stop	Stop tmr at its current value.
reset	Reset tmr to 1.
ext lo/hi	Put a low or high value on the external trigger signal.

Trigger Window Menus

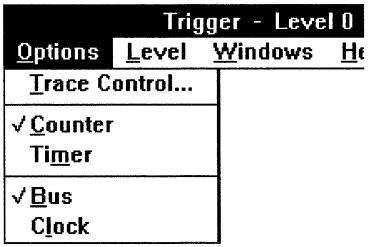
Menu	Use To:
File	Exit the Trigger window.
Edit	Specify an event using the Event window.
Options	Configure the trace buffers; toggle counter/timer conditions and actions; toggle bus/clock cycle triggering.
Level	View a specified trigger level.
Windows	Open another SLD window.
Help	Open a window for help with SLD.

Edit Menu

Events... opens the Event window

Options Menu

Following is a sample Options menu.



Trace Control... opens the Trace Control dialog box, described in the "Trace Window Reference" chapter.

Counter configures two 10-bit counters for use in trigger conditions and actions.

Timer configures a 20-bit timer for use in trigger conditions and actions.

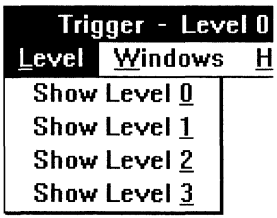
Bus lets the trigger recognizes conditions on valid bus cycles only. Choose Bus mode except when:

- tracking hardware bus problems possibly caused by processor cycles between valid address, data, or status cycles
- triggering on the initial transition of a hardware signal

Clock uses clock cycles as trigger conditions. Address, data, and status events occur at different clocks. Chose Clock mode for a single event that tests conditions including address, data, and status.

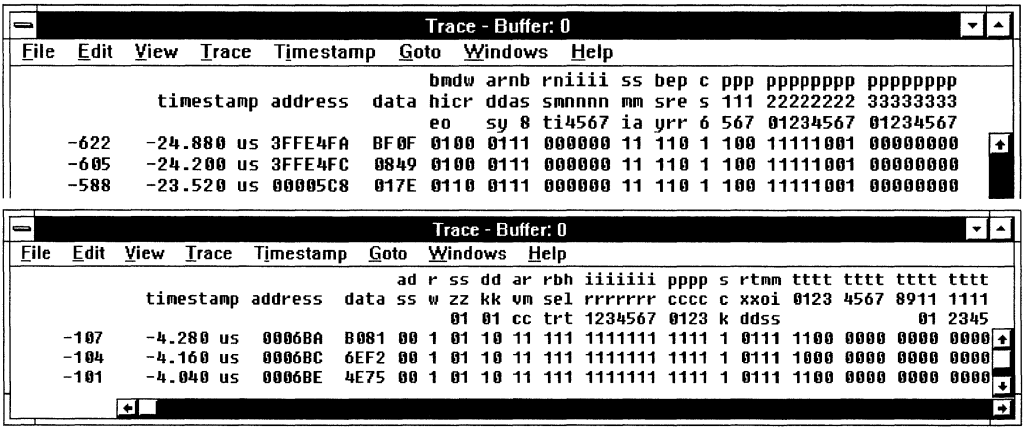
Level Menu

Choosing a level displays the conditions and actions for that trigger. Following is a sample Level menu.



Trace Window Reference

The following figure shows two sample Trace windows. The first is for an Intel386 EX processor; the second is for a Motorola 68332 processor. Different signals, address formats, and instruction formats are available for different processors. For some Motorola processors, the available signals also depend on your chip select configurations



This chapter describes the Trace window contents, menus, and dialog boxes.

The Trace window has three view modes:

Bus	displays every cycle of bus activity.
Clock	displays address, data, and processor status signals aligned on clock cycles.
Instruction	displays disassembled instructions. To find the beginning of the first instruction to display, SLD looks for a discontinuity caused by a change in execution flow (a branch trace message). No instructions can be disassembled before such a discontinuity is found.

