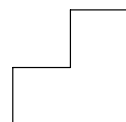


M16C v2.1

CROSSVIEW PRO DEBUGGER USER'S GUIDE



A publication of
TASKING
Documentation Department
Copyright © 2001 TASKING, Inc.

All rights reserved. Reproduction in whole or part is prohibited
without the written consent of the copyright owner.

The following trademarks are acknowledged:

FLEXlm is a registered trademark of Globetrotter Software, Inc.
HP and HP-UX are trademarks of Hewlett-Packard Co.
Motorola is a trademark of Motorola, Inc.
MS-DOS and Windows are registered trademarks of Microsoft Corporation.
IBM is a trademark of International Business Machines Corp.
SUN is a trademark of Sun Microsystems, Inc.
UNIX is a registered trademark of X/Open Company, Ltd.

All other trademarks are property of their respective owners.

Data subject to alteration without notice.

E-mail: support@tasking.com
WWW: <http://www.tasking.com>

The information in this document has been carefully reviewed and is believed to be accurate and reliable. However, TASKING assumes no liabilities for inaccuracies in this document. Furthermore, the delivery of this information does not convey to the recipient any license to use or copy the software or documentation, except as provided in an executed license agreement covering the software and documentation.

TASKING reserves the right to change specifications embodied in this document without prior notice.

CONTENTS

TABLE OF CONTENTS



CONTENTS

OVERVIEW 1-1

1.1	Introduction	1-3
1.2	CrossView Pro's Features	1-3
1.3	Source Level Debugging	1-7
1.4	How CrossView Pro Works	1-9
1.5	M16C Program Development	1-11
1.6	Getting Started	1-13
1.6.1	Before Starting	1-13
1.6.2	Setting Up the Execution Environment	1-13
1.6.3	Starting CrossView Pro	1-14
1.6.3.1	CrossView Pro Startup Settings	1-15
1.6.3.2	Configuring CrossView Pro	1-16
1.6.3.3	Loading Symbolic Debug Information	1-17
1.6.4	Executing an Application	1-20
1.6.5	Debugging an Application	1-22
1.6.6	CrossView Pro Output	1-24
1.6.7	Exiting CrossView Pro	1-25
1.6.8	What You May Have Done Wrong	1-25
1.6.9	Building Your Executable	1-26
1.6.9.1	Using EDE	1-26
1.6.9.2	Using the Control Program	1-33
1.6.9.3	Using the Makefile	1-35

SOFTWARE INSTALLATION 2-1

2.1	Introduction	2-3
2.2	Note about Filenames	2-3
2.3	Installation for Windows	2-3
2.3.1	Requirements	2-4
2.4	Installation for Linux	2-4
2.4.1	RPM Installation	2-5
2.4.2	Tar.gz Installation	2-6
2.5	Installation for UNIX Hosts	2-7
2.6	Configuring the X Windows Motif Environment	2-9
2.7	Using X Resources	2-10

2.8	Licensing TASKING Products	2-13
2.8.1	Obtaining License Information	2-13
2.8.2	Installing Node-Locked Licenses	2-14
2.8.3	Installing Floating Licenses	2-15
2.8.4	Starting the License Daemon	2-17
2.8.5	Setting Up the License Daemon to Run Automatically	2-18
2.8.6	Modifying the License File Location	2-19
2.8.7	How to Determine the Hostid	2-21
2.8.8	How to Determine the Hostname	2-21

COMMAND LANGUAGE **3-1**

3.1	Introduction	3-3
3.2	CrossView Pro Expressions	3-3
3.3	Constants	3-4
3.4	Variables	3-7
3.5	Formatting Expressions	3-13
3.6	Operators	3-17
3.7	Special Expressions	3-18
3.8	Conditional Evaluation	3-19
3.9	Functions	3-20
3.10	Case Sensitivity	3-21

USING CROSSVIEW PRO **4-1**

4.1	Introduction	4-3
4.2	Using the CrossView Pro Interface	4-3
4.3	Invoking CrossView Pro	4-4
4.4	Startup Options	4-5
4.4.1	What You May Have Done Wrong	4-9
4.5	The CrossView Pro Desktop	4-11
4.5.1	Menus	4-13
4.5.1.1	Local Popup Menus	4-14
4.5.2	Window Operation	4-15

4.5.3	Dialog Boxes	4-16
4.5.4	Customizing CrossView Pro	4-17
4.5.5	CrossView Pro Messages	4-19
4.6	CrossView Pro Windows	4-20
4.6.1	Opening Windows from the View Menu	4-20
4.6.2	Command Window	4-21
4.6.3	Source Window	4-23
4.6.4	Trace Window	4-26
4.6.5	Stack Window	4-27
4.6.6	Register Window	4-28
4.6.7	Data Window	4-29
4.6.8	Memory Window	4-31
4.6.9	Virtual I/O Window	4-35
4.6.10	Simulated I/O Window	4-36
4.6.11	Pop-Up Windows	4-36
4.7	Control Operations for CrossView Pro	4-37
4.7.1	Echoing Commands	4-37
4.7.2	Mouse/Menu/Command Equivalents	4-37
4.7.3	Button Selection	4-37
4.7.4	Text Selection	4-38
4.8	Using the On-Line Help System	4-39
4.8.1	Accessing On-line Help	4-39
4.8.2	Components of MS-Windows Help	4-39
4.8.2.1	Using MS-Windows Help	4-40

CONTROLLING PROGRAM EXECUTION

5-1

5.1	Source Positioning	5-3
5.1.1	Changing the Viewing Position	5-4
5.1.2	Changing the Execution Position	5-5
5.1.3	Synchronizing the Execution and Viewing Positions	5-7
5.2	Controlling Program Execution	5-8
5.2.1	Starting the Program	5-8
5.2.2	Halting and Continuing Execution	5-9
5.2.3	Single-Step Execution	5-9

5.2.4 Stepping through at the Machine Level 5-12

5.3 Notes About Program Execution 5-14

5.4 Searching through the Source Window 5-14

5.4.1 Searching for a Function 5-14

5.4.2 Searching for a String 5-15

5.4.3 Jumping to a Source Line 5-16

ACCESSING CODE AND DATA 6-1

6.1 Introduction 6-3

6.2 Accessing Variables 6-3

6.2.1 Viewing Variables, Structures and Arrays 6-3

6.2.2 Changing Variables 6-7

6.2.3 The l Command 6-8

6.3 Expressions 6-10

6.3.1 Evaluating Expressions 6-10

6.3.2 Monitoring Expressions 6-11

6.3.3 Formatting Data 6-13

6.3.4 Displaying Memory 6-14

6.3.5 Displaying Memory Addresses 6-16

6.4 Displaying Disassembled Instructions 6-17

6.4.1 Intermixed Source and Disassembly 6-18

6.5 The Stack 6-19

6.5.1 How the Stack is Organized 6-19

6.5.2 The Stack Window 6-20

6.5.3 Listing Locals and Parameters of a Function 6-22

6.5.4 Low-level Viewing the Stack 6-23

6.6 Trace Window 6-24

6.6.1 Trace Window Setup 6-24

6.7 Register Window 6-26

6.7.1 Register Window Setup 6-26

6.7.2 Editing Registers 6-27

BREAKPOINTS AND ASSERTIONS **7-1**

7.1	Introduction to Breakpoints	7-3
7.1.1	Code Breakpoints	7-3
7.1.2	Data Breakpoints	7-6
7.1.3	Listing Breakpoints	7-7
7.2	Setting Breakpoints	7-8
7.2.1	Data Breakpoints over a Range of Addresses	7-10
7.2.2	Temporary Breakpoints	7-11
7.2.3	Setting the Count	7-12
7.3	Deleting Breakpoints	7-13
7.4	Enabling/Disabling Breakpoints	7-14
7.5	Breakpoint Commands	7-15
7.5.1	Attaching Conditionals to a Breakpoint	7-16
7.5.2	Attaching Macros to a Breakpoint	7-17
7.5.3	Attaching Strings to a Breakpoint	7-17
7.6	Suppressing Breakpoint Messages	7-18
7.7	Low-level Breakpoints	7-19
7.8	Up-level Breakpoints	7-20
7.9	Patches	7-22
7.9.1	Patching Code out of a Program	7-22
7.9.2	Patching Code into a Program	7-23
7.9.3	Replacing Code in a Program	7-23
7.10	Diagnostic Output and Statistical Information	7-24
7.11	Assertions	7-25
7.11.1	Global Assertion Mode	7-25
7.11.2	Defining an Assertion	7-26
7.11.3	Editing an Assertion	7-28
7.11.4	Activating and Suspending Assertions	7-28
7.11.5	Deleting Assertions	7-29
7.11.6	Using Assertions	7-30
7.11.7	Gathering Statistics with Assertions	7-32

DEFINING AND USING MACROS **8-1**

8.1	CrossView Pro Macros	8-3
8.2	Defining Macros	8-3
8.2.1	Listing Macros	8-5
8.2.2	Redefining a Macro	8-5
8.2.3	Saving Macro Definitions to a File	8-6
8.2.4	Loading Macro Definitions from a File	8-7
8.2.5	Deleting Macros	8-8
8.3	Macro Parameters	8-9
8.4	Redefining Existing CrossView Pro Commands	8-10
8.5	Using the Toolbox	8-11
8.5.1	Opening the Toolbox	8-11
8.5.2	Connecting Macros to the Toolbox	8-11
8.5.3	Removing a Macro Connection	8-12

COMMAND RECORDING & PLAYBACK **9-1**

9.1	Recording Commands	9-3
9.1.1	Entering Comments	9-4
9.1.2	Suspend Recording	9-4
9.1.3	Resume Recording	9-5
9.1.4	Check Recording Status	9-5
9.1.5	Close File for Recording	9-6
9.1.6	Command Recording Example	9-6
9.2	Playing Back Command Files	9-7
9.2.1	Setting the Type of Playback	9-8
9.2.2	Calling Other Playback Files	9-9
9.2.3	Quitting Playback Mode	9-9
9.3	Command Line Batch Processing	9-10
9.4	Logging	9-12
9.4.1	Setting up Logging	9-13
9.4.2	Recording Commands and Logging Screen Output	9-14
9.4.3	Command Window Log File Example	9-14
9.4.4	Suspending and Resuming Output Log	9-14
9.4.5	Closing the Output Log File	9-16

9.5	Startup Options	9-17
9.6	CrossView Pro Command History Mechanism	9-18

SPECIAL FEATURES

10-1

10.1	Transparency Mode	10-3
10.2	RTOS Aware Debugging	10-4
10.3	Coverage	10-5
10.4	Profiling	10-7
10.5	Virtual I/O Channels	10-9
10.5.1	ROM Monitor	10-9
10.5.2	Keyboard Mappings Virtual I/O	10-10
10.6	Simulated Input/Output	10-14
10.6.1	Setting Up Simulated I/O	10-15
10.6.2	Viewing Current Stream Settings	10-17
10.6.3	Changing Stream's Properties	10-17
10.6.4	Changing the Simulated Input Prompt	10-19
10.6.5	Directing I/O to a File	10-20
10.7	The Simulated I/O Window	10-20
10.8	Background Mode	10-21
10.8.1	Configuration	10-21
10.8.2	Manual Refresh	10-22
10.8.3	Entering Background Mode	10-23
10.8.4	Leaving Background Mode	10-24
10.8.5	The Stack in Background Mode	10-25
10.8.6	Local and Global Variables	10-25
10.8.7	Refresh Limitation	10-25
10.8.8	Assertions	10-26

DEBUGGING NOTES

11-1

11.1	Debugging Assembly Language	11-3
11.2	Debugging Multiple Programs	11-3

COMMAND REFERENCE **12-1**

12.1	Conventions Used in this Chapter	12-3
12.2	Commands: Summary	12-4
12.2.1	Startup Options	12-4
12.2.2	Viewing Commands	12-7
12.2.3	Data Monitoring	12-8
12.2.4	Execution Control Commands	12-9
12.2.5	Record & Playback	12-13
12.2.6	Macros	12-13
12.2.7	Simulated Input/Output	12-14
12.2.8	Target System Control	12-14
12.2.9	Help Commands	12-14
12.2.10	Search Commands	12-15
12.3	Commands: Detailed Descriptions	12-15

ERROR MESSAGES **13-1**

13.1	What this Chapter Covers	13-3
13.2	Error Messages	13-3

GLOSSARY **14-1**

14.1	What this Chapter Covers	14-3
14.2	Glossary Terms	14-3

FLEXIBLE LICENSE MANAGER (FLEXlm) **A-1**

1	Introduction	A-3
2	License Administration	A-3
2.1	Overview	A-3
2.2	Providing For Uninterrupted FLEXlm Operation	A-5
2.3	Daemon Options File	A-7
3	License Administration Tools	A-8
3.1	lmcksum	A-10
3.2	lmdiag (Windows only)	A-11

3.3	lmdown	A-12
3.4	lmgrd	A-13
3.5	lmhostid	A-15
3.6	lmremove	A-16
3.7	lmreread	A-17
3.8	lmstat	A-18
3.9	lmswitchr (Windows only)	A-20
3.10	lmver	A-21
3.11	License Administration Tools for Windows	A-22
3.11.1	LMTOOLS for Windows	A-22
3.11.2	FLEXlm License Manager for Windows	A-23
4	The Daemon Log File	A-25
4.1	Informational Messages	A-26
4.2	Configuration Problem Messages	A-29
4.3	Daemon Software Error Messages	A-31
5	FLEXlm License Errors	A-33
6	Frequently Asked Questions (FAQs)	A-37
6.1	License File Questions	A-37
6.2	FLEXlm Version	A-37
6.3	Windows Questions	A-38
6.4	TASKING Questions	A-39
6.5	Using FLEXlm for Floating Licenses	A-41

SOUND SUPPORT (MS-Windows)

B-1

SIMULATOR		Sim-1
1	Introduction	Sim-3
2	Executable Name	Sim-3
3	Supported Features	Sim-3
3.1	Mapping Memory	Sim-3
3.2	State Counter	Sim-4
3.3	Coverage	Sim-4
4	Restrictions	Sim-4



ROM MONITOR		Rom-1
1	What is a ROM Monitor?	Rom-3
2	Setting up the Target Environment	Rom-5
3	Restrictions	Rom-5
4	Resources used by the ROM Monitor	Rom-6
5	Supported Targets	Rom-7

INDEX

MANUAL PURPOSE AND STRUCTURE

PURPOSE

This manual is aimed at users of the CrossView Pro debugger for the M16C. It assumes that you are familiar with programming the M16C.

MANUAL STRUCTURE

Related Publications

Conventions Used In This Manual

1. Overview
Highlights specific CrossView Pro features and capabilities, and shows how to compile code for debugging.
2. Software Installation
Describes how to install CrossView Pro on your system.
3. Command Language
Details the syntax of CrossView Pro's command language.
4. Using CrossView Pro
Describes the basic methods of invoking, operating, and exiting CrossView Pro.
5. Controlling Program Execution
Describes the various means of program execution.
6. Accessing Code and Data
Describes how to view and edit the variables in your source program.
7. Breakpoints and Assertions
Describes breakpoints and assertions.
8. Defining and Using Macros
Describes how to simplify a complicated procedure by creating a "shorthand" macro which can be used to execute any sequence of CrossView Pro or C language commands and expressions.
9. Command Recording & Playback
Describes the record and playback functions of CrossView Pro.

10. Special Features

Describes special features of CrossView Pro, such as the Transparency Mode, RTOS Aware Debugging, Coverage, Profiling and the Background Mode.

11. Debugging Notes

Contains some notes about debugging in special situations.

12. Command Reference

An alphabetical list of all CrossView Pro commands. Consult this chapter for specifics and the exact syntax of any CrossView Pro command.

13. Error Messages

Contains CrossView Pro error messages and gives advice for correcting them.

14. Glossary

Defines the most common terms used in embedded systems debugging.

APPENDICES

A. Flexible License Manager (FLEXlm)

Contains a description of the Flexible License Manager.

B. Sound Support (MS-Windows)

Describes how to add sound to CrossView Pro events under MS-Windows.

ADDENDUM

Execution Environment

Contains information specific to your particular type of target system.

INDEX

RELATED PUBLICATIONS

- The C Programming Language (second edition) by B. Kernighan and D. Ritchie (1988, Prentice Hall)
- ANSI X3.159-1989 standard [ANSI]
- M16C Cross-Assembler, Linker/Locator, Utilities User's Guide [TASKING, MA499-000-00-00]
- M16C C Cross-Compiler User's Guide [TASKING, MA499-002-00-00]

CONVENTIONS USED IN THIS MANUAL

The notation used to describe the format of call lines is given below:

{ }	Items shown inside curly braces enclose a list from which you must choose an item.
[]	Items shown inside square brackets enclose items that are optional.
	The vertical bar separates items in a list. It can be read as OR.
<i>italics</i>	Items shown in italic letters mean that you have to substitute the item. If italic items are inside square brackets, they are optional. For example: <i>filename</i> means: type the name of your file in place of the word <i>filename</i> .
...	An ellipsis indicates that you can repeat the preceding item zero or more times.
screen font	Represents input examples, screen output examples, filenames and keywords.
bold font	Represents a command name, an option or a complete command line which you can enter.

For example

command [*option*]... *filename*

This line could be written in plain English as: execute the command *command* with the optional options *option* and with the file *filename*.

Illustrations

The following illustrations are used in this manual:



This is a note. It gives you extra information.



This is a warning. Read the information carefully.



This illustration indicates actions you can perform with the mouse.



This illustration indicates keyboard input.



This illustration can be read as “See also”. It contains a reference to another command, option or section.



MANUAL STRUCTURE

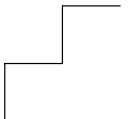
CHAPTER

1

OVERVIEW



TASKING



1

CHAPTER

1.1 INTRODUCTION

This chapter highlights many of the features and capabilities of CrossView Pro, including an Introduction to Source Level Debugging and the M16C Development Environment.

This chapter also contains the section *Getting Started*, which shows you how to compile a program to work with the debugger.

1.2 CROSSVIEW PRO'S FEATURES

CrossView Pro is TASKING's high-level language debugger. CrossView Pro is a real-time, source-level debugger that lets you debug embedded microprocessor systems at your highest level of productivity. Its powerful capabilities include:

- Multi-Window Graphical User Interface
- C and Assembly level debugging
- C Expression Evaluation including Function Calls
- Code and Data Breakpoints
- Assertions (software data breakpoints)
- C-trace, Instruction Trace
- Simulated I/O
- Data Monitoring
- Single Stepping
- Coverage
- Profiling
- Macros
- Flexible Record & Playback Facilities
- Real-Time Kernel Support
- On-line context sensitive Help
- Documentation

Multi-Window Interface

This interface uses your host's native windowing system, so that you already know how to open, close and resize windows. With windows you can keep track of information concerning registers, the stack, and variables. CrossView Pro automatically updates each window whenever execution stops.

You have great freedom in designing a suitable display. You can hide and resize the various windows if you choose.

Statement Evaluation

You can enter C expressions, CrossView Pro commands or any combination of the two for CrossView Pro to evaluate. You may also call functions defined in your source code from the command line. Expression evaluation is an ideal way to test subroutines by passing them sample values and checking the results.

Code Breakpoints

Code breakpoints let you halt the program at critical junctures of program execution and observe values of important variables.

Data Breakpoints

You may place data breakpoints to determine when memory addresses are read from, written to, or both. With data breakpoints, you can easily track the use and misuse of variables. Data breakpoints are not supported by all execution environments.

Assertions

A powerful assertion mechanism lets you catch hard-to-find errors. An assertion is a command, or series of commands, executed after every line of source code. You may use assertions to test for all sorts of error conditions throughout the entire length of your program.

C-Trace

CrossView Pro has a separate window that displays the most recently executed C statements or machine instructions. This feature uses the execution environment's trace buffer along with symbolic information generated during compilation. This feature is depending on the execution environment.

Simulated Input/Output

With Simulated I/O you can debug programs before the actual input and output devices are present. Input data from the keyboard or file, or output to a window or a file. You can view the data in several formats, including hexadecimal and character. You can have up to eight separate simulated I/O ports, which can be associated with the screen and displayed in windows.

Data Monitoring

You may place variables and expressions in the Data window, where CrossView Pro updates their values when execution stops.

Single Stepping

With CrossView Pro, you can single step through your code at source level or at assembly level, into or over procedure calls. Running your program one line at a time lets you check variables and program flow.

Coverage

When a command such as StepInto or Continue executes the application, CrossView Pro traces all memory access, i.e. memory read, memory write and instruction fetch. Through code coverage you can find executed and non-executed areas of the application program. Areas of unexecuted code may exist because of programming errors or because of unnecessary code. It may be that your program input, your test set, is incomplete; It does not cover all paths in the program. Data coverage allows you to verify which memory locations, i.e. which variables, are accessed during program execution. Additionally, you can see stack and heap usage. The availability of this feature depends on the execution environment.

Profiling

Profiling allows you to perform timing analysis on your software. Two forms of profiling are implemented in CrossView Pro.

Function profiling, also called cumulative profiling, gives you timing information about a particular function or set of functions. CrossView Pro shows: the number of times a function is called, the time spent in the function, the percentage of time spent in the function, and the minimum/maximum/average time spent in the function. The timing results include the time spent in functions called by the profiled function.

Code range profiling presents timing information about a consecutive range of program instructions. CrossView Pro displays the time consumed by each line (source or disassembly) in the Source Window. Next to this, the Profile Report dialog shows the time spend in each function. The timing results do not include the time consumed in functions called by the profiled function.

The availability of profiling depends on the execution environment. Function profiling can be supported if the execution environment provides a clock that starts and stops whenever execution starts and stops. Code range profiling heavily relies on special profiling features in the execution environment. Normally code range profiling is only supported by instruction set simulators.

Macros

Macros let you store and recall complex commands and expressions with a minimal number of keystrokes. You can store macros in a "toolbox", making it possible to execute complex functions with the touch of a mouse button. You can also place macros in command lists of breakpoints and assertions. You can use flow control statements within macros, and macros can call other macros, allowing you to construct arbitrarily complex sequences. Macros can accept multiple parameters, be saved and loaded from files and can even rename existing CrossView Pro commands.

Record & Playback

At any time, you can record the commands you type, and optionally their output, to a file. You can also play back files of commands all at once or in a single-step playback mode. These functions are helpful for setting up standardized debugging tests or to save results for later study or comparison.

Kernel Support

CrossView Pro supports RTOS (Real-Time Operating System) aware debugging for various kernels. Since each kernel is different, the RTOS aware features are not implemented in the CrossView Pro executable, but in a library that will be loaded at run-time by CrossView Pro. The amount of windows and dialogs and their contents is kernel dependent.

On-Line Help

All the major windows and dialog boxes contain a **Help** button. Clicking on this button wherever it appears, or pressing the function key **F1**, opens the CrossView Pro help system at the appropriate section. From this point, you can also access the rest of the help system. The MS-Windows version of CrossView Pro uses the native help system.

Documentation

CrossView Pro has a comprehensive set of documentation for both new and experienced users. The manual includes an installation guide, description of debugging with CrossView Pro, error messages, and a command reference section. The documentation tries to cover a wide range of expertise, by making few assumptions about the technical experience of the reader.

1.3 SOURCE LEVEL DEBUGGING

CrossView Pro is a source level debugger. **Source level** means that debugging works on the actual C code. Traditional debuggers are symbolic, not source, level. Generally speaking, symbolic debuggers are limited to dealing with global, non-dynamic variables and know nothing of data types. Symbolic debuggers translate global names and global subroutines into addresses. A symbolic debugger usually cannot deal with variables that are placed on the stack, since stack resident data do not have absolute addresses.

CrossView Pro, unlike a symbolic debugger, can deal with global and local variables that are both statically and dynamically allocated variables. Therefore, it can deal with compiled addresses of variables that move around the stack. CrossView Pro knows the compiler's addressing conventions for variables of any type.

The Debugging Environment

All debugging configurations follow a similar pattern. There is a **host** system where the debugger runs, and a **target** system (usually an execution environment), where the program being debugged runs. There may also be a **probe** that can plug into the actual hardware of the embedded system being designed.

CrossView Pro provides a high-level interface between you, the user, working at the host system and a program running at the target system (execution environment). This means that you may issue commands that refer directly to the variables, source files, and line numbers as they appear in the source program. You can do this because CrossView Pro uses symbol information generated during compilation to translate the high-level commands that you type into a series of low level instructions that the target system understands. Using a connection (usually an RS-232 cable or LAN) between the host and target, CrossView Pro finds out information about the state of the target program and then tells the target to perform the requested actions.

A host-target arrangement can perform functions beyond the reach of traditional software-based debuggers. Since the target contains the actual chip, CrossView Pro can observe its operations without interfering. The existence of CrossView Pro and the host is invisible to the target program. This means that the program under debug runs exactly the same as the final program will in a real embedded system (except for real-time situations like timings).

With CrossView Pro, you may also take advantage of any advanced capabilities of your target hardware through **emulator mode** (transparency mode). In transparency mode you can communicate with the target as if the host system were a terminal directly connected to the target. You can enter and leave transparency mode freely without restarting the debugger or the target system. CrossView Pro therefore does not interfere with the normal operation of the target hardware. Thus the debugger is a powerful accessory to the machine-level debugging that you might do with the target system alone. The transparency mode is not available for all execution environments.

1.4 HOW CROSSVIEW PRO WORKS

Although it is not necessary to know how CrossView Pro performs its debugging, you may be curious how CrossView Pro works.

Whenever you enter a debugger command, CrossView Pro obtains information from or controls the execution environment by sending appropriate commands over the host-target link. A typical session may go something like this:

1. Highlight `initval` and click on the Show selected source expression button in the Source Window.

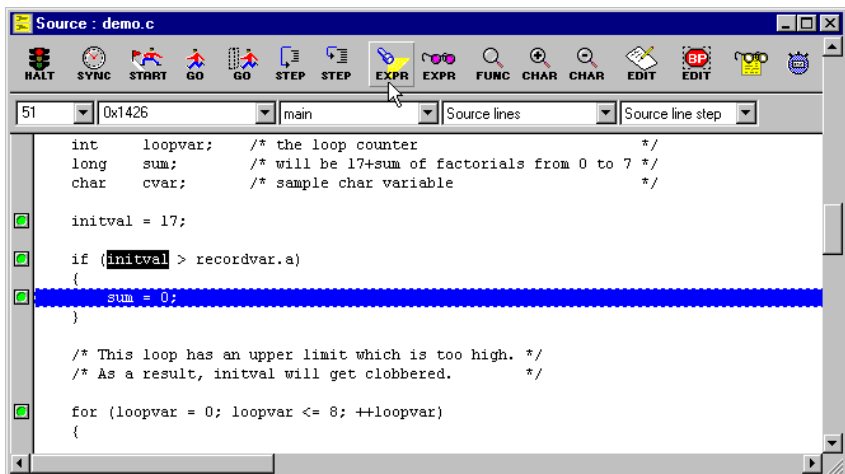


Figure 1-1: Inspect a variable

2. CrossView Pro converts this action into a command. Depending on preferences you have set, the variable is shown in the Data Window or the Expression Evaluation dialog is shown.
3. CrossView Pro consults the symbol table to deduce the type and address of `initval`. Suppose `initval` is a variable of type `int` which lies at absolute location 100.
4. The debugger forms a command asking the target system to read two bytes starting at address 100 (the size of an `int` equals 2).
5. CrossView Pro then transmits the command to the target system and receives the response.

6. CrossView Pro interprets the response, and for example determines that `initval` equals 17.
7. CrossView Pro then displays `initval=17` since it knows `initval`'s type.

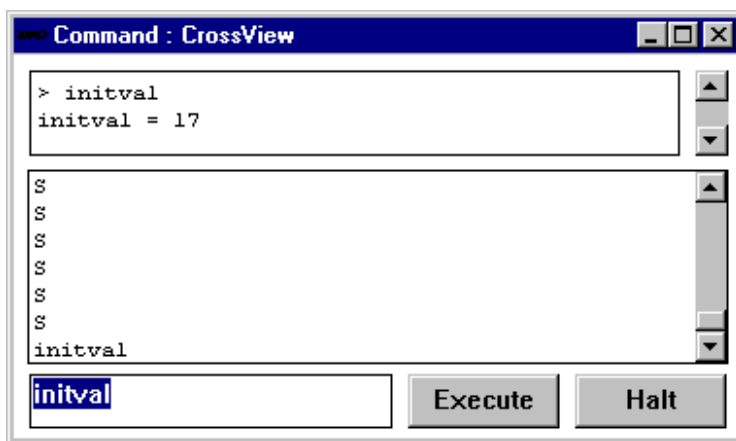


Figure 1-2: CrossView Pro Command Output

This is a simplified example, many CrossView Pro commands require several complex transactions, but all take place without you being aware of them.

1.5 M16C PROGRAM DEVELOPMENT

The CrossView Pro debugger package is part of a toolchain that provides an environment for modular program development and debugging. The figure below shows the structure of the toolchain. The toolchain contains the following programs:

- | | |
|---------------|---|
| ccm16 | The control program which activates the C compiler, assembler, linker and/or locator depending on its input. |
| cm16 | The M16C C compiler. This is a dedicated M16C C compiler which translates a C source program into a highly optimized assembly source file, using the M16C assembly language specification. |
| asm16 | The assembler program which produces a relocatable object file from a given assembly file. |
| lkm16 | A linker combining objects and object libraries into one relocatable object file. |
| lcm16 | A locator that links an arbitrary number of linker output files to one absolute load file in the IEEE Std. 695 debugging connection format. This program can also produce files in the Intel Hex format or Motorola S-record. |
| arm16 | A librarian program, which can be used to create and maintain object libraries. |
| prm16 | A utility to view the contents of a relocatable object file or an absolute file. |
| mkm16 | A program builder which uses a set of dependency rules in a 'makefile' to build only the parts of an application which are out of date. |
| xfwm16 | The CrossView Pro debugger using M16C execution environments. |

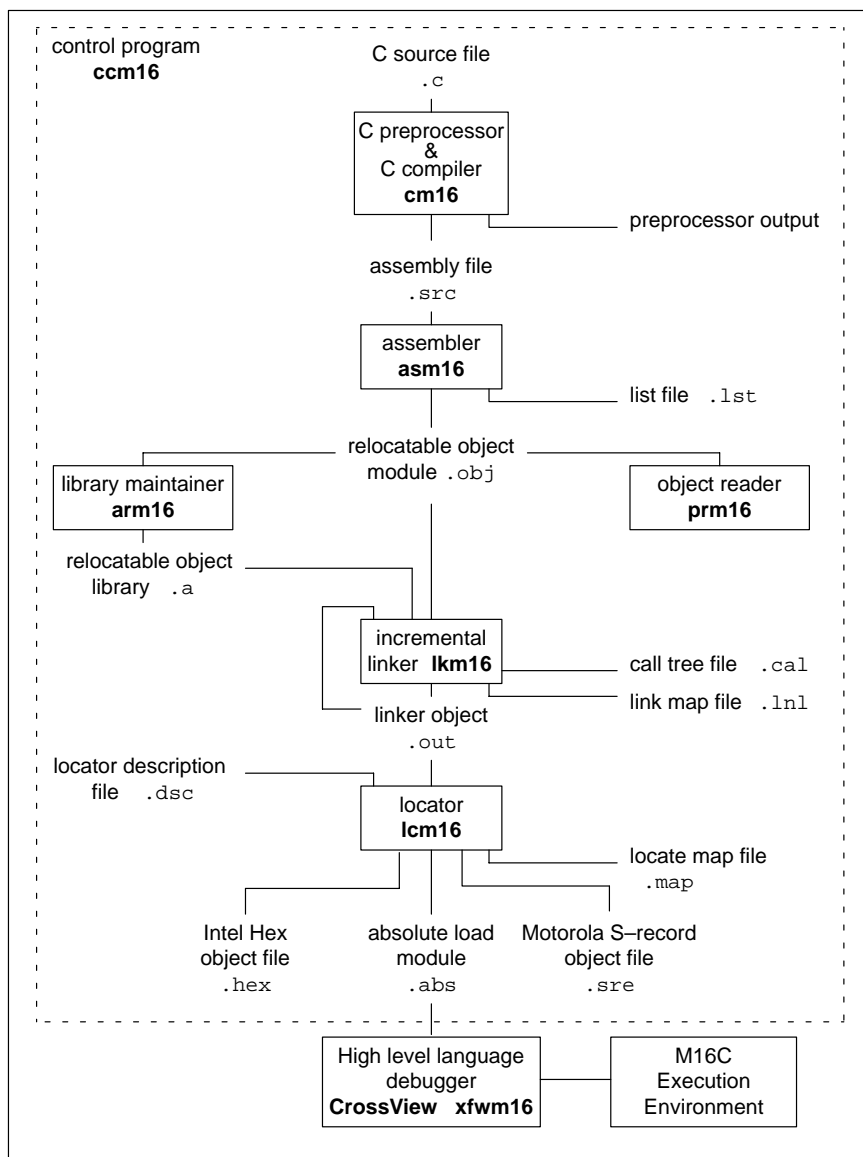


Figure 1-3: M16C development flow

For a full description of all available utility programs see the chapter *Utilities* in the *M16C Cross-Assembler User's Guide*.

1.6 GETTING STARTED

1.6.1 BEFORE STARTING

Before using CrossView Pro, there are several things that you must do:

- Install the CrossView Pro software. Directions for your particular system are found in the *Software Installation* chapter.
- Configure your execution environment as described in the *Execution Environment* addendum.
- Compile the program that you want to debug. A brief description of this process is outlined in the section *Building Your Executable* later in this chapter.

For the purpose of getting you started quickly, we have supplied you with a demo program that you can debug. The demo program is `demo.abs`.

1.6.2 SETTING UP THE EXECUTION ENVIRONMENT

The following text only applies to ROM monitor and emulator versions of CrossView Pro. Within CrossView Pro simulator versions the execution environment is part of the CrossView Pro executable.

In order for the host and execution environment to communicate, a proper connection must exist between the two machines. Here are some important considerations:

- Use the correct kind of RS-232 cable. Note there are at least two types of cables, *null modem* and *direct*. Consult the execution environment's manual for the correct type.
- Make sure the execution environment is configured to communicate with the host at the baud rate that CrossView Pro expects. Usually, the baud rate is 9600, but this is not always the case.
- Use the correct ports on both the execution environment and host. Many machines have two ports. If you use a different port on the host than the default (COM1 for PC), you will have to use a special startup switch, **-D**. See the startup options of the *Using CrossView Pro* chapter.
- See the addendum for details on the connection to the execution environment.

1.6.3 STARTING CROSSVIEW PRO

To invoke CrossView Pro, simply double-click on its icon. CrossView Pro starts up and opens the command window, source window and other windows.

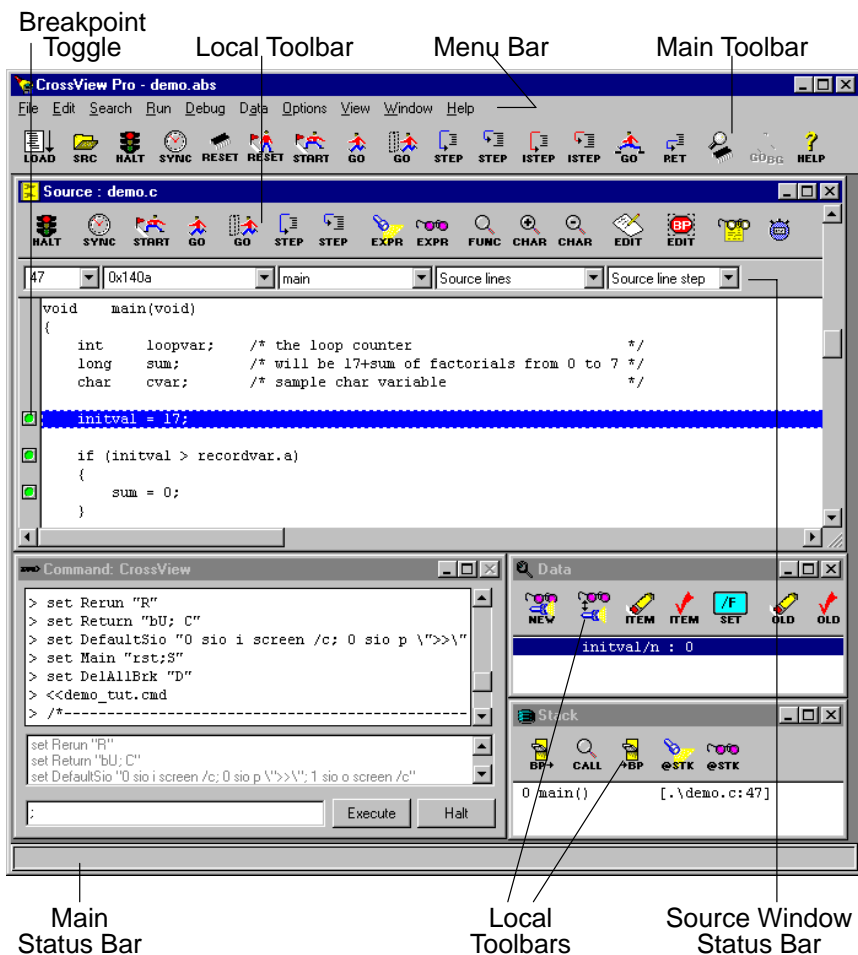


Figure 1-4: Command Window

CrossView Pro can be passed the name of an execution (*.abs) file. This can be done from a command line, but the native windowing system often provides alternatives. Usually this involves dragging the program to be debugged onto the CrossView Pro executable from the Windows Explorer for Windows 95/98/NT/2000, and dropping it there or associating CrossView Pro to be the application to start when double-clicking an .abs icon. CrossView Pro will start and load the symbol information from that file.

1.6.3.1 CROSSVIEW PRO STARTUP SETTINGS

You can specify specific CrossView Pro startup settings in the CrossView Startup dialog.

To open the CrossView Startup dialog:

- Select the Options | Startup | CrossView... item from the menu. This option opens up the CrossView Startup dialog box as shown in figure 1-5.

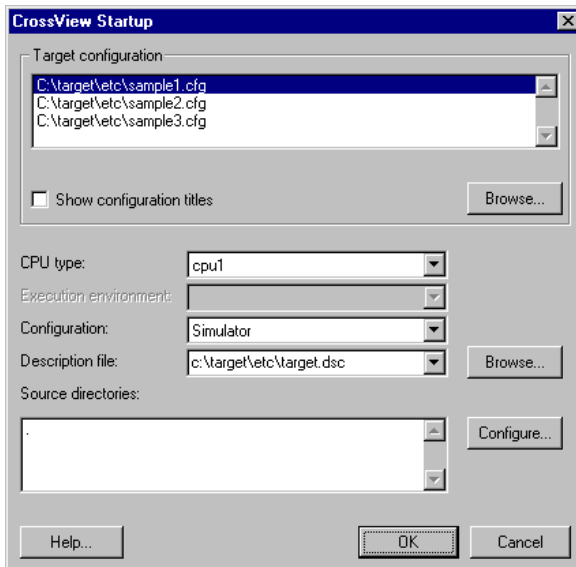


Figure 1-5: CrossView Pro Startup Settings

You can set the following items in this dialog:

- Select a target configuration containing some target specific configuration items. See the text below for more information.
- Select the CPU type.
- Specify a description file (*.dsc). The description file must be the same file as you used to locate your application.
- Select the execution environment (not available for M16C).
- Specify the source directories for CrossView Pro. Click on the `Configure...` button to change the list of source directories.

Target Configuration

The available targets are described by the target configuration files (*.cfg in the `etc` subdirectory). The target configuration files are text files and can be edited with any text editor.

Empty lines, lines consisting of only white space are allowed. Comment starts at an exclamation-sign (!) and ends at the end of the line.

The following configuration items are defined in a target configuration file:

<code>title:</code>	title of the configuration file.
<code>cpu_type:</code>	cpu type. You can specify multiple cpu types separated by white space.
<code>debug_instrument_module:</code>	name of the DLL used for debugging.
<code>kasm_dll_name:</code>	name of the DLL used for RTOS aware debugging (optional).

For each target a different set of extensions can be added to this list.

1.6.3.2 CONFIGURING CROSSVIEW PRO

You may have to configure CrossView Pro to talk to the emulator or ROM monitor. If you have a simulator version this step is not needed and the associated menu item is grayed. To configure CrossView Pro:

- Select the `File | Communication Setup...` item from the menu. This option opens up the `Communication Setup` dialog box as shown in figure 1-6.

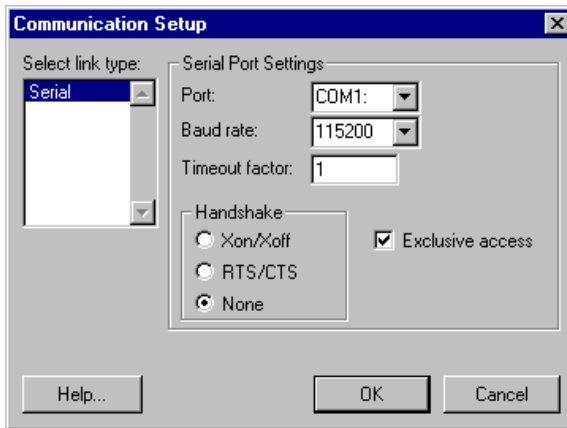


Figure 1-6: Setting up CrossView Pro Communications

- Adjust the communication parameters (baud rate and I/O port) to match your hardware configuration.
- Close the dialog box by clicking on the OK button.
- The settings in this dialog (and other dialogs) will be saved on exiting CrossView Pro, when the Desktop and target settings check box in the Save Options dialog is set. This dialog always appears on exiting CrossView Pro.



From EDE you can set the communication parameters in the Communications tab of the EDE | CrossView Pro Options... menu item.

1.6.3.3 LOADING SYMBOLIC DEBUG INFORMATION

You must tell CrossView Pro which program that you want to debug. To do this:

- Select File | Load Symbolic Debug Info... from the menu. This opens up the Load Symbolic Debug Info dialog box, as shown in figure 1-7.
- Type in the path and file name of the program that you want to debug, or click on the File... button to bring up a file selection dialog box. In our example we are using demo.abs. Note that in most cases you will want to set the code bias field to 0x0000.

- Set the `Download image too` check box by clicking on it, if you want to download the image of your absolute object file to the target. You can decide to postpone downloading to the target. In that case you can select the menu item `File | Download Image` any time afterwards.
- Set the `Target system reset` check box by clicking on it, if you want to reset the target system to its initial state. You can decide to postpone resetting the target. In that case you can select the menu item `Run | Target System Reset` afterwards.
- Set the `Goto main` check box by clicking on it, if you want to execute the startup code. This automatically enables the `Program reset` check box. You can decide to postpone to goto the main function. In that case you can execute a high-level single step afterwards.
- Clicking on the `Communication setup...` button, if not grayed, opens the `Communication Setup` dialog box as shown in figure 1-6. With the `startup options...` buttons you can open the `CrossView Startup` dialog and, if the button is not grayed, the `Emulator Startup` dialog. Please check the information in these dialogs before downloading an application.
- Clicking on the `Load` button will load the program's symbol file into the debugger, and will download the image of your absolute object file if you have set the `Download image too` check box.
- Clicking on `Cancel` ignores all actions.

CrossView Pro remembers all previously saved settings. In this case, the Load Symbolic Debug Info dialog already contains the previously saved configuration, so you only have to click the Load button to perform your actions.

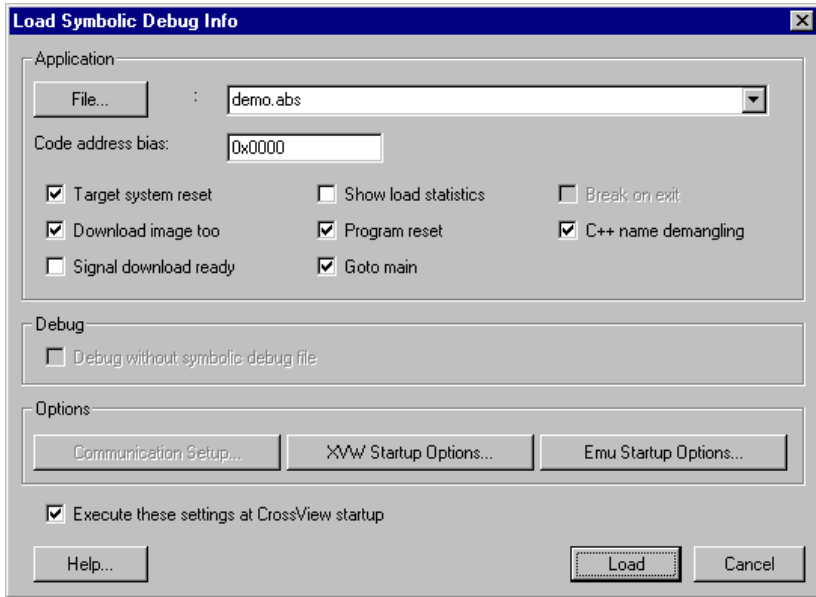


Figure 1-7: Loading Symbolic Debug Information

1.6.4 EXECUTING AN APPLICATION

To view your source while debugging, the Source Window must be open. To open this window,

- Select View | Source->Source lines from the menu.

Before starting execution you have to reset the target system to its initial state. The program counter, stack pointer and any other registers must be set to their initial value. The easiest way to do this is:

- Set the Target system reset check box and the Goto main check box in the Load Symbolic Debug Info dialog box. (See the previous section) Goto main automatically enables the Program reset



Depending on your execution environment a target system reset may have undesired side effects. For this reason, the target system reset is executed before the code is downloaded to the target.

If you have not checked these items:

- Select Run | Target system Reset.
- Select Run | Program Reset.
- Execute a high-level single step (either into or over) using the accelerator bar in the Source Window (or F8/F10).

The first single step executes the startup code and stops at the first line of code in `main()`. You should see your program's source code.

Another way of getting there is:

- Set a breakpoint at the entry of `main()` by clicking on a breakpoint toggle at the left side of the text in the Source Window. See figure 1-8.
- Start the application using Run | Program Reset and Run | Run.

To set a breakpoint you can:

- Click on a breakpoint toggle (as shown in figure 1-8) to set or to remove a breakpoint. A green colored toggle shows that no breakpoint is set. A red colored toggle shows that a breakpoint is installed. An orange colored toggle shows that an installed breakpoint is disabled.

Due to compiler optimizations it is possible that a C statement does not translate in any executable code. In this case you cannot set a breakpoint at such a C statement. No breakpoint toggle is shown in this case.

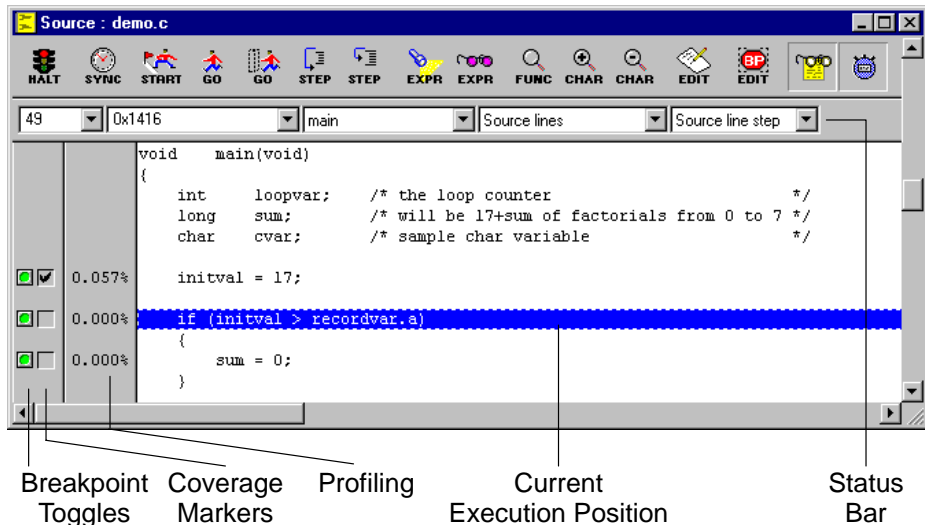


Figure 1-8: Getting Control

Now it is time to execute your program:

- Select Run | Run from the menu.

In the Source Window the current execution position (i.e. the statement at the address identified by the current value of the program counter) is highlighted in blue. As a result, when execution stops, the line you set a breakpoint on is highlighted. You can now single step through your program using the Step Into and Step Over buttons in the Source Window. Or you may choose to execute the rest of the program (or at least until the next breakpoint) with the Run button.

At any point you can interrupt the emulator and regain control by clicking on the Halt button in either the Source Window or the Command Window.

For more information on executing a program, see the chapter *Controlling Program Execution*.

1.6.5 DEBUGGING AN APPLICATION

When debugging your application you probably want to see the calling sequence of your program, and inspect the contents of variables and data structures used within your program.

To see the calling sequence of your program the Stack Window must be open. The stack window shows the functions that are currently on the stack. To open the stack window,

- Select View | Stack from the menu.

To see the value of the local variables of a function,

- Select View | Data | Watch Locals Window from the menu.

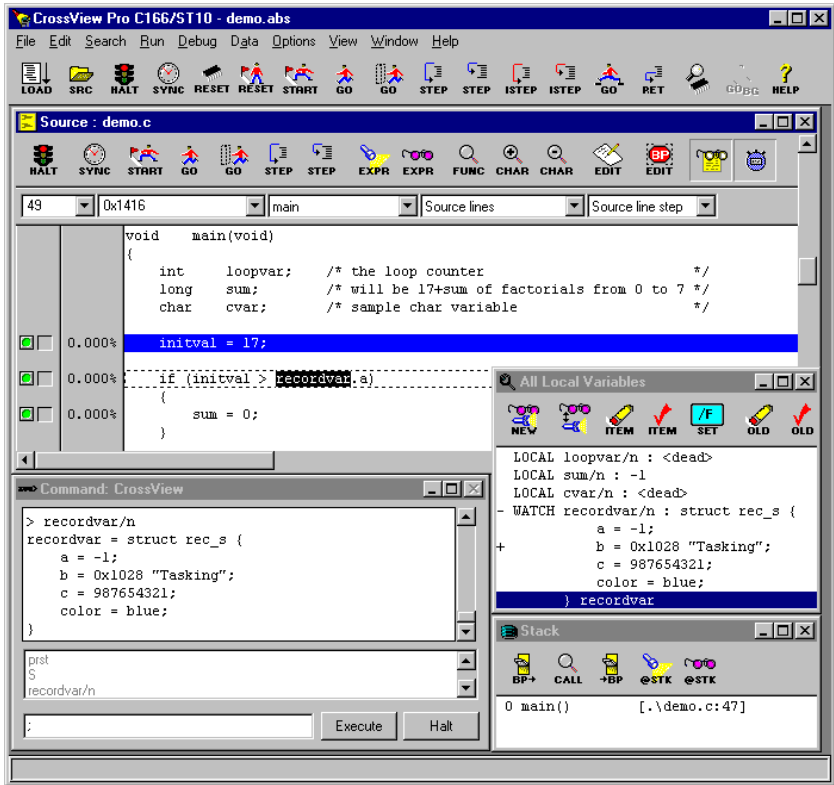


Figure 1-9: Watch variables

To inspect the value of global variables and data structures,

- Double-click on the variable name in the Source Window.

Depending on preferences you have set, the variable is shown in the Data Window as shown in figure 1-9 or the dialog displayed in figure 1-10 is shown.

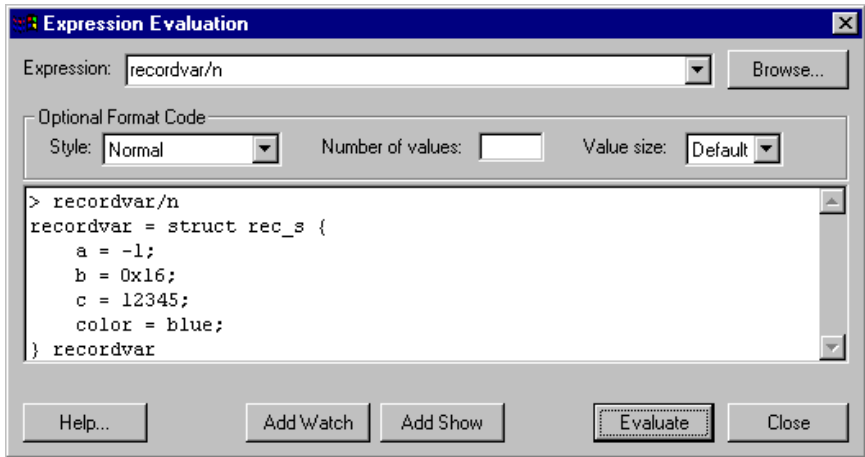


Figure 1-10: Expression evaluation

Pointers, structures and arrays displayed in the data window have a compact and expanded form. The compact form for a structure is just `<struct>`, while the expanded form shows all the fields. The compact form of a pointer is the value of the pointer, while the expanded form shows the pointed-to object. The compact form is indicated by putting a '+' at the start of the display. (i.e., the object is expandable), while a '-' indicates the expanded form (i.e., the object is contractible). Nesting is supported, so structures within structures can likewise be expanded, ad infinitum.

To expand a pointer or a structure:

- Click on the '+' in the Data Window

1.6.6 CROSSVIEW PRO OUTPUT

Nearly every CrossView Pro command can be given using the graphical user interface. These commands and the debugger's response is logged in the Command Output Window which is the upper part of the Command Window. Alternatively, CrossView Pro commands can be entered directly (without using the menu system) in the command edit field of the command window.

To open the Command Window:

- Select View | Command->CrossView from the menu.

Figure 1-11 shows an example of the Command Window. Commands can be typed into the command edit field (bottom field) or selected from the command history list (middle field) and edited then executed. The top field is referred to as the Command Output Window. Each command, echoed from the command edit field, is displayed with a '>' prefix. CrossView's response to the command is displayed below the command.

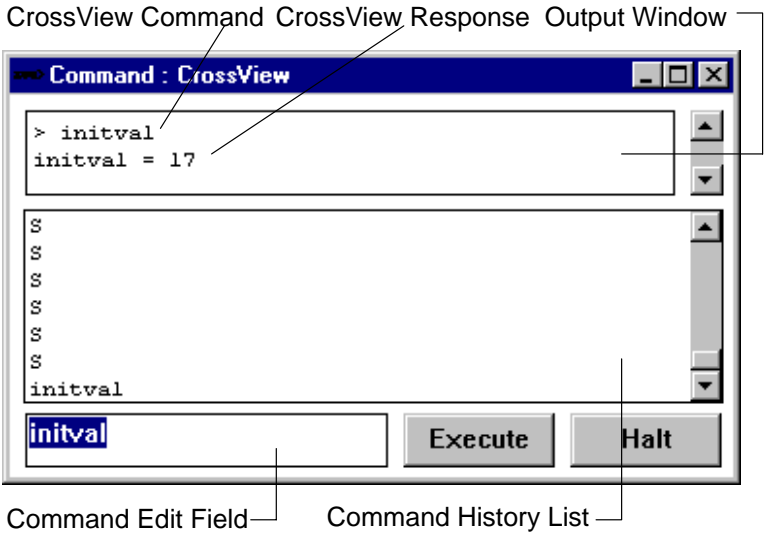


Figure 1-11: CrossView Pro Command Output

You can change the behavior of the command edit field in the Options | Desktop Setup dialog.

1.6.7 EXITING CROSSVIEW PRO

To quit a debugging session:

- Select the **File | Exit** menu item or close the Command Window.
- In the Save Options dialog that appears, select the options you want to be saved for another debug session.
- Click on the **Exit** button in the Save Options dialog.

CrossView Pro will exit immediately. If you selected one or more items in the Save Options dialog your settings will be saved in the initialization file `xvw.ini`. This file is located in the startup directory.

1.6.8 WHAT YOU MAY HAVE DONE WRONG

Most problems in starting up CrossView Pro for a debugging session stem from improperly setting up the execution environment or from an improper connection between the host computer and the execution environment. Some targets will require you to enter transparency mode to set the execution environment for a debugging session. Check the notes for your particular execution environment.

Here are some other common problems:

- Specifying the wrong device name when invoking the debugger.
- Specifying a baud rate different from the one the execution environment is configured to expect.
- Not supplying power to the execution environment or an attached probe.
- Using the wrong kind of RS-232 cable.
- Plugging the cable into an incorrect port on the execution environment or host. Some target machines and hosts have several ports.
- Installation of a device driver or resident application that uses the same communications port on the host system.
- The port may already be in use by another user on some UNIX hosts, or being allocated by a login process.
- Specifying no or an invalid cpu type with the **-C** option.

1.6.9 BUILDING YOUR EXECUTABLE

The subdirectory `xvw` in the `examples` subdirectory contains a demo program for the M16C toolchain.

In order to debug your programs, you will have to compile, assemble, link and locate them for debugging using the TASKING M16C tools. You can do this with one call to the control program or you can use EDE, the Embedded Development Environment (which uses a project file and a makefile) or you can call the makefile from the command line.

1.6.9.1 USING EDE

EDE stands for "Embedded Development Environment" and is the Windows oriented Integrated Development Environment you can use with your TASKING toolchain to design your application.

To use EDE on the demo program, located in the subdirectory `xvw` in the `examples` subdirectory of the M16C product tree, follow the steps below.

A detailed description of the process creating the sample program `demo.abs` is described below. This procedure is outlined as a guide for you to build your own executables for debugging.



The dialog boxes shown in this manual serve as an example. They may slightly differ from the ones in your product.

How to Start EDE

You can launch EDE by double-clicking on the EDE shortcut on your desktop.



The EDE screen provides you with a menu bar, a toolbar (command buttons) and one or more windows (for example, for source files), a status bar and numerous dialog boxes.

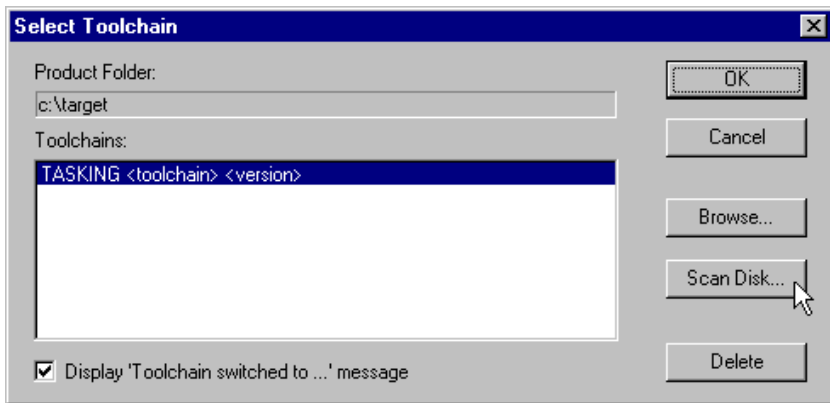


How to Select a Toolchain

EDE supports all the TASKING toolchains. When you first start EDE, the toolchain of the product you purchased is selected and displayed in the title of the EDE desktop window.

If you selected the wrong toolchain or if you want to change toolchains do the following:

1. Access the EDE menu and select the `Select Toolchain...` menu item. This opens the `Select Toolchain` dialog.
2. Select the toolchain you want. You can do this by clicking on a toolchain in the `Toolchains` list box and press OK.



If no toolchains are present, use the `Browse...` or `Scan Disk...` button to search for a toolchain directory. Use the `Browse...` button if you know the installation directory of another TASKING product. Use the `Scan Disk...` button to search for all TASKING products present on a specific drive. Then return to step 2.

How to Open an Existing Project

Follow these steps to open an existing project:

1. Access the `Project` menu and select `Set Current...`
2. Select the project file to open and then click OK. For the demo program select the file `sim.pjt`, located in the subdirectory `xvw` in the `examples` subdirectory of the M16C product tree. If you have used the defaults, the file `sim.pjt` is in the directory `c:\cm16c\examples\xvw`.

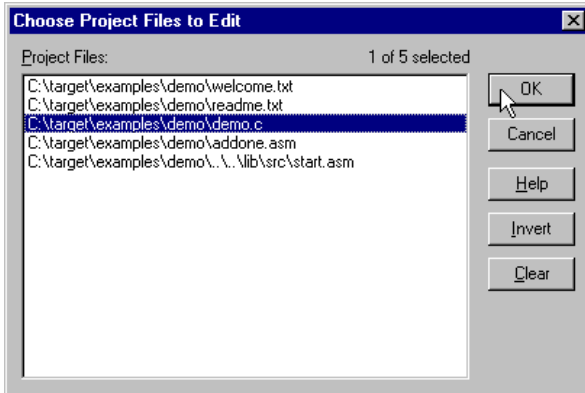
How to Load/Open Files

The next two steps are not needed for the demo program because the files `addone.src` and `demo.c` are already open. To load the file you want to look at:

1. In the **Project** menu click on **Load files...**

This opens the **Choose Project Files to Edit** dialog.

2. Choose the file(s) you want to open by clicking on it. You can select multiple files by pressing the **<Ctrl>** or **<Shift>** key while you click on a file. With the **<Ctrl>** key you can make single selections and with the **<Shift>** key you can select everything from the first selected file to the file you click on. Then press the **OK** button.



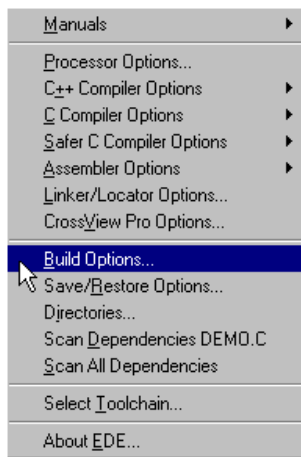
This launches the file(s) so you can edit it (them).

How to Build the Demo Application

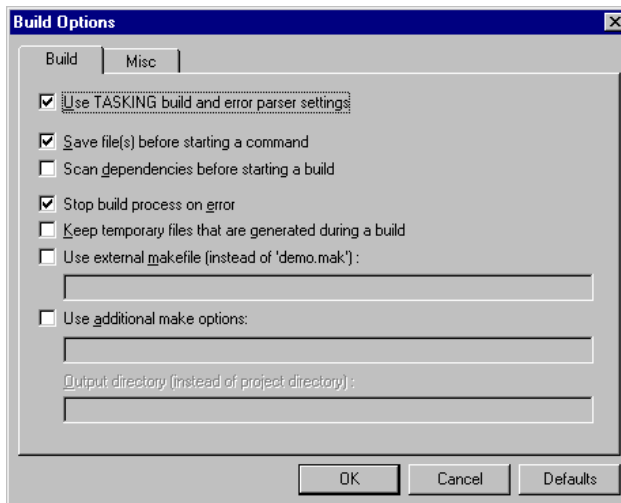
The next step is to compile the file(s) together with its dependent files so you can debug the application.

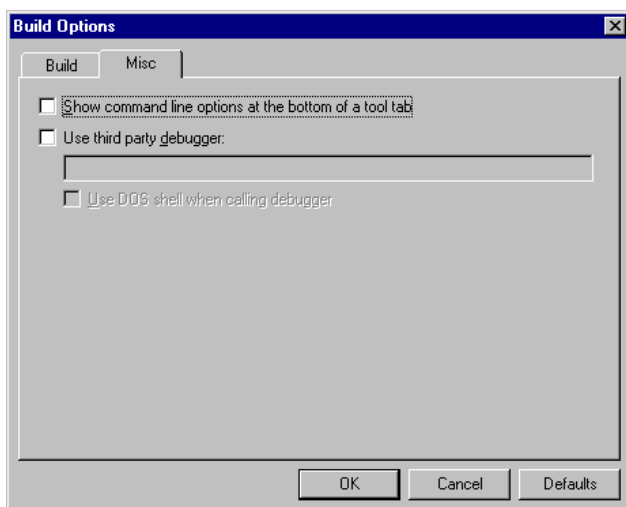
Steps 1 and 2 are optional. Follow these steps if you want to specify additional build options such as to stop the build process on errors and to select a command to be executed as foreground or background process.

1. Access the **EDE** menu and select the **Build Options...** menu item.

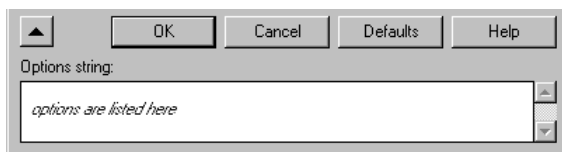


This opens the Build Options dialog.

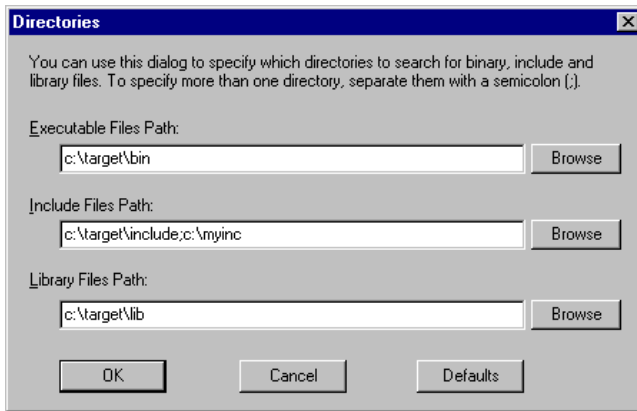




If you set the Show command line options at the bottom of a tool tab check box EDE shows the command line equivalent of the selected tool option. You can also click on the arrow button (left of the OK button) in a tool options dialog.



2. Make your changes and press the OK button.
3. Select the EDE | Directories menu item and check the directory paths for programs, include files and libraries. You can add your own directories here, separated by semicolons.



4. Access the EDE menu and select the Scan All Dependencies menu item.
5. Click on the Execute 'Make' command button. The following button is the execute Make button which is located in the toolbar.



If there are any unsaved files, EDE will ask you in a separate dialog if you want to save them before starting the build.

How to View the Results of a Build

Once the files have been processed you can inspect the generated messages in the Build tab:

```
TASKING program builder vx.y rz          SN00000001-020 (c) year TASKING, Inc.
Compiling "demo.c"
Assembling "demo.src"
Assembling "addone.src"
Linking to "sim.out"
Locating "sim.out" to "sim.abs" (IEEE-695)
```

How to Start the CrossView Pro Debugger

Once the files have been compiled, assembled, linked, located and formatted they can be executed by CrossView Pro.

To execute CrossView Pro:

1. Click on the `Debug` application button. The following button is the `Debug` application button which is located in the toolbar.



CrossView Pro is launched. CrossView Pro will automatically download the compiled file for debugging.

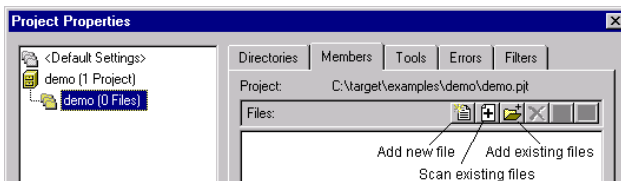
How to Start a New Project

When you first use EDE you need to setup a project space and add a new project:

1. Access the `Project` menu and select `Project Space | New...`
2. Give your project space a name and then click OK.
3. Click on the `Add new project to project space` button.
4. Give your project a name and then click OK.

The `Project Properties` dialog box then appears for you to identify the files to be added.

5. Add all the files you want to be part of your project. Then press the OK button. To add files, use one of the 3 methods described below.



- If you do not have any source files yet, click on the `Add new file to project` button in the `Project Properties` dialog. Enter a new filename and click OK.

- To add existing files to a project by specifying a file pattern click on the `Scan existing files into project` button in the `Project Properties` dialog. Select the directory that contains the files you want to add to your project. Enter one or more file patterns separated by semicolons. The button next to the `Pattern` field contains some predefined patterns. Next click `OK`.
- To add existing files to a project by selecting individual files click on the `Add existing files to project` button in the `Project Properties` dialog. Select the directory that contains the files you want to add to your project. Add the applicable files by double-clicking on them or by selecting them and pressing the `Open` button.

The new project is now open.

6. Click `Project | Load Files` to open files you want on your EDE desktop.

EDE automatically creates a makefile for the project. EDE updates the makefile every time you modify your project.

1.6.9.2 USING THE CONTROL PROGRAM

A detailed description of the process creating the sample program `demo.abs` is described below. This procedure is outlined as a guide for you to build your own executables for debugging.

1. Make the subdirectory `xvw` of the `examples` directory the current working directory.
2. Be sure that the directory of the binaries is present in the `PATH` environment variable.
3. Compile, assemble, link and locate the modules using one call to the control program **`ccm16`**:

```
ccm16 -g -M demo.c addone.src -o demo.abs
```

The **`-g`** option specifies to generate symbolic debugging information. This option must always be specified when debugging with CrossView Pro.

The **`-M`** option specifies to generate map files.

The **`-o`** option specifies the name of the output file.

The command in step 3 generates the object files `demo.obj` and `addone.obj`, the linker map file `demo.lnl`, the locator map file `demo.map` and the absolute output file `demo.abs`. The file `demo.abs` is in the IEEE Std. 695 format, and can directly be used by CrossView Pro. No separate formatter is needed.

Now you have created all the files necessary for debugging with CrossView Pro using one call to the control program.

If you want to see how the control program calls the compiler, assembler, linker and locator, you can use the **-v0** option or **-v** option. The **-v0** option only displays the invocations without executing them. The **-v** option also executes them.

```
ccml6 -g -M demo.c addone.src -o demo.abs -v0
```

The control program shows the following command invocations without executing them (UNIX output):

```
M16C control program vx.y rz          SN00000000-003 (c) year TASKING, Inc.
demo.c:
+ cml6 -e -g -Ms -o /tmp/cc26583b.src demo.c
+ asml6 /tmp/cc26583b.src -e -g -o demo.obj
addone.src:
+ asml6 addone.src -e -g -o addone.obj
+ lkm16 -e -M demo.obj addone.obj -lcs -lms -lfps -lrts -odemo.out -Odemo
+ lcm16 -e -M -odemo.abs -dm16c.dsc demo.out
```

The **-e** option removes output files after errors occur. The **-Ms** option selects the small memory model. The **-lcs**, **-lfps** and **-lrts** options of the linker specify to link the appropriate C libraries. The **-O** option of the linker specifies the basename of the map file. The **-d** option of the locator specifies the name of the locator description file.

As you can see, the tools use temporary files for intermediate results. Also the file `demo.out` will be removed afterwards. If you want to keep the intermediate files you can use the **-tmp** option. The following command makes this clear.

```
ccml6 -g -M demo.c addone.src -o demo.abs -v0 -tmp
```

This command produces the following output:

```
M16C control program vx.y rz          SN00000000-003 (c) year TASKING, Inc.
demo.c:
+ cm16 -e -g -Ms -o demo.src demo.c
+ asm16 demo.src -e -g -o demo.obj
addone.src:
+ asm16 addone.src -e -g -o addone.obj
+ lkm16 -e -M demo.obj addone.obj -lcs -lms -lfps -lrts -odemo.out -Odemo
+ lcml6 -e -M -odemo.abs -dm16c.dsc demo.out
```

As you can see, if you use the **-tmp** option, the assembly source files and linker output file will remain in your current directory.

Of course, you will get the same result if you invoke the tools separately using the same calling scheme as the control program.

As you can see, the control program automatically calls each tool with the correct options and controls.

1.6.9.3 USING THE MAKEFILE

The subdirectories in the `examples` directory each contain a `makefile` which can be processed by **mk16**. Also each subdirectory contains a `readme.txt` file with a description of how to build the example.

To build the demo example follow the steps below. This procedure is outlined as a guide for you to build your own executables for debugging.

1. Make the subdirectory `xvw` of the `examples` directory the current working directory.

This directory contains a `makefile` for building the demo example. It uses the default **mk16** rules.

2. Be sure that the directory of the binaries is present in the `PATH` environment variable.
3. Compile, assemble, link and locate the modules using one call to the program builder **mk16**:

```
mk16
```

This command will build the example using the file `makefile`.

To see which commands are invoked by **mkm16** without actually executing them, type:

```
mkm16 -n
```

This command produces the following output:

```
M16C program builder vx.y rz          SN00000000-003 (c) year TASKING, Inc.  
ccm16 -c -o demo.obj -s -g demo.c  
ccm16 -c -o addone.obj addone.src  
ccm16 -o demo.abs demo.obj addone.obj
```

The **-s** option in the makefile instructs the compiler to include C source lines as comments in the assembly output. The **-g** option is used to instruct the C compiler to generate symbolic debug information. This information makes debugging an application written in C much easier to debug.

To remove all generated files type:

```
mkm16 clean
```

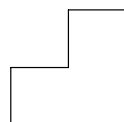
CHAPTER

2

SOFTWARE INSTALLATION



TASKING



2

CHAPTER

2.1 INTRODUCTION

This chapter describes the procedure for the installation of the TASKING CrossView Pro debugger for the M16C on Windows, Linux and several UNIX hosts.

2.2 NOTE ABOUT FILENAMES

Members of the CrossView Pro family of debuggers use the following name convention for their executables:

xfwm16c

2.3 INSTALLATION FOR WINDOWS

Step 1

Start Windows (95/98/NT/2000), if you have not already done so.

Step 2

Insert the CD-ROM into the CD-ROM drive.

If the TASKING Welcome dialog box appears, skip to Step 5. Otherwise, continue from Step 3.

Step 3

Select the Start button and select the Run . . . menu item.

Step 4

On the command line type:

d:\setup

(substitute the correct drive letter for your CD-ROM drive) and press the **<Return>** or **<Enter>** key or click on the OK button.

The TASKING Welcome dialog box appears.

Step 5

Select a product and click on Install.

Step 6

Follow the instructions that appear on your screen.



You can find your serial number on the *Certificate of Authenticity* or *Product Update Form* delivered with the product.

Step 7

Make sure that the directory containing the installed executable files is present in the PATH environment variable, when you invoke the tools from a command prompt.

Step 8

License the software product as explained in section 2.8, *Licensing TASKING Products*.

2.3.1 REQUIREMENTS

The hardware/software requirements are:

- 486 PC or higher
- Windows 95/98, NT or 2000

2.4 INSTALLATION FOR LINUX

Each product on the CD-ROM is available as an RPM package and as a gzipped tar file. For each product the following files are present:

```
SWproduct-version-RPMrelease.i386.rpm  
SWproduct-version.tar.gz
```

Both files contain exactly the same information. When your Linux distribution supports RPM packages, you can install the .rpm file. Otherwise, you can install the product from the .tar.gz file.

2.4.1 RPM INSTALLATION

Step 1

In most situations you have to be "root" to install RPM packages, so either login as "root", or use the **su** command.

Step 2

Insert the CD-ROM into the CD-ROM drive. Mount the CD-ROM on a directory, for example /cdrom. See the Linux manual pages about **mount** for details.

Step 3

Go to the directory on which the CD-ROM is mounted:

```
cd /cdrom
```

Step 4

To install or upgrade all products at once, issue the following command:

```
rpm -U SW*.rpm
```

This will install or upgrade all products in the default installation directory /usr/local. Every RPM package will create a single directory in the installation directory.

The RPM packages are 'relocatable', so it is possible to select a different installation directory with the **--prefix** option. For instance when you want to install the products in /opt, use the following command:

```
rpm -U --prefix /opt SW*.rpm
```



For Red Hat 6.0 users: The **--prefix** option does not work with RPM version 3.0, included in the Red Hat 6.0 distribution. Please upgrade to RPM version 3.0.3 or higher, or use the .tar.gz file installation described in the next section if you want to install in a non-standard directory.

Step 5

Make sure that your path is set to include all of the executables you have just installed.



X Windows is required to run CrossView Pro.

2.4.2 TAR.GZ INSTALLATION

Step 1

Login as a user.

Be sure you have read, write and execute permissions in the installation directory. Otherwise, login as "root" or use the **su** command.

Step 2

Insert the CD-ROM into the CD-ROM drive. Mount the CD-ROM on a directory, for example /cdrom. See the Linux manual pages about **mount** for details.

Step 3

Go to the directory on which the CD-ROM is mounted:

```
cd /cdrom
```

Step 4

To install the products from the .tar.gz files in the directory /usr/local, issue the following command for each product:

```
tar xzf SWproduct-version.tar.gz -C /usr/local
```

Every .tar.gz file creates a single directory in the directory where it is extracted.

Step 5

Make sure that your path is set to include all of the executables you have just installed.



X Windows is required to run CrossView Pro.

2.5 INSTALLATION FOR UNIX HOSTS

Step 1

Login as a user.