Each trace frame (one line in the Trace window) contains the following information, in columns from left to right:

Cycle number	The clock cycle number of the trace frame relative to the cycle of the triggering event. In instruction and bus view modes, the frame numbers are discontinuous because multiple clock frames make up a single bus or instruction frame.

Timestamp	The time the trace frame occurred, relative either to the beginning of trace or to the previous frame.
Address	The value on the address bus.

In bus or clock view mode:

Data	The value on the data bus
Signals	The values of processor-specific signals. The signal mnemonic labels are formatted vertically.

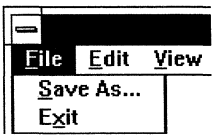
In instruction view mode, disassembly is shown instead of data and signals. Also, the number of clock cycles between instruction frames describes how many cycles have elapsed between signals appearing on the target processor external pins (for example, the number of cycles between successive prefetches); this number does not, for example, report how many clocks the processor used to execute an instruction.

Trace Window Menus

Menu	Use To:
File	Save trace to a buffer; close the Trace window.
Edit	Open the Event window; search for an event; clear trace.
View	Configure the trace display; link the Source window display to scroll with the Trace window cursor.
Trace	Start and stop trace; configure Trace Control.
Timestamp	Configure the timestamp and the system clock frequency.
Goto	Navigate through the Trace buffer.
Windows	Open another SLD window.
Help	Open a window for help on SLD.

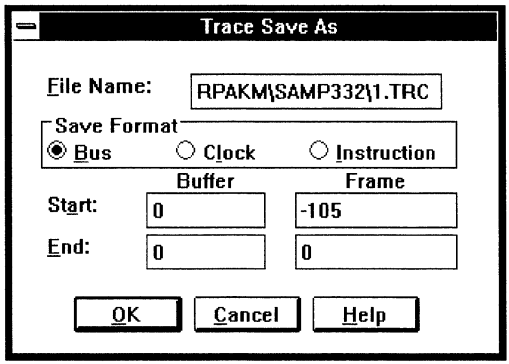
File Menu

The following is a sample File menu.



Save As... opens a dialog box to save the trace buffer to a file. Enter the filename. If a file with the specified name already exists, it will be overwritten. A Trace Save As dialog box appears:

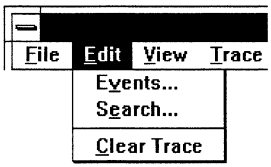
File Name:	is the drive, directory, and filename you specified in the first dialog box. You can edit this string.
Save Format	saves the trace in bus, clock, or instruction format.
Buffer	saves a specified range of buffers.
Frame	saves a specified range of frames.



Exit closes the Trace window.

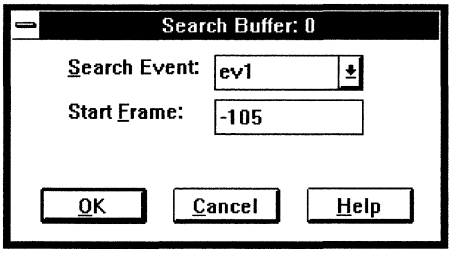
Edit Menu

The following shows a sample Edit menu.



Events... opens the Event window.

Search... opens a dialog box to find an event in the currently displayed trace buffer. The following figure shows a Search Buffer dialog box.