Be sure you have read, write and execute permissions in the installation directory. Otherwise, login as root or use the **su** command.

Step 2

If you are a first time user decide where you want to install the debugger (By default it will be installed in `/usr/local`).

Step 3

For CD-ROM install: insert the CD-ROM into the CD-ROM drive. Mount the CD-ROM on a directory, for example `/cdrom`. Be sure to use a ISO 9660 file system with Rock Ridge extensions enabled. See the UNIX manuals page about **mount** for details.

Or:

For tape install: insert the tape into the tape unit and create a directory where the contents of the tape can be copied to. Consider the created directory as a temporary workspace that can be deleted after installation has succeeded. For example:

```
mkdir /tmp/instdir
```

Step 4

For CD-ROM install: go to the directory on which the CD-ROM is mounted:

```
cd /cdrom
```

For tape install: copy the contents of the tape to the temporary workspace using the following commands:

```
cd /tmp/instdir
tar xvf /dev/tape
```

where *tape* is the name of your tape device.



If you have received a tape with more than one product, use the non-rewinding device for installing the products.

Step 5

Run the installation script:

```
sh install
```

and follow the instructions appearing on your screen.

First a question appears about where to install the software. The default answer is **/usr/local**. On certain sites you may want to select another location.

On some hosts the installation script asks if you want to install SW000098, the Flexible License Manager (FLEXlm). If you do not already have FLEXlm on your system, you must install it; otherwise the product will not work on those hosts. See section 2.8, *Licensing TASKING Products*.

If the script detects that the software has been installed before, the following messages appear on the screen:

```
*** WARNING ***
SWxxxxxx xxxx.xxxx already installed.
Do you want to REINSTALL? [y,n]
```

Answering **n** (no) to this question causes installation to abort and the following message being displayed:

```
=> Installation stopped on user request <=
```

Answering **y** (yes) to this question causes installation to continue. And the final message will be:

```
Installation of SWxxxxxx xxxx.xxxx completed.
```

Step 6

For tape install: remove the temporary installation directory with the following commands:

```
cd /tmp
rm -rf instdir
```

Step 7

Make sure that the directory containing the installed executable files is present in the PATH environment variable.

Step 8

If you purchased a protected TASKING product, license the software product as explained in section 2.8, *Licensing TASKING Products*.



X Windows is required to run CrossView Pro.

2.6 CONFIGURING THE X WINDOWS MOTIF ENVIRONMENT

To run the Motif version of CrossView Pro on a Sun, you must define the environment variable **LD_LIBRARY_PATH** to where the library file `libMrm.a` resides. For example:

```
LD_LIBRARY_PATH=/usr/dt/lib  
export LD_LIBRARY_PATH
```

CrossView Pro uses a binary resource file for appearance-related specifications for windows, menus, dialog boxes, and strings to be accessed at run-time. The name of the resource file has the same name as the executable but with `.uid` extension. Be sure that the `.uid` file is present in one of the following directories:

1. the current directory
2. the directory specified by the **UIDPATH** environment variable

The environment variable **UIDPATH** specifies the path used by Motif to locate the resource (`.uid`) file. If not set, it is set to a default value. The resource file is installed in the same directory as the associated executable. So, you should set **UIDPATH** as follows (Bourne shell syntax):

```
UIDPATH=path_to_uid/%U  
export UIDPATH
```

Replace `path_to_uid` by the path to the directory in which the resource file is installed. The `%U` is required.

For more details refer to `MrmOpenHierarchy` in the *OSF/Motif Programmer's Reference* manual.

2.7 USING X RESOURCES

X toolkit resources specify GUI object (widget) attributes. Resources are specified in either the `.xdefaults` file or in application class-specific files.

The `.xdefaults` file is (typically) loaded into the X server at the start of the session. Any changes take effect only in a new session, or after using **xrdb**. Alternatively, application class resource files may be used. Application resource files have the same name as the executable CrossView Pro version they refer to (first letter NOT capitalized). Application resource files must be present either in the directory specified by the **HOME** environment variable, or in the `app-defaults` directory. The `app-defaults` directory is typically located under `/usr/lib/X11`.

X recognizes various environment variables for specifying paths to the application resource files. For more information, consult the chapter on X resources in *O'Reilly's X Toolkit Intrinsic Programming Manual* and your system documentation.

The X resource specification allows either global (loosely) bound specifications (`*foreground: black`) or per-widget instance specifications (`*button.foreground: black`).

The following list shows the relevant widgets used by the Motif version of CrossView Pro:

Windows:

TOP-LEVEL	– XmMainWindow	=> XmDrawingArea
CHILD	– XmScrolledWindow	=> XmDrawingArea

Dialog:

MODAL	– XmBulletinBoard
MODELESS	– XmBulletinBoard

Menu:

MENUBAR	– XmMenuShell
PULLDOWN	– XmCascadeButton

Controls:

CHECKBOX	- XmToggleButton
RADIOBUTTON	- XmToggleButton
TEXT	- XmLabel
EDIT	- XmText
LISTBOX	- XmScrolledWindow => XmList
SCROLLBAR	- XmScrollBar
PUSHBUTTON	- XmPushButton
LISTBUTTON	- XmText & XmArrowButton & XmScrolledWindow => XmList
LISTEDIT	- XmText & XmArrowButton & XmScrolledWindow => XmList
GROUPBOX	- XmFrame => XmLabel
ICON	- XmLabel with pixmap
FILESELECTION	- XmFileSelectionBox
ERRORPOPUP	- XmMessageBox

CrossView Pro repaints its windows in the default color as specified with the Motif widget resource settings. It is possible to overrule this behavior with a resource setting like: `*XmDrawingArea.background: blue`.

CrossView Pro uses a non proportional font in all of its windows. The font size is selected using the "Desktop Setup dialog". You can use the "font" resource (`*fontList` on Motif) to select the font to be displayed in the menubar and dialogs, it won't affect the font displayed in the CrossView Pro windows.

The CrossView Pro stack and data windows are implemented using a `XmScrolledWindow` widget on Motif.

The following list show the contents of an example `app-defaults` file intended for Motif environments. Of course you may adjust the colors and font to your preferences. Sample `app-defaults` files are delivered with the product in the `etc` directory (`app_def.mwm` for Motif).

```
*fontList:                7x13bold
*foreground:              black
*XmMainWindow.background: white
*XmScrolledWindow*background: white
*XmDrawingArea.background: white
*XmBulletinBoard.background: DarkSeaGreen
*XmToggleButton*background: gray
*XmLabel*background:      gray
```

```
*XmText*background:           white
*XmScrollBar*background:      gray
*XmPushButton*background:     gray
*XmFrame*background:          SeaGreen
*XmArrowButton*background:    gray
*XmForm.background:           SeaGreen
*XmMenuShell*background:      DarkSeaGreen
*XmCascadeButton*background:   SeaGreen
```

If you encounter any problems due to incorrect resource settings, like invisible text caused by identical text and background color, clear the `RESOURCE_MANAGER`. Use the following procedure to clear the `RESOURCE_MANAGER`:

1. Save a copy of the `.Xdefaults` file located in your home directory.
2. Install an empty `.Xdefaults` file.
3. Execute **xrdb -all .Xdefaults** to actually clear the `RESOURCE_MANAGER` property.
4. Restart CrossView Pro and check if windows and dialogs are displayed correctly.
5. Now you add the saved resources (one by one) back into the `.Xdefaults` file and execute **xrdb** to install them in the server. Restart CrossView Pro and check the influence of the new resource settings. Adapt your saved resources when necessary.

2.8 LICENSING TASKING PRODUCTS

TASKING products are protected with license management software (FLEXlm). To use a TASKING product, you must install the licensing information provided by TASKING for the type of license purchased.

You can run TASKING products with a node-locked license or with a floating license. When you order a TASKING product determine which type of license you need (UNIX products only have a floating license).

Node-locked license (PC only)

This license type locks the software to one specific PC so you can use the product on that particular PC only.

Floating license

This license type manages the use of TASKING product licenses among users at one site. This license type does not lock the software to one specific PC or workstation but it requires a network. The software can then be used on any computer in the network. The license specifies the number of users who can use the software simultaneously. A system allocating floating licenses is called a **license server**. A license manager running on the license server keeps track of the number of users.



See Appendix A, *Flexible License Manager (FLEXlm)*, for detailed information on FLEXlm.

2.8.1 OBTAINING LICENSE INFORMATION

Before you can install a software license you must have a "License Information Form" containing the license information for your software product. If you have not received such a form follow the steps below to obtain one. Otherwise, you can install the license.

Node-locked license (PC only)

1. If you need a node-locked license, you must determine the hostid of the computer where you will be using the product. See section 2.8.7, *How to Determine the Hostid*.

2. When you order a TASKING product, provide the hostid to your local TASKING sales representative. The License Information Form which contains your license key information will be sent to you with the software product.

Floating license

1. If you need a floating license, you must determine the hostid and hostname of the computer where you want to use the license manager. Also decide how many users will be using the product. See section 2.8.7, *How to Determine the Hostid* and section 2.8.8, *How to Determine the Hostname*.
2. When you order a TASKING product, provide the hostid, hostname and number of users to your local TASKING sales representative. The License Information Form which contains your license key information will be sent to you with the software product.

2.8.2 INSTALLING NODE-LOCKED LICENSES

Keep your "License Information Form" ready. If you do not have such a form read section 2.8.1, *Obtaining License Information*, before continuing.

Step 1

Install the TASKING software product following the installation procedure described in section 2.3, *Installation for Windows*.

Step 2

Create a file called "license.dat" in the c:\flexlm directory, using an ASCII editor and insert the license information contained in the "License Information Form" in this file. This file is called the "license file". If the directory c:\flexlm does not exist, create the directory.



If you wish to install the license file in a different directory, see section 2.8.6, *Modifying the License File Location*.



If you already have a license file, add the license information to the existing license file. If the license file already contains any SERVER lines, you must use another license file. See section 2.8.6, *Modifying the License File Location*, for additional information.

The software product and license file are now properly installed.



See Appendix A, *Flexible License Manager (FLEXlm)*, for more information on FLEXlm.

2.8.3 INSTALLING FLOATING LICENSES

Keep your "License Information Form" ready. If you do not have such a form read section 2.8.1, *Obtaining License Information*, before continuing.

Step 1

Install the TASKING software product following the installation procedure described earlier in this chapter on the computer or workstation where you will use the software product.

As a result of this installation two additional files for FLEXlm will be present in the `flexlm` subdirectory of the toolchain:

Tasking	The Tasking daemon (vendor daemon).
<code>license.dat</code>	A template license file.

Step 2

If you already have installed FLEXlm v6.1 or higher for Windows or v2.4 or higher for UNIX (for example as part of another product) you can skip this step and continue with step 3. Otherwise, install SW000098, the Flexible License Manager (FLEXlm), on the license server where you want to use the license manager.

The installation of the license manager on Windows also sets up the license daemon to run automatically whenever a license server reboots. On UNIX you have to perform the steps as described in section 2.8.5, *Setting Up the License Deaemon to Run Automatically*.



It is not recommended to run a license manager on a Windows 95 or Windows 98 machine. Use Windows NT instead (or UNIX).

Step 3

If FLEXlm has already been installed as part of a non-TASKING product you have to make sure that the `bin` directory of the FLEXlm product contains a copy of the **Tasking** daemon (see step 1).

Step 4

Insert the license information contained in the "License Information Form" in the license file, which is being used by the license server. This file is usually called `license.dat`. The default location of the license file is in directory `c:\flexlm` for Windows and in `/usr/local/flexlm/licenses` for UNIX.



If you wish to install the license file in a different directory, see section 2.8.6, *Modifying the License File Location*.

If the license file does not exist, you have to create it using an ASCII editor. You can use the license file `license.dat` from the toolchain's `flexlm` subdirectory as a template.



If you already have a license file, add the license information to the existing license file. If the `SERVER` lines in the license file are the same as the `SERVER` lines in the License Information Form, you do not need to add this same information again. If the `SERVER` lines are not the same, you must use another license file. See section 2.8.6, *Modifying the License File Location*, for additional information.

Step 5

On each PC or workstation where you will use the TASKING software product the location of the license file must be known. If it differs from the default location (`c:\flexlm\license.dat` for Windows, `/usr/local/flexlm/licenses/license.dat` for UNIX), then you must set the environment variable **LM_LICENSE_FILE**. See section 2.8.6, *Modifying the License File Location*, for more information.

Step 6

Now all license information is entered, the license manager must be started (see section section 2.8.4). Or, if it is already running you must notify the license manager that the license file has changed by entering the command (located in the `flexlm bin` directory):

lmreread

On Windows you can also use the graphical FLEXlm Tools (**lmttools**): Start **lmttools** (if you have used the defaults this can be done by selecting Start | Programs | TASKING FLEXlm | FLEXlm Tools), fill in the current license file location if this field is empty, click on the Reread button and then on OK. Another option is to reboot your PC.

The software product and license file are now properly installed.

Where to go from here?

The license manager (daemon) must always be up and running. Read section 2.8.4 on how to start the daemon and read section 2.8.5 for information how to set up the license daemon to run automatically.

If the license manager is running, you can now start using the TASKING product.



See Appendix A, *Flexible License Manager (FLEXlm)*, for detailed information on FLEXlm.

2.8.4 STARTING THE LICENSE DAEMON

The license manager (daemon) must always be up and running. To start the daemon complete the following steps on each license server:

Windows

1. Start the license manager tool by (Start | Programs | TASKING FLEXlm | FLEXlm License Manager).
2. In the Control tab, click on the Start button.
3. Close the program by clicking on the OK button.

UNIX

1. Log in as the operating system administrator (usually root).
2. Change to the FLEXlm installation directory (default /usr/local/flexlm):

```
cd /usr/local/flexlm
```

3. For C shell users, start the license daemon by typing the following:

```
bin/lmgrd -2 -p -c licenses/license.dat >>& \  
/var/tmp/license.log &
```

Or, for Bourne shell users, start the license daemon by typing the following:

```
bin/lmgrd -2 -p -c licenses/license.dat >> \
/var/tmp/license.log 2>&1 &
```

In these two commands, the **-2** and **-p** options restrict the use of the **lmdown** and **lmremove** license administration tools to the license administrator. You omit these options if you want. Refer to the usage of **lmgrd** in Appendix A, *Flexible License Manager (FLEXlm)*, for more information.

2.8.5 SETTING UP THE LICENSE DAEMON TO RUN AUTOMATICALLY

To set up the license daemon so that it runs automatically whenever a license server reboots, follow the instructions below that are appropriate for your platform. steps on each license server:

Windows

1. Start the license manager tool by (Start | Programs | TASKING FLEXlm | FLEXlm License Manager).
2. In the Setup tab, enable the Start Server at Power-Up check box.
3. Close the program by clicking on the OK button. If a question appears, answer Yes to save your settings.

UNIX



In performing any of the procedures below, keep in mind the following:

- Before you edit any system file, make a backup copy.

HP-UX

1. Log in as the operating system administrator (usually root).
2. In the directory `/etc/rc.config.d` create a file named `rc.lmgrd` with the following contents. Replace *FLEXLMDIR* by the FLEXlm installation directory (default `/usr/local/flexlm`):

```
#!/sbin/sh
FLEXLMDIR/bin/lmgrd -2 -p -c FLEXLMDIR/licenses/license.dat >> \
/var/tmp/license.log 2>&1 &
```

After the **-c** option, you have to specify the correct location of the license file.

SunOS4

1. Log in as the operating system administrator (usually root).
2. Append the following lines to the file `/etc/rc.local`. Replace *FLEXLMDIR* by the FLEXlm installation directory (default `/usr/local/flexlm`):

```
FLEXLMDIR/bin/lmgrd -2 -p -c FLEXLMDIR/licenses/license.dat >> \
/var/tmp/license.log 2>&1 &
```

SunOS5 (Solaris 2)

1. Log in as the operating system administrator (usually root).
2. In the directory `/etc/init.d` create a file named `rc.lmgrd` with the following contents. Replace *FLEXLMDIR* by the FLEXlm installation directory (default `/usr/local/flexlm`):

```
#!/bin/sh
FLEXLMDIR/bin/lmgrd -2 -p -c FLEXLMDIR/licenses/license.dat >> \
/var/tmp/license.log 2>&1 &
```

3. Make it executable:

```
chmod u+x rc.lmgrd
```

4. Create an 'S' link in the `/etc/rc3.d` directory to this file and create 'K' links in the other `/etc/rc?.d` directories:

```
ln /etc/init.d/rc.lmgrd /etc/rc3.d/Snumrc.lmgrd
ln /etc/init.d/rc.lmgrd /etc/rc?.d/Knumrc.lmgrd
```

num must be an appropriate sequence number. Refer to your operating system documentation for more information.

2.8.6 MODIFYING THE LICENSE FILE LOCATION

The default location for the license file on Windows is:

```
c:\flexlm\license.dat
```

On UNIX this is:

```
/usr/local/flexlm/licenses/license.dat
```

If you want to use another name or directory for the license file, each user must define the environment variable **LM_LICENSE_FILE**. Do this in `autoexec.bat` (Windows 95/98), from the Control Panel -> System | Environment (Windows NT) or in a UNIX login script.

If you have more than one product using the FLEXlm license manager you can specify multiple license files to the **LM_LICENSE_FILE** environment variable by separating each pathname (*lppath*) with a ';' (on UNIX also ':'):

Example Windows:

```
set LM_LICENSE_FILE=c:\flexlm\license.dat;c:\license.txt
```

Example UNIX:

```
setenv LM_LICENSE_FILE  
/usr/local/flexlm/licenses/license.dat:/myprod/license.txt
```

If the license file is not available on these hosts, you must set **LM_LICENSE_FILE** to *port@host*; where *host* is the host name of the system which runs the FLEXlm license manager and *port* is the TCP/IP port number on which the license manager listens.

To obtain the port number, look in the license file at *host* for a line starting with "SERVER". The fourth field on this line specifies the TCP/IP port number on which the license server listens. For example:

```
setenv LM_LICENSE_FILE 7594@elliott
```



See Appendix A, *Flexible License Manager (FLEXlm)*, for detailed information.

2.8.7 HOW TO DETERMINE THE HOSTID

The hostid depends on the platform of the machine. Please use one of the methods listed below to determine the hostid.

Platform	Tool to retrieve hostid	Example hostid
HP-UX	lanscan (use the station address without the leading '0x')	0000F0050185
SunOS/Solaris	hostid	170a3472
Windows	tkhostid (or use lmhostid)	0800200055327

Table 2-1: Determine the hostid



If you do not have the program **tkhostid** you can download it from our Web site at: <http://www.tasking.com/support/flexlm/tkhostid.zip> . It is also on every product CD that includes FLEXlm.

2.8.8 HOW TO DETERMINE THE HOSTNAME

To retrieve the hostname of a machine, use one of the following methods.

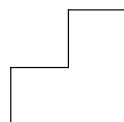
Platform	Method
HP-UX	hostname
SunOS/Solaris	hostname
Windows 95/98	Go to the Control Panel, open "Network", click on "Identification". Look for "Computer name".
Windows NT	Go to the Control Panel, open "Network". In the "Identification" tab look for "Computer Name".

Table 2-2: Determine the hostname

CHAPTER

3

COMMAND LANGUAGE



3

CHAPTER

3.1 INTRODUCTION

The syntax and semantics of CrossView Pro's command language is discussed here. This language is mainly used to enter textual commands in the command edit field of the Command Window. The mouse and menus allow you to access most actions without knowing the command language, although the command language is more powerful. The command language is also used when evaluating expressions and in commands associated with assertions, breakpoints and macros. For information about specific CrossView Pro commands, refer to Chapter 12, *Command Reference*.

3.2 CROSSVIEW PRO EXPRESSIONS

There are several methods that you can use to input an expression into CrossView Pro:

It is possible to display both monitored and unmonitored expressions in the Data Window. Monitored expressions are updated after every halt in execution. Unmonitored expressions are just one-shot inspections of the expressions value. Refer to section 4.6, *CrossView Pro Windows* for a detailed description of the Data Window.

To evaluate a simple expression:



Double click on a variable in the Source window. The result of the expression appears in the data window. Alternatively, depending on the preferences you set in the Data Display Setup dialog, the expression appears in the Evaluate Expression dialog. Select the Watch or Show button to display the result of the expression in the Data Window. Select the Evaluate button to display the result of the expression in the output field of the Evaluate Expression dialog.

To evaluate a complex expression:



Select Data | Evaluate Expression... from the menu and type in any C expression in the Evaluate Expression dialog box. Optionally select a format code. Click the Evaluate button.



Type the expression into the command edit field of the Command Window followed by a return or press the Execute button.

Expressions can be any length in most windows and dialog boxes; CrossView Pro provides a horizontal scroll bar if an expression exceeds the visible length of the entry field.

In CrossView Pro, C expressions may consist of a combination of numeric constants, character constants, strings, variables, register names, C operators, function names, function calls, typecasts and some CrossView Pro-specific symbols. Each of these is described in the next sections.

Evaluation Precision

CrossView Pro evaluates expressions using the same data types and associated precision as used by the target architecture when evaluating the same expression.

3.3 CONSTANTS

CrossView Pro, like C, supports integer, floating point and character constants.

Integers

Integers are numbers without decimal points. For example, CrossView Pro will treat the following as integers:

5 9 23

The following number, however, are not treated as integers:

5.1 9.27 0.23

Negative integers, if they appear as the first item on a line, must have parentheses around the number:

(-5) * 4

This is to prevent confusion with CrossView Pro's own - (minus sign) command.

In addition, CrossView Pro supports standard C octal, hexadecimal and binary notation. You can specify a hexadecimal constant using a leading **0x** or a trailing **H** (or **h**). The first character must be a decimal digit, so it may be necessary to prefix a hexadecimal number with the '0' character. The hexadecimal representation for decimal 16 is:

0x10 or 10H

For the hexadecimal digits **a** through **f** you can use either upper or lower case. The following are all correct hexadecimal representations for decimal 43981:

0xabcd 0xABCD 0abCdH 0AbcDh

You can specify a binary constant using a trailing **B** or **Y** (or **b** or **y**). The following are all binary representations for decimal 5:

0101b 101Y 00000101B

You can specify an octal constant using a leading '0'. The octal representation for 8 decimal is:

010

You can use an **L** to indicate a **long integer constant**. For example, CrossView Pro will recognize the following as long integers:

0L 57L 0xffL



CrossView Pro uses the same ANSI C integral type promotion scheme as the C compiler.

Floating Point

A floating point number requires a decimal point and at least one digit before the decimal point. The following are valid examples of floating point numbers:

12.34 5.6 7.89

Exponential notation, such as 1.234e01, is not allowed. The following are *not* valid floating point numbers:

.02 1.234e01 5

As with integers, bracket a negative number with parentheses:

(-54.321)

Expressions combining integers and floating point numbers will evaluate to floating point values:

2.2 * 2
4.4



Character

Character constants are single characters or special constants that follow the C syntax for special characters. Examples of valid character constants include:

```
'm'  'x'  '\n'
```

Character constants must be a single byte and are delimited by ' ' (single quotation marks). For instance:

```
$mychar='m'
```

Remember not to confuse character constants with *strings*. A character constant is a single byte, in this example, the ASCII value of m.

Strings

Strings are delimited by " " (double quotation marks). In C all strings end with a null (zero) character. Strings are referenced by pointer, not by value. This is standard C practice. In CrossView Pro, you may assign a string literal to a variable which is of type char* (pointer to character):

```
$ystring = "name"
```

CrossView Pro supports the standard C character constants shown below:

Code	ASCII	Hex	Function
\b	BS	08	Backspace
\f	FF	0C	Formfeed
\n	NL (LF)	0A	Newline
\r	CR	0D	Carriage return
\t	HT	09	Horizontal tab
\\	\	5C	Back slash
\?	?	3F	Question mark
\'	'	27	Single quote
\"	"	22	Double quote
\ooo			3-digit octal number
\xhhh			hexadecimal number

Table 3-1: C character codes

Trigraph sequences are not supported.

3.4 VARIABLES

CrossView Pro lets you use variables in the C expressions you type. You may reference two classes of variables: variables defined in the source code and *special variables*.

Variables defined in your source code fall into two categories: *local variables* and *global variables*.

Storage Classes

Variables may be of any C storage class. The size of each class is target dependent. Consult your M16C C Cross-Compiler User's Guide for specific sizes.

You may cast variables from one class to another:

```
(long) $mychar
```

Local Variables

You define local variables within a function; their values are maintained on the stack or in registers. When the program exits the function, you lose local variable values. This means that you can only reference local variables when their function is active on the stack.

Local variables of type `static` retain values between calls. Therefore, you can reference `static` variables beyond their functions, but only if their function is active on the stack.

CrossView Pro knows whether the compiler has allocated a local variable on the stack or directly in a register and whether the register is currently on the stack. The compiler may move some local variables into registers when optimizing code.

If a part of your source code looks like this:

```
x = 5;  
y = x;
```

and you stopped the program after the assignment to `x`, and set `x` to another value, this may not prevent the second statement from setting `y` to 5 due to "constant folding" optimizations performed by the compiler.

Global Variables

Global variables are defined outside every function and are not local to any function. Global (non-static) variables are accessible at any point during program execution, after the system startup code has been executed.

Global variables can be defined `static` in a module. These variables can only be accessed when a function in this module is active on the stack, or when that file is in the Source Window using the `e` command.

Specifying Variables in C expressions

The following table specifies how CrossView Pro treats different variables in C expressions. The left column is the variable's syntax in the expression, the right column is the CrossView Pro semantics.

Variable Syntax	CrossView Pro Behavior
<i>variable</i>	CrossView Pro performs a scope search starting at the current viewing position and proceeding outwards. The debugger first checks locals, local statics and parameters, followed by statics and globals explicitly declared in the current file. Finally, globals in other files are checked.
<i>function#variable</i>	CrossView Pro searches for the first instance of <i>function</i> . If found, the debugger uses the frame's address to perform a scope search for <i>variable</i> . Variables are available only if the specified function is active. That is, the stack frame for that function can be found on the run-time stack.
<i>number#variable</i>	The frame at stack level <i>number</i> is used by the debugger for the scope search. The current function is always at stack level 0. This format is very useful if you are debugging a recursive function and there are multiple instances of a variable on the stack.
<i>:variable</i>	CrossView Pro searches for a global variable named either <i>variable</i> or <i>_variable</i> , in that order.
<i>\$variable</i>	CrossView Pro searches the list of special variables for <i>\$variable</i> .

Table 3-2: Variables in C expressions

Variables and Scoping Rules

A variable is in scope at any point in the program if it is visible to the C source code. For instance, if you have a local variable `initval` declared in `main()`, and then step (or move the viewing position) into `factorial`, `initval` will be out of scope. You can still find the value of `initval` by typing:

```
main#initval
```

In this case CrossView Pro will search the stack for the function `main()`, then look outwards from that function for the first occurrence of `initval` in scope and report its value. Note that `main()` must be active, that is, program execution must have passed through `main()` and not yet returned, in order for `initval` to have a value.

You can also use the `Browse...` button in the Expression Evaluation dialog box. This dialog box is called by either the `Show/Watch new expression` button in the Toolbar or the `Data | Evaluate Expression...` item in the menu. The Variable Lists dialog box lists both global and local variables for you.

Special Variables

CrossView Pro maintains a set of variables that are separate from those defined in your program being debugged. These special variables reside in memory on the host computer, not on the target system. They contain the values of the target processor's registers, information about the debugger's status, and user-defined values. Special variables are case insensitive. Use the **opt** command to display and set these variables (without using the '\$'-sign).

The following is a list of the reserved special variables for CrossView Pro:

Reserved Variable	Description
<code>\$ARG(<i>n</i>)</code>	Contains the value of the <i>n</i> th int-sized argument of the current function. Allows access to arguments of variable argument list functions without knowing the name of the argument.
<code>\$FILE</code>	Contains the name of the file that holds the current viewing position.
<code>\$IN(<i>function</i>)</code>	Contains the value 1 if the current pc is inside the specified <i>function</i> , otherwise 0.

Reserved Variable	Description
\$LINE	Contains the line number of the current viewing position. This variable is often used in assertions to monitor program flow.
\$PROCEDURE	Contains the name of the procedure at the current viewing position.
\$ASMHEX	Contains a string "ON" or "OFF". The value "ON" specifies that the disassembled code as displayed in the assembly window will display hexadecimal opcodes. Default is "OFF".
\$AUTOSRC	Contains a string "ON" or "OFF". The value "ON" specifies that the debugger will automatically switch between the source window and the assembly window display depending on the presence of symbolic debug information at the current location. The value "OFF" prevents the automatic window switching. Default is "OFF".
\$CPU	Contains a string indicating the current CPU type.
\$DSC	Contains a string indicating the current locator description file. Default is m16c (etc/m16c.dsc). See the -dsc option.
\$FP	Contains the value of the frame pointer.
\$MIXEDASM	Contains a string "ON" or "OFF". The value "ON" specifies that the disassembled code as displayed in the assembly window will be intermixed with the corresponding source lines. The value "OFF" suppresses this intermixing. Default is "ON".
\$MORE	Contains a string "ON" or "OFF". The value "ON" specifies that the more output pager is enabled. The value "OFF" disables the more output pager. Default is "ON".
\$PC	Contains the value of the program counter.
\$PIPELINE	Contains a string "ON" or "OFF". The value "ON" specifies that the pipeline should be displayed in the assembly window. Default is "OFF".
\$register	Contains the value of the specified <i>register</i> .
\$SP	Contains the value of the stack pointer.
\$SYMBOLS	Contains a string "ON" or "OFF" indicating if local symbols and symbolic addresses (e.g. main:56+0x4) or absolute addresses are present in disassembly. Default is "ON".

Reserved Variable	Description
\$SRCLINENRS	Contains a string "ON" or "OFF". The value "ON" specifies that line numbers should be printed in the source window. The value "OFF" suppresses printing of line numbers. Default is "OFF".
\$SRCMERGELIMIT	Contains the value for the source merge limit in the assembly window, the number of source lines to be intermixed in the assembly window. Value 0 indicates that there is no limit. Default is 0.

Table 3-3: Reserved special variables

Registers

For CrossView Pro, a fixed set of registers is always available. You can add additional M16C derivative specific SFRs in a `.sfr` file. See the *C Cross-Compiler User's Guide* for more information.

You can configure which (and in which order) registers must appear in the register window, using the `Debug | Register Window Setup` menu item.

It is possible to request the address of an SFR by using the address operator `&`.

```
&$sp
Location of $SP is reg [SP]
Operand for '&' incorrect
```

```
&$psw
0x578218
```

In addition to the standard register special variables, CrossView Pro supplies the special variables: `$sp` (the stack pointer), `$pc` (the program counter) and `$fb` (the current frame pointer) for all targets.

The values of Reserved special variables cannot be changed interactively (i.e., on the CrossView Pro command line).

User-defined Special Variables

During a debugging session, you may need some new variables for your own debugging purposes, such as counting the number of times you encounter a breakpoint. CrossView Pro allows you to create and use your own special variables for this purpose. CrossView Pro does not allocate space for these variables in target memory; it maintains them on the host computer.

The names of these variables, which must begin with a **\$** (dollar sign), are defined when they are first used. For instance:

```
$count = 5
```

defines a variable named `$count` of type `int` with a value of 5. Special variables are of the same type as the last expression they were assigned. For example:

```
$name="john"
```

then:

```
$name=3*4
```

creates a special variable `$name` of type `(char *)`. The second statement creates a special symbol `$name` and assigns it the value of 12 of type `int`.

Special variables are just like any other variables, except you can not meaningfully take the address of them. CrossView Pro allows as a default 26 user-defined special variables. You may change this limit with the **-s** option at startup, or by selecting the **Options | Initialization...** menu item.



See the startup options in Chapter 4, *Using CrossView Pro*.

3.5 FORMATTING EXPRESSIONS

By default, CrossView Pro displays the value of an expression using the appropriate format for the type of expression. CrossView Pro follows several simple rules for displaying variables:

- The defaults are: addresses appear in hexadecimal format, characters as ASCII and integers as decimal.
- There are four possible formats to show one integer value: **decimal**, **hexadecimal**, **octal**, and **ASCII**.
- There are two different formats to display one floating point value: **decimal real** and **hexadecimal**. If the absolute value is either too big or too small (with too many non-significant zeroes), the debugger automatically converts the format to one with fixed decimal point and exponent.
- **ASCII** is the only format to display a string. Note that you can opt for the array format. Unpredictable characters are output as `\xbh`, where `bh` is a hexadecimal value. Control characters are output as `^C`.
- All the values in an array appear in the same format. You are free to select this format from the available options.
- If All the values of a structure appear in the same format. You are free to select this format from the available options.



You can determine in which format a variable is displayed. Once the format has been selected, however, you must enter values or change values in the appropriate format. When editing is finished, the debugger interprets all values in terms of the currently selected formats.

You may, however, tell CrossView Pro to display an expression in a particular format other than the default format. The format code follows the variable, in one of two ways:



The simplest method of specifying display formats is from the Evaluate Expression dialog box. To access this dialog box:

- Select the Data | Evaluate Expression menu option.



In the Command Window, you can use several **format codes** shown in the next table to specify the variable display. The format codes can be entered as:

variable/format



to display the variable in format *format*, or:

```
variable@format
```

to display the variable’s address in format *format*.

The structure of the formatting code is:

```
[count] style [size]
```

Count is the number of times to apply the format style *style*. *Size* indicates the number of bytes to be formatted. Both *count* and *size* must be numbers, although you may use **c** (char), **s** (short), **i** (int), and **l** (long) as shorthand for *size*. Legal integer format sizes are 1, 2, and 4; legal float format sizes are 4 and 8.



Be sure not to confuse CrossView Pro format codes with C character codes, e.g. \a. CrossView Pro uses a forward slash / not a backward slash \.

Style	Description
a	Print the specified number of characters of the character array; any positive <i>size</i> is OK. Use the expression’s value as the address of the first byte.
c	Print a character; any positive <i>size</i> is OK; default <i>size</i> is sizeof(char).
D	Print in decimal; needs NO <i>size</i> specifier; <i>size</i> is sizeof(long).
d	Print in decimal; can have a <i>size</i> specifier; default <i>size</i> is sizeof(expression).
E	Print in “e” floating point notation; needs NO <i>size</i> specifier; default <i>size</i> is sizeof(double).
e	Print in “e” floating point notation; the <i>size</i> specifier can be sizeof(float) or sizeof(double); default <i>size</i> is sizeof(expression).
F	Print in “f” floating point notation; needs NO <i>size</i> specifier; default <i>size</i> is sizeof(double).
f	Print in “f” floating point notation; the <i>size</i> specifier can be sizeof(float) or sizeof(double); default <i>size</i> is sizeof(expression).
G	Print in “g” floating point notation; needs NO <i>size</i> specifier; default <i>size</i> is sizeof(double).

Style	Description
g	Print in “g” floating point notation; the <i>size</i> specifier can be sizeof(float) or sizeof(double); default <i>size</i> is sizeof(expression).
l	Print the function, source line, and disassembled instruction at the address.
i	Print the disassembled instruction at address.
n	Print in the “natural” format, based on type; use it for printing variables that have the same name as an CrossView Pro command.
O	Print in octal; needs NO <i>size</i> specifier; <i>size</i> is sizeof(long).
o	Print in octal; can have a <i>size</i> specifier; default <i>size</i> is sizeof(expression).
P	Print the name of the function at the address.
p	Print the names of the file, function, and source line at the address.
s	Print the specified number of characters of the string, using the expression’s value as the address of a pointer to the first byte. Equivalent to <i>*expression/a</i> . If no <i>size</i> is specified the entire string, pointed to by expression, is printed (till nil-character).
t	Display the type of the indicated variable or function.
U	Print in unsigned decimal; needs NO <i>size</i> specifier; <i>size</i> is sizeof(long).
u	Print in unsigned decimal; can have a <i>size</i> specifier; default <i>size</i> is sizeof(expression).
X	Print in hexadecimal; needs NO <i>size</i> specifier; <i>size</i> is sizeof(long).
x	Print in hexadecimal; can have a <i>size</i> specifier; default <i>size</i> is sizeof(expression).

Table 3-4: Format style codes

For example, typing:

```
initval/4xs
```

displays four, hexadecimal two-byte memory locations starting at the address of `initval`.

The following piece of C-code can be accessed in CrossView Pro using the string format codes:

```
char  text[] = "Sample\n";
char *ptext = text;
```

```
text           What is the address of this char array
text = 0x8200
```

```
text/a         Print it as a string
text = "Sample^J"
```

```
ptext          What is the contents of this pointer
string = 0x8200
```

```
ptext/s        Print it as a string
string = "Sample^J"
```

```
&ptext         Where does ptext itself reside
0x8210
```

With format codes, you may view the contents of memory addresses on the screen. For instance, to dump the contents of an absolute memory address range, you must think of the address being a pointer. To show (dump) the memory contents you use the C-language indirection operator `*`. Example:

```
*0x4000/2x4
0x4000 = 0x00DB0208 0x5A055498
```

This command displays in hexadecimal two long words at memory location 0x4000 and beyond. Instead of using the size specifier in the display format, you can force the address to be a pointer to unsigned long by casting the value:

```
*(unsigned long *)0x4000/2x
0x4000 = 0x00DB0208 0x5A055498
```

To view the first four elements of the array `table` from the `demo.c` program, type:

```
table/4d2
table = 1      1      2      6
```

This command displays in decimal the first four 2-byte values beginning at the address of the array `table`.

3.6 OPERATORS

Standard C Operators

CrossView Pro supports the standard C operators in the ANSI defined order of precedence. The order of precedence determines which operators execute first.

The semicolon character (;) separates commands on the same line. In this way, you may type multiple commands on a single line. Comments delimited by /* and */ are allowed; CrossView Pro simply ignores them.

Order of Precedence (in descending order)
() [] -> .
! ~ ++ -- + - * & (type) sizeof
* // %
+ -
<< >>
< <= > >=
== !=
&
^
&&
? : = += -= *= /= %= &= ^= = <<= >>=

Table 3-5: Order of precedence of standard C operators

The *, - and + operators appear twice since they exist as both unary and binary operators and unary operators have higher precedence than binary.



Division is represented by // (two slashes) not / (one slash). This is to avoid confusion with CrossView Pro's format specifier syntax.