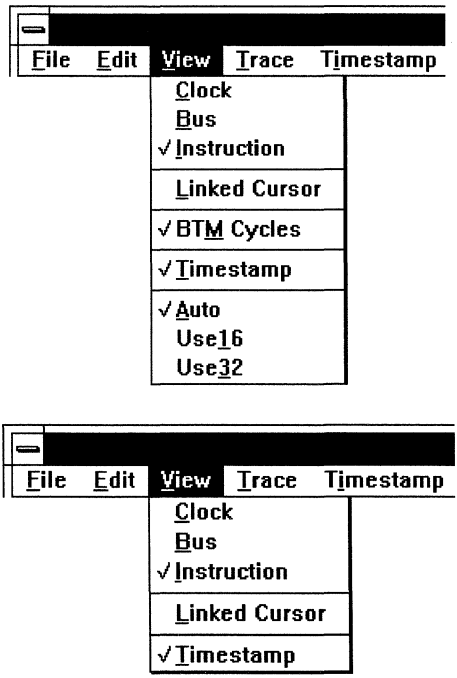
Search Event select an event from the list of defined events.

Start Frame select the frame to start searching.

Clear Trace clears all trace buffers and resets the buffer pointer to zero. (The current trace buffer is automatically cleared and reset when you start emulating or tracing.)

View Menu

The following figure shows two sample View menus. The first is for an Intel386 EX processor; the second is for a Motorola 68332 processor. Different processors have different signals and address formats.



Clock displays trace as clock cycles.

Bus displays trace as bus cycles.

Instruction displays trace as disassembly (instruction cycles). In instruction mode, a branch trace message (BTM) must be collected before disassembly can be constructed. Instructions in the trace before any such execution flow change cannot be displayed.

Linked Cursor to link the cursors in the Source and Trace windows, so when you scroll through the Trace window the Source window scrolls synchronously. This item is available only in instruction view mode.

BTM Cycles, when checked, generates BTM cycles and collects them in trace. A BTM cycle is a special bus cycle executed by the bondout processor when execution is discontinuous (e.g., at a jump, call, interrupt, return, etc.). Their occurrence degrades real-time execution slightly. For trace to be displayed as instructions, BTM cycles must be collected. Toggling BTM Cycles clears the trace buffer.

Timestamp displays the timestamps.

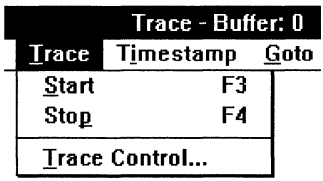
Auto uses the Intel386 processor pmode to determine whether operands and addresses are interpreted as 16-bit or 32-bit values.

Use16 interprets Intel386 operands and addresses as 16-bit values.

Use32 interprets Intel386 operands and addresses as 32-bit values.

Trace Menu

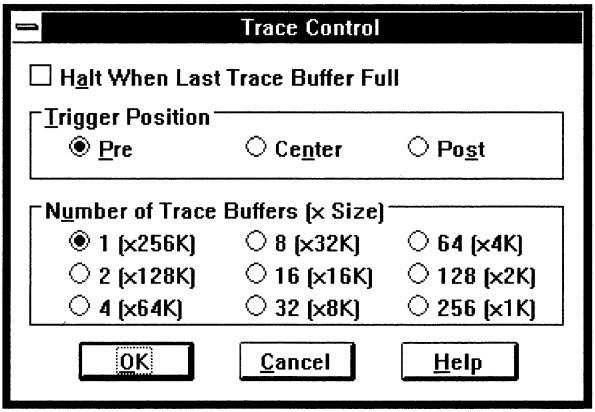
The following shows a sample Trace menu.



Start (or pressing the F3 key) starts trace collection. This occurs automatically when emulation begins.

Stop (or pressing the F4 key) stops trace collection.

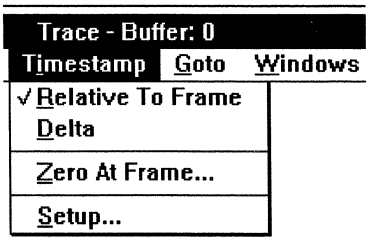
Trace Control... opens a dialog box to configure the number of buffers, the trigger location, or a breakpoint on a full buffer.