Using Addresses

To specify an address, you may use the & operator. To determine the address of `initval`, type:

```
&initval
```

If you try to use the & operator on a local variable in a register, CrossView Pro issues an error message and tells you which register holds the variable.

3.7 SPECIAL EXPRESSIONS

String Commands

Whenever CrossView Pro encounters an expression consisting solely of a string by itself, it simply echoes the string. For example:

```
"hello, world\n"
hello, world
```

Use this technique to place helpful debugging messages on breakpoints. For example, setting the following breakpoint:

```
60 b {"now in for loop\n"; sum; C }
```

this cause CrossView Pro to echo the message `now in for loop`, to display the value of `sum` in the Command Window, and to continue when line 60 is encountered. You can also enter this breakpoint and the associated commands via the Breakpoints dialog box, which you can open by selecting the `Breakpoints` menu item from the `Debug` menu.

The Period Operand

As a shorthand, CrossView Pro supports a special operand, period '.', that stands for the value of the last expression CrossView Pro calculated. For instance, in the following example, the period in the second command equals the value 11, which is the result of the previous expression:

```
5 + 6
11
4 * .
44
```

The period operand assumes the same size and format implied by the specifier used to view the previous item. Thus if you look at a long as a char, a subsequent '.' is considered to be one byte. Use this technique to alter specified pieces of a larger data item, such as the second highest byte of a long, without altering the rest of the long. The period operand may be used in any context valid for other variables.



'.' is the *name* of a location. When you use it, it is dereferenced like any other name. If you want the address of something that is 30 bytes farther on in memory, do not type `.+30` as this takes the contents of dot and adds 30 to it. Type instead `&.+30` which adds 30 to the address of the period operand.

3.8 CONDITIONAL EVALUATION

CrossView Pro supports the `if` construct. Use this construct in breakpoints and assertions to alter program flow conditionally. For example, if you reset the following breakpoint:

```
60 b {if (sum<=5931){C}{sum}}
```

CrossView Pro compares the value of `sum` with 5931 when the program stops at line 60. If `sum` is less than or equal to 5931, CrossView Pro continues. Otherwise, CrossView Pro displays the value of `sum` with 5931 when the program stops at line 60.

You can also use the `exp1 ? exp2 : exp3` C ternary operator for conditional expressions. For example:

```
$myvar = (5 > 2) ? 1 : -1
```

assigns the value 1 to `myvar`.

3.9 FUNCTIONS

In CrossView Pro expressions, you can include functions defined in the program's code.



Command line function calls are not supported for the M16C.



You can call functions through the Call a Function dialog box. Note that shows only the results of the function call. You cannot enter expressions in this field. If you want to use the results of the function call in an expression, then type the expression into the Evaluate Expression dialog box or type in the command into the Command Window (described in the keyboard method below).

- Select `Run | Call a Function...` item from the menu to open this box.
- List all functions by clicking the `Browse...` button.
- You can place parameters in the `Parameters` field of the Call a Function dialog box, separated by commas, but without the usual parentheses or select from the drop-down history list.

The Command Window receives the results of the function call.



Type in the expression containing a function call directly into the Command Window.

To execute a function on the target type the function name and the arguments as you would do in your C program. For example,

```
do_sub(2, 1)
```

or

```
a = do_add(3,4)
```

3.10 CASE SENSITIVITY

The absolute file supplies the case sensitivity information for variable names. It is initially case *sensitive* for the C language. You may toggle case sensitivity by:



Selecting the Search | Search String... menu item to view the Search String dialog box. This dialog contains the Case Sensitive check box.



Typing the (capital) **Z** command in the Command Window.

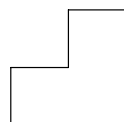
CHAPTER

4

USING CROSSVIEW PRO



TASKING



4

CHAPTER

4.1 INTRODUCTION

This chapter and the following 8 chapters give you a comprehensive picture of CrossView Pro's features. In order to address the broadest range of expertise, the contents range from introductory examples to the more technical aspects and techniques of debugging with CrossView Pro. While it is not necessary for you to read the chapters straight through, you may find it especially helpful to do so. All of the examples are from the sample program `demo.c` which comes with CrossView Pro. For a complete description of the commands presented in this chapter, consult the *Command Reference* chapter.

Each CrossView Pro command introduced in the text has a matching box summarizing its syntax and semantics. The command description follows these general rules:

Items in **bold** font are the actual CrossView Pro commands: **save**, **set**. Items in *italics* are names for the things you should type: *filename*, *commands*. In addition, the | symbol means *or*. For instance, **screen** | *filename* means you can use the word "screen" or a filename in the syntax.

4.2 USING THE CROSSVIEW PRO INTERFACE

This manual uses the word "Windows" to generically refer to the host computer system's windowing system. On IBM-PCs and compatibles, this is equivalent to Microsoft Windows (95/98/NT or 2000). On UNIX workstations, this refers to the X Window System. Generally, this manual makes no distinctions between the various windowing systems unless needed to clarify the discussion.

This manual assumes you possess a basic familiarity with Windows software. For this reason, discussion focuses on how CrossView Pro works, rather than how to use the Window interface. For more information on your Windows system, consult the Windows documentation provided with your host system.

You can execute most CrossView Pro commands using either mouse or textual commands. Mouse commands are executed by means of buttons and pull-down menus in each of the separate CrossView Pro windows. Text commands are typed at the prompt in the Command Window. In most cases, there is no difference in functionality between mouse and text equivalents.

This manual discusses both methods of performing CrossView Pro functions. For a quick-reference guide to all CrossView Pro commands, refer to the *Command Reference* chapter.

4.3 INVOKING CROSSVIEW PRO

Once an absolute file has been made it can be executed by CrossView Pro. There are several ways to invoke CrossView Pro.

From EDE

To start CrossView Pro from EDE (the Embedded Development Environment), click on the `Debug application` button. The following button is the `Debug application` button which is located in the ribbon bar.



From the desktop

With Windows 95/98/NT/2000 you can start CrossView Pro through the Start menu. Or in the Windows Explorer you can double-click on an absolute file if the `.abs` extension is associated with the CrossView Pro executable.



On the PC, CrossView Pro is a Microsoft Windows application. As such, you must invoke it from the Windows environment.

From the command line

To begin the debugging session, type the name of the CrossView Pro debugger and optionally the name of the target program (absolute file).

4.4 STARTUP OPTIONS

CrossView Pro allows you to specify several options when you invoke the program. Type these startup options (or switches as they are sometimes called) after the optional basename of the application. The basename can also contain a path specification. In this case, CrossView Pro sets its current directory to the specified path. A minus sign proceeds each option; the options can appear in any order.

Note that some versions of CrossView Pro have different startup options and procedures than the ones described here. Please consult the Addendum (at the end of this manual), for precise information about starting up CrossView Pro with your target hardware.

You can set many of CrossView Pro's options by using the dialog boxes called by the `Options | Startup->CrossView...` and `Options | Initialization...` menu items from the menu. Some targets and environments have an additional menu option of `Options | Startup->Emulator...`. You can save the options in the `xvw.ini` file and they are automatically used upon startup.

In Microsoft Windows, add startup options to the program's property sheet:

Windows 95/98/2000 or Windows NT 4.0 (or higher):

- Right-click on the CrossView Pro shortcut icon, shown in your program installation folder.
- Select `Properties`. The Program Item Properties dialog box will open up.
- Enter the startup options after the executable's name in the Target field of the shortcut.



Use menus to set options. After setting the options in the menus and selecting the appropriate options in the `Save Options` dialog on exit, CrossView Pro saves the settings in the file `xvw.ini` for future debug sessions.

To start up CrossView Pro type:

xfwml6

When your execution environment itself has a human-oriented ASCII interface, you can use transparency mode with the **-T** option. In transparency mode you can configure the execution environment's memory. Check the Addendum, the hardware-specific section of this manual. In-circuit emulators generally require you to map the address space, allocating memory ranges to the execution environment and/or the target system. Fortunately, this generally does not mean you need to learn your emulator's command set, just a rote sequence of startup commands. When your CrossView Pro version does not support transparency mode, you do not need to configure the memory, and the **-T** option is not needed.

If your target system supports serial communication and if the target system is connected to a port other than the default port (see the *Overview* chapter to determine the default port for your host), you can use the **-D** option to specify the port name. The default baud rate is 9600. You may use the **-D** option to specify the baud rate if the execution environment is not the same as the default. For example:

```
xfwm16 -D rs232,com2,19200
```

instructs CrossView Pro to use the COM2 port at 19200 baud. See your execution environment in the Addendum of this manual for specific communication information.

When you specify a startup option in CrossView Pro, the option overrules the corresponding value in the current `xvw.ini` file.

There are many different options you can invoke when starting up CrossView Pro. The listing below gives an overview of all startup options.

There are several startup options having to do with the recording and playing back of CrossView Pro command files. See also chapter 9, *Command Recording & Playback*.

Startup Option	Description
-a <i>number</i>	Sets the maximum number of assertions (the default is 100).
-b <i>number</i>	Sets the maximum number of code breakpoints (the default is 200).
-c <i>number</i>	Sets the maximum number of instruction trace for the trace buffer (the default is 32).
-C <i>cpu</i>	Forces CPU type selection. This option also determines which register file (<i>regcpu.dat</i>) will be used. This option overrides the CPU type selection in both <i>xvw.ini</i> and a target configuration file.
-dsc <i>dsc</i>	Forces locator description file selection. The default is <i>m16c.dsc</i> .
-D <i>device_type,opt1[,opt2]</i>	Selects a device and specifies device specific options, such as communication port and baud rate. The allowed combinations for your execution environment are described in the manual addendum for that specific execution environment. The following combinations are possible:
-D rs232,port,speed	Select RS-232 communication. <i>port</i> For PC this is COM1, COM2, COM3 or COM4. A colon should not be added. For UNIX this is the full path of the RS-232 device driver (e.g., <i>/dev/tty01</i>). By default CrossView Pro uses the first RS-232 port. <i>speed</i> This is the baud rate used for the specified <i>port</i> . The default is 9600.
-D parallel,port	Select parallel communication. <i>port</i> For PC this is LPT1 or LPT2. Do not add a colon. For UNIX this is the full path of the parallel device driver. By default CrossView Pro uses the first parallel port.
-D tcp,host,port	Select TCP/IP communication. On UNIX the standard TCP/IP implementation is used. On MS-Windows the <i>WINSOCK.DLL</i> implementation is used. <i>host</i> The name of the host to be accessed via TCP/IP. <i>port</i> The port number on <i>host</i> to be accessed.

Startup Option	Description
-D dev,device-file	Use a UNIX device driver as communication channel. For RS-232 devices use the -D rs232 option, described above. <i>device-file</i> The full path of the UNIX device file.
-D isa,io-port,address	Select communication channel to an (E)ISA interface card in the PC. <i>io-port</i> PC I/O port number or I/O channel used for accessing the (E)ISA card. <i>address</i> The memory address used to access the (E)ISA card.
-em macro[=def]	Add macro for pre-processing the description file. If <i>def</i> is not given ('=' is omitted), '1' is assumed.
-f file	Read command line options from <i>file</i> .
-G path	Specify startup directory for CrossView Pro.
-i	Has CrossView Pro download the image of the absolute object file.
-L file	Keeps a log of CrossView-to-target communications in a file. Not available for all execution environments.
-n address	Informs CrossView Pro that the program was loaded into memory at an address other than zero.
-p file	Starts playing back commands from file.
-P file	Starts playing back commands from file with commands single step.
-r file	Starts recording commands in file.
-R file	Starts recording screen output in file.
-s number	Sets the maximum number of special variables (variables independent of the program that CrossView Pro provides for your use). The default is 26.

Startup Option	Description
-sd <i>directory</i>	Specifies the directories CrossView Pro should search for source files. Relative paths are allowed. When the x command is used to load a new symbol file, the current directory is set to the directory containing the symbol file and CrossView Pro now searches for source files relative to this directory. Directories must be separated by semicolons.
-tcfg <i>file</i>	Specify a target configuration file. This overrides the filename specified in <code>xvw.ini</code> . See section <i>CrossView Pro Startup Settings</i> in the <i>Overview</i> chapter.
-T [<i>file</i>]	Starts CrossView in transparency mode if present; if <i>file</i> is given, commands in <i>file</i> are sent to the execution environment.

Table 4-1: CrossView Pro Startup Options

4.4.1 WHAT YOU MAY HAVE DONE WRONG

Most problems in starting up CrossView Pro for a debugging session stem from improperly setting up the execution environment or from an improper connection between the host computer and the execution environment. Some execution environments require you to enter transparency mode to set the execution environment for a debugging session. Check the notes for your particular execution environment and the Addendum of this manual.

Here are some other common problems:

- Specifying the wrong device name when invoking the debugger.
- Specifying a baud rate different from the one the execution environment is configured to expect.
- Not supplying power to the execution environment or an attached probe.
- Using the wrong kind of communication cable.
- Plugging the cable into an incorrect port. Some target machines have several ports.
- Installation of a device driver or resident applications that use the same communications port on the host system.

- The port is already in use by another user or login process on some UNIX hosts.
- Specifying no or an invalid cpu type with the **-C** option.

4.5 THE CROSSVIEW PRO DESKTOP

The CrossView Pro desktop is the screen background in which all windows, icons and dialog boxes appear (see figure 4-1). Under some windowing systems, the desktop is itself a window that does not contain all other CrossView Pro windows.

The desktop always has the Command Window opened or iconized.

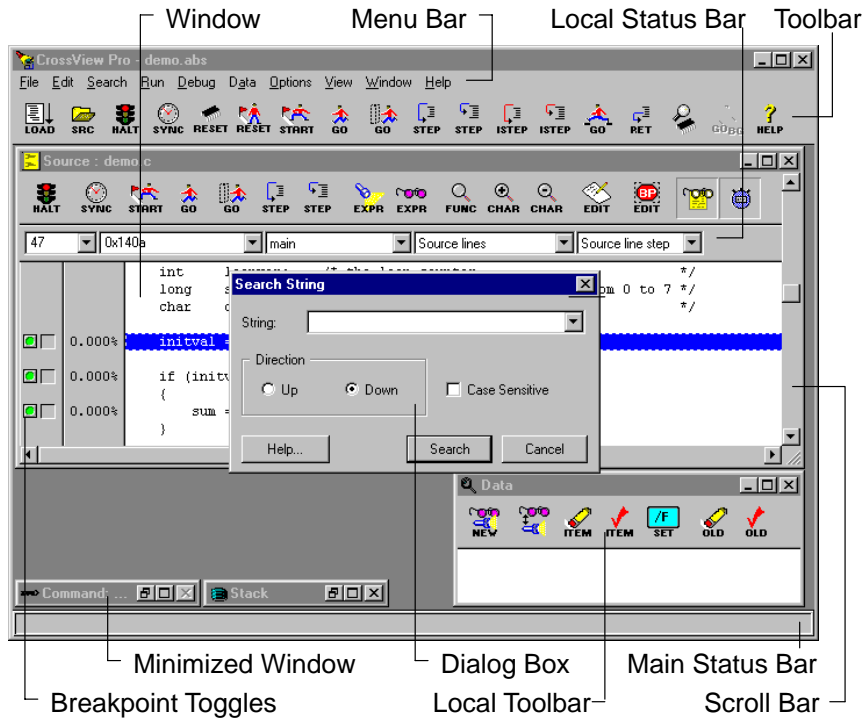


Figure 4-1: CrossView Pro Desktop

At the top of the desktop is the **Menu Bar**, which contains the menus applicable to the currently active window. Below the menu bar is the main **Toolbar**, from which you can execute commands to control program execution as button functions. Except for the Command Window, the desktop can contain other windows as well.

Along the bottom of the desktop there is a **Main Status Bar**. The status bar displays messages such as short “help messages” when you move the cursor over any button in any CrossView Pro window.

Menus

Each CrossView Pro window may have a menu associated with it. Under Microsoft Windows, the active window's menu is displayed in the menu bar of the desktop.

Depending on your execution environment some menu items are always grayed. For example, `Communication Setup` is grayed if your target is an instruction set simulator.

Windows

The debugger supports two types of windows: *primary windows* and *dialog boxes*. Dialog boxes are the windows you access from a primary window. For the remainder of this manual, the term “window” denotes a primary window.

This manual also uses the term *pop-up window*. A pop-up window is a primary window that contains supplemental information such as on-line help.

CrossView Pro Windows are used to display information and to get user input through either buttons, commands typed in input fields, or menu selections. Windows may be moved around the desktop, sized, or iconized. All windows can be opened from the `View` menu. The section on *CrossView Pro Windows* provides more detail about each window.



A window is considered opened even if it is iconized (under Microsoft Windows, this is called minimized). A window is considered closed if it does not exist on the desktop in any form.

Dialog Boxes

Certain menu items or push buttons may call up a dialog box to complete an action, display information, or get additional data. No other actions can be performed until the dialog box is closed.

4.5.1 MENUS

Each window in CrossView Pro uses the same menu. The active window's menu appears in the CrossView Pro desktop menu bar as shown in figure 4-2. The method of selection of a menu item varies depending on the windowing system being used. See your Windowing System's manual for details of how to do this.

Each window has a hidden Control menu, to manipulate the window, as part of the menu bar. The menu item Control | Close closes the current window. Your implementation of the windowing system may have additional features. See your documentation for further details.

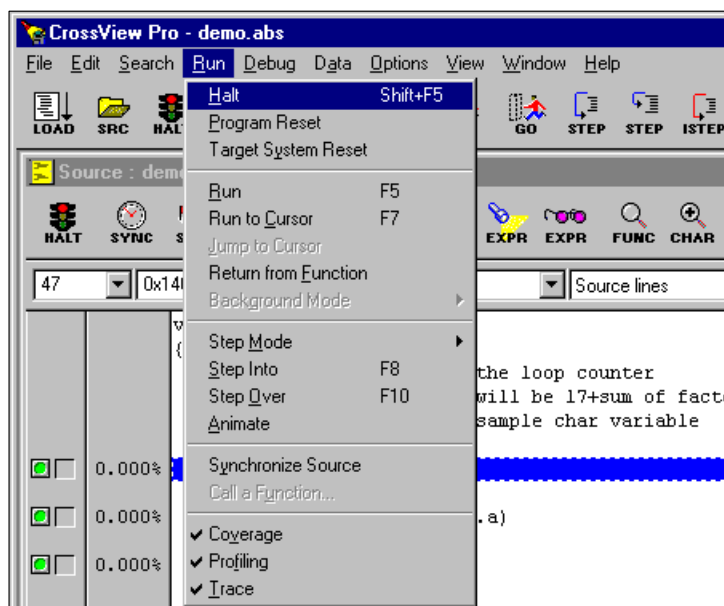


Figure 4-2: CrossView Pro Menus

4.5.1.1 LOCAL POPUP MENUS

On MS-Windows environments CrossView Pro supports local popup menus. Local popup menus are invoked by clicking the right mouse button. The menu contents is context sensitive. If the mouse pointer is on top of the global (main) toolbar the **Configure Toolbar** dialog is shown. If the mouse pointer is located in the MDI window (task window or background) the **View Menu** is shown which allows you to open new windows.

Within the Source Window four different local popup menus may appear. If the cursor is within the display area of the window the **Run Menu** is shown. The **Run Menu** contains commands associated with program execution. If your cursor is at a breakpoint indicator, the **New Code Breakpoint** or **Edit Code Breakpoint** dialog is shown. If the cursor is on a code coverage marker then the local popup menu contains commands to move the cursor to the next or previous block of (not)covered statements. If your cursor is in the profile column you can change the format of the timing figures. All other windows have their own local popup menu. The exception to the rule is the command window which does not have a local popup. See figure 4-3 for an example of the local popup menu of the **Memory Window**.

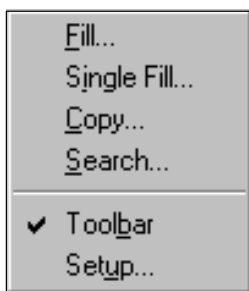


Figure 4-3: CrossView Pro Local Popup Menu (Memory Window)

4.5.2 WINDOW OPERATION

Windows can be opened, made active, and closed.

Opening Windows

The View menu of the menu bar lists all windows. Selecting a window name from this list causes the window to open up. Selecting a window that is already open brings that window to the front.

Selecting a Window

At any one time, a particular window is *active*. Most operations act (by default) on the active window. The active window is distinguished by highlighting the title bar. Only one window may be active at a time. There are several ways to select a window (that is, make a window active).

- Open the window from the View menu. If the window is already open it will be brought to the front.
- Click on the window's border (or on any portion of the window in some windowing systems). It will be brought to the front.
- Select the window name from the Window menu. The window will be made active and is brought to the front. (This option is available under Microsoft Windows only).

Closing a Window

Windows are closed by selecting Control | Close, Control | Quit or Control | Delete menu item, or by clicking a Close button, as shown in figure 4-4. Selecting this item from the Command Window will exit CrossView Pro.

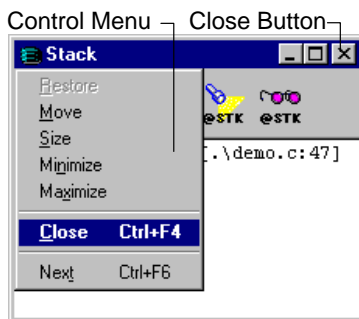


Figure 4-4: Closing a Window

4.5.3 DIALOG BOXES

The debugger uses dialog boxes to acquire information needed to complete a requested operation. The debugger also uses dialog boxes to display information. If a button or menu item displays an ellipsis (...) after its name, then there is an associated dialog box or pop-up window.

For example, the dialog box shown in figure 4-5 displays a list of Virtual I/O streams. This dialog box uses a scrollable list box, radio buttons to display the state, edit fields to enter information and push buttons to allow certain functions to be performed. Note that the *Browse...* button calls another dialog box. The *Help...* button causes the help pop-up window to be displayed.

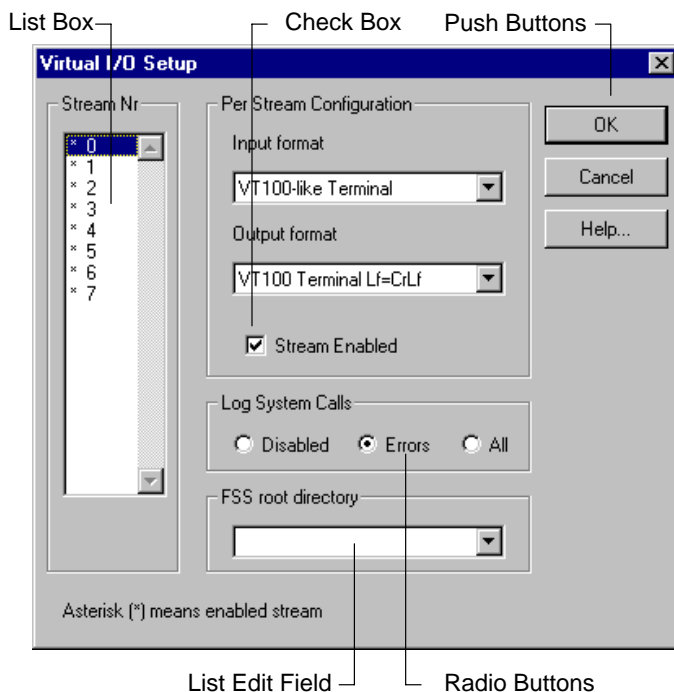


Figure 4-5: Dialog Box

4.5.4 CUSTOMIZING CROSSVIEW PRO

You can customize CrossView Pro's visual appearance and operative parameters to best suit your debugging environment.

Changing the Visual Appearance

Windows can be organized by resizing and moving them around the desktop (see your Windowing System's manual for details on how to do this). All windows under Microsoft Windows have an additional **Window** menu item. This menu allows the user to arrange all opened windows in a tiled or cascaded format. In the tiled format, selected by **Window | Tile**, all windows become the same size. All windows are the visible, the same size and do not overlap. In the cascaded format, selected by **Window | Cascade**, all open windows are changed to the same size and overlapped in a cascade with a constant vertical and horizontal offset. Iconized (minimized) windows can be automatically rearranged by selecting **Window | Arrange Icons** from the **Window** menu.

See the section *Using X Resources* in the chapter *Software Installation* for details on changing the visual appearance of CrossView Pro under X Windows.

Changing Operative Parameters

Operative parameters for CrossView Pro are adjusted by using the menu items from the **Options** menu, the **File | Communication Setup...** menu item and the **Setup** items of the **Debug** menu:

- **Record, Playback, and Log:** Allow you to set command recording and playback options.
- **Toolbox Setup, and Macro Definitions:** Allow you to define macros, and assign them to a push button in the **Toolbox**.
- **Startup:** Allows you to specify the source directories for CrossView Pro, the execution environment and the CPU type. The values are processed at CrossView Pro startup before executing commands entered in the **Command Window** or before the target is accessed as a result of opening a window. So, first edit this dialog when you start CrossView Pro. If you have not loaded a symbol file yet, you do not have to restart CrossView Pro.

- **Initialization...**: Allows you to specify the maximum number of breakpoints, assertions, special variables, C-trace instructions, command history lines, command output lines, emulator output lines, simulated I/O lines. The values are processed at CrossView Pro startup.
- **Desktop Setup**: Allows you to specify color settings for the execution position in the Source Window and the colors used in the Memory Window to show how a memory location has been accessed by the application program. You can also specify font sizes to be used in output windows.
- **Toolbar Setup**: Allows you to configure the main toolbar to your personal preferences.
- **Background Mode Setup**: Allows you to specify which windows to automatically refresh when running in background mode.
- **Communication Setup**: Allows you to set communications parameters.
- **Data Display Setup**: Allows you to specify how CrossView Pro displays data. This dialog also determines if the Expression Evaluation dialog box must be bypassed or not.
- **Memory Window Setup**: Allows you to specify the mode and size of the data and the number of data rows and columns to be shown in the Memory Window. It also allows you to automatically refresh the Memory Window and to display data coverage information.
- **Virtual I/O Setup**: Allows you to specify the Virtual I/O streams to be used in the Virtual I/O Windows. These windows are used by File System Simulation (FSS).
- **Simulated I/O Setup**: Allows you to specify the simulated I/O streams to be used in the Simulated I/O Windows.
- **Source Window Setup**: Allows you to specify the step mode, symbolic disassembly, automatically switching between source lines and disassembly source to be displayed in the Source Window and display code coverage information.
- **Register Window Setup**: Allows you to specify the registers that appear in the Register Window.

Saving Changes on Exit

If you find yourself using a particular configuration, you may want to save your configuration when you exit CrossView Pro:

- Select the **File | Exit** menu item or close the Command Window.

- In the Save Options dialog that appears, select the options you want to be saved for another debug session.
- Click on the **Exit** button in the Save Options dialog.

CrossView Pro exits immediately. If you selected one or more items in the Save Options dialog your settings are saved in the initialization file `xvw.ini`. This file is in the startup directory.

4.5.5 CROSSVIEW PRO MESSAGES

CrossView Pro communicates with you in a variety of ways. The command window displays the results of commands. Important messages, such as errors, appear in dialog boxes that pop up.

4.6 CROSSVIEW PRO WINDOWS

The two prominent windows used in CrossView Pro are the Command Window and the Source Window. From the Command Window you can type CrossView Pro and emulator commands, and gain access to all other windows. You can accomplish most global operations from either the menu bar or the Command Window. Only from the Command Window can you accomplish Single step playback. When you close the Command Window, you exit CrossView Pro.

The Source Window focuses on the program being debugged. This window controls most of the commonly-used execution operations, such as breakpoints and searching functions.

4.6.1 OPENING WINDOWS FROM THE VIEW MENU

You can open all CrossView Pro windows from the View menu by selecting the name of the window. Selecting a window in this case brings the window to front and makes it the active window. Available windows are:

- **Command Window** supporting two modes: CrossView or Emulator. Displays all CrossView Pro commands and responses or Emulator commands and responses.
- **Source Window**: Controls the execution of the program and displays the source file or disassembly.
- **Trace Window**: Displays the most recently executed lines.
- **Stack Window**: Displays the application's stack trace.
- **Register Window**: Displays the current state of the processor's registers.
- **Data Window**: Displays the values of data that are being monitored.
- **Memory Window**: Always you to display and modify target memory.
- **Simulated I/O Windows**: Allow simulated I/O for an application.
- **Kernel Windows**: Display real-time kernel information.

Improving CrossView Pro Performance

CrossView Pro updates every window that is open, even if it is iconized (minimized). Keeping a window up to date usually involves extra communication with the emulator, slowing CrossView Pro down. For instance, if the Register Window is open, CrossView Pro asks the emulator to dump the contents of all displayed registers after each single step. Thus it is a good idea to keep only those windows open that you are interested in.

4.6.2 COMMAND WINDOW

The Command Window allows you to:

- Enter CrossView Pro and emulator commands from the keyboard.
- View a history of CrossView Pro commands or emulator commands.
- View the result of CrossView Pro commands or emulator commands.
- Execute playback files (in single step mode).

From the View menu you can specify if you want the Command Window to be a CrossView Pro Command Window or an Emulator Command Window. This way you can specify whether CrossView Pro interprets commands or they go directly to the emulator.

Figure 4-6. shows the Command Window. You can type commands into the command edit field (bottom field) or select them from the command history list (middle field), edit and execute them. The command history field displays previously entered commands. You can select and execute one or more commands. The command history list provides you with a clear, comfortable way to re-execute specific commands or sequences of commands by preserving them in a scrollable list.

You can switch between the history list and the command edit field by hitting the **<Tab>** key. Hitting the **<Esc>** key (escape) returns you to an empty edit field.

The top field is the Command Output Window or the Emulator Output Window, depending on the type of Command Window you choose. Each command, echoed from the command edit field, appears with a '**>**' prefix. CrossView Pro displays its response (or the emulator's response if the window is an Emulator Command Window) to the command immediately following the command.

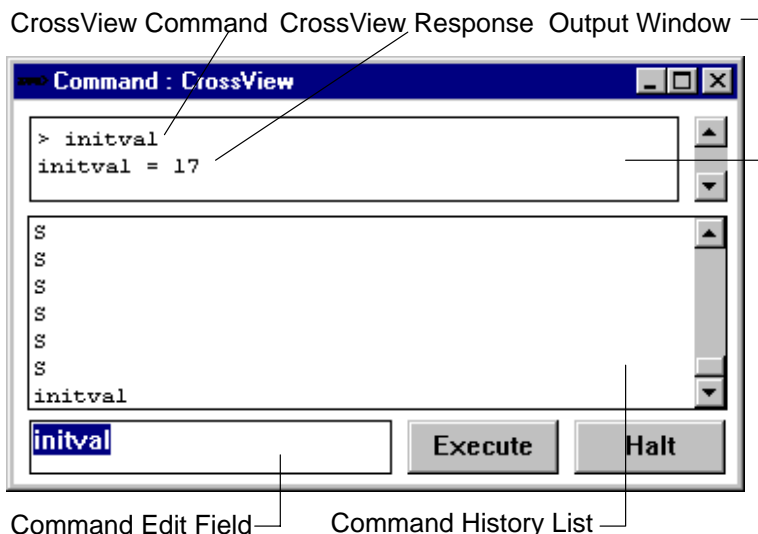


Figure 4-6: CrossView Pro Command Window

The Command Window also has two push buttons that provide rapid access to frequently used actions. The **Execute** button executes the current command (or sequence of commands if more than one command is selected). Note that the **<Enter>** or **<Return>** key is equivalent. Use the **Halt** button to interrupt commands executing in continuous mode, or to stop the emulator.

The Command Window maintains a history of recently executed commands. To re-perform previously executed commands simply double-click on it or select the command(s) from the command history list in the Command Window and press the **Execute** button. By hitting the **<Tab>** key, it is also possible to select one or more entries. Hitting **<Tab>** or **<Esc>** will return you to the command edit field.



The maximum number of lines saved to the CrossView Pro command buffer list is set during debugger startup. The default is 100 lines. To change the default select the **Options | Initialization...** item from the menu. This number can also be modified via a startup option.

4.6.3 SOURCE WINDOW

The Source Window offers most of the debugging functions you will need on a regular basis. It allows you to:

- View the source file (source lines, disassembly or both).
- Set and clear assertions (not in Toolbar).
- Set and clear breakpoints.
- Monitor and inspect variables.
- Search for strings, functions, lines, addresses.
- Control execution.
- Call functions (not in Toolbar) and evaluate expressions.
- View code coverage information.
- View profiling/timing information.

An example of the source window is shown in figure 4-7.

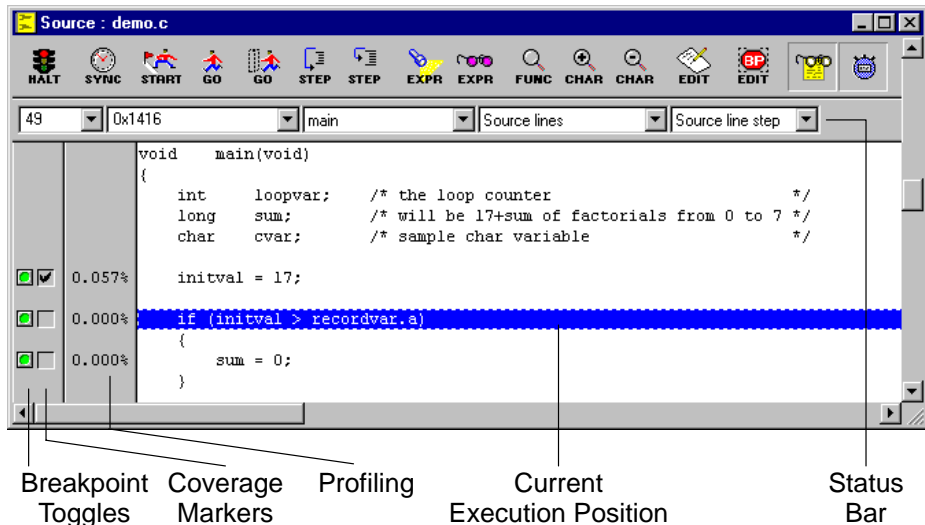


Figure 4-7: CrossView Pro Source Window

You can specify the step mode, symbolic disassembly and source lines / disassembly with the Source Window Setup dialog box (Debug | Source Window Setup...) or with Run | Step Mode. Alternatively, you can use the drop-down buttons in the Source Window's status bar.



The default step modes are:

Source lines Window:	Source line step
Disassembly Window:	Instruction step
Source and Disassembly Window:	mode of previous window!
(assumes the step mode of the previous Source Window setting)	

The location of the cursor is also the viewing position. The line number and address of the viewing position, appears at the top-left position of the Source Window. This does NOT represent the current execution position (\$pc). The current execution position appears in reverse or blue color. The cursor appears as a dotted line.

On MS-Windows the so-called "quick watch" feature is supported. When you position the mouse cursor over a variable or a function, a bubble help box appears showing the value of the variable or the type information of the function respectively.

A green colored toggle shows that no breakpoint is set. A red colored toggle indicates an installed breakpoint. An orange colored toggle indicates an installed but disabled breakpoint. If code coverage is enabled, coverage markers appear to the right of the breakpoint toggles. If a checkmark appears next to a line, it has been executed. If no checkmark appears next to a line, it has not been executed.

The Source Window provides a local Toolbar containing the following buttons, nearly all of which are shortcuts (using selected text) to operations that you can perform via the menu bar:



Stop program or command



Synchronize source














Restart program



Continue execution (same as F5)



Run to cursor (same as F7)

	Step (over function calls)
	Step (into function calls)
	Show selected source expression
	Watch selected source expression
	Find function
	Repeat search down for string
	Repeat search up for string
	Edit current source file
	Edit breakpoint at cursor
	Display code coverage
	Display profiling

You can toggle the appearance of this local toolbar by selecting the View
| Local Toolbars | Source menu item.

Edit Source

To edit the current source file, which appears in the Source Window, select **Edit | Edit Source**, or press the **Edit current source file** button. On MS-Windows the Codewright editor will be called with the filename and line number of the file that is currently in the debugger. on UNIX systems the **xvwdedit** program will be called with the filename and line number of the file that is currently in the debugger. The editor will be started and the file will be loaded.

The **xvwdedit** program is a shell script. You can adapt it to your specific requirements.

4.6.4 TRACE WINDOW

The Trace Window, shown in figure 4-8, allows you to:

- Display the most recently executed lines of code.

CrossView Pro automatically updates the Trace Window each time you halt execution, as long as the window is open, allowing you to check the progress and flow of your program throughout the debugging session.

The Trace Window is only supported if your execution environment supports the trace facility.

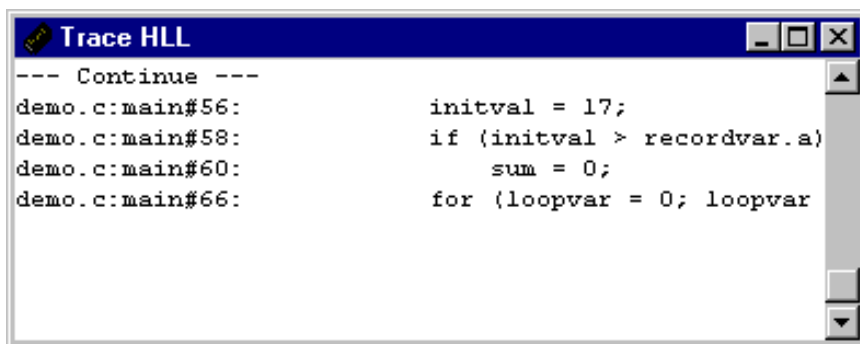


Figure 4-8: CrossView Pro Trace Window

4.6.5 STACK WINDOW

The *stack* records the return addresses of all functions the application has called, and CrossView Pro can use this information to reconstruct the path to the current execution position. The Stack Window, shown in figure 4-9, displays the function calls on the stack with the values of the parameters passed to them in an easily accessible and understandable form.

The Stack Window can help you assess program execution and allows you to view parameter values. The stack window allows you to:

- View the stack trace which includes information about function names, parameter values, source line numbers and stack level.
- Easily switch to the call statement of a stack level by clicking on it once.
- Set temporary and permanent breakpoints at any level of the stack, by double-clicking on the desired level.

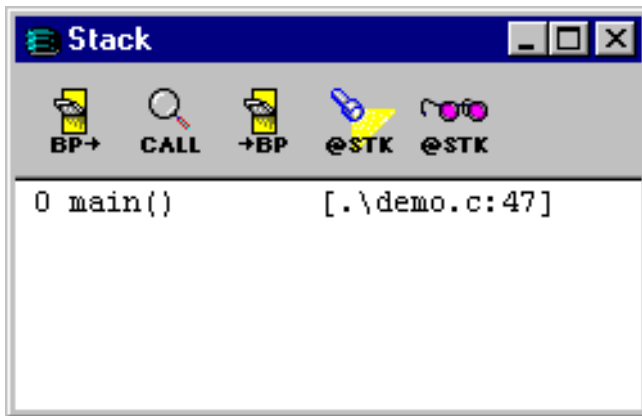


Figure 4-9: CrossView Pro Stack Window with Toolbar

The Stack Window provides a local Toolbar containing the following buttons:



Set stack breakpoint after function call point



Find call site



Set stack breakpoint at function entry point



Show variables in selected stack frame



Watch variables in selected stack frame

You can toggle the appearance of this local toolbar by selecting the View | Local Toolbars | Stack menu item.

4.6.6 REGISTER WINDOW

Figure 4-10 shows the Register Window. This window allows you to view and edit register contents.

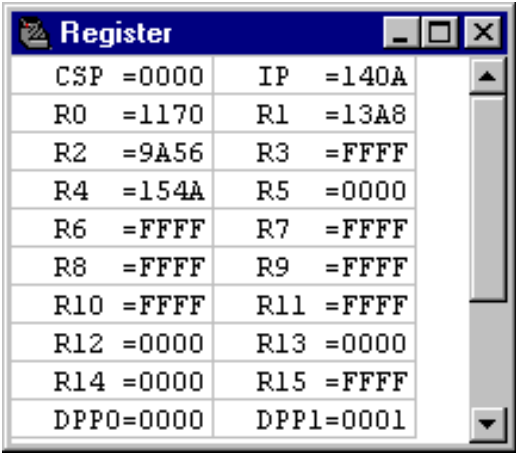


Figure 4-10: CrossView Pro Register Window



Note that the contents of the Register Window for your particular target may be different from the one show in figure 4-10.

You can specify which register set definition appears in the Register Window with the Register Window Setup dialog box (Debug | Register Window Setup...).

CrossView Pro supports multiple Register Windows. Register Windows either have the title "Register" or "Register – *register set name*". The "Register" title indicates the default register set.

In-situ editing allows you to change the registers contents directly by clicking on the corresponding cell.

4.6.7 DATA WINDOW

The Data Window is shown in figure 4-11. This window allows you to show the value of monitored expressions and variables.

The Data Window updates the values shown every time the program stops, and after an **o** command.

It is possible to display both monitored and unmonitored data expressions in the Data Window. CrossView Pro monitors and updates WATCH expressions after every halt in execution, and marks them with the text "WATCH" at the start of the display line in the Data Window. SHOW expressions, on the other hand, are one-shot inspections of an expression's value, and CrossView Pro does not update them except by direct user action. Initially, SHOW expressions appear as normal text until they are no longer known to be correct, at which time they appear with the word "OLD" at the start of the display line

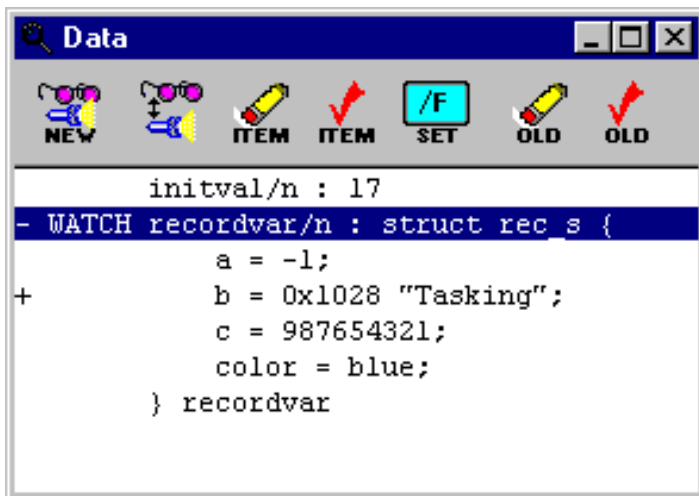


Figure 4-11: CrossView Pro Data Window

To inspect the value of global variables and data structures, double-click on the variable name in the Source Window.

Depending on preferences you set in the Data Display Setup dialog, the variable appears immediately in the Data Window, see figure 4-11, or the Expression Evaluation dialog appears first.

In-situ editing allows you to change the contents of everything in this window by clicking the value you want to change.

If you have set the `Display addresses` check box in the Data Display Setup dialog box the addresses of the variables are also shown.

Pointers, structures and arrays displayed in the data window have a compact and expanded form. The compact form for a structure is just `<struct>`, while the expanded form shows all the fields. The compact form of a pointer is the value of the pointer, while the expanded form shows the pointed-to object. Indicate the compact form by putting a '+' at the start of the display. (i.e., the object is expandable), and indicate the expanded form with (i.e., the object is contractible). Nesting is supported, so you can expand structures within structures ad infinitum.

To expand a pointer, structure or an array, double-click on the '+' in the Data Window

The Data Window provides a local Toolbar containing the following buttons:



Show/Watch new expression



Toggle watch attribute on selected item



Delete selected item



Update selected item



Reformat selected item



Delete old data items



Update old data items

You can toggle the appearance of this local toolbar by selecting the `View | Local Toolbars | Data` menu item.

The auto-watch locals feature may be activated or deactivated. When active, a selected Data Window becomes the "auto-watch" window, and all local variables from the current top-of-stack frame appear in that Data Window. The text "LOCAL" appears at the start of the display for variables displayed in this manner. As the execution position changes, the auto-watch window deletes and adds locals as necessary, so that the locals on the current top-of stack frame always appear.

To see the value of the local variables of a function, Select `View | Data | Watch Locals Window` from the menu.

CrossView Pro supports multiple Data Windows. Data Windows either have the title "Data Window #n" or "All Local Variables". The "All Local Variables" title indicates the auto-watch window if it exists (as explained above).

4.6.8 MEMORY WINDOW

The Memory Window is shown in figure 4-12. This window allows you to view and edit the target memory.

Depending on the setting of the `Auto Refresh` check box in the Memory Window Setup dialog, CrossView Pro updates the displayed values every time the program is stopped or only updates the values by user request. For example, by pressing the `Refresh memory window` button located on the toolbar.

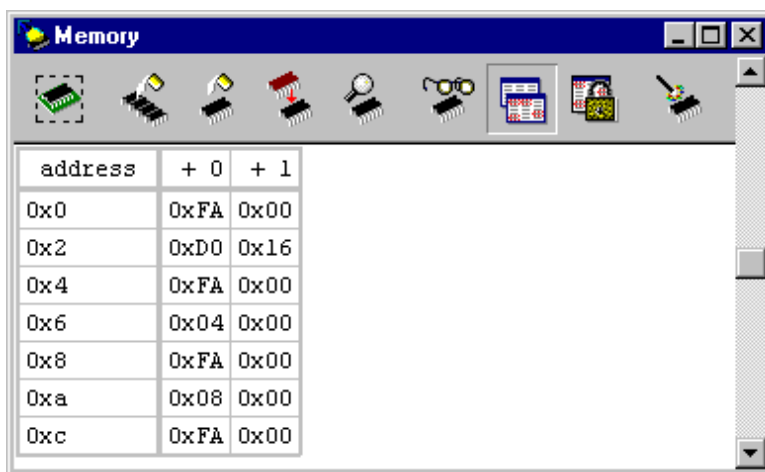


Figure 4-12: CrossView Pro Memory Window

To edit the target memory, click on a memory cell and type a new value. To display another memory region: click on an address cell and type a new address. CrossView Pro accepts input in symbolic format, so you can enter expressions instead of just values.

CrossView Pro supports multiple instances of the Memory Window. If your target supports multiple memory spaces, the Memory Window supports them all. Refer to the section about memory space keywords to become familiar with the memory space keywords and associated syntax your target system uses.

You can specify the way data appears in the Memory Window by opening the Memory Window Setup dialog. Select **Debug | Memory Window Setup...** to open this dialog. The memory contents can appear in many formats including ASCII character, hexadecimal, decimal, signed, unsigned, and floating point formats. You can specify the size of the memory window. You specify the number of memory cells that appear within the window. The number of cells is fixed in the sense that if you re-size the window the number of cells does not change.

Besides the current value of memory locations, the Memory Window also displays whether memory locations have been accessed during program execution. This is called 'data coverage'. An application program may read from, write to, or fetch an instruction from a memory location. Of course all combinations may be legal. Although writing data to a memory location from which an instruction has been fetched is suspicious. All types of accesses, read, write, fetch or combinations of these, can be shown using different foreground and background colors. The color combination used to show "rwx" access are specified in the `Desktop Setup` dialog. Change the background color if instructions are fetched from a memory location, and change the foreground color to show read and write access.

You can display data coverage information in the Memory Window by clicking on the `Display coverage` button in the Memory Window or by setting the `Display Data Coverage` check box in the Memory Window Setup dialog.

The Memory Window has the ability to highlight memory cells of which the contents have been changed. Click on the `Highlight changed values` button in the Memory Window to see the changed cells. With the `Set reference` button you can enter a new reference point for highlighting. All the cells that have been changed since that reference point are highlighted.

The Memory Window provides a local Toolbar containing the following buttons:



Setup memory display



Fill memory



Single fill memory



Copy memory



Search memory



Display data coverage



Highlight changed values



Set reference



Refresh memory window

You can toggle the appearance of this local toolbar by selecting the View
| Local Toolbars | Memory menu item.

4.6.9 VIRTUAL I/O WINDOW

The CrossView Pro Virtual I/O windows provide an interface to exchange data with the application on the target. This I/O facility can be implemented in various ways. Using standard stream I/O function calls like `printf()` in your source, you can test I/O to and from the target system or simulator.

The debugger supports up to eight separate Virtual I/O windows simultaneously.

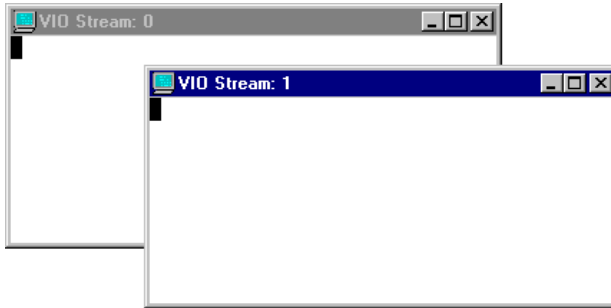


Figure 4-13: CrossView Pro Virtual I/O Windows

You can setup the virtual I/O streams with the Virtual I/O Setup dialog box (Debug | Virtual I/O Setup...).

4.6.10 SIMULATED I/O WINDOW

The Simulated I/O Windows, shown in figure 4-14, let you observe and simulate the input and output of your program before the hardware peripherals are in place.

Using special function calls in your source, you can simulate I/O to and from the target system. The debugger supports up to eight separate Simulated I/O windows simultaneously.

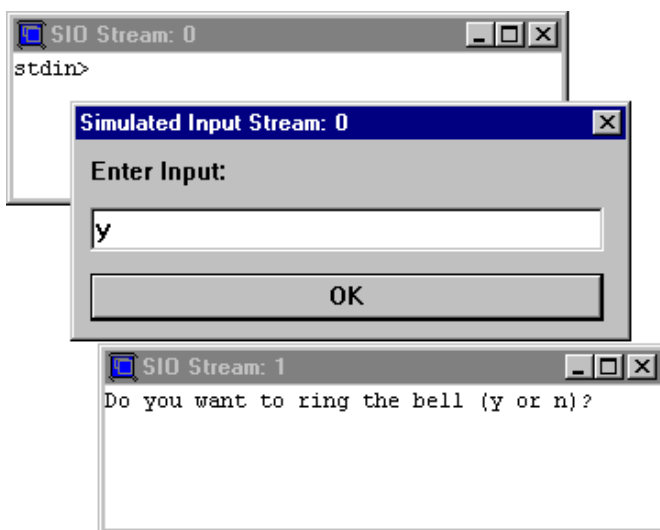


Figure 4-14: CrossView Pro Simulated I/O Windows

You can setup the simulated I/O streams with the Simulated I/O Setup dialog box (Debug | Simulated I/O Setup...).

4.6.11 POP-UP WINDOWS

Finally, two more windows appear in certain situations:

Help Window: Activated with function key F1 or when a Help button is pressed inside a dialog.

Toolbox: This window contains user definable buttons.

4.7 CONTROL OPERATIONS FOR CROSSVIEW PRO

All control operations can take place in any CrossView Pro Window. You can select and save startup options. You can record and play back playback files. You can define macros and assign them a button in the toolbox (allowing you to configure up 16 buttons).

4.7.1 ECHOING COMMANDS

The Command Window echoes every command given to CrossView Pro. CrossView Pro translates most button actions and menu selections into the CrossView Pro keyboard command equivalents. The Command Window echoes the equivalent commands just as if you had typed them there.

4.7.2 MOUSE/MENU/COMMAND EQUIVALENTS

Actions in CrossView Pro are performed by using keyboard commands typed into the Command Window, selecting a menu item, by clicking on a push button and sometimes by direct manipulation of objects with the mouse. Many actions can be accomplished several ways. For instance there are three different ways to set a breakpoint. You can:

1. Use the ***line b*** command in the command entry field.
2. Click on a breakpoint toggle in the Source Window.
3. Select Debug | Breakpoints... menu item to open up the Breakpoints dialog box.

4.7.3 BUTTON SELECTION

There are three types of buttons available with CrossView Pro:

- **Radio Buttons:** These buttons are grouped together. You can only select one button from a group at a time (just like a car radio button). This type of button is commonly used to select a mode of operation.
- **Check Boxes:** These are special buttons which you can turn on or off by selecting them. If more than one check box is present, you can select as many as you like.

- **Push Buttons:** These buttons do something. They either produce a CrossView Pro command or perform some action related to the Windows interface. A special form of push buttons are the **Accelerator Buttons** of the **Toolbar**. Some windows, such as the Source Window and Data Window, have a local toolbar.

4.7.4 TEXT SELECTION

Several windows allow you to select text using the mouse. You can use this text in a variety of ways.

Most windows operate on a line of information at a time. To select a line of text, move the mouse pointer to the text and click. The entire line appears highlighted indicating that it is selected (not in the Source Window). You can select multiple lines by either not releasing the mouse button and dragging the mouse pointer over several lines or by holding down a modifier key and clicking on the new line, depending on the windowing system that you use.

Other windows, such as the Source Window, allow you to select portions of a line (such as a variable name). To select a portion of a line, drag the mouse over the desired text while holding down the left mouse button. Double-clicking text selects and monitors an expression.

4.8 USING THE ON-LINE HELP SYSTEM

CrossView Pro has an extensive on-line help system to aid you. CrossView Pro uses a windowing system's Help system. This Help system uses pop-up windows for definitions and hypertext jumps to cross reference different topics. Topics for Help cover all CrossView Pro Windows, commands, and dialog boxes.

4.8.1 ACCESSING ON-LINE HELP

You can access it by one of several ways:



All dialog boxes have a Help button. Pressing this button brings you directly to the subject matter related to the dialog box in a Help Pop-up Window. Also, CrossView Pro has F1=Help on the menu bar (MS-Windows only). For the X Windowing systems (Motif) the File menu contains a Help entry. Selecting this item opens up a Help Pop-up Window containing information related to the window that called it.



CrossView Pro maps the F1 key to the help function. Pressing this key at any time gives you access to the on-line help system.

4.8.2 COMPONENTS OF MS-WINDOWS HELP

The Help Pop-up Window is composed of text, definition boxes, and jumps to other topics, as shown in figure 4-15.

- **Text:** Main body of the subject.
- **Definition Boxes:** Contain the definition for a word or phrase in the main text body. Text with a broken underline has a definition box associated with it. Clicking on this text causes the definition of that term to appear in an informational dialog box.
- **Jumps:** A jump is a link to another subject area. Clicking on text underlined with a solid line jumps you to the new subject.

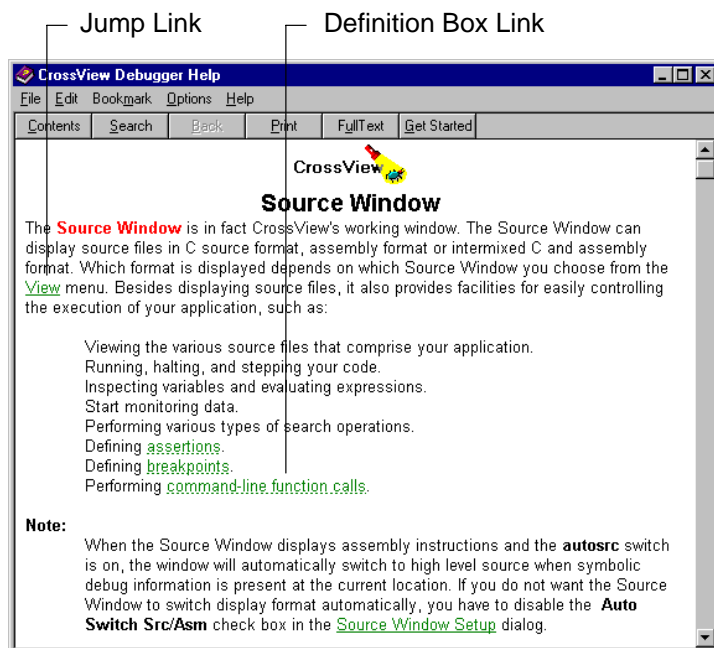


Figure 4-15: CrossView Pro Help Window

4.8.2.1 USING MS-WINDOWS HELP

Help, as mentioned previously, is context sensitive. When you first enter Help, you are at a topic related to the current window or dialog box. By clicking on jump links, you can follow different paths. You can return to your starting point by clicking the Back button or by using the History button and clicking on the node that you want to return to.

You can also use Help to browse different subjects. At any time in help, you can click on the Contents button. This displays a list of main subjects. Alternatively, you can click the Search button. A dialog box appears that allows you to search for a subject string, or to select a subject from a scrolling list. Clicking on the Show Topic button lists all topics pertaining to that subject.

To save time, you can iconize the Help Window and maximize it when necessary.

CHAPTER

5

CONTROLLING PROGRAM EXECUTION



5

CHAPTER

5.1 SOURCE POSITIONING

When you have the Source Window open and it displays a source file, there are two points of reference to keep in mind: the execution position and the viewing position.

The **execution position** refers to the line of source at the Program Counter address. This line is always the next statement or instruction to be executed. When you load a file into the Source Window, CrossView Pro automatically displays the portion of the source code that contains the execution position.

The **viewing position** (also called 'cursor') is the line currently being examined in the displayed source file. Since many Source Window operations act on this line, you can think of the viewing position as the 'current line'. For instance, if you set a breakpoint without specifying a line number, CrossView Pro sets the breakpoint at the line marked by the viewing position. Please note that it is the viewing position that appears to the left of the Source Window (NOT the execution position!).

The execution position and the viewing position refer to the same line when a source file is first loaded into the Source Window. You can then change the viewing position, if you wish.

The execution position and the viewing position appear different to distinguish them from the rest of the source code. The execution position line appears in the execution position highlight colors, while the viewing position appears as a broken-line frame, also called the **cursor**. Note that a line containing a breakpoint appears in the breakpoint highlight colors.

A combination of execution position, cursor and breakpoint (all of which are potentially active on the same line) appear accordingly.

5.1.1 CHANGING THE VIEWING POSITION

When a program is active the viewing position is always visible in the Source Window. You can change the viewing position to move throughout the source file. Usually, whenever the execution position changes, the viewing position automatically follows suit. But you may easily change the viewing position without affecting the execution position.



To change the viewing position use any of the following possibilities:

- Use the vertical scroll bar to move a line or a page at a time. The view point stays on the same line until it is no longer visible. It then stays on the first or last line of the display, depending on the direction of scrolling.
- Click on the desired, unmarked source line.
- Select the **Search | Find Line...** menu item to specify to which particular line you wish to move.

In the upper-left corner of the Source Window, there are two text fields. These fields show the line number of the current viewing position and the address of the first machine instruction for that line. CrossView Pro updates the **Line** and **Address** values each time the viewing position changes.



You can change the viewing position to the first executable line of a particular function with the **e** command. For instance:

```
e main
```

will make the first executable line of `main()` the current viewing position and display it in the Source window. You may also use the stack depth as an argument, if you place it before the **e**:

```
1 e
```

This will change the viewing position to stack depth 1, that is, the line that called the current function.

FUNCTION: Change the viewing position.

COMMAND: *stack e*
 e function

To change the viewing position to a specified address, you can use the **ei** command. This command is useful for viewing some code in the assembly window, without changing the program counter, since the execution position is not changed.

FUNCTION: Change the viewing position to address.

COMMAND: *address ei*

5.1.2 CHANGING THE EXECUTION POSITION

There may be times when you want to start or resume execution at a different line than the one marked by the current execution position.

Exercise caution when changing the execution position. Often each line of C source code compiles into several machine language instructions. Moving the program counter to a new address in the middle of a series of related assembly instructions is sometimes risky. Moreover, even though you change the program counter, registers and variables may not have the expected values if you bypass parts of the code.



In the Source Window you can change the execution position to the viewing position with the menu entry **Run | Jump to Cursor**. This menu entry is disabled in Source file window mode to prevent problems by skidding pieces of C code which are required to be executed. See also the **g** and **gi** commands below.



When the program halts, you can change the execution position with the **g** command in the Command Window. The **g** command moves the execution position, but does not continue the program. To resume execution from your new execution position, use the **C** command.

Although risky, the **g** command does have its uses, especially in conjunction with breakpoints to patch code. Refer to the *Breakpoints and Assertions* chapter for more information.

For example, to change the execution position from the current line, 54, to line 62, enter:

g 62

When you resume execution in this program, it is from line 62 instead of line 54.

FUNCTION: Change the execution position to a specified C source line

COMMAND: **g** *line_number*

You can also change the execution position to a specified address directly, although the same warnings apply. To do so, use the **gi** command. For instance:

0x800 gi

FUNCTION: Change the execution position to address.

COMMAND: *address* **gi**

Of course, moving the program counter (**gi** command) is even more potentially reckless than using the **g** command. Use both with caution especially when debugging a program which has instructions re-ordered due to optimizations.

To determine the address of a line of source, use the **P** command:

80 P
80:(0x1486): sum = sum + 1;

The hexadecimal number in parentheses is the instruction address for line 80.

FUNCTION: Print a source line and its instruction address.

COMMAND: *line_number* **P**

5.1.3 SYNCHRONIZING THE EXECUTION AND VIEWING POSITIONS

The viewing position is always visible in the Source Window when a program is active. The execution position, however, can disappear from view when you scroll through the program or load a new file. Use the synchronize source commands to bring the execution position back into view after scrolling or loading a new source file.

Each time you stop execution, the execution position and the viewing position synchronize. The viewing position always moves to match the execution position. To synchronize the positions manually:



Click on the `Synchronize source` button in the Source Window or select the `Run | Synchronize Source` menu item.



From the Command Window, use the **L** command.

The L Command

The **L** command is shorthand for **oe**. It synchronizes the viewing and the execution positions, adjusting the viewing position if the two are different. The **L** command never affects the execution position. The **L** command is useful if you have changed your viewing position and do not remember where your execution position is.

FUNCTION: Synchronize viewing and execution position.

COMMAND: **L**

5.2 CONTROLLING PROGRAM EXECUTION

Using the mouse in the Source Window, you can direct the execution of your source programs. Among your options are:

- Starting execution from the first instruction or from the current execution position.
- Manually stopping execution whenever you want.
- Executing the program a single line at a time.
- Executing functions by calling them directly.

5.2.1 STARTING THE PROGRAM

To restart a program from its first instruction:



Click on the Restart program button in the Source Window.

or:

- Select Run | Program Reset menu item.
- Select Run | Run menu item, or click on the Continue execution button.



Type the **R** command from the Command Window.



This is NOT a target system reset. Refer to the **rst** command for information about side effects that may be introduced due to a target system reset.

After restarting a program, you can stop execution only by a breakpoint, an assertion or a halt operation from the user.

FUNCTION: Reset program; run program.

COMMAND: **R**

5.2.2 HALTING AND CONTINUING EXECUTION

To stop or continue execution:



Click on the **Halt** button in the Source Window to stop execution. Click on the **Continue** execution button to resume execution.



Select the **Run | Halt** menu item to stop execution. Select the **Run | Run** menu item to resume execution.



Use the **C** command or function key **F5** to resume execution.

When you halt the program, all the active windows update automatically to reflect the program's current status. For instance, if you have any expressions monitored in the Data Window, their current value appears.

Note that when you use any of the above methods to stop the program, CrossView Pro halts at the machine instruction that was on when interrupted. While this is a convenient way to stop the program, it is hardly an accurate one — you may stop execution in the middle of a C source statement.

To stop a program at a precise line of C source code, set a breakpoint. For more about breakpoints see the *Breakpoints and Assertions* chapter.

When continuing, CrossView Pro resumes execution as if the program had never stopped.

FUNCTION: Continue execution from the current execution position.

COMMAND: **C**

5.2.3 SINGLE-STEP EXECUTION

When the program stops, you can continue execution, or you can step through it one line or instruction at a time. This is called **single-step execution**.

Single-stepping is a valuable tool for debugging your programs. The effect is to watch your programs run in stop motion. You can observe the values of variables, registers, and the stack at a precise point in a program's execution. You can catch many potential bugs by watching a program run line by line.

When you single step, CrossView Pro normally executes one line of your source and advances to the next sequential line of the program. If you single step to a line that contains a function call, however, you have two options: step into the function or step over the function call.

Source Single-Step Into

There are several methods you can use to single step into:



Click on the Step Into button in the Source Window or select the Run | Step Into menu item.



Press function key F8 or type the **s** command in the Command Window. You have the option of setting the number of lines you want to execute. For example, to execute 2 lines of the program, type: **2 s**.

FUNCTION:	Step through a program one source line at a time.
COMMAND:	<i>number s</i>

Stepping Into Functions

Stepping into a function means that CrossView Pro enters the function and executes its prologue machine instructions, halting at the first C statement. When you reach the end of the function, CrossView Pro brings you back to the line after the function call and continues with the flow of the program. The debugger changes the source code file displayed in the Source Window, if necessary.



If you accidentally step into a function that you meant to step over, you can select the Run | Return from Function menu item to escape quickly.

For example, suppose you are at line 59 of a file, which contains a call to the function `factorial()`:

```
main#59:    table[ loopvar ] = factorial(loopvar)
```

By performing one **Step Into** action, you can step into the source code for `factorial()`. Your Execution and viewing position change to:

```
factorial#103:  char locvar = 'x';
```

CrossView Pro shows you the current function and line number and the C source code for the current execution position.

Source Single-Step Over

To step *over* a statement or a function call:



Click on the **Step Over** button in the Source Window or select the **Run | Step Over** menu item.



Press function key **F10** or enter the **S** command in the Command Window. You have the option of setting the number of lines you want the debugger to execute. For example, to execute three lines of source, single stepping over functions, enter: **3 S**.

FUNCTION:	Single step, but treat function calls as single statements.
COMMAND:	<i>number S</i>

Stepping over Functions

Stepping over a function means that CrossView Pro treats function calls as a single statements and advances to the next line in the source. This is a useful operation if a function has already been debugged or if you do not want to take the time to step through a function line by line.

For example, suppose you reach line 59 in `demo.c`, which calls the function `factorial()`, as in the example above. If you give a **Step Over** command, the execution position moves to line 60 of the source code in the `main()` function immediately, without entering the source code for `factorial()`. CrossView Pro has executed the function call as a single statement.

If you try to step over a function that contains a breakpoint or that calls another function with a breakpoint, CrossView Pro halts at that breakpoint. Once execution stops, the step over command is complete. Therefore, if you resume execution by clicking on the Run button or with the **C** command, you do not regain control at the entrance to the function with the breakpoint. You can either single step through the rest of the function, or select the Run | Return from Function menu item to return to the line after the point of entry.

5.2.4 STEPPING THROUGH AT THE MACHINE LEVEL

While single stepping through code at the source level is informative, you might need a lower level approach. CrossView Pro can step through a program at the assembly language instruction level.

While more time-consuming than a source level step-through, an instruction level step-through allows you to examine how your code has been compiled. As you advance through the assembly instructions, notice how CrossView Pro translates data addresses to variable names, and correlates branch addresses to points in the source code. This makes it much easier to follow the source at the instruction level.



The default step modes are:

Source lines Window:	Source line step
Disassembly Window:	Instruction step
Source and Disassembly Window:	mode of previous window!
(assumes the step mode of the previous Source Window setting)	



Mouse and menu actions:

- The Step Into and Step Over buttons, and Run | Step Over and Run | Step Into menus can be set to step by instructions by selecting Run | Step Mode | Instruction step from the menu bar.
- To change back to stepping by source lines, select Run | Step Mode | Source line step.
- Another way to set the step mode is to select the Source line step or Instruction step radio button in the Debug | Source Window Setup dialog box.



To control this function from the Command Window, use the **Si** and **si** commands. The **Si** and the **si** commands are analogous to the **S** and **s** commands, **Si** will treat function calls (more precisely, jump to subroutine instructions) as single statements, while **si** will enter the function.

FUNCTION: Single step at instruction level. Step into functions.

COMMAND: *number si*

FUNCTION: Single step at instruction level. Step over functions.

COMMAND: *number Si*

As an example of stepping through instruction level code, restart the program. Then select Run | Step Mode | Instruction Step. Once it stops at the breakpoint you installed, advance execution one assembly language instruction at a time by using the Step Over and Step Into buttons. Or give the **Si** or **si** commands.

CrossView Pro will display disassembly of the next machine instruction that forms part of the C code in the Command Output Window:

```
main#47+0x4:      disassembled instruction
```

Different types of targets, of course, have different assembly code, so debugging at the assembly level is hardware dependent.

Notice that a single C statement is usually compiled into several, sometimes many, machine instructions.



CrossView Pro supports debugging on machine instruction level using the Intermixed or Assembly mode of the Source Window.

5.3 NOTES ABOUT PROGRAM EXECUTION

If you stop the program in a module without debug symbols, then an **S** or **s** command attempts to step to a module with symbols. CrossView Pro does this by searching the run-time stack for a return address in a module with symbols, then setting a temporary breakpoint there, and running. This process relies on two assumptions: that the stack layout is uniform, and that each function eventually returns. In the unlikely event that these assumptions are violated, the program may run away when you attempt to single step.

5.4 SEARCHING THROUGH THE SOURCE WINDOW

CrossView Pro can search for addresses and functions in the entire application and for line numbers, and strings in the current source file. A string search starts from the current viewing position and "wraps around" the end (or begin) of the current source file. The string search ends when a matching string is found or when it returns to the starting point.

5.4.1 SEARCHING FOR A FUNCTION

There are several ways to initiate a search for a function:



Using the mouse:

- Open up the Browse Function dialog box by clicking on the Find Function accelerator button or by selecting the Search | Browse Function... menu item. Type in the name of the function, select a function name from the drop-down history list or click on the Browse button to select the function name.
- Alternatively, you can highlight the name of the function in the source code by clicking and dragging the mouse. To search for the highlighted function name, click on the Find Function accelerator button.
- Selecting a function in the Stack Window shows the line that called it.



From the Command Window, you can either specify **c** followed by the function name, or a stack position followed by **c**. For example:

```
e main      Find the function main().
1 e         Find the line that called the current function.
```

CrossView Pro searches through all the relevant source code files to find the one containing the body of the function. The part of the file containing the function appears in the Source Window.

5.4.2 SEARCHING FOR A STRING

CrossView Pro allows you to search for a particular string in the current source file. CrossView Pro searches the Source Window from the current viewing position. If it finds the string, it moves the viewing position to the corresponding line. This does not affect the execution position.

To search for a string:



Open the Search String dialog box by clicking on the Repeat search down or Repeat search up accelerator buttons, or selecting the Search | Search String... menu item. Enter the string to search for in the box or select one from the drop-down history list. You can turn case sensitivity on or off by clicking on the Case Sensitive check box.

Alternatively, you can highlight the string to search for in the source code, by clicking and dragging the mouse. To search for the highlighted string, click on either the Repeat search up or Repeat search down accelerator button.



In the Command Window, use the / or ? commands. The / command searches forwards and the ? command searches backwards. For example, to find the string `initval`, enter:

```
/initval    Search forward for the string "initval"
```

CrossView Pro's searches "wrap around" beyond the top or bottom of the file if necessary.

FUNCTION: Search forward for a string.

COMMAND: / *string*

FUNCTION: Search backward for a string.

COMMAND: ? *string*

If no string is supplied to the / or ? command, or if you hit carriage return, or press the function key F3 or select the `Search | Search Next String` menu item, CrossView Pro searches again for the last string requested.

5.4.3 JUMPING TO A SOURCE LINE

As mentioned earlier in the *Changing the Viewing Position* section, you can use the scroll bar to scroll through the source code or use the arrow keys or the + and – keys. To find a specific line, you can use one of several methods:



Select the `Search | Find Line...` menu item to open the Find Line dialog box.

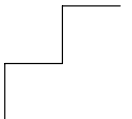
After you enter a line number (or select one from the history list) in this dialog box and click on the `Find` button, CrossView Pro will change the viewing position to the indicated line number. At the first use, the Find Line dialog box contains no line number, but on subsequent invocations it will show the line number you entered before.



Enter the line number on the command line.

CHAPTER 6

ACCESSING CODE AND DATA



6

CHAPTER

6.1 INTRODUCTION

This chapter discusses topics related to viewing and editing the variables in your source program and execution environment, including accessing variables and registers, viewing and modifying the data space, using monitors, viewing the source file, and disassembling code.

6.2 ACCESSING VARIABLES

This section describes how to view and edit your program variables using the debugger. You can monitor data so that every time you stop the program, CrossView Pro updates the current value.

The Data Window displays the values of variables and expressions. As long as the this window is open, CrossView Pro automatically updates the display for each monitored variable and expression each time the program stops.



Uninitialized variables will not have meaningful values when you first start the debugger, since your program's startup code has not been executed. Also note that global data is initialized at load time. Re-running a program may produce unexpected results. To guarantee that global data is initialized properly, download the program again.

6.2.1 VIEWING VARIABLES, STRUCTURES AND ARRAYS

You may view variable values, and change them, from the Source Window and the Command Window. CrossView Pro returns the variable in the format *var_name = value* in the Command Window.

It is possible to display both monitored and unmonitored expressions in the Data Window. After every halt in execution, CrossView Pro updates monitored expressions. Unmonitored expressions are just one-shot inspections of the expressions value. Refer to section 4.6, *CrossView Pro Windows* for a detailed description of the Data Window.

To show the contents of a variable or to show the type information of a function:



Position the mouse cursor over a variable or a function in the Source Window. A bubble help box appears showing the value of the variable or the type information of the function, respectively.

To evaluate a simple expression:



Double-click on a variable in the Source window. The result of the expression is shown in the Data Window. Alternatively, depending on the preferences you set in the Data Display Setup dialog, the expression appears in the Evaluate Expression dialog. Select the Watch or Show button to display the result of the expression in the Data Window. Select the Evaluate button to display the result of the expression in the output field of the Evaluate Expression dialog.

To evaluate a complex expression:



Select **Data | Evaluate Expression...** from the menu and type in any C expression in the Evaluate Expression dialog box. Optionally select a format code. Click the Evaluate button.



Type the expression into the command edit field of the Command Window followed by a return or press the Execute button.

For example, to find the value of `initval` in `demo.c` type:

```
initval
```

and CrossView Pro will display:

```
initval = 17
```

FUNCTION: Display the value of a variable.

COMMAND: *variable's_name*



For variables having the same name as an CrossView Pro command, use `/n` as format style code.



Any expression that can be typed into the Command Window can also be typed in the Expression field of the Expression Evaluation dialog box. Throughout this discussion, expressions can be typed in either location, depending on what is convenient.

Viewing Structures

You can also view structures.

By using any of the methods described above, you can print out the entire structure. For example:

recordvar

and CrossView Pro prints out the structure of `recordvar` and values of `recordvar`'s fields in correct C notation:

```
recordvar = struct rec_s {  
    a = -1;  
    b = 0x1028 "TASKING";  
    c = 987654321;  
    color = blue;  
} recordvar
```

Displaying Individual Fields

Similarly, you can instruct the debugger to print the value of an individual field.

In the Source Window, highlight `recordvar.color` and press the Show selected source expression button. Or, in the Expression edit field of the Expression Evaluation dialog box or in the Command Window, type the structure name followed by a period and the field name. For instance, to see the field `color` for the structure `recordvar`, enter:

```
recordvar.color Command  
color = blue    Output
```

Note that CrossView Pro returns the value in the form *field_name = value*. CrossView Pro also displays enumerated types correctly.



Variables will not have meaningful values when you first start CrossView Pro, since your program's startup code has not been executed.

Displaying the Address of an Array

If you enter the name of an array in the **Expression Evaluation** dialog box or in the **Command Window**, the debugger returns its address. For instance, to find the address for the array `table`, select `table` from the browse list in the dialog box or type the name in the **Command Window**:

table	<i>Command</i>
table = 0x200	<i>Output</i>

Note that **CrossView Pro** returns the address in the form *array_name = address*.

The debugger can also display the address and value of an individual element of an array. Enter the name of the array and the number of the element in brackets. For instance, to find the address and value of the third element of array `table`, enter:

table[3]	<i>Command</i>
0x20C = 0	<i>Output</i>

Note that **CrossView Pro** returns the information in the form *address = value*.

Displaying Character Pointers and Character Arrays

The following piece of C code can be accessed in **CrossView Pro** using the string format codes:

```
char text[] = "Sample\n";
char *ptext = text;
```

text	<i>What is the address of this char array</i>
text = 0x8200	

text/a	<i>Print it as a string</i>
text = "Sample^J"	

ptext	<i>What is the contents of this pointer</i>
string = 0x8200	

ptext/s	<i>Print it as a string</i>
string = "Sample^J"	

&ptext	<i>Where does ptext itself reside</i>
0x8210	

Sizing Structures

With structured variables, it is especially useful to know the size of a variable.