Halt When Last Trace Buffer Full	stops emulation after the last trace buffer has been filled. This overwrites the first trace buffer.
Trigger Position	specifies whether the triggering event will be recorded in the trace buffer:
	Pre — collects cycles before the trigger. The event appears near the end of the buffer.
	Center — collects cycles before and after the trigger. The event appears in the middle of the buffer.
	Post — collects cycles after the trigger. The event appears near the beginning of the buffer.
Number of Trace Buffers (x Size)	configures a single trace buffer 256K bytes long, or 256 trace buffers each of which is 1K byte long, or any of various combinations in between.

Timestamp Menu

The following shows a sample timestamp menu.

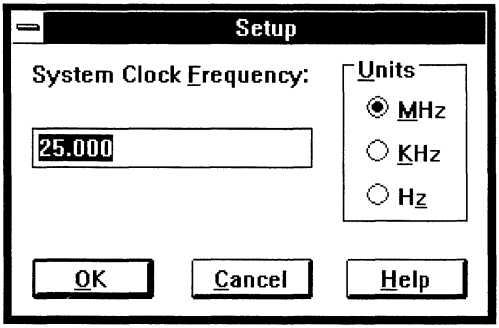


Relative To Frame computes each frame's timestamp relative to the beginning of trace.

Delta computes each frame's timestamp relative to the previous frame's timestamp.

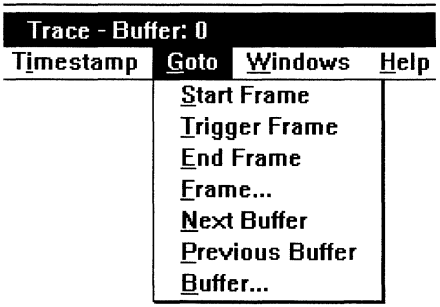
Zero At Frame sets the base frame for calculating the Relative To Frame timestamp. The zero frame is marked with dashes (--).

Setup... opens a Setup dialog box to set the system clock frequency. Enter a floating-point value from 0.01 Hz to 40 MHz.



Goto Menu

The following shows a sample Goto menu.

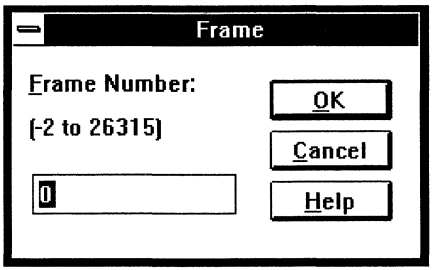


Start Frame scrolls to the first trace frame in the displayed trace buffer.

Trigger Frame scrolls to the trigger frame in the displayed trace buffer.

End Frame scrolls to the last frame in the displayed trace buffer.

Frame... opens a dialog box to scroll to a specified frame in the displayed trace buffer. The following shows a Frame dialog box.

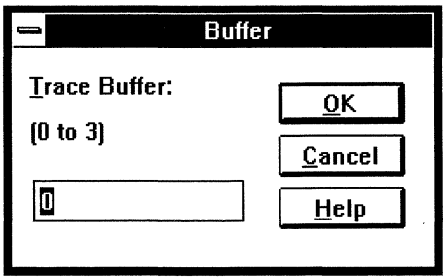


With multiple buffers, the following Goto menu items are also available:

Previous Buffer displays the next lower numbered buffer.

Next Buffer displays the next higher numbered buffer.

Buffer... opens a dialog box to display the specified buffer. The following shows a Buffer dialog box.



Glossary

address

Unsigned value identifying a location in memory. An address can be a hexadecimal number or a symbol (if symbols have been loaded). See the Address Formats section in the "Debugging with Triggers and Trace" chapter.

alarm limit

User-specified percentage of the stack area. If the stack usage exceeds the alarm limit when emulation halts, a message appears.

alias

Symbol defined in the Shell window to represent a character string. For example, used to shorten long commands.

alignment

See **trace alignment**.