In the Command Window, you can determine the size of a variable with the `sizeof()` function. For instance, to determine the size of the structure `recordvar`, enter:

```
sizeof(recordvar)  
24
```

6.2.2 CHANGING VARIABLES

With CrossView Pro, you can not only view your variables, but change them. This function allows you to easily test your code by single-stepping through the program and assigning sample values to your variables. For instance, to set the variable `initval` to 100, enter:

```
initval=100
```

and CrossView Pro confirms `initval`'s new value:

```
initval = 100
```

Note that CrossView Pro returns the values of variables with the syntax: *var_name = value*, with any right-hand side expression evaluated to a single value.

Changing variables in the Data Window



To change a variable in the Data Window, follow these steps:

- In the Data Window, double-click on the variable you wish to edit. In-situ editing will be activated.
- Specify the new value in the edit control and hit the **Enter** key.

When in-situ editing is active, you can use the **Tab** key to move the edit field to the next variable value or use the **Shift+Tab** key combination to move the edit control to the previous variable.



Assigning Structures

CrossView Pro also allows you to assign whole structures to one another.

You can use a simple equation to assign the structures. For instance, to assign `statrec` to `recordvar`, enter:

```
statrec = recordvar
```

6.2.3 THE l COMMAND

CrossView Pro’s windows contain a great deal of information about the current debugging session. Occasionally, however, you have a few closed windows, or wish the information to appear in the Command window (for instance, when you are recording output). Using the **l** (list) command, you can find out all sorts of things about the current state of the debugger and have the information appear in the Command window.

Arguments of the l Command	
a assertions	k kernel state data
b breakpoints	m memory map (of application code sections)
d directory	p procedures (functions)
f files (modules)	r registers
g globals	s special variables

For configurations that support real-time kernels the **l k** command can have additional arguments. See the description of the **l** command in the *Command Reference* for details.

You may for example view the contents of the registers:

```
l r
```

Or the list of procedures (that is, functions):

```
l p
```

a complete list of global variables:

```
l g
```

The **lf** command (list files) also shows the address where CrossView Pro placed the first procedure in the module. If the module is a data module then the address reflects the first item's placement.

With all of these **l** commands you can specify a string:

l g record

and CrossView Pro searches the globals for a match with the same initial characters; in this case global variables that begin with `record`.

6.3 EXPRESSIONS

6.3.1 EVALUATING EXPRESSIONS

CrossView Pro expressions use standard C syntax, semantics, and allow special variables. You can calculate and show the values of expressions in CrossView Pro by using a variety of methods:

It is possible to display both monitored and unmonitored expressions in the Data Window. CrossView Pro updates monitored expressions after every halt in execution. Unmonitored expressions are just one-shot inspections of the expressions value. Refer to section 4.6, *CrossView Pro Windows* for a detailed description of the Data Window.

To evaluate a simple expression:



Double-click on a variable in the Source window. The result of the expression appears in the data window. Alternatively, depending on the preferences you set in the Data Display Setup dialog, the expression appears in the Evaluate Expression dialog. Select the Watch or Show button to display the result of the expression in the Data Window. Select the Evaluate button to display the result of the expression in the output field of the Evaluate Expression dialog.

To evaluate a complex expression:



Select Data | Evaluate Expression... from the menu and type in any C expression in the Evaluate Expression dialog box. Optionally select a format code. Click the Evaluate button.

CrossView Pro calculates the result and displays the value in the appropriate format. For details about expression formats see the section *Formatting Expressions* in the chapter *CrossView Pro Command Language*.



Type the expression in the Command Window.

Expressions can contain variable names as arguments. For instance, if the variable `initval` has a value of 17 and you enter:

```
initval * 2
```

CrossView Pro displays:

34

The expression can contain names of variables, constants, function calls with parameters, and so forth; anything that you can write directly at the Command Window, you can use in the Evaluate Expression dialog box. For more information on expressions and the CrossView Pro command language, refer to the section *CrossView Pro Expressions* in the *Command Language* chapter.

The Dot Operand

Using the dot shorthand “.” can save you some typing. The dot stands for the last value CrossView Pro displayed. For instance:

```
initval  
initval = 17
```

Now you can use the value 17 in another expression by typing:

```
. * 2  
34
```

The value is the result of the new expression.

Naturally, using the dot operand saves you from retyping complex expressions.

6.3.2 MONITORING EXPRESSIONS

CrossView Pro allows you to monitor any variable or expression. Monitoring means that the debugger evaluates a particular expression and displays the result each time the program stops. If you are in window mode, CrossView Pro displays the values of the monitored variables and expressions in the Data window.

Monitor Set Up

To set up a monitor you can:



Select the **Data | Evaluate Expression...** menu item or double-click on a variable in the Source Window, or click on the Watch selected source expression accelerator button to view the Expression Evaluation dialog box. From this dialog box, you can enter an expression and monitor (watch) its value in the Data Window. You can skip the Expression Evaluation dialog if you activate the **Bypass Dialog** check box in the Data Display Setup dialog.

Alternatively, click on the Show/Watch new expression accelerator button from the Data Window.



The Data Window must be open to display the result. Otherwise CrossView Pro does not monitor the expression. Therefore, CrossView Pro opens the Data Window automatically if you give a Show/Watch expression command.



Type the **m expression** command in the Command Window.

To place the variable `initval` in the Data window type:

```
m initval
```

`initval` remains in the Data window. You may run the program, step through it, and the display updates continually. Even if you are not in window mode, CrossView Pro still displays the value of `initval` after every CrossView command.

FUNCTION: Monitor an expression or variable.

COMMAND: **m expression**

Similarly, if you want twice the value of `initval` you could type:

```
m initval*2
```

And the expression `initval*2` is monitored.



Monitored expressions are evaluated exactly as if you had typed them in from the command line; therefore, if you are monitoring a variable, say `R`, identical to an CrossView Pro command, use the `/n` format, in this example `R/n`.

Monitor Delete

To remove a monitored expression you can:



Select the item in the Data Window and click on the Delete selected item accelerator button from the Data Window, or select Data | Delete | Item.

Selecting Data | Delete | All will remove all expressions from the Data Window.



Type the ***number m d*** command in the Command Window.

To remove `initval` from your Data Window #1, type the number of the expression (first item of the Data Window has number 0) and ***m d*** (monitor delete):

0 m d

and CrossView Pro removes `initval` (in this case, assuming it is the first variable listed in the window) from the Data Window.

FUNCTION: Remove an expression from the Data Window

COMMAND: ***number m d***

Since local variables have no meaning beyond their range, CrossView Pro issues error messages if you try to evaluate local variables beyond their scope. Some variables also become invisible when the program call another function. For instance, if you are in `main()`, monitoring `sum`, and `main()` calls `factorial()`, the unqualified name `sum` is no longer visible inside `factorial()`. You can get around this problem, however, by monitoring `main#sum` instead.

6.3.3 **FORMATTING DATA**

When you display a particular variable, CrossView Pro displays it in the format the symbolic debug information defines for it. You may, however, easily specify another format using dialogues or keyboard commands. See the section *Formatting Expressions* in the chapter *CrossView Pro Command Language*.

Examples

To print the value of `initval` in hexadecimal format, enter

initval/x

Be sure not to confuse CrossView Pro format codes with C character codes. CrossView Pro uses a `/` (forward slash) not a `\` (backward slash).

Don't worry about trying to memorize the list, you probably won't have occasion to use all these formats. Notice, however, that the `/t` format code give information about a particular value. For instance, if you wanted to find out what the type of `initval` is, type:

```
initval/t
global long initval
```

You can also take more low-level actions, such as finding out which function contains the hexadecimal address `0x100`.

```
0x100/P
main
```

CrossView Pro tells you that address `0x100` is in the function `main()`.

6.3.4 DISPLAYING MEMORY

CrossView Pro supports several methods to display memory contents. The Memory Window provides a very user-friendly yet powerful way to display the raw contents of the target memory.

Refer to section 4.6.8 for a description of the Memory Window.

Format codes also give you control over the number and size of multiple pieces of data to display beginning at a particular address. The debugger accepts format codes in the following form:

```
[count] style [size]
```

Count is the number of times to apply the format style *style*. *Size* indicates the number of bytes to be formatted. Both *count* and *size* must be numbers, although you may use **c** (char), **s** (short), **i** (int), and **l** (long) as shorthand for *size*. Legal integer format sizes are 1, 2, and 4; legal float format sizes are 4 and 8.

For instance:

```
initval/4xs
```

displays four, hexadecimal two-byte memory locations starting at the address of `initval`.

With format codes, you may view the contents of memory addresses on the screen. For instance, to dump the contents of an absolute memory address range, you must think of the address being a pointer. To show the memory contents you use the C language indirection operator `*`. Example:

```
*0x4000/2x4
0x4000 = 0x00DB0208 0x5A055498
```

This command displays in hexadecimal two long words at memory location 0x4000 and beyond. Instead of using the size specifier in the display format, you can force the address to be a pointer to unsigned long by casting the value:

```
*(unsigned long *)0x4000/2x
0x4000 = 0x00DB0208 0x5A055498
```

To view the first four elements of the array `table` from the `demo.c` program, type:

```
table/4d2
table = 1      1      2      6
```

This command displays in decimal the first four 2-byte values beginning at the address of the array `table`.

By typing the a space followed by a carriage return you can advance and see the succeeding values in the same format:

```
[Enter]
0x11 = 24  120  720  5040
```

You may recognize that the array `table` contains the factorials for the integers 0 through 7.

Displaying memory in this way is particularly effective when you have two-dimensional arrays. In this case you can display each row by specifying the appropriate count. For instance, if `myarr` is defined as `int myarr[5][8]`:

```
myarr/8ds
```

displays the values for the eight elements in the first row of `myarr`. Typing the carriage return repeatedly then display subsequent rows in the same format.

To scroll back in memory, type the ^ (caret) sign:

```
^
0x9 = 1 1 2 6
```

FUNCTION: Display value(s) at previous memory location.

COMMAND: ^

6.3.5 DISPLAYING MEMORY ADDRESSES

The **f** command lets you specify in which notation CrossView Pro displays memory addresses. It takes the same arguments as the `printf()` function in C.

FUNCTION: Specify memory address notation.

COMMAND: **f** [*printf-style-format*]

For instance, if you wish to display all memory addresses in octal, type:

```
f "%o"
```

Now all addresses appear in octal. To return to the default hexadecimal, type:

```
f "%x"
```

Using the **f** command without an argument also returns to hexadecimal address display.

6.4 DISPLAYING DISASSEMBLED INSTRUCTIONS

To show disassembled instructions:



Select the View | Source | Disassembly menu item to open the Disassembly Source Window.



Use the /i format switch to display disassembled code in the Command Window.

By using an address and the /i format it is possible to display disassembled code at any point. Suppose you wish to see how the `factorial()` function has been compiled. One method would be to examine the instructions displayed as you single step through a program at the assembly language level. There is however a quicker method that does not require you to execute the instructions. Type:

```
factorial/10i
```

This command displays the first ten assembly language instructions of `factorial()`. Remember that in C a function's name is also its address. Thus `factorial` is the address of the function `factorial()`.

Note that CrossView Pro keeps track of variable and function names for you in the disassembled code. You can also disassemble from the current execution position by using the program counter:

```
$pc/5i
```

This command disassembles five assembly language instructions from the current execution line.

You can display disassembled code for any function:

```
main#56/7i
```

disassembles seven instructions from line 56.

See also the **ei** command for displaying disassembly in a window.

Labels in Disassembly



If you desire labels in disassembly you can:

- Select the Debug | Source Window Setup... menu item to open the Source Window Setup dialog box.
- Toggle the Symbolic disassembly check box until it is set.



If you desire labels in disassembly you can:

- Turn the `$symbols` special variable "ON" by typing the following command in the Command Window:

```
opt symbols=on
```

6.4.1 INTERMIXED SOURCE AND DISASSEMBLY

To show intermixed source and disassembly:



Select the View | Source | Source and Disassembly menu item to open the Source and Disassembly Window.



Use the `/I` format switch to display intermixed C and disassembled code in the Command Window.

The `/I` format works exactly as the `/i` format, except CrossView Pro intermixes the pseudo-assembly listing with the original C source. This feature is often helpful in displaying long portions of code.

Auto Switch between Source and Disassembly

To automatically switch between source and disassembly window depending on the presence of symbols:



Select the Debug | Source Window Setup... menu item to open the Source Window Setup dialog box.

Toggle the Show assembly when SDI is missing check box until it is set.



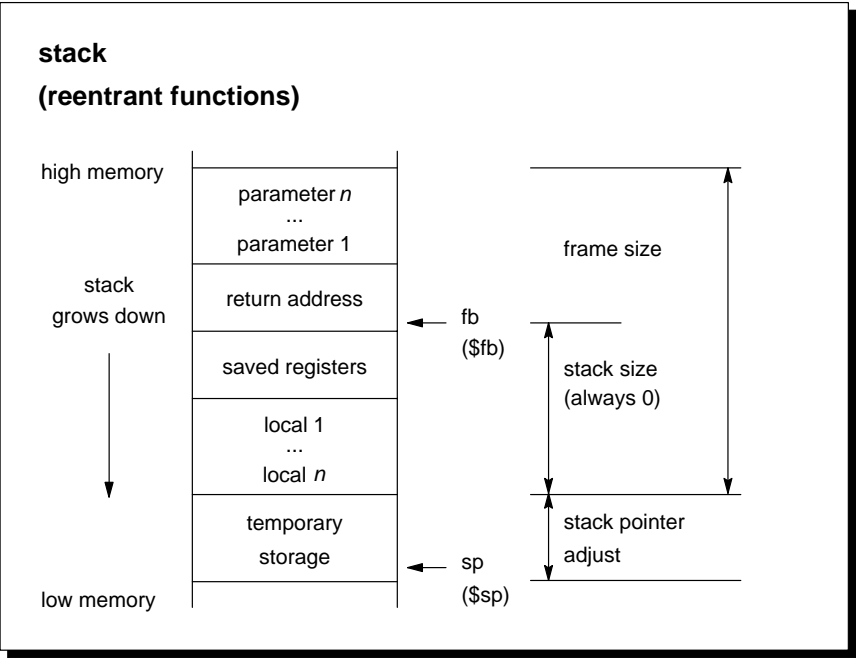
Turn the `$autosrc` special variable "ON" by typing the following command in the Command Window:

```
opt autosrc=on
```

6.5 THE STACK

During debugging, you frequently find yourself lost or unable to pinpoint your location through a series of function calls. The **stack** helps you with the problem by recording the return addresses of all functions you have passed through. CrossView Pro can use this information to reconstruct the path to your current location.

The following diagram show the structure of the stack.



6.5.1 HOW THE STACK IS ORGANIZED

The **stack** saves the return addresses of functions, non-register automatic and parameter variables of reentrant functions.

The stack is defined in the locator description file (*.dsc in directory etc) with the keyword **stack**, which results in a segment called **stack**. The description file tells **lcm16** to allocate the stack after all other data segments and the heap.

The stack size can be controlled with the keyword `length=size` in the description file. If you do not specify the stack size, the locator will allocate the rest of the available memory for the stack, as done in the startup code. You can use the locator defined labels `__lc_bs` and `__lc_es` in your application to retrieve the beginning and end address of the stack. Please note that the locator will only allocate a stack segment if the application refers to one of the locator defined symbols `__lc_bs` or `__lc_es`. Remember that there must be enough space allocated for the stack, which grows downwards.

Automatics and parameters are all accessed using the stack pointer register. The stack pointer `sp` points to the last item pushed on the stack.

The stack frame also contains the frame base register (`fb`). The virtual frame pointer points to the lower byte of the function's return address. In case of an `_interrupt` function `fb` points to PSWL. No on-chip register is allocated to serve as frame pointer. It is the debuggers task to calculate the virtual frame pointer for a function. All stack offsets in the debug info are relative to this virtual frame pointer. To be able to access automatic variables, the debugger needs to know two offsets, the stack size and the stack pointer adjust.

The stack size is passed as a function constant by the compiler. The stack size is always 0 (zero), because stack pointer adjust information is also generated in the function prologue. The stack pointer adjust reflects the number of pushes/pops done since the functions prologue.

Be aware that an interrupt function pushes both the PC and the current value of the PSW on the stack.

6.5.2 THE STACK WINDOW

The Stack Window shows the current contents of the stack after the program has been stopped. This window helps you assess program execution and allows you to view program values. You can also set breakpoints for different stack levels from this window, as described in the chapter *Breakpoints and Assertions*.

The Stack Window displays the following information for each stack level:

- The name of the function that was called
- All parameters specified to the function

- The line number in the source code from which the function was called

Each stack level shown in the Stack Window is displayed with its level number first. The levels are numbered sequentially from zero. That is, the lowest/last pushed level in the function call graph is always assigned zero.

When you first see stack information, the lowest level appears against a darker background than the other lines in the window. The marked line in the Stack Window is the **selected stack level**, meaning that this line is selected for window operations. You can change the selected stack level by clicking on a different line.

Checking the Stack from the Command Window

The stack information is also accessible from the Command Window with the **t** and **T** commands. The **t** command reconstructs the program's calling path. For instance, if you stepped into the function `factorial()` and issue a **t** (trace) command:

t

CrossView Pro displays:

```
0 factorial(num=0) [demo.c:105]
1 main() [demo.c:59]
```

The numbers to the left indicate the depth of each function on the stack. The function at the zero stack level is your current function. CrossView Pro tells you the line number where the function was called (`[demo.c:line_nr]`) and the value of the argument passed (`(num=value)`). With this information it is fairly easy to reconstruct your calling path, and see what parameter values your functions have received.

FUNCTION: Trace stack to reconstruct program's calling path.

COMMAND: **t**

There is a slight variation on the **t** command called the **T** command. The two are identical, except that the **T** command also displays the local variables for each function. For instance:

```
T
0 factorial(num=0)    [demo.c:105]
    locvar = 'x'
1 main()              [demo.c:59]
    loopvar = 0
    sum = 0
    cvar = '\xff'
```

FUNCTION: Trace stack and display local variables.

COMMAND: **T**

6.5.3 LISTING LOCALS AND PARAMETERS OF A FUNCTION

As mentioned in the previous section, CrossView Pro displays all parameters of a function. You can view the local variables and parameters of any single function active on the stack. To do this:



Follow these steps:

- Open up the Expression Evaluation dialog box by clicking on the Show/Watch new expression accelerator button from the toolbar or selecting the Data | Evaluate Expression... menu item.
- Click on the Browse... button.



In the Command Window, use the **l** (lowercase L) command.

For example, assuming you are still in `factorial()`, issue an **l** command:

```
l factorial
num = 0
locvar = 'x'
```

You can accomplish the same task by specifying the stack depth instead of a function name:

```
1 0
```

6.5.4 LOW-LEVEL VIEWING THE STACK

You can directly view the contents of the stack. Although CrossView Pro provides several high level methods of tracing functions on the stack, you can view its contents directly with the frame pointer special variable, `$fp`. For instance, the command:

```
$fp[0]/4xb
```

displays the four one-byte values in hexadecimal to which the frame pointer points. Notice that the stack frame is not really an array, but by pretending it is, you can display the memory much as you did with the `table` array. Refer to the *Accessing Variables* section in this chapter for more information.

6.6 TRACE WINDOW



C level trace is not available for all execution environments. Please check the Addendum for details.

The Trace Window displays the most recently executed lines of code each time program execution stops. CrossView Pro automatically updates the Trace Window each time execution halts, as long as the window is open.

For each executed line of code, the Trace Window displays:

- The name of the source file
- The name of the function
- The line number and corresponding source code
- The window shows all the code executed since the the last time the program halted.

6.6.1 TRACE WINDOW SETUP

The Trace Window’s only function is to display the contents of the emulator’s/ simulator’s trace buffer. The only operation you can perform in this window that directly affects the contents is to set the maximum number of instructions in the display.



To set the displaying limit, select the Options | Initialization... menu item to view the Initialization dialog box. You can change the maximum number of C-Trace machine instructions to fetch from the execution environment’s trace buffer and the maximum number of trace output lines in the Trace Window.



To view the most recently executed source statements from the Command Window, use the **ct** command preceded by the number of machine instructions you want to list. For example, to view the last source lines corresponding to the last ten machine instructions, enter:

10 ct

FUNCTION:	Display in the Command window the most recently executed C statements.
COMMAND:	<i>number</i> ct

To activate the source level trace window:



Select the View | Trace | Source Level menu item to view the Trace Source Window.



You can view the last machine instructions executed with the **ct i** command. For example:

15 ct i

displays the last 15 machine instructions in disassembled form in the Command Window.

FUNCTION: Display the most recently executed machine instructions.

COMMAND: *number* **ct i**

To activate the instruction level trace window:



Select the View | Trace | Instruction Level menu item to view the Trace Instructions Window.



You can view a raw trace with the **ct r** command. For example:

20 ct r

displays the last 20 trace frames in the Command Window.

FUNCTION: Display a raw trace.

COMMAND: *number* **ct r**

To activate the raw trace window:



Select the View | Trace | Raw menu item to view the Trace Raw Window.

6.7 REGISTER WINDOW

The Registers Window shows you the values of internal registers on your target processor.

You can create multiple Register Windows and each Registers Window contains the names and contents of all currently selected registers in the selected register set definition. Values are displayed in hexadecimal format. As long as the window is open, the debugger automatically updates the values when the program stops.



To show the list of current registers and their contents in the Command Window, enter the list registers command (**lr**).

CrossView Pro also supplies the following special variables:

\$sp stack pointer
\$pc program counter
\$fp current frame pointer

for all targets. For more information, refer to the *Command Language* chapter.

6.7.1 REGISTER WINDOW SETUP

You can configure which register set definition with which (and in which order) registers must be displayed in the Register Window; using the Debug | Register Window Setup... menu item. Since you can have more than one Register Window, the last active Register Window will be configured when you select this menu item.



To configure a Register Window follow these steps:

- Select a Register Window.
- Select the Debug | Register Window Setup... menu item to view the Register Window Setup dialog box.

The dialog will show the active register set definition and the list of available and selected registers for this particular register set definition.

- You can create a new register set definition by entering an unique register set definition name in the Name edit field and using the Add button.
- You can delete a register set definition by selecting an item from the defined register set definition list and using the Delete button. Note that when you delete a register set definition, any Register Window displaying a deleted register set will be closed.
- You can select a register set definition by selecting an item from the defined register set definition list. The list of available and selected registers will be updated according to the configuration of the selected register set definition.

Once you have selected a register set definition, follow these steps to configure this register set definition:

- You can add registers to the list of selected registers by selecting registers from the list of available registers by highlighting those registers in the left list box and using the Add->, Add All button or by double-clicking on the register you want to add.
- You can remove registers from the list of selected registers by highlighting those registers in the right list box and using the <-Remove, Rm All button or by double-clicking on the register you want to remove.
- By using the Move Up and Move Down buttons you can change the display order of the selected registers in the Register Window.

CrossView Pro automatically updates all Register Windows and places the registers in each Register Window starting at the top-left position on one line, wrapping to the next line if the next register does not fit.

6.7.2 EDITING REGISTERS

CrossView Pro lets you change the contents of registers in a simple and direct manner.



Follow these steps:

- In the Register Window, click on the register value you wish to edit. In-situ editing will be activated.
- Specify the new value in the edit control and hit the Enter key.

If the edited value is not acceptable, the debugger will emit an error message and reset the old value.

When in-situ editing is active, you can use the **Tab** key to move the edit field to the next register value or use the **Shift+Tab** key combination to move the edit control to the previous register. Use the **Esc** key to cancel in-situ editing. When a register is not in view the contents of the Register Window will be updated automatically.



You can enter any expression in the Registers Window.

Registers which can be edited symbolically have a special marker just before the register name. You can click on this marker to activate the Assign Register Symbolically dialog.



To access registers from the Command Window, use the **\$** designation and the register name in the format:

`$register = value`

CHAPTER 7

BREAKPOINTS AND ASSERTIONS



TASKING



7 | CHAPTER

You can use breakpoints to stop program execution at specified locations and return control to the user. An assertion is a number of statements executed by the debugger each time the target executes a program line. Use assertions to track down bugs, the cause of which is very hard to find.

7.1 INTRODUCTION TO BREAKPOINTS

Breakpoints halt program execution and return control to you. There are two types of breakpoints, code and data. A code breakpoint halts the program on a particular statement or instruction; a data breakpoint stops the program when a particular memory address (or range of addresses) is accessed. Data breakpoints are not available for all execution environments, please check the Addendum.

7.1.1 CODE BREAKPOINTS

A **code breakpoint** is set on a line in the code and makes the program halt exactly before that line executes. When you define a code breakpoint, you can include three elements:

- A *count*, which is the number of times the breakpoint must be encountered before it stops the program (default is 1).
- A *reset count*, which is the value assigned to the *count* after the program has stopped on a breakpoint (default is 1).
- A list of commands, which will be executed when the program hits the breakpoint.

In the Source Window, a green colored toggle shows that no breakpoint is set. A red colored toggle shows that a breakpoint is installed. An orange colored toggle indicates an installed but disabled breakpoint. If coverage is enabled, coverage markers are present to the right of the breakpoint toggles. An executed line is marked and not executed lines are not marked.

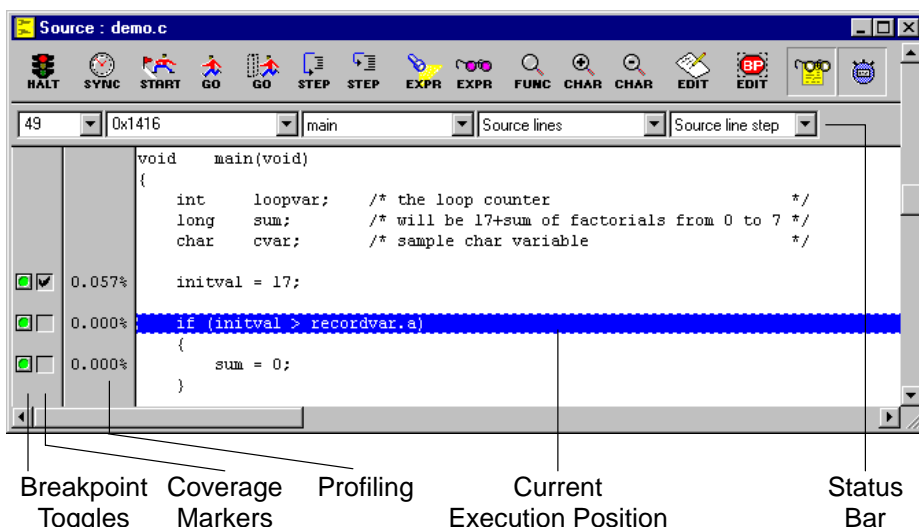


Figure 7-1: Code Breakpoint

Permanent/Temporary Code Breakpoints

Code breakpoints can be: *permanent* or *temporary*. A permanent breakpoint exists until explicitly deleted. A temporary breakpoint only exists until it stops the program once.

How CrossView Pro Sets Code Breakpoints

CrossView Pro depends on the symbol table for information about how machine instructions map to lines of source. In general, the C compiler issues line symbols at the start of each statement or line, whichever comes first. This can lead to some surprising results. If you look carefully, you can tell on which line CrossView Pro set the breakpoint, since CrossView Pro tells you on which line the program stopped, a line that may be different from the one you expected. To find out what happens if you install a code breakpoint, use single stepping and watch the order in which the source lines print out.

Multiple Statements on a Single Source Line

If you frequently include multiple statements on a single line in your source code, you may have difficulties setting code breakpoints at certain locations. For instance, suppose you have a source line containing:

```
a = 0; b = 1
```

Suppose you want to halt execution after the assignment to `a` and before the one to `b`. A normal code breakpoint does not work here, because execution stops at the first instruction of the source line. CrossView Pro provides you with the capability of disassembling the code and inserting breakpoints at the machine level. You can use the Assembly Source Window or the Intermixed Source Window to spot the right location.

For more information on machine level breakpoints, see below.

Setting Breakpoints for Multi-line Statements

Code breakpoints have a special behavior for multiple-line statements, such as a multiple-line `if`. In an `if` clause, a line symbol is generated at the beginning of the list of conditions, and the other lines of the conditions are generally associated with the first line of the clause. In an `if-then-else` construct, the `}` character before the `else` is associated with the branch-around to the end of the statement.

Consider the following example:

```
22: if ((a == b)&&
23:  (c == d)) {
24:   x = 2;
25: } else {
26:   y = 3;
27: }
```

If you try to set a code breakpoint at line 23, CrossView Pro sets the breakpoint on the preceding statement. If you try to set a breakpoint on line 22, CrossView Pro highlights line 23. If you set a breakpoint on line 25, it hits after the assignment to `x`, but before the jump to line 27. Notice that it is not hit unless the `if` clause is true. In other words, a breakpoint on line 25 is really a break on the `}`, not on the `else {`. The same behavior applies when the `else {` statement is on the next source line.

Breakpoints and For Loops and While Loops

The code generated for a C `for` statement has three parts: the initialization; the body of the loop; and the increment, test, and branch. The initialization part and the increment, test, and branch are different parts of code, but are both associated with the `for` statement itself. For example consider:

```
99: for (i = 0; i < 9; i++) {
100:   myfunction(i);
101: }
```

A breakpoint placed on line 99 will only be hit once, because it is hit at the initialization code. The code for the increment, test, and branch is associated with line 101, not 99, as you might expect.

The same applies to 'while' loops.

Breakpoints and Emulator Mode

Upon entering emulator mode, the debugger removes any breakpoints it established in the target code. Removing breakpoints ensures that you can access unmodified target code. When emulator mode ends, CrossView Pro reestablishes breakpoints as necessary.

As long as you avoid the debugger's own breakpoint trap, you may establish arbitrary breakpoint conditions while in emulator mode. These will not be removed by CrossView Pro and thus remain active, however, after you exit emulator mode. If one of these breakpoints is hit during normal debugging, CrossView Pro will issue a message such as:

```
Stopped on breakpoint not set by debugger.
```

System Startup Code

It is possible (for example, by using the **si** command) to debug system level startup code that initializes the target environment. You should not use any global variables in CrossView Pro expressions until the data area has been initialized. CrossView Pro assertions and other CrossView Pro commands that examine C variables may deliver erroneous information or cause memory access errors if used before the C environment is established.

7.1.2 DATA BREAKPOINTS

A **data breakpoint** instructs the execution environment to watch a particular data address or address range and halt execution if the program reads from or writes to that address. Data breakpoints are a powerful feature for tracking the use, and possible misuse, of pointers, global variables and memory mapped I/O ports.



Data breakpoints are not available for all execution environments, please check the Addendum.

When setting a data breakpoint, you can specify whether the breakpoint stops the program when data is read from, written to, or both.

Data breakpoints are implemented in hardware. As a consequence, the number of allowable data breakpoints is limited by your execution environment. Refer to the environment-specific Addendum for more information.

You may set a data breakpoint on a local variable, but only if the local variable is active. CrossView Pro notifies you when program execution passes beyond a local variable's scope, and a breakpoint set on such a variable deletes automatically. Data breakpoints for static variables do not have this restriction.

Note that any local variables placed in registers cannot be tracked with data breakpoints. In this case, you must use an assertion. Refer to the *Assertions* section later in this chapter for more information.

7.1.3 LISTING BREAKPOINTS

To see a listing of all of the currently defined breakpoints:



Select the **Debug | Breakpoints...** menu item from the menu bar to view the Breakpoints dialog box.



In the Command Window, enter the **lb** or **B** commands. The list appears in the Command Window.

For example entering the **B** command can result in:

```
B  
0 CODE main:59 count: 1: 1
```

The breakpoint's number (used when deleting breakpoints) is listed first, then its type: **CODE** for code breakpoints, **READ** for read data breakpoints, **WRITE** for write data breakpoints and **READ & WRITE** for read and write (sometimes called memory access) data breakpoints. Next, CrossView Pro lists the function and line number, its count and reset count, and finally any attached commands enclosed by **<** and **>**.

FUNCTION: View all breakpoints in the Command window.

COMMAND: **B**

CrossView Pro decrements the count each time the breakpoint is hit. When the breakpoint's count reaches 0, CrossView Pro halts the program.

7.2 SETTING BREAKPOINTS

You may set a code or data breakpoint by:

- Using the mouse to open the Breakpoints dialog box.
- Using the mouse in the Source Window.
- Using the Stack Window.
- Using the command line in the Command Window.

When you set a new breakpoint using the mouse, without using the Breakpoint dialog box, the type is always permanent, the count 1 and the location corresponds to the current viewing position, if the Source Window is open. These variables are described in more detail below.

Setting Breakpoints from the Menu

To set a code or data breakpoint from the menu, select the `Debug | Breakpoints...` menu item to view the Breakpoints dialog box. From this dialog box, you have the option of defining either type of breakpoint.

Setting Breakpoints from the Source Window

You can set or remove a code breakpoint directly from the Source Window by clicking on:

- The breakpoint toggle at the left side of the text in the Source Window.

To set data breakpoints use the menu as described above.

Setting Breakpoints from the Stack Window

See the section *Up-level Breakpoints* later in this chapter.

Setting Breakpoints from the Command Window

You can set a code breakpoint from the Command Window using the **b** command, and set a data breakpoint using the **bd** command.

When setting a code breakpoint, you may specify a line number, followed by the **b** command and any commands you want to attach to the breakpoint. For example, to set a code breakpoint at line 51 in your source, enter:

```
51 b
```

If you do not specify a line number, a breakpoint will be set at the current viewing position.

FUNCTION: Set a code breakpoint.
 COMMAND: *[line_number]* **b** *[commands]*

To set a data breakpoint, you must specify an address, followed by the **bd** command and any commands you want to attach to the breakpoint. There are three types of data breakpoints:

- A data read breakpoint to see if a variable is read from (**bd r** command)
- A data write breakpoint to watch if a variable is written to (**bd w** command)
- A data read or write breakpoint to check if a variable is either read from or written to (**bd b** command)

For example, to set a data breakpoint to watch the lowest byte in memory of the global variable `initval`, enter:

```
&initval bd w
```

This command instructs CrossView Pro to set a data breakpoint that will halt execution if the program writes to the lowest byte in memory of the variable `initval`. Note that you have to specify the variable's address.

FUNCTION: Set a data read breakpoint.
 COMMAND: *address* **bd r** *commands*

FUNCTION: Set a data write breakpoint.
 COMMAND: *address* **bd w** *commands*

FUNCTION: Set a data read and write breakpoint.

COMMAND: *address* **bd b** *commands*

7.2.1 DATA BREAKPOINTS OVER A RANGE OF ADDRESSES

You can also use data breakpoints to watch a contiguous range of memory. As with standard data breakpoints, data breakpoints over a range of addresses can be set to watch for reading, writing or both. To set a data breakpoint of this type:



Using mouse and menu:

- Click on the Debug | Breakpoints... menu item to open the Breakpoints dialog box.
- Select the data breakpoint you want to edit and click on the Edit... button.
- Alternatively, click on the New Data... button to define a new data breakpoint.
- Set the Read check box or the Write check box or both.
- Specify an address range by entering a start address and an end address.



From the Command Window:

- Type **bd r** to set a data read breakpoint over a range.
- Type **bd w** to set a data write breakpoint over a range.
- Type **bd b** to set a data breakpoint for both reading and writing over a range.

For example, to ensure that the program stops if any of `recordvar`'s fields are either written to or read from:

```
&recordvar bd b (int) \
    &recordvar+sizeof(recordvar)-1
```

FUNCTION: Set a data read breakpoint over a range of addresses.

COMMAND: *address* **bd r** *address commands*

FUNCTION: Set a data write breakpoint over a range of addresses.

COMMAND: *address* **bD w** *address commands*

FUNCTION: Set a read and write breakpoint over a range of addresses.

COMMAND: *address* **bD b** *address commands*

7.2.2 TEMPORARY BREAKPOINTS

Code breakpoints can be: **permanent** or **temporary**. A breakpoint exists until it is manually deleted. A temporary breakpoint is automatically removed by CrossView Pro after it halts the program once.

To set a temporary breakpoint:



Follow these steps:

- Open the Source Window by selecting the View | Source | Source lines menu item.
- Open the Breakpoints dialog by selecting the Debug | Breakpoints... menu item.
- Click on the New Code... button to open the Code Breakpoint dialog box. The values displayed in the Location group box correspond to the current viewing position if the Source Window is open.
- Select the Temporary option in the Type field when defining or editing a breakpoint in the Code Breakpoint dialog box.
- Click on the Run button in the Source Window when the program halts. This sets a temporary breakpoint at the viewing position and the program starts again.
- Alternatively, scroll to the line that you want to stop at and click once (to establish a viewing position). Select Run | Run to Cursor menu item to continue execution until you reach this temporary breakpoint.



Type the **C** command followed by a line number in the Command Window.

C 51

sets a temporary breakpoint at line 51 and resumes execution at the current execution position.

7.2.3 SETTING THE COUNT

CrossView Pro allows you to set a breakpoint's **count**. The count defines how many times you encounter the breakpoint before it halts the program. For example, a breakpoint with a count of 3 means the program stops on the third hit. Each time the breakpoint is hit, CrossView Pro decrements the count. When the count reaches 0, CrossView Pro halts the program, and resets the count to the value of the **reset count**. The default reset count is 1.

To set a breakpoint's count,



Follow these steps:

- Click on the Debug | Breakpoints... menu item to view the Breakpoints dialog box.
- When you define or edit a code breakpoint from this box, you can set the breakpoint's count.
- Select one of the radio buttons Reset to 1 or Reset to Value.



From the Command Window, use the **bc** command.

For example, suppose you have a breakpoint set at line 59 of your source code. The first time the program halts at line 59, enter:

bc 3 4

This command sets the breakpoint's count to 3 and the reset count to 4. You can observe a breakpoint's current count and reset count when you list the breakpoints in the Command Window with the **lb** command.

FUNCTION: Set the count and reset count for a breakpoint.

COMMAND: *[breakpoint_number]* **bc** *[count]* *[reset_count]*

7.3 DELETING BREAKPOINTS

You can delete a breakpoint directly from the source code, using the menu items, or through the Command Window. To see a list of active breakpoints, click on the `Debug | Breakpoints...` menu item or use the **1 b** command in the Command Window.

To delete a code breakpoint:



Click on the corresponding red breakpoint toggle at the left side of the text in the Source Window. This deletes the code breakpoint and the breakpoint toggle will be green.