BDM

Background Debug Mode available in Motorola CPU32 processors.

branch trace message (BTM)

Trace information recording a change in execution flow.

break cause

Why emulation is halted.

breakpoint

Location where emulation halts. Also see: **software breakpoint**, **hardware breakpoint**, **permanent breakpoint**, **temporary breakpoint**.

browse

Select a module to view in the Source window.

browser history

In the Source window, you can view up to two modules simultaneously. When you browse more than two modules, the emulator keeps a chronological list in a browser history buffer of the modules you have browsed. You can specify a buffer depth of the number of entries to save. To review a sequence of modules, use the File menu Previous Browsed Module and Next Browsed Module entries.

buffer

See **browser history**, **command history**, **loadfile history**, **trace buffer**.

bus event

One or more data, address, or status signals occurring during a single target bus cycle.

bus mode

Displays trace aligned in frames by the bus-cycle termination signals; or, collects trace for each target bus cycle. The display mode and the collection mode are set separately using the Trace window View and Options menus. Also see **clock mode**, **instruction mode**.

call stack

Current nesting of calls in the executing program, including information about each function's name, stack address, return address, local variables, and parameters.

case sensitive

Distinguishes lower-case letters from upper-case letters.

cause

See **break cause**.

clock mode

Displays trace aligned in frames by clock cycle; or, collects trace for each target clock cycle. Clock cycles are based on the external speed of the processor. The display mode and the collection mode are set separately using the Trace window View and Options menus. Also see **bus mode**.

command entry pane

Bottom part of the Shell window. Type Shell commands on the command entry pane command lines; press <Enter> to execute the commands. Separate multiple commands with semicolons. Also see **transcript pane**.

command history

As you enter Shell commands, the emulator keeps a chronological list in a command history buffer of all your entries. You can specify a buffer depth of the number of entries to save. To recall commands from the buffer to the command entry pane, use the <Ctrl><Up Arrow> and <Ctrl><Down Arrow> key combinations.

command script

See **script**.

compress display

Display only the first line of a variable, peripheral register, or peripheral group.

control processor

Located in the main chassis; controls emulation processing. Also see **emulation processor**, **target processor**.

current module and function

Code location where the emulator has most recently halted.

cursor

Highlight, vertical or horizontal bar, or other symbol showing the current focus point in a window display. Move the cursor with the <Up Arrow> and <Down Arrow> keys or by pointing and clicking with the mouse.

data breakpoint

Hardware breakpoint causing a break when a specified address is read or written.

debug environment

The debug environment includes the control options (such as overlay memory), user-defined aliases or debug variables, and the SLD desktop.

demangle

To demangle is to strip C++ mangling from symbol names during load.

disabled breakpoint

Encountering a disabled breakpoint does not halt emulation. The disabled breakpoint is ignored. Also see **enabled breakpoint**, **temporary breakpoint**, **permanent breakpoint**.

disassembly

Memory contents or trace information interpreted by the emulator as assembly language instructions.

double word

32 bits (four bytes).

emulation breakpoint

See **hardware breakpoint**.

emulation pod (EPOD)

Contains emulation and overlay circuits; attached by cables to the emulator chassis and probe head.

emulation processor

Located in the Probe, the emulation processor replaces the processor in the target system, providing the emulator with information about the program execution. Also see **control processor**, **target processor**.

emulation status

Whether the emulator is running or halted. This information appears in the Status window or icon. Also see **break cause**.

emulator

Uses a special version of the processor to monitor and control your target's software and hardware activity involving the processor. In the PowerPack™ emulator documentation, **emulator** refers to the PowerPack emulator and SLD software.

enabled breakpoint

Encountering an enabled breakpoint halts emulation. Also see **disabled breakpoint**, **temporary breakpoint**, **permanent breakpoint**.

event

Condition arising in program execution that can be used to trigger an emulator action during emulation or to find specified activity in the trace buffer.

execution breakpoint

Hardware breakpoint causing a break when an instruction at a particular address is executed.