You can also follow these steps:

- Select the `Debug | Breakpoints...` menu item to view the Breakpoints dialog box. This box contains a delete function.
- Select the Breakpoint.
- Click the Delete button.
- Leave the dialog by clicking on the OK button.



Use the *breakpoint_number* **d** command in the Command Window. You need to know the breakpoint's number for this command.

For example, to delete the breakpoint numbered 1, enter:

```
1 d
```

FUNCTION: Delete a breakpoint.

COMMAND: *breakpoint_number* **d**

To clear all the breakpoints in the program, type:

```
D
```

```
Do you want to delete all breakpoints?y
```

FUNCTION: Delete all breakpoints.

COMMAND: **D**

7.4 ENABLING/DISABLING BREAKPOINTS

You can enable or disable a breakpoint directly from the source code, using the menu items, or through the Command Window. To see a list of active breakpoints, click on the `Debug | Breakpoints...` menu item or use the **1b** command in the Command Window.

To enable or disable a code breakpoint:



Follow these steps:

- Select the `Debug | Breakpoints...` menu item to view the Breakpoints dialog box. This box contains an edit function.
- Select the Breakpoint.
- Click the `Edit` button.
- Check the `Enabled` check box to enable or uncheck it to disable a breakpoint.
- Leave the dialog by clicking on the `OK` button.



Use the *breakpoint_number* **bena** command or *breakpoint_number* **bdis** command in the Command Window to enable or disable a breakpoint. You need to know the breakpoint's number for these commands.

For example, to disable the breakpoint numbered 1, enter:

```
1 bdis
```

FUNCTION: Disable a breakpoint.

COMMAND: *breakpoint_number* **bdis**

To enable the breakpoint numbered 1, enter:

```
1 bena
```

FUNCTION: Enable a breakpoint.

COMMAND: *breakpoint_number* **bena**

7.5 BREAKPOINT COMMANDS

CrossView Pro allows you to attach commands to code and data breakpoints. When execution halts at a breakpoint, CrossView Pro executes the commands. Valid commands are almost any C statements and CrossView Pro commands, giving you a very powerful tool for manipulating a debugging session. To do this:



Follow these steps:

- Open up the Breakpoints dialog by selecting the Debug | Breakpoints... menu item.
- Select an existing breakpoint and edit it by clicking on the Edit... button.
- Alternatively, click on the New Code... button or the New Data... button.
- Click on the Advanced >> button in the resulting dialog box. Note that the advanced part of this dialog will be visible when there is already a command defined.
- Click in the Command edit area.
- Type in the commands to be executed when the breakpoint is reached.



You do not need to enclose a group of commands in brackets. However, each command must be delimited by a semicolon.

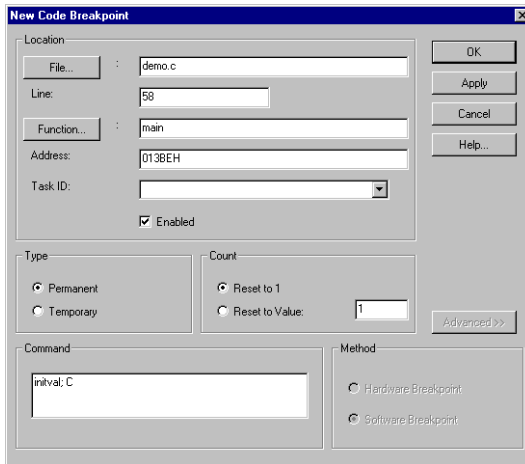


Figure 7-2: Breakpoint Commands



Type the commands, enclosed in brackets and delimited by semicolons, after the **b** command in the Command Window.

For instance, suppose you want a program to stop at a breakpoint, display a variable's value, and resume execution all in one stroke. To perform this function, you need to attach the appropriate commands to a breakpoint. Enter:

```
51 b {initval;C}
```

This places a breakpoint at line 51. When execution stops at the breakpoint, CrossView Pro displays the value of `initval` and immediately resumes execution.

You can attach almost any valid CrossView Pro commands or C statement to breakpoints. This latitude allows you to use breakpoints in powerful ways. Later on you find out how breakpoints can create patches in your program.



CrossView Pro does not check the syntax of attached commands until the breakpoint is hit.

Data breakpoints accept command lists the same way as code breakpoints. For instance, to set a data breakpoint that monitors the lowest byte in memory of the value of `initval`, enter:

```
&initval bd w {initval; C}
```

Every time the program writes to the lowest byte in memory of the variable `initval`, this breakpoint halts the program, prints the value of `initval` and continues execution.



For more information on the use of attached commands, see the *Patches* and *Diagnostic Output and Statistical Information* sections later in this chapter.

7.5.1 ATTACHING CONDITIONALS TO A BREAKPOINT

You can pass standard C conditionals to a breakpoint.

For example:

```
63 b {if (initval==17) {C} {initval/n}}
```

stops the program at line 63, checks to make sure the variable `initval` is 17, and resumes execution if it is. If `initval`'s value does not equal 17, CrossView Pro prints the value, and the program remains halted.

7.5.2 ATTACHING MACROS TO A BREAKPOINT

You can attach any currently defined macro to a breakpoint in a command list. For example, suppose you define a macro named `rg` that checks the value of the variable `initval`. The command to define this macro is:

```
set rg "if (initval != 17) {initval/n} {C}"
```

If the value does not equal 17, the macro prints the value and halts the program. Otherwise, execution continues.

You can include this macro at any point by attaching it to a breakpoint. Entering:

```
51 b {rg}  
63 b {rg}
```

this is a very efficient way to insert the macro with breakpoints at lines 51 and 63.



For more information on macros, refer to *Defining and Using Macros* chapter.

7.5.3 ATTACHING STRINGS TO A BREAKPOINT

You can attach strings to a breakpoint's command list. This feature is useful for placing comments and reminders within your breakpoints. Attaching a string to a breakpoint also eliminates the need for diagnostic `printf()` statements in your compiled code.

For example, you could place a breakpoint on line 49 such as:

```
49 b {"Passed line 47\n";C}
```

Whenever the breakpoint on line 49 is hit, CrossView Pro prints the string and continues execution.

7.6 SUPPRESSING BREAKPOINT MESSAGES

Whenever a breakpoint is hit, CrossView Pro displays in the Command Window, the name of the function, line number and file in which the breakpoint appears. You can suppress this information by setting breakpoint “silent” mode. In the silent mode, the current location is not printed out.

To set silent mode you can use the **Q** (for quiet) command as part of the command attached to a breakpoint definition.

Pass the **Q** command to a breakpoint first. For example:

```
51 b {Q; initval = 5}
```

stops the program on line 51, but does not print a message stating where the break occurred.

7.7 LOW-LEVEL BREAKPOINTS

CrossView Pro allows you to place breakpoints at individual machine instructions.



When you define a new breakpoint, specify a particular address in the Address field in the Code Breakpoint dialog box.



In the Command Window, enter the *address* **bi** or *address* **BI** command. The **bi** command sets a permanent breakpoint. The **BI** command sets a temporary breakpoint.

If you do not specify an address, CrossView Pro uses the current viewing position. For example:

factorial bi

instructs the debugger to place a breakpoint at the first machine instruction in the function `factorial()`.

When you use hexadecimal addresses, be sure the number is actually the start of a machine instruction because, in most implementations, the actual opcode at the breakpoint address is replaced by a target-dependent value that ultimately causes the breakpoint to occur. If this value appears in the middle of an instruction, results are unpredictable.

FUNCTION: Set a instruction level breakpoint

COMMAND: [*address*] **bi** [*commands*]

7.8 UP-LEVEL BREAKPOINTS

Up-level breakpoints are breakpoints set at the entrance and/or exit of functions. Basically, up-level breakpoints are code breakpoints that are directly connected to the current HLL stack handling.

To see the current HLL stack, open the Stack Window or enter the **t** command in the Command Window.

You can set up-level breakpoints via the Stack Window or in the Command Window. You cannot set up-level breakpoints in the Source Window:



Double-click on the function in the Stack Window to install a stack breakpoint after the function call.



You can also follow these steps:

- Click on the function in the Stack Window.
- Select either Debug | Stack Breakpoint | After Function Call or Debug | Stack Breakpoint | At Function Entry from the menu.

You have the option of setting the breakpoint before (function entry) or after (up-level) a selected function.



All breakpoints set through the Stack Window are temporary by default. This can be toggled using the **permanent** radio button in the Code Breakpoint dialog box. Using the **bU** and **bB** commands (explained below), you can also set temporary breakpoints at the beginning and end of functions.



In the Command Window, use the following commands:

Command	Function	Type
bU	Sets breakpoint <i>after</i> return of function	temporary
bu	Sets breakpoint <i>after</i> return of function	permanent
bB	Sets breakpoint at <i>beginning</i> of function	temporary
bb	Sets breakpoint at <i>beginning</i> of function	permanent

For example, suppose you have accidentally single-stepped into a function called `factorial()`. If you do not want to single step through the function, an up-level breakpoint can help you. Enter:

bU

The **bU** command sets a temporary breakpoint after return of the function. Now, instead of having to single step all the way through the function, you can start continuous execution, which stops when it hits the new breakpoint at the function's return. Note that it makes no difference whether the function has several possible points of return; the up-level breakpoint works at all points of return. Note that when the function that contains the breakpoint is called from one of the functions that are located below it on the stack, the execution may be stopped before returning at the desired stack level, for example with recursive functions.

When setting up-level breakpoints from the Command Window, you can specify how deep in the stack the function's address is located. For example, if you are two functions down from the `main()` program, enter:

2 bU

This command breaks when you return to the top level of the call graph.

FUNCTION: Set a temporary breakpoint after function call.

COMMAND: `[stack] bU [commands]`

FUNCTION: Set a permanent breakpoint after function call.

COMMAND: `[stack] bu [commands]`

FUNCTION: Set a temporary breakpoint at function entry.

COMMAND: `[stack] bB [commands]`

FUNCTION: Set a permanent breakpoint at function entry.

COMMAND: [*stack*] **bb** [*commands*]

7.9 PATCHES

A **patch** is a means of using CrossView Pro to change the execution of your program without recompiling. Patches involve manipulating breakpoints to skip code, include code, or replace existing code with new code.

Basically, a patch is a breakpoint with certain associated commands that enable you to alter program execution. This capability is a useful debugging tool.

You can associate the commands used to patch code with a breakpoint through either the Command Window or through the Commands edit box in the Breakpoint dialog box. The examples below set breakpoints using CrossView Pro commands typed in the Command Window. Breakpoints can also be set using the Debug | Breakpoints... menu item. In this case the commands between the brackets are entered into the Command edit area.

7.9.1 PATCHING CODE OUT OF A PROGRAM

To patch code out of a program, you can set a breakpoint that changes the execution position. For instance, suppose you want to patch an infinite loop out of your source.

```
78:  while (loopvar)
79:  {
80:      sum = sum + 1;
81:  }
82:
83:  sum = sum + 5;
```

On line 78, place a breakpoint that jumps to line 83, effectively bypassing the loop. In the Command Window, enter:

```
78 b {g 83; C}
```

This creates a breakpoint on line 78 that does nothing more than move the execution position beyond the loop and issue a **C** command. Remember that the breakpoint on line 78 is hit before the **C** statement on that line executes.

7.9.2 PATCHING CODE INTO A PROGRAM

You can also patch code into a program by just including the code in the breakpoint command. For example, suppose you want to add an equation with the variable `loopvar`.

```
78: while (loopvar)
79: {
80:     sum = sum + 1;
81: }
82:
83: sum = sum + 5;
```

In the Command Window, enter:

```
78 b {loopvar = 0;C}
```

This command halts execution at line 78, adds the statement `loopvar=0` to the program, and continues execution.

7.9.3 REPLACING CODE IN A PROGRAM

Finally, you can combine the two techniques described above to replace code in a program. For instance, suppose you want to replace an infinite loop with new code.

```
78: while (loopvar)
79: {
80:     sum = sum + 1;
81: }
82:
83: sum = sum + 5;
```

In the Command Window, enter:

```
78 b {Q; if (sum<100) {sum++; g 78; C} {g 83; C}}
```


This command sets a breakpoint that halts execution (quietly) at line 78 and inserts an `if` statement into the program. If `sum` is less than 100, `sum` increments and line 78 executes again. If `sum` equals 100, CrossView Pro moves the execution position to line 83 (beyond the infinite loop) and resumes execution.

7.10 DIAGNOSTIC OUTPUT AND STATISTICAL INFORMATION

Breakpoints with attached commands allow you to report on various variables while the program executes. In the past, one inefficient method of tracking variables was to litter code with `printf()` statements. Using breakpoints makes that process unnecessary.

For instance, suppose you want to keep track of the variable `loopvar` at line 59 of a program. Install a breakpoint with the following command:

```
59 b {Q; loopvar; C}
```

The breakpoint halts the program, prints the value of `loopvar`, and resumes execution. The **Q** command suppresses the listing of where the break occurred. This breakpoint does not affect the source code and no recompilation is necessary.

Using special variables, you can also keep statistics about your program, such as how many times a line of code executes or how many times a variable is accessed.

For example, suppose you want to know how many times line 60 executes. You must define a special variable to keep track of your statistical data, and set a breakpoint to accumulate the data for you.

First, define the special variable. In the Command Window, enter:

```
$test = 0
```

This command defines the special variable `$test` and sets it to zero. For convenience, you can also set a breakpoint at the beginning of the program that initializes `$test`.

Secondly, set a breakpoint at line 60 that increments `$test` and continues execution every time the program hits line 60:

```
60 b {$test++ ; C}
```

7.11 ASSERTIONS

An **assertion** is a collection of debugger commands executed by the debugger after each program line. When you execute a program using assertions, the debugger is in **assertion mode**. Running the debugger in assertion mode is a way of executing continuous control of certain data.

Using assertions, you can have continuous control of certain data and stop program execution if any of the set conditions are fulfilled. In this respect, assertions are similar to data breakpoints. Assertions, however, are more versatile than data breakpoints. For instance, a data breakpoint can only detect when a variable is accessed. An assertion, on the other hand, can check that the variable's value falls within a certain range. Also, an assertion can monitor variables whose values are kept in registers.

The default limit for the number of assertions you can define is 16. It is possible to increase the number of assertions by selecting the **Options | Initialization...** menu item. Each individual assertion can be activated or deactivated. In addition, you can also choose to suppress all assertions by turning off the global assertion mode.

Opening the Assertions Dialog Box



Select **Debug | Assertions...** menu item.

The Assertions dialog box contains scrollable lists of all defined assertions, and provides functions for defining, activating, suspending, editing and deleting assertions.

7.11.1 GLOBAL ASSERTION MODE

The debugger is running in assertion mode when there is at least one active assertion. A program executing in assertion mode is actually being single-stepped very quickly, to ignore breakpoints. Because the program is single-stepping, however, it runs significantly slower than at normal speed.

A global assertion mode is available that suppresses all assertions, regardless on whether they are marked as activated or deactivated. To set the global active assertion mode:



Open the Assertions dialog box and globally activate all assertions by clicking on the **Assertion Mode Active** button.



In the Command Window, enter the **A** command:

- **A a** — activates assertion mode
- **A s** — suspends assertion mode
- **A** — (by itself) toggles the assertion mechanism

The Global Active state activates all assertions. Globally activating the assertion mode, however, does not change how each assertion is marked.

FUNCTION: Activate assertion mechanism.

COMMAND: **A a**

FUNCTION: Suspend assertion mechanism.

COMMAND: **A s**

FUNCTION: Toggle assertion mechanism.

COMMAND: **A**

7.11.2 DEFINING AN ASSERTION

To define or edit an assertion:



Follow these steps:

- Click on the **Debug | Assertions...** menu item to view the Assertions dialog box.
- Click on the **New...** button to open up a text edit dialog box as shown in figure 7-3 to type in commands.

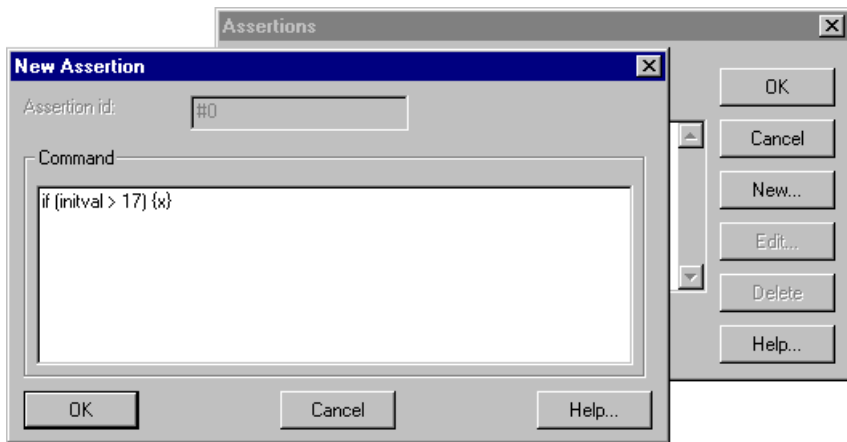


Figure 7-3: Defining Assertions



Use the **a** command followed by a list of commands.

FUNCTION: Create an assertion.
 COMMAND: **a** *commands*

Assertions accept standard C statements and certain CrossView Pro commands as arguments.

An assertion usually contains a conditional. For example, suppose you want to create an assertion that watches the value of the global variable `initval` to see that its value does not exceed a certain limit. In this case, you enter in the Assertion dialog box (or into the Command Window after the **a** command):

```
if (initval > 17) {x}
```

This command creates an assertion with the condition that if `initval` exceeds 17, CrossView Pro halts the program. The `{x}` is a special assertion command that tells CrossView Pro to halt the program and return control to you.

7.11.3 EDITING AN ASSERTION

To edit the contents of an assertion:



Follow these steps:

- Click on the `Debug | Assertions...` menu item to view the Assertions dialog box.
- Click on the assertion to edit.
- Click on the `Edit...` button. A text edit dialog box opens allowing you to edit the assertion. Click on `OK` or `Cancel` when finished.



You must delete the specific assertion (section 7.11.5) and define a new assertion (previous section) with the desired command.

7.11.4 ACTIVATING AND SUSPENDING ASSERTIONS

A particular assertion is either **active** or **suspended**. A suspended assertion does not execute before every line, but it retains its definition.

You may find it helpful to use activate and suspend assertion commands in conjunction with code breakpoints, since assertions tend to slow the target program. By attaching commands to a breakpoint to activate and suspend assertions, you can turn assertions on only for certain sections of code where a particular value needs checking. This method can dramatically speed up the program.



Open up the Assertions dialog box from the `Debug` menu and double-click on the assertion's number.



To activate or suspend an assertion from the Command Window, you must know the assertion's number. To see a list of assertions and their assigned numbers:

- Enter `!a`, the list assertions command, in the Command Window.

To activate an assertion:

- Enter `assertion_number a a` command. For example:

```
2 a a      activates assertion 2
```

To suspend an assertion:

- Enter the *assertion_number* **a s** command. For example:

2 a s *suspends assertion 2*

FUNCTION: Activate an assertion.

COMMAND: *assertion_number* **a a**

FUNCTION: Suspend an assertion.

COMMAND: *assertion_number* **a s**

7.11.5 DELETING ASSERTIONS

Deleting an assertion removes its definition. It is important to note the difference between suspending an assertion and deleting an assertion: deleting an assertion removes its definition for good, while suspending it retains the definition but prevents its execution.



Follow these steps:

- Open the Assertions dialog box from the Debug menu.
- Click on the assertion that you want to delete.
- Click the Delete button.



Follow these steps:

- List the assertion numbers with **l a** command in the Command Window.
- In the Command Window, enter the assertion number followed by the **a d** command. For example:

2 a d *Deletes assertion 2.*

FUNCTION: Delete an assertion.

COMMAND: *assertion_number* **a d**

7.11.6 USING ASSERTIONS

You can use assertions for almost any type of debugging task. For example, if you want to check the value of a global variable, `global_val`, during the execution of a certain function, `f()`. A data breakpoint or a straightforward CrossView Pro assertion does not suffice for this task since there is no way to make either method limited to that function's code range. The solution lies in creating an assertion that is active only over a specific range of lines. In this case, you could solve your problem with the following steps:

```

110: void f(void)
111: {
112:     if ( global_flag )
113:     {
114:         ++global_val;
115:     }
116:     else
117:     {
118:         global_val = g();
119:     }
120: }
```



Using the mouse and menu:

1. Open up the Assertions dialog box from the Source Window.
2. Click on the New... button.
3. Set up the assertion to check the value of `global_val`. Enter:

```
if (global_val == 17) {x}
```

This assertion halts program execution if the value of `global_val` equals 17.

4. Open up the Breakpoints dialog box by selecting **Debug | Breakpoints...** from the menu. Click the **New Code...** button.
5. We want to establish a breakpoint at line 112, the first line of the function `f()` and attach commands to the breakpoint to activate assertion mode and continue execution. Change the **Line** number to 112. Click in the **Command** edit area and enter:

```
A a; C
```

Activate the assertion and continue.

6. Create an assertion whose only function is to check that the current line number is still valid for assertion mode. To do this, use the reserved special variable `$LINE`, which contains the line number of the current execution position. In the Assertions dialog box, click on `New...` and enter:

```
if ($LINE >= 120) {A s; 1 x; C}
```

If the line number exceeds 120, the program is about to leave the function `f()` and CrossView Pro deactivates assertion mode. Normally, the **x** command would make the program stop, but the non-zero value tells CrossView Pro to execute the rest of the commands in the list, in this case, **C** for continue.



You must enter all commands in the Command Window.

1. First set up the assertion you want:

```
a if (global_val == 17) {x}
```

2. Now set a breakpoint on the first line of the function `factorial()` that will activate assertion mode, and continue execution:

```
110 b {A a; C}
```

3. Now create an assertion that does nothing but make sure that the current line number is still valid for assertion mode. If the line number exceeds 120, you know you have left the function `f()` and assertion mode should be suspended.

```
a if ($LINE >= 120) {A s; 1 x; C}
```

`$LINE` is a reserved special variable that CrossView Pro maintains containing the number of the line currently executing. If it becomes equal to 120, assertion mode is turned off. Normally, the **x** would make the program stop, but the non-zero value 1 tells CrossView Pro to execute the rest of the commands in the list, in this case, **C** for continue.

In this manner you have created an assertion that is only active over a limited range of source lines.

7.11.7 GATHERING STATISTICS WITH ASSERTIONS

You can also use assertions to gather statistics about your code. For instance, you can find out how many lines of C code execute in a particular session:

```
a {$numlines++}
```

`$numlines` is a user-defined special variable that increments on each line of C code. When the program stops, type:

```
$numlines
```

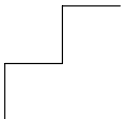
and CrossView Pro gives the result. To start again, you may want to re-initialize `$numlines` to zero:

```
$numlines = 0
```

Or just set a breakpoint on the first line of code to do the same.

CHAPTER 8

DEFINING AND USING MACROS



8

CHAPTER

8.1 CROSSVIEW PRO MACROS

A **macro** is a user-created shorthand for any sequence of CrossView Pro or C commands and expressions. Macros allow you to debug more efficiently when using CrossView Pro by substituting a short string for a longer combination of words and evaluators.

You can use a macro anywhere an CrossView Pro or C expression is valid: in a breakpoint's command list, with assertions, from the keyboard, among other places. CrossView Pro also allows you to save macro definitions, so they are always available. By passing parameters to a macro, you can create powerful and flexible macros to debug your code more efficiently.

You can use macros in the Command Window, or connect them to the graphic interface in a feature called the **toolbox**. You can have this toolbox visible as a CrossView Pro window and use it to execute a macro by clicking a button. You control which macros have corresponding buttons, making the toolbox easy to adapt to different situations.

8.2 DEFINING MACROS

You can create as many macros as you want:



Select the Options | Macro Definitions... menu item to view the Macro Definitions dialog box and click on the New... button.

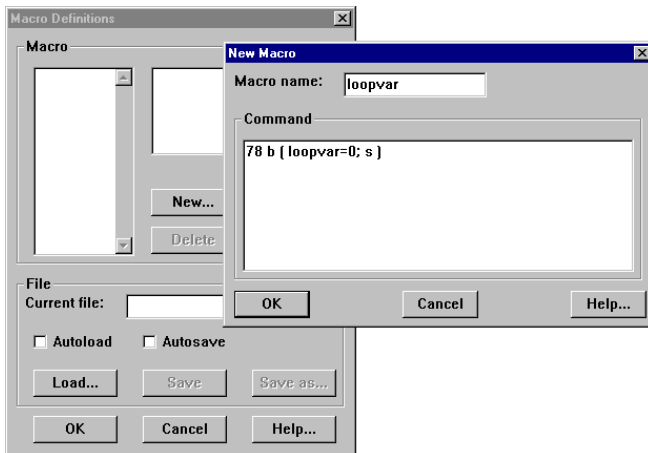


Figure 8-1: Macro Definitions



In the Command Window, use the **set** command followed by the macro's invocation name and the list of commands. Note that the list of commands must be in (double) quotation marks. For example, the command:

```
set st "e main; R"
```

creates a macro call **st** that tells CrossView Pro to change the viewing position to be the first executable line in the function `main()` and restart the program from the beginning. Each time you enter **st** in the Command Window, CrossView Pro substitutes the lengthier list of commands in the definition.

FUNCTION:	Create a macro.
COMMAND:	set <i>name</i> "commands"

Note that there is no rule that the macro definition must be shorter than the commands it represents. For instance, you could substitute **break** for the **b** command, to make CrossView Pro's command language more expressive:

```
set break "b"
```

Now instead of typing **74 b** to set a breakpoint, you can also type:

```
74 break
```

Macros defined using either the command line or the graphic interface are accessible both from the Command Window and the Toolbox.

Macros may call other macros, so it is possible to use simple macros as building blocks for more complex functionality. No macro, however, can call itself, or another macro that refers to the calling macro, since this type of action results in infinite recursion.



Because of the order in which CrossView Pro parses statements, you may not use the CrossView Pro commands **#** or **%** in a macro.

8.2.1 LISTING MACROS



Open up the Macro Definitions dialog box by selecting the Options | Macro Definitions... menu item. This dialog box contains a scrollable list of the macros.

To see the current definition of a macro:



Follow these steps:

- Open up the Macro Definitions dialog box by selecting the Options | Macro Definitions... menu item.
- Click on the macro that you want to see.
- Much of the macro is shown in the box in the center of the dialog. If you need to see more, click on the Edit... button.



Type the **echo name** command in the Command Window. For instance, to see the definition for the `st` macro:

```
echo st      Command.
e main; C 56 Output.
```

FUNCTION: Display macro expansion.

COMMAND: **echo name**

8.2.2 REDEFINING A MACRO

If you want to change the definition of a macro:



Open up the Macro Definitions dialog box by selecting the Options | Macro Definitions... menu item. Click on the name of the macro you wish to change and click on the Edit... button.



In the Command Window, use the **set** command again, but enter an exclamation point after the macro name. For instance, to redefine the macro `st`, which was defined in the example above, use the command:

```
set st! "e main; C 56"
```

Now, the `st` macro changes the viewing position and restarts program execution, placing a temporary breakpoint at line 56. Be sure you do not include a space before the exclamation point. Otherwise, CrossView Pro may interpret the `!` as the C “not” operator.

8.2.3 SAVING MACRO DEFINITIONS TO A FILE

You can save all the macros you define in a debugging session in an external file. This way, you do not lose the definitions when the program ends.

To save macros to an external file:



Follow these steps:

- Open up the Macro Definitions dialog box by selecting the `Options | Macro Definitions...` menu item.
- Click on the `Save as...` button. A Save Macro File dialog box opens.
- If you want to save a file previously opened, click on the `Save` button. This saves the file without opening the Save Macro File dialog box.
- Alternatively, you can use the `Autosave` check box. When `Autosave` is checked, all macros are saved in the ‘current file’ when you leave CrossView Pro.



Type the **`save file`** command in the Command Window. This command saves your macros to the file of your choice. For instance:

`save macro.mac` *writes all your macros to macro.mac*

FUNCTION: Save macros to a file.

COMMAND: **`save filename`**

8.2.4 LOADING MACRO DEFINITIONS FROM A FILE

You can load saved macros anytime you want to re-use a definition. There is no limit to the number of times you can load macros.

To load a macro file:



Follow these steps:

- Open up the Macro Definitions dialog box by selecting the Options | Macro Definitions... menu item.
- Click on the Load... button and select the macro file you want to load.
- Alternatively, you can use the Autoload check box. When Autoload is checked, the macros saved in the 'current file' are loaded at startup.



To reinstate your macro definitions from the Command Window, use:

< filename.mac

You must load a program before you can read a macro definition file. Autoload will be ignored when the Execute these settings at CrossView startup check box in the Load Symbolic Debug Info dialog box is not checked.



For more information on record and playback functions, see the next chapter, *Command Recording & Playback*.

8.2.5 DELETING MACROS

To delete a specific macro:



Follow these steps:

- Select the Options | Macro Definitions... menu item to view the Macro Definitions dialog box.
- Highlight the name of the macro.
- Click on the Delete button. To delete all the macro definitions at the same time, click on the Delete all button. CrossView Pro prompts you for confirmation.



Type the **unset** command in the Command Window. For example, to remove the `st` macro, enter:

```
unset st!
```

When you are removing a macro definition in this manner, you must place an exclamation point after the macro name to prevent CrossView Pro from expanding the name to its full macro definition. To update your macro definition files, issue a **save** command after using **unset**.



You can remove *all* existing macro definitions by entering the **unset** command by itself. CrossView Pro prompts you for confirmation before deleting the macros:

```
unset
```

```
Do you want to delete all macros?y
```

FUNCTION: Delete a macro.

COMMAND: **unset** *name!*

8.3 MACRO PARAMETERS

Macros can accept arguments. Parameters are labelled sequentially in a macro definition: \$1, \$2, \$3, etc. Note that \$0 has no meaning. When you invoke a macro with parameters, enclose the parameters with parentheses and separate them with commas.

CrossView Pro macros can accept any number of parameters, so it is possible to create very complex command shortcuts. You may use any type of parameter when defining a macro, including integers, strings, or addresses. Note, however, that you must pass the macro the correct type at invocation.

For instance, suppose you want to set a detailed breakpoint on any number of lines and a parameter is to specify each line number on which to install a breakpoint. Defining a macro named `brk`, type in the Macro Definitions dialog box:

```
$1 b {Q; initval; recordvar.a; if (initval > 1) {C}}
```

or type in the Command Window:

```
set brk "$1 b {Q; initval; recordvar.a; if (initval > 1) {C}}"
```

In this case, the argument \$1 represents the intended line number. To use the `brk` macro, type:

```
brk(72)     From the Command Window
```

CrossView Pro replaces every instance of \$1 with the value 72. For this example, that means a breakpoint is set at line 72.

8.4 REDEFINING EXISTING CROSSVIEW PRO COMMANDS

Using macros, you can even redefine an existing CrossView Pro command.

For instance, you could redefine the breakpoint command **b** to always place a breakpoint at line 72 of your source code. To do this, enter the command:

```
set b "72 b!"
```

CrossView Pro now interprets the **b** command as **72 b**.

The exclamation point in the definition is necessary to prevent infinite recursion. It tells CrossView Pro to take the command literally and to not expand it into a macro definition. For example:

```
66 b!
```

CrossView Pro interprets this command as the standard breakpoint command and places a breakpoint at line 66, despite the macro definition for **b**.



Be sure not to have any space between the command and the exclamation point. Otherwise CrossView Pro may interpret the **!** as the C *not* operator.

8.5 USING THE TOOLBOX

The CrossView Pro toolbox, shown in figure 8-2, is controlled from the View menu. Using the Options menu, you can configure the toolbox and define the macros for it. You can resize the toolbox to the size you want.

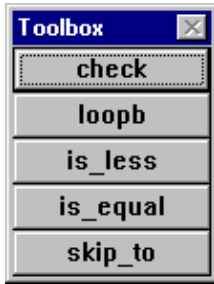


Figure 8-2: CrossView Pro Toolbox

8.5.1 OPENING THE TOOLBOX

To open the toolbox:



Select the View | Toolbox menu item.



The Toolbox is a pop-up window that remains on top of the CrossView Pro Desktop while you work in other windows.

8.5.2 CONNECTING MACROS TO THE TOOLBOX

To configure the toolbox, select the Options | Toolbox Setup... menu item to view the Toolbox Setup dialog box, shown in figure 8-3. This dialog box displays the toolbox buttons and an alphabetized list of the current macro definitions.

To connect a macro to a toolbox button:



Follow these steps:

- Click on the button you wish to change
- Scroll through the macro list to highlight the desired function
- Click on the Assign button or press the Enter key

Note that double clicking on the macro name in the alphabetized list performs the third step automatically. The name of the new function appears on the selected button and the connection is performed.

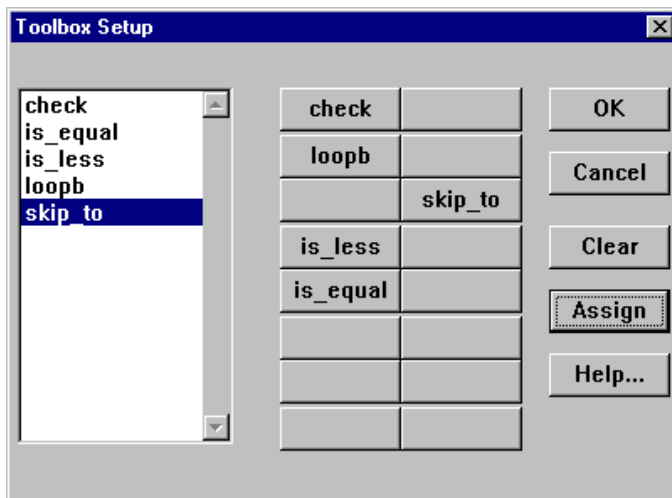


Figure 8-3: Setting Up the Toolbox



Do not assign parameterized macros to the toolbox since there is no way to pass in parameter values.

8.5.3 REMOVING A MACRO CONNECTION

To delete a macro definition from the toolbox:



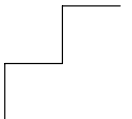
Follow these steps:

- Select the Options | Toolbox Setup... menu item to view the Toolbox Setup dialog box.
- Select the desired button.
- Click on Clear.

This deletes the macro definition from the toolbox.

CHAPTER 9

COMMAND RECORDING & PLAYBACK



9

CHAPTER

9.1 RECORDING COMMANDS

CrossView Pro lets you save a series of CrossView Pro commands to the file of your choice. This is **record mode**. You can re-load a saved file to repeat parts of debugging tasks or replay a debugging session (up to the point where you left the last time).

Record mode means that all CrossView Pro commands from the keyboard, mouse or menu are recorded to a disk file. The debugger can read this file and execute the commands as if they were entered into the Command Window. This is called **playback mode**, see more about playback mode later in this chapter.



Record and playback modes can never be active at the same time.

You can only record CrossView Pro commands. When recording on CrossView Pro command level, all commands that you type in the Command Window, as well as the CrossView Pro command language equivalents of dialog actions and menu selections are saved in a file. When you want to record commands entered in the Emulator Command Window, you have to turn on CrossView-Emulator I/O logging (see the section *Logging* in this chapter).

From the Command Window you control record mode using either the mouse or keyboard commands. To start or setup recording:



From the menu system:

- Select the **Options | Record...** menu item.

The Record dialog box contains an **Automatically at CrossView startup** check box. If you select this check box the debugger enters record mode at every startup.

- Enter the name of the file in the **Filename:** edit field, or click on the **Browse...** button to select an existing file. The default filename extension is **.cmd**.
- Click on the **Setup** button to save the current settings into the initialization file *xvw.ini* for following debugging sessions.
- Click on the **Start** button to start recording.



Enter the **>** command with the name of the file to start recording. For example, enter:

```
>session.cmd
```


After you invoke this command, CrossView Pro saves every executed command, whether using the mouse or manually typed into the Command Window, to the file `session.cmd`.

FUNCTION: Save CrossView Pro commands to a file.

COMMAND: **>filename**

9.1.1 ENTERING COMMENTS

Every command, whether typed into the Command Window or the result of a mouse or menu action goes into the recording file. To add comments to a file recording CrossView Pro commands, enclose text typed in the Command Window with C comments delimiters, “/*” and “*/”. When logging emulator commands, refer to your emulator documentation for the appropriate comment characters.

9.1.2 SUSPEND RECORDING

This function acts like the pause button on a tape recorder: the recording mechanism stays in place, but suspends temporarily. CrossView Pro does not save to file any commands you enter while you suspend recording, but the file remains open and ready to accept input. To suspend recording:



Select the Options | Record... menu item to view the Record dialog box. Click on the Suspend button.



In the Command Window, use the **>f** command (for “false”).

FUNCTION: Suspend recording.

COMMAND: **>f**

9.1.3 RESUME RECORDING

This function is the counterpart of the suspend recording function. CrossView Pro resumes adding commands to the current record file. Any new command you enter appears in the file; they do not affect the commands already saved.



Select the Options | Record... menu item to view the Record dialog box. Click on the Resume button to resume recording.



In the Command Window, use the **>t** command (for “true”).

FUNCTION: Resume command recording.

COMMAND: **>t**

9.1.4 CHECK RECORDING STATUS

If at any point you do not remember whether recording is on or off, check by:



Selecting the Options | Record... menu item. If record mode is active, the Stop button is enabled. If the Start and Setup buttons are enabled, record mode is off.



Enter the **>** command in the Command Window.

This command shows the status of the recording and logging mechanism. For example, if you enter **>** you might see:

```
>
Output logging is OFF
Command recording is ON
Target communication logging is OFF
```

The **>** command gives you the status for the different recording mechanisms. Output logging and target communication logging are described below.

9.1.5 CLOSE FILE FOR RECORDING

Closing a file for recording differs from suspending recording in that when you close a file, you may not add any more commands to it. If you were to start recording again using the same filename, the old commands in the file would be deleted. (Note that this does not exclude editing the file manually by some other means, since the file is saved as ASCII text.)



Select the Options | Record... menu item to view the Record dialog box. Click on the Stop button to stop recording.



Enter the **>c** command to close the file:

```
>c
```

FUNCTION: Close command recording file.