frame

See **trace frame**, **stack frame**.

go

Emulate until halted by a predefined condition or by a halt request.

granularity

In the Source and Shell windows, the step granularity can be set to source line or source statement. With the granularity set to line, stepping emulates one or more source lines. With the granularity set to statement, stepping emulates one or more source statements.

hardware breakpoint

Breakpoint using a processor register rather than a software interrupt. Also see **software breakpoint**.

high-water mark

The greatest percentage of the stack area used during program execution.

history buffer

See **command history**, **browser history**, **loadfile history**.

host

Your workstation or PC, where you run SLD.

include file

See **script**.

initialization code

See **startup code**.

initialization script

The script run automatically when you start SLD, to configure the emulator. Also see **script**.

initialization file

File named powerpak.ini, which is placed in your Windows directory by the SLD installation.

instruction mode

Displays trace as disassembly instructions. Also see **clock mode**, **bus mode**. A branch trace message must be collected for the emulator to disassemble the instructions.

line numbers

Sequential source line numbers in each independently compiled high-level language module.

linked cursor

You can link the Source and Trace displays so that, when you scroll or browse in the Trace window in instruction mode, the Source window scrolls automatically to display the corresponding source.

load

Write executable code and/or symbolic information from your host system to target or emulator memory.

load status

Optional dynamic display of loading progress. The final status can be redisplayed with the Source window File menu Load Information item. Load information includes: the loadfile pathname; the module source file pathname; the number of bytes, modules, symbols, types, functions, and lines loaded; the program counter; and the stack base and size. The load status information box also displays a bar graph that fills to indicate the percent of loading complete.

loadfile

File containing executable code and/or symbolic information in OMF86, OMF386, IEEE-695, or S-record format.

loadfile history

When you load a file, the emulator keeps a chronological list in a loadfile history buffer of the most recent four loadfiles. To load one of these files, in the Source window File menu choose one of the last entries numbered from 1 to 4.

log file

You can record Shell commands and their results to a file called a logfile.

long

See **double word**.

main chassis

Houses the PowerPack emulator motherboard, trace and communications modules, and power supply.

mangle

A compiler mangles C++ overloaded names by adding a prefix or suffix to uniquely identify the names for type-safe linkage.

map

Configure overlay and target memory to control access and emulation response to memory accesses.

map file

File containing a saved map configuration.

memory access size

Number of memory locations read or written in a single access: byte, word, or double word.

module

Independently compiled source file.

motherboard

Circuit board, in the PowerPack emulator main chassis, containing the system processor, memory, communications, and analysis circuits.

null target

Board supplied with your Motorola PowerPack emulator for use as a target board when you run the emulator startup tests. If you have code ready to test but no hardware (and no special hardware needed to run the code), you can run the code with the emulator attached to the null target instead of to your target hardware. For Intel emulators, see **SAST board**.

on-demand loading

Defers loading symbolic information for an individual module until either the module is displayed in the Source window or a breakpoint is set in the module. On-demand loading saves time when the file is loaded and saves space if some symbols are never needed.

overlay memory

RAM used and controlled by the emulator in place of your target system memory. Also see **target memory**.

permanent breakpoint

A breakpoint which remains defined after causing emulation to halt. Also see **temporary breakpoint**, **enabled breakpoint**, **disabled breakpoint**.

probe

Plugs into the target system, replacing the target processor, and provides the hardware interface between the EPOD and the target.

program counter

Register used by the processor to find the next instruction to be executed. On Intel, this register is CS:EIP (code segment extended instruction pointer); on Motorola, PC (program counter).

SAST board

Board supplied with your Intel emulator for use as a target board when you run the emulator stand-alone self-tests. If you have code ready to test but no hardware (and no special hardware needed to run the code), you can run the code with the emulator attached to the SAST board instead of to your target hardware. For Motorola emulators, see **null target**.

script

Text file of Shell commands separated by semicolons. Execute a script with the Include Shell command.

shell variable

Symbol starting with $, defined in the Shell window or in a script for use with Shell commands.

SLD

Source Level Debugger, the PowerPack and PowerScope user interface.

software breakpoint

Breakpoint using a software interrupt inserted as the instruction at the address where you set the breakpoint. Also see **hardware breakpoint**.

source line

Single line of executable code in a source file.

source statement

Single statement of executable code in a source file. Some C compilers allow multiple statements per line, separated by semicolons.

split box

Windows object that you can drag to split a window into two panes. In SLD, such a box is located above the top arrow of the Shell and Source window vertical scroll bars.

stack frame

When a function is called, information about the call (return address, parameters, local variables) is stored in a record on the stack. One such record is a stack frame. The frames on the stack change as calls and returns execute.

startup code

Executable code that runs before main() to set up the processor registers for your target system. The startup code is usually written in assembly language. Some compilers automatically add startup code; for some target designs, you may need to write the startup code.

status

See **load status, emulation status, tracing status**.

step

Execute a line, statement, or instruction; then break.

system clock (CLKOUT)

Internal system clock signal used as the bus timing reference by external devices.

system processor

See **control processor**.

tab

Single character interpreted as a specified number of spaces.

tab width

Number of spaces replacing a tab character. Ensure your emulator tab width matches your compiler tab width.

target memory

RAM or ROM available on your target system.

target processor

The processor in your target system. When the emulator is attached to your target system, the emulation processor in the emulator probe head replaces the target processor. Physically, this replacement is done either by removing your target processor and plugging the probe head into the socket on your target board, or by using a clip-over adapter to attach the probe head on top of your target processor, tri-stating your target processor.

target system

Hardware of your design to which you connect the emulator. Also see **SAST board, null target**.

temporary breakpoint

A breakpoint which is removed after it causes emulation to halt. Also see **permanent breakpoint, enabled breakpoint, disabled breakpoint.**

timestamp

Number associated with each trace frame indicating how many clock cycles have elapsed since a specified frame or since the previous frame. Clock cycles are based on the external speed of the processor.

toggle

Specify or choose one of a set of two or more mutually exclusive values or items.

toolchain

The compiler, assembler, linker/locator, and translator you use to generate a loadfile from your source code. A supported toolchain is one Microtek International has tested and approved for generating emulator-loadable files. The emulator is not guaranteed to work with unsupported toolchains.

trace

Record of the emulation processor activity and signals collected at the emulation processor clock rate. These signals can be displayed in frames based on clock cycles, bus cycles, or as disassembled instructions.

trace buffer

Buffer containing a snapshot of the collected trace. The snapshot can be taken relative to a specified event occurring during emulation. You can partition trace into one or more buffers; the size of each buffer depends on the number of buffers.

trace frame

A trace frame is one line of information in the trace buffer. Each frame starts at a consistent point relative to a bus cycle, clock cycle, or instruction fetch.

tracing status

Whether tracing is on or off; if on, which trace buffer is active. This information appears in the Status window and icon.

transcript pane

Top pane of the Shell window. Optionally, you can configure the transcript pane to display commands entered in the command entry pane and the associated emulator responses. Also see **command entry pane**.

trigger

Defines the action taken by the emulator in response to the occurrence of one or more events.

trigger frame

First frame collected after a trigger is reached.

word

16 bits (two bytes).

Index

A

B

C

D

E

F

G

H

I

M

S

U

V

W

X

Z

MICROTEK INTERNATIONAL
Development Systems Division
3300 N.W. 211th Terrace
Hillsboro, OR 97124-7136
Phone: (503) 645-7333
Fax: (503) 629-8460

6, Industry East Road 3
Science-based Industry Park
Hsinchu 30077
Taiwan, ROC
Tel: +886 35 772155

SLD User's Manual For the PowerPack™ Development Tool

Part Number 14913-000