COMMAND: **>c**

9.1.6 COMMAND RECORDING EXAMPLE

For example, consider the following command sequence (from the Command Window):

```
>session.cmd          ----- Start Recording to File
initval
p 12
>f                    ----- Carriage Return
l b                   ----- Suspend Recording
sum
>t                    ----- Resume Recording
/* This is a comment! */
>c
```

This series starts with a command to record to a file named `session.cmd`. The blank line above represents a carriage return. After the last command, **c**, if you were to view this file, it contains:

```
initval
p 12
/* This is a comment! */
```

The saved command file contains simply the commands, without any output. Note that commands entered while recording was suspended (**lb** and **sum**) do not appear in the file. Carriage returns are not recognized as commands.

9.2 PLAYING BACK COMMAND FILES

Once you have recorded a set of CrossView Pro commands, you can play them back to recreate a debugging session or repeat often-used sequences. Running the debugger while reading commands from a file is **playback mode**.



Remember that for a file to be played back, it can only contain CrossView Pro or emulator commands. For this reason, screen output files cannot be used in playback mode. Refer to the *Recording Commands* section earlier in this chapter for more information.

As with recording, the Command Window controls playback mode. To playback a command file:



Follow these steps:

1. Select the **Options | Playback | CrossView...** menu item to view the **CrossView Playback** dialog box, or select the **Options | Playback | Emulator...** menu item to view the **Emulator Playback** dialog box.



You can choose to playback either CrossView Pro commands or Emulator commands. Open the **Emulator Command Window** if the playback file contains commands sent directly to your emulator.

2. Type the playback filename or use the **Browse...** button to select the file. The default filename extension is **.cmd**.



In the Playback dialog box, you have two additional options: **Playback at XVW startup** and **Continuous playback**. CrossView Pro enters playback mode automatically when you start the debugger if you click on the **Playback at XVW startup** check box in the Playback dialog box. The entire playback file executes if you check the **Continuous playback** check box.

3. Click on the **Execute** button to start the playback.



In the Command Window, use the `<` or `<<filename` command to playback CrossView Pro commands.



On the command line of CrossView Pro give the option `-T filename` to start CrossView Pro in transparency mode and playback emulator commands. Not available for all execution environments.

9.2.1 SETTING THE TYPE OF PLAYBACK



Using the mouse, you can toggle continuous playback by clicking on the Continuous playback check box in the CrossView Playback dialog box.



In the Command Window, there are two commands for the type of playback. The `<filename` command starts playback. Commands are read from a file and executed without any stop. For example:

`<session.cmd` *load and execute all the commands*



The `<<` command causes CrossView Pro to playback commands one at a time, similar to single-stepping through code. For example:

`<<session.cmd` *read a command from the file.*

Clicking the Execute button or pressing the Enter key executes the next command.

FUNCTION:	Play back a file of CrossView Pro commands.
COMMAND:	<code><filename</code>
FUNCTION:	Play back a file of CrossView Pro commands, one command at a time.
COMMAND:	<code><<filename</code>

9.2.2 CALLING OTHER PLAYBACK FILES

A playback file can call another playback file in the course of its execution.

When CrossView Pro creates a command file, it saves all commands in their textual form, whether entered by the mouse or as text. You must edit this file to use the `<` and `<<` commands.

When the debugger reaches a `<` or `<<` command in a playback file, playback execution switches to the new file, but does not return to the original file. In other words, you chain playback files but not nest them.

9.2.3 QUITTING PLAYBACK MODE

Playback mode stops automatically when CrossView Pro reaches the end of the command file. If you want to end playback mode before this point, click the `HalT` button.

9.3 COMMAND LINE BATCH PROCESSING

CrossView Pro supports command line batch file processing, but CrossView Pro will halt if a modal dialog is encountered or if the target program contains an endless loop. The command line option **--timeout=*n*_seconds** switches CrossView Pro to a different mode of operation, without the two drawbacks mentioned above.

In order to process files in batch mode you have to do the following:

1. Create a temporary directory.
2. Start CrossView Pro from this temporary directory. For Windows 95/98/NT/2000 you can create a separate icon or shortcut to start CrossView Pro, which has the working directory (Start in:) set to the temporary directory.
3. Close all CrossView Pro windows except the Command Window.
4. Exit CrossView Pro (with Save Desktop enabled).

You now have generated an `xvw.ini` file with minimal GUI overhead.

5. Save the `xvw.ini` file and remove the temporary directory.

For each batch run of CrossView Pro you have to do the following:

1. Create a temporary directory.
2. Copy the saved `xvw.ini` file to the temporary directory.
3. Create a command file in the temporary directory.

The following command file `session.cmd` loads the `.abs` file, downloads the code, runs the code and exits.

N hello.abs	<i>load the symbols</i>
dn	<i>download the program</i>
__exit bi	<i>set a breakpoint at the exit point</i>
R	<i>run the program</i>
\$pc	<i>optional: show the program counter</i>
q y	<i>exit CrossView Pro</i>

where `hello.c` contains

```
#include <stdio.h>

void main()
{
    printf("Hello World!\n");
}
```

4. Copy the `.abs` file to the temporary directory. This is needed because CrossView Pro changes its working directory when the **N** command is used.
5. The following line executes CrossView Pro in batch mode and waits for it to finish:

Windows 95/98/NT/2000:

```
start /wait c:\cml6\bin\xfwml6 --timeout=120 -tcfg sim.cfg
-p session.cmd -R session.log
```

UNIX:

```
xfwml6 --timeout=120 -tcfg sim.cfg -p session.cmd -R session.log
```

This command must be issued in the temporary directory! After the execution has ended, the file `session.log` contains a transcript of the commands.

6. Save the output files and clean up (or remove) the temporary directory. This must be done because the `xvw.ini` file has been modified now. If CrossView Pro would be started again in the temporary directory, the file `session.cmd` would be executed again.

The **--timeout=*n*_seconds** command activates the batch operation mode of CrossView Pro. It causes CrossView Pro to terminate when the specified amount of time has elapsed, which is crucial in batch processing: if a program does not terminate, the timeout will terminate CrossView Pro, so that the next program in the batch can be executed. CrossView Pro will also terminate in the batch mode if a modal dialog pops up, since this requires user interaction to continue. Before CrossView Pro exits, the text in the dialog will be written to the log file. A special case of this dialog is the 'End of program reached' dialog. For this reason, the line `has to be added to the .cmd file`, so it is possible to do some things (for example, read registers modified by a machine code program) after the program is finished. If the breakpoint at `is absent`, CrossView Pro immediately exits

after having executed the **R** command, so any consecutive commands will be ignored.

9.4 LOGGING

Logging means that all output text to a particular window is saved in a file for later use. Two windows allow logging:

- Command Output Window
(upper part of the CrossView Command Window)
- Emulator Output Window
(upper part of the Emulator Command Window)

“GDI Accesses” can also be logged. This is the information transferred between CrossView Pro and the Debug Instrument (DI).

You can control logging from the Options menu or from the Command Window.

You can also determine the status of each logging function:



Select the Options | Log | Command Input/ Output... menu item, the Options | Log | CrossView-Emulator I/O... menu item or the Options | Log | CrossView-GDI Accesses... menu item.

If a logging function is active, the Stop button is enabled. If the Start and Setup buttons are enabled, logging is off.



Enter the **>>** or **>&** command in the Command Window.

Each type of logging is described in the following section.



The Emulator Output Window is primarily a diagnostic tool. It should be used wisely, since it generates substantial amounts of output, the format of which is emulator dependent. For emulators that have an ASCII interface, the actual command/response dialogue will be displayed. For emulators with a binary interface, CrossView Pro will generate a record of function calls with their associated input and output parameters. This also applies to the GDI Accesses output logging.

9.4.1 SETTING UP LOGGING

To setup logging:



From the menu system:

- Select Options | Log | Command Input/ Output..., Options | Log | CrossView- Emulator I/O... or Options | Log | CrossView-GDI Accesses... to open the appropriate dialog box.
- Type in the name of the log file or use the Browse... button to select a filename. The default filename extension is .log.

Each Log dialog box has an Autostart check box. This check box instructs CrossView Pro to start recording the output of a particular window upon starting up of CrossView Pro.

- Click on the Setup button to save the current settings into the initialization file *xvw.ini* for following debugging sessions.
- Click on the Start button to start logging.



You can open up a log file for CrossView Command Output by using the **>>filename** command as in:

>>screen.log



You can open up a log file for Emulator Output by using the **>&filename** command as in:

>&target.log

FUNCTION: Save CrossView Pro commands and command window output to a file.

COMMAND: **>>filename**

FUNCTION: Log target communications.

COMMAND: **>&filename**

9.4.2 RECORDING COMMANDS AND LOGGING SCREEN OUTPUT

It is possible to have command recording, command output logging and target communication logging on at the same time. That is, you can have one file recording just the CrossView Pro commands, and another file concurrently recording both the commands and the computer responses. Refer to the previous section for information on command record files.

Since the Command Window log file contains both your commands and the computer responses, you cannot use it in playback mode.

9.4.3 COMMAND WINDOW LOG FILE EXAMPLE

For example, if you entered the following commands:

```
>>screen.log
initval
l a
```

The output file, `screen.log`, contains:

```
> initval
initval = 0
> l a
no assertions
```

9.4.4 SUSPENDING AND RESUMING OUTPUT LOG

You can resume and suspend the Logging process from the menu or from the Command Window:



Select Options | Log | Command Input/ Output..., Options | Log | CrossView- Emulator I/O... or Options | Log | CrossView-GDI Accesses... to select the appropriate dialog box.

To suspend logging:



Click on the Suspend button.



In the Command Window, use the **>>f** command for suspending the logging of the Command Output Window. Type **>&f** to suspend the Emulator Output Window. After you issue this command, CrossView Pro does not save all subsequent commands and their computer responses.

To resume logging:



Click on the **Resume** button.



In the Command Window, use the **>>t** command to resume logging the Command Output Window. Type **>&t** to resume the Emulator Output Window. After you issue this command, CrossView Pro saves all subsequent commands and their computer responses.

FUNCTION: Suspend output logging (logging is false).

COMMAND: **>>f**

FUNCTION: Resume output logging (logging is true).

COMMAND: **>>t**

FUNCTION: Suspend target logging (logging is false).

COMMAND: **>&f**

FUNCTION: Resume target logging (logging is true).

COMMAND: **>&t**

9.4.5 CLOSING THE OUTPUT LOG FILE

To close the output file:



Select Options | Log | Command Input/ Output..., Options | Log | CrossView- Emulator I/O... or Options | Log | CrossView-GDI Accesses... to select the appropriate dialog box. Click on the Stop button to stop logging.



Enter the **>>c** or **>&c** command in the Command Window to close the Command Output and Emulator Output log files. These commands end the recording for the currently specified output log file.

FUNCTION:	Close output log file.
COMMAND:	>>c
FUNCTION:	Close target log file.
COMMAND:	>&c

9.5 STARTUP OPTIONS

When starting up CrossView Pro you may immediately start recording or playing back files. For instance,

```
xfwm16 fact -p session
```

plays back the commands in the file `session`. A **-P** switch single-steps through each command, prompting you for a return after each. You can also start recording:

```
xfwm16 fact -r session
```

This command records all your commands (just like the **>** command) to the file `session`, while:

```
xfwm16 fact -R session
```

logs your commands and screen output to the file `session` (just like the **>>** command).

You can also use the **Automatically at CrossView startup** option in the Record, Playback, and Log dialogs to immediately start recording, playback or logging at CrossView Pro startup.



You can also enter record and playback files via EDE. Select the **EDE | CrossView Pro Options...** menu item. Enter your filenames in the Logging tab.

9.6 CROSSVIEW PRO COMMAND HISTORY

MECHANISM

CrossView Pro stores the command history in the list box of the Command Window.

You can select a command from the history list by clicking on it or jumping with the **<Tab>** key to the history listing and using the arrow keys.. The command appears in the edit field of the Command Window. You may edit the command if you want.

To execute the command, click on the **Execute** button.

If you do not want to edit the command, double-click on the selected command in the list box to execute the command, or hit the **<Return>** key.

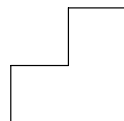
CHAPTER

10

SPECIAL FEATURES



TASKING



10

CHAPTER

10.1 TRANSPARENCY MODE

Transparency mode allows you to communicate directly with the execution environment. Most of the time CrossView Pro will handle all the low level communications, freeing you to concentrate on the high level C code. Depending on the type of execution environment, however, you may have to enter transparency mode to set up the execution environment when the machine is first turned on.

To enter transparency mode:



Select the View | Command | Emulator menu item.

All commands entered in the Emulator Command Window are passed directly to the execution environment.

To exit transparency mode:



Select the View | Command | CrossView menu item.

In CrossView Pro, you can pass a string directly to the execution environment without leaving CrossView Pro with the **o** command:

o map

This passes the command map directly to the execution environment, while you remain in CrossView Pro. Naturally you will have to learn your execution environment's command set to make use of the **o** command.

FUNCTION: Pass a command to the execution environment.

COMMAND: **o** *string*



Do not issue one-shot transparency commands that result in large output (or otherwise require intervention other than a carriage return to terminate output). Instead, enter transparency mode first, then issue the command.

You may also enter transparency mode upon startup with the **-T** option. See the section on startup options.

10.2 RTOS AWARE DEBUGGING

CrossView Pro supports RTOS (Real-Time Operating System) aware debugging for various kernels. Since each kernel is different, the RTOS aware features are not implemented in the CrossView Pro executable, but in a library that will be loaded at run-time by CrossView Pro. The amount of windows and dialogs and their contents is kernel dependent.

Within the CrossView Startup dialog (Options | Startup | CrossView) you select the CrossView configuration you will use by selecting a "target configuration file". These target configuration files are normal ASCII text files. The name of the shared library that contains the kernel aware code can be specified in the target configuration. The "radm" configuration item specifies the name of the shared library that contains the kernel aware code.

The syntax of a target configuration file is:

[! comment] field : field-value

field one of the defined keywords

field-value the value assigned to the field

comment optional comment

Empty lines, lines consisting of only white space are allowed. Comments start at an exclamation-sign (!) and end at the end of the line.

The line for the shared library that supports RTOS aware code could be:

```
radm: yourrtos.dll
```

10.3 COVERAGE



You can only use this feature if it is supported by the execution environment (see the addendum).

When the application program is executed as a result of a command such as StepInto or Continue, CrossView Pro traces all memory access, i.e. memory read, memory write and instruction fetch. Through code coverage, executed and not executed areas of the application program can be found. Areas of unexecuted code may exist in case of programming errors or simply dead code which could be eliminated. Alternatively, your program input, your test set, is incomplete. It does not cover all paths in the program. Data coverage allows you to verify which memory locations, i.e. which variables, are accessed during program execution. Additionally, stack and heap usage can be shown.

To enable/disable coverage:



Select the Run | Coverage checked menu item.

When the menu item is checked, coverage is enabled. Select the menu item again to disable coverage.



Type the **ce** or **cd** command on the command line:

ce

FUNCTION: Enable coverage.

COMMAND: **ce**

FUNCTION: Disable coverage.

COMMAND: **cd**

Two dialogs are present to give you coverage information. The code coverage dialog shows the percentage of executed code within application, module and function scope. Code coverage information can also be displayed in the Source Window. The data coverage dialog shows the data access of HLL variables in the executed program. Data coverage can also be displayed in the Memory Window. The coverage dialogs can be opened via the Debug menu.

You can display code coverage information in the Source Window by clicking on the `Display coverage` button in the Source Window. In this case an extra column appears to the right of the breakpoint toggles (to the left of the source line). For each source code line that is executed (*covered*), the source line is marked. The not executed lines are not marked. CrossView Pro has special commands to move the cursor to the next or previous covered or uncovered line:

FUNCTION:	Move cursor to next covered line.
COMMAND:	nC
FUNCTION:	Move cursor to next uncovered line.
COMMAND:	nU
FUNCTION:	Move cursor to previous covered line.
COMMAND:	pC
FUNCTION:	Move cursor to previous uncovered line.
COMMAND:	pU

You can display data coverage information in the Memory Window by clicking on the `Display coverage` button in the Memory Window. Besides the current value of memory locations, the memory window also displays whether memory locations have been accessed during program execution. An application program may read from, write to, or fetch an

instruction from a memory location. Of course all combinations may be legal. Although writing data to a memory location from which an instruction has been fetched is suspicious. All types of accesss, read, write, fetch or combinations of these, can be shown using different foreground and background colors. The color combination used to show "rwx" access are specified in the Desktop Setup dialog. It is advised to change the background color if instructions are fetched from a memory location, and to change the foreground color to show read and write access.

10.4 PROFILING



You can only use this feature if it is supported by the execution environment (see the addendum).

Profiling allows you to perform timing analysis on your software. Two forms of profiling are implemented in CrossView Pro. Both forms of profiling are fully implemented in the CrossView Pro debugger. You do not have to recompile your source code to enable the profiling features.

Function profiling, also called cumulative profiling, gives timing information about a particular function or set of functions. The time spent in functions called by the function being profiled is included in the timing results. Within the Cumulative Profiling Setup dialog you select one or more functions to be profiled. The gathered profile is shown in the Cumulative Profiling Report dialog. For each function the number of calls, the minimum/maximum/average and total time spent in the function are shown. Also, the relative amount of time consumed by a function in respect to the time consumed by the application is shown.

Function profile data is gathered whenever the program is executed using the Continue command (not single stepped). Function profiling can be supported if the execution environment provides a clock that starts and stops whenever execution starts and stops. Basically function profiling is implemented by using a special type of breakpoint. Breakpoints are inserted at the function entry address and all it's return addressed. Whenever execution stops due to a profile-breakpoint hit, CrossView Pro will read the clock, update the internal profile tables, and restart execution.

To specify the functions to be profiled:



Select the Debug | Cumulative Profiling Setup... menu item.

To view the profiling results:



Select the `Debug | Cumulative Profiling Report...` menu item.

Code range profiling presents timing information about a consecutive range of program instructions. CrossView Pro displays the time consumed by each statement, C or assembly, in the source window. The timing data can be displayed in three different formats: absolute, relative to program, and relative to function. To change the display format: position the cursor on the profile column and click the right mouse button. Select the appropriate format from the popup menu.

Next to the source window, the profile report dialog shows the time spend in each function. The time consumed by functions called from the function being profiled is not included in the displayed time.

Code range profiling data is gathered whenever the program is executed. It does not matter if the program executes due to a continue, step-over or step-into command. Code range profiling heavily relies on special profiling features in the execution environment. Normally code range profiling is only supported by instruction set simulators.

To enable/disable profiling:



Select the `Run | Profiling checked` menu item.

When the menu item is checked, code range profiling is enabled. Enabled means that the execution environment starts gathering profiling data. Select the menu item again to disable profiling.



Type the **pe** or **pd** command on the command line:

pe

FUNCTION:	Enable profiling.
COMMAND:	pe

FUNCTION:	Disable profiling.
COMMAND:	pd

Select the `Display profiling` button in the Source Window to display profile data in the Source Window. (If profiling is not enabled, clicking the accelerator button also enables the gathering of profiling data.)



Normally both function and code range profiling will slow down the execution speed of the application being debugged. Therefore, switch off profiling whenever the timing information is not required.

10.5 VIRTUAL I/O CHANNELS

The CrossView Pro Virtual I/O windows provide an interface to exchange data with the application on the target. This I/O facility can be implemented in various ways. The debugger supports up to eight separate Virtual I/O windows simultaneously.

10.5.1 ROM MONITOR

If your CrossView Pro environment is a ROM monitor, the TASKING ROM monitor supports so-called virtual I/O channels. These channels can be used by the application for reading or writing data on the same channel as the ROM monitor uses for its debug communications. The channels will however, as opposed to the regular input and output USRs, be logically separated from the ROM monitor's input and output. This way it is possible to communicate with the application, while the ROM monitor communications remain undisturbed, and debugging control over the application remains intact.

The channels are to be used in combination with CrossView Pro. CrossView Pro will turn on the new *virtchan* configuration item of the ROM monitor. This tells the ROM monitor to treat virtual channel service requests to be logically separated from the ROM monitor's own communications. This way CrossView Pro will be able to display these communications in separate windows.

When you want to run your application without CrossView Pro, then leave the *virtchan* configuration turned off. This tells the ROM monitor to treat any virtual channel service request as a regular I/O request. Now you can use a simple terminal to communicate with your application, even if it is reading and writing using the virtual channel service requests instead of using the regular service requests such as `USR_INCHR` and `USR_OUTCHR`. See the file `vio_test.c` for an example of virtual I/O.

10.5.2 KEYBOARD MAPPINGS VIRTUAL I/O

The following keyboard mappings, being both control codes and escape sequences, are supported by the VT100-like terminal mode of the virtual I/O windows:

Key	Character Sequence and/or Decimal Value
Backspace	8d
TAB	9d
DEL	127d
ESC	27d
Insert	ESC [2 ~
Prev/Page Up	ESC [5 ~
Next/Page Down	ESC [6 ~
Arrow Up	ESC [A
Arrow Right	ESC [B
Arrow Left	ESC [C
Arrow Down	ESC [D

Table 10-1: General Keyboard Mappings

Display Control Virtual I/O

The VT100-like terminal mode of the virtual I/O windows comprises the following control codes and escape sequences for displaying:

ASCII Code	Decimal Value	Operation
BELL	7	Ring the bell
BS	8	Move cursor one position back
TAB	9	Move cursor to next tab stop
LF	10	Move cursor one line down
CR	13	Move cursor to start of line
ESC	27	Start escape sequence (see below)

Table 10-2: Control Codes

Escape Sequences

Escape Sequence	Operation
<code>ESC D</code>	Cursor one line down (scrolls if already at last line)
<code>ESC E</code>	Cursor one line down and to left margin (scrolls)
<code>ESC M</code>	Cursor one line up (scrolls if already at top line)
<code>ESC [n1 A</code>	Cursor <i>n1</i> lines up
<code>ESC [n1 B</code>	Cursor <i>n1</i> characters right
<code>ESC [n1 C</code>	Cursor <i>n1</i> characters left
<code>ESC [n1 D</code>	Cursor <i>n1</i> lines down
<code>ESC [H</code>	Cursor home
<code>ESC [n1 ; n2 H</code>	Move cursor to (<i>n1</i> , <i>n2</i>) with <i>n1</i> =row, <i>n2</i> =col

Table 10-3: Cursor Motion

Parameters *n1* and/or *n2* may be left out, in which case a value of 1 is assumed.

Escape Sequence	Operation
<code>ESC [J</code>	Clear screen from cursor till bottom-right
<code>ESC [p1 J</code>	0: Clear screen from cursor till bottom-right 1: Clear screen from top-left till cursor 2: Clear entire screen
<code>ESC [K</code>	Clear line from cursor till end
<code>ESC [p1 K</code>	0: Clear line from cursor till end 1: Clear line from begin to cursor 2: Clear entire line

Table 10-4: Erasing

For example, to clear the entire screen in the C programming language, you can enter:

```
printf("\033[H\033[2J");
fflush(stdout);
```

Escape Sequence	Operation
<code>ESC [n1 @</code>	Insert characters
<code>ESC [n1 P</code>	Delete <i>n1</i> characters
<code>ESC [n1 L</code>	Insert <i>n1</i> lines
<code>ESC [n1 M</code>	Delete <i>n1</i> lines

Table 10-5: Inserting and Deleting



Parameter *n1* may be left out, in which case a value of 1 is assumed.

Escape Sequence	Operation
<code>ESC [m</code>	Turn off all attributes
<code>ESC [n1 m</code>	0: turn off all attributes 1: bold 4: underline 5: blinking 7: reverse 8: invisible 22: turn off bold 24: turn off underline 25: turn off blinking 27: turn off reverse 28: turn off invisible

Table 10-6: Character Attributes

Multiple parameters may be specified simultaneously:

```
ESC [ n1 ; ... ; nN m
```

Some attributes or combinations of attributes are mapped to a regular standout mode.

Parameters may be left out, in which case a value of 0 is assumed.

Escape Sequence	Operation
ESC [12 l	Local echo on
ESC [12 h	Local echo off
ESC [? 7 h	Wrap around on
ESC [? 7 l	Wrap around off
ESC [? 25 h	Cursor on
ESC [? 25 l	Cursor off
ESC [? 92 l	Enquire after the window's size Response: ESC [? rows, columns c

Table 10-7: Miscellaneous

10.6 SIMULATED INPUT/OUTPUT

Simulated I/O allows you to observe the input and output of your program before the hardware is in place. In CrossView Pro, simulated I/O operates through special function calls.

I/O calls in your M16C C program like `printf()` and `getchar()` call the low-level function `_iowrite()` and `_ioread()`. In the distributed C libraries these functions call `_simo()` and `_simi()`, respectively. So, all your M16C programs support simulated I/O by default. Stream number 0 is for input, for example by `getchar()`; stream number 1 is for output functions like `printf()`; stream number 2 is for error messages.

To use simulated I/O, you will need to add two special function calls to your application. The `_simi` and `_simo` routines allow CrossView Pro to trap I/O for debugging purposes. The `_simi` and `_simo` routines, found in the C library or supplied with CrossView Pro as source code in a module called `lib/src/_simio.c`, are simple stubs as shown below:

```
int _simi(int stream, char *port, int len)
{
    return len + stream + *port;
    /* names used by CrossView Pro */
}

int _simo(int stream, char *port, int len)
{
    return len + stream + *port;
    /* names used by CrossView Pro */
}
```

The parameters for this function are:

- | | |
|--------|--|
| stream | The stream number used to identify the particular I/O. |
| port | The address of the input or output buffer where your program reads or writes its input/output data (the name 'port' is used for historical reasons). |
| len | The length of the input data or size of the output data. |

Example:

```
_simo(OUTPUT_STREAM, outbuf, 80);
```

OUTPUT_STREAM is the stream number used to identify this I/O; outbuf is the address of the output buffer where your program writes its output data; and 80 is the size of the **output buffer**. Note that outbuf must always be the same physical buffer. So, copy your data into it before calling _simo.

The _simi call (for simulated input) has a similar form:

```
_simi(INPUT_STREAM, inbuf, 2)
```

INPUT_STREAM is the identifying I/O number; inbuf is the address where the input data will be received; and 2 is the size of the **input buffer**: two bytes.

If you have several I/O routines that you want to simulate, be sure to give each one a different stream number. Valid stream numbers are 0 through 7.

10.6.1 SETTING UP SIMULATED I/O

Once your code has been compiled with the appropriate _simi/_simo calls, and CrossView Pro is up and running, you have to define the simulated streams.

You can set up an input or output stream. For input you may specify either a file or the keyboard, for output either a file or the screen. Each stream has its own identifying number. There can be as many as eight streams.

You may also specify the format of the stream's values. The default is character, but you may want to use hexadecimal or octal values when directing data to or from a file.

To set up a simulated I/O stream:



From the menu system:

- Select the Debug | Simulated I/O Setup... menu item to open the Simulated I/O Setup dialog box.
- Select a stream number from the Stream Nr list selection box.
- Select the Input or Output radio button.

- Select the I/O device: either **Screen** / **Keyboard** or **Filename**. If you want file I/O enter the name of the file or use the **Browse...** button to select a file. The default filename extension is **.sio**.
- Optionally, select one of the **Format** radio buttons.
- Optionally, specify an alternative prompt for an input stream.
- Click on the **Activate** button to activate the stream.
- Specify another stream by repeating the steps above or click on the **OK** button to close this dialog box.



Enter the **sio o** or **sio i** command in the Command Window.

FUNCTION: Create an output stream.

COMMAND: *stream* **sio o** {file | **screen**} [/format]

To set up a simulated output stream, you could type:

```
1 sio o screen
```

This creates an output stream number 1 (corresponding to the `OUTPUT_STREAM #define` in the source). The output is directed to the screen. CrossView Pro automatically assigns the stream a simulated I/O window where the output appears.

To create a simulated input stream, type:

```
0 sio i screen
```

FUNCTION: Create an input stream.

COMMAND: *stream* **sio i** {file | **screen**} [/format]

This creates an input stream number 0 (corresponding to the `INPUT_STREAM #define` in the source), and directs the input from the keyboard while prompted on the screen.



Using the hexadecimal or octal format, CrossView Pro requires a space between each hexadecimal/octal value as input.

10.6.2 VIEWING CURRENT STREAM SETTINGS

To view the number and types of current streams:



Follow these steps:

- Select the **Debug | Simulated I/O Setup...** menu item to open the Simulated I/O Setup dialog box.
- If a stream in the **Stream Nr** list selection box contains an asterisk (*) before its number the stream is active. The **Memory** box shows the address and length of the currently selected stream.
- Select a stream number from the **Stream Nr** list selection box for which you want to see the current settings.



Enter the **sio** command in the Command Window.

```
sio
stream: 0 screen input format x   adr: 0x190 len: 2
stream: 1 screen output          adr: 0x140 len: 80
```

CrossView Pro shows each stream's characteristics: file or screen, input or output, its format, location in memory, and its length (defined in the `_simi` and `_simo` calls). Note that the program must run and call a simulated I/O function at least once in order for a stream to have the address and length information displayed.

FUNCTION: View simulated I/O status.

COMMAND: **sio**

10.6.3 CHANGING STREAM'S PROPERTIES

When CrossView Pro requests the simulated input from the keyboard, it displays the prompt and waits for your typed input. If you want, you may also input hexadecimal characters by changing the format of the I/O stream.



To change stream properties:

- Select the **Debug | Simulated I/O Setup...** menu item to open the Simulated I/O Setup dialog box.

- Select the stream from the `Stream Nr` list selection box for which you want to make some changes.
- Note that when a stream is active and has already been used, you must deactivate the stream before editing.
- Make the changes and click on the OK button to accept your changes.



Delete the previous stream with the **sio d** command in the Command Window and create a new stream.

For example, to change stream 0 first, delete the previous stream:

```
0 sio d
```

FUNCTION: Delete an I/O stream.

COMMAND: *stream* **sio d**

Then create a new stream. For example, to change stream 0 to hexadecimal format:

```
0 sio i screen /x
```

Now the values you enter must be hexadecimal numbers. For instance, to enter the ASCII value of 'y' you would type:

```
SIO_input>79
```

You can specify one of three types of formats: **c** (character), **x** (hexadecimal) and **o** (octal). The default format is character.

Simulated I/O Buffers

Each of the eight simulated I/O streams has a corresponding dedicated buffer. For instance, output stream 1 corresponds to a different buffer than output stream 2. Once a stream corresponds to a buffer, it is always with that buffer. You cannot select a new buffer after the initial call to either `_simi` or `_simo`.



The address and size of each simulated I/O stream buffer is determined the first time a call to either `_simi` or `_simo` is made. Thereafter, every call to either `_simi` or `_simo` using that stream will use the same buffer size and address as the first call. The buffer size and address for a stream cannot be changed once it has been initially set. You have to delete the stream first.

I/O streams that correspond to files instead of the screen also have their buffers set to a fixed size on the first call.

Any output to a buffer that is too small results in truncated output at the limit of the buffer. If 40 characters go to a 20 character buffer, it discards 20 characters. Any input to a buffer that is too small results in limiting the input to the size of the buffer.

10.6.4 CHANGING THE SIMULATED INPUT PROMPT

To change the prompt for simulated input:



Follow these steps:

- Select the `Debug | Simulated I/O Setup...` menu item to open the Simulated I/O Setup dialog box.
- Select the stream from the `Stream Nr` list selection box for which you want to change the prompt.
- Enter the new prompt in the `Prompt` edit field and click on the `OK` button to accept your changes.



Enter the **`sio p`** command in the Command Window.

To change the prompt for the input stream number 0, type:

```
0 sio p "Enter>"
```

The quotation marks are not strictly necessary, but they help to distinguish the prompt from other CrossView Pro commands.

FUNCTION: Change prompt of an I/O stream.

COMMAND: `stream sio p prompt`

10.6.5 DIRECTING I/O TO A FILE

For more complicated I/O you may want to direct information to or from a file. Enter the filename in the corresponding edit field of the Simulated I/O Setup dialog box. When you use the keyboard, type the filename after the **sio** command.

In the demo example, you direct the output stream to file `myfile`, by typing:

```
1 sio o myfile
```

Now CrossView Pro sends the simulated output to the file `myfile`. Similarly, if you want the information recorded in hexadecimal format, you type:

```
1 sio o myfile /x
```

In order for CrossView Pro to close the output file, you must delete the stream:

```
1 sio d
```

If the output from the program does not fill the length specified, CrossView Pro fills the additional space with null (zero) values.

Inputting data from a file is an identical process. To input octal data from the file `myfile`, you type:

```
0 sio i myfile /o
```

10.7 THE SIMULATED I/O WINDOW

If you direct simulated I/O to the screen, CrossView Pro displays the output in the Simulated I/O window. Depending on the number of streams, the window shows from one to eight streams at a time.

Similarly, if you direct input from the keyboard; whatever you input appears in the appropriate simulated I/O window.

See chapter *Using CrossView Pro* for examples of Simulated I/O Windows.

10.8 BACKGROUND MODE

Background mode is a feature for running the application under debug and CrossView Pro at the same time. This allows you to monitor the target application using CrossView Pro, while the application is running. Depending on the target hardware and/or debug instrument connected to CrossView, target execution can even be real-time.

Since CrossView's monitoring of the target hardware must be non-intrusive, not all functions of the debugger are enabled while running in background mode.



You can only use this feature if it is supported by the execution environment (see the addendum).

10.8.1 CONFIGURATION

CrossView can be instructed to automatically refresh one or more windows of the debugger periodically while running in background mode. You can use the Background Mode Setup dialog for specifying the desired set of windows to be refreshed.



Use menu item **Options | Background Mode Setup...** to open the Background Mode Setup dialog.



A distinction has been made between updating the Source lines window and updating the Disassembly window. Updating the Disassembly window may be to time-consuming, so you may want to disable its updating in Background mode, while still keeping the Source lines window up-to-date when that is displayed on screen.



Use the **u** command to toggle the updating of windows in background mode.

FUNCTION: Toggle update of window in background mode.

COMMAND: `[interval] u [d | k | r | cd | ck | cr | s | a | mem | t]`

The following windows can be updated in background mode:

d (Data), **k** (Stack), **r** (Register),
cd (Data, composite), **ck** (Stack, composite), **cr** (Register, composite),
s (Source), **a** (Assembly), **mem** (Memory), **t** (Trace)

Initially only the data window will be updated. CrossView Pro repeatedly looks at the execution environment to react on changes. It pseudo-simultaneously looks for user commands from the keyboard (or from the playback file), and periodically it updates the windows.

If all windows would be updated the update frequency would drop. That is why you can toggle a switch for each window. To toggle the updating of the register window, you can type:

```
xvw% u r
```

If the switch for a window is 'on', it will be updated, otherwise it will be skipped.

You can also specify a new update interval.

Without arguments, CrossView Pro displays all windows updated periodically plus the update interval.



Notice that simulated I/O is done through 'invisible' breakpoints, and these must be handled inside the loop. Hence, if updating the windows takes a lot of time (many monitor commands), it will also slow down simulated I/O.

10.8.2 MANUAL REFRESH

If you have windows which you do not want to refresh periodically, you can disable them in the Background Mode Setup dialog's refresh list, and refresh these windows manually.



Select the View | Background Mode menu item and select one of the refresh options.



Use the **ubgw** command.

FUNCTION: Update the appropriate window when the target runs in the background.

COMMAND: **ubgw** [**s** | **a** | **k** | **r** | **d** | **mem** | **t** | **all**]



Section *Refresh Limitation* in this chapter.

10.8.3 ENTERING BACKGROUND MODE

To run a program in background mode:



Select menu item Run | Background Mode | Run in Background.



Type the **CB** command on the command line.

FUNCTION: Run a program in background mode.

COMMAND: [count] **CB** [linenumber]

This will start the application under debug to run continuously (as with the **C** command), and switch CrossView Pro from Halted to Background Mode. *count* is assigned to the breakpoint at the current execution position as the number of times to hit this breakpoint before execution to stop. *linenumber* specifies the source line to place a temporary breakpoint.

The mouse pointer changes to an arrow with a small watch face underneath. This indicates that CrossView Pro is now in background mode. Some commands are treated a little different in this mode, because they can otherwise influence the running program badly. Commands that need information from the stack (like **bU**, **bu**, **bb** or **BB**) are not allowed because that information is not reliable. Other commands require great care, for example the **o** command.

For example if you type the **g** while in background mode you will see:

```
xvw% g 56
```

```
Command "g" is not allowed while the emulator is
running in background.
```

10.8.4 LEAVING BACKGROUND MODE

You can leave Background Mode in three ways:

- 7. Stop the target immediately:



Select menu item Run | Background Mode | Halt Target.



Enter the **st** command:

```
xvw% st
```

- 8. Let CrossView wait for the target to stop:



Select menu item Run | Background Mode | Wait for Target to Stop.



To wait to come to a breakpoint, you can use the **wt** command:

```
xvw% wt
```

- 9. A program running in background mode also stops when it encounters a breakpoint.

FUNCTION:	Stop a program in background mode.
COMMAND:	st

The **wt** command behaves just as if you have typed the **C** command. CrossView Pro returns with a prompt, after the program hits a breakpoint. However, there is an interesting difference with the **C** command. If you push the **Halt** button, it returns with the background prompt. The program that runs in the execution environment continues without interruption.

FUNCTION:	Wait for the running process to stop
COMMAND:	wt

10.8.5 THE STACK IN BACKGROUND MODE

While the execution environment runs in background, CrossView Pro does not allow the use of information that comes from the stack. The reason is that the running program must be stopped in order to get consistent information from the stack. Stopping (and afterwards continuing) the program conflicts with the "real-time" nature of the background mode.

If there is a need for it, you can make a macro that performs the desired operations.

10.8.6 LOCAL AND GLOBAL VARIABLES

In background mode you can continuously monitor variables. However, realize that local variables (in CrossView Pro variables are called 'local' if they reside on the stack) cannot be monitored. Instead you will see "unknown name". Global variables have a fixed address, so CrossView Pro knows where to get their contents from.

If you are very anxious to see local variables you can first get an address and then use that address to monitor the contents. For example:

```
$adr_sum = &sum  
m *(adr_sum)/x4
```

In this example sum is a long (4 bytes). You must be sure that sum remains at that address while the program is running.



The values you get this way are only valid under specific conditions. Local variables from the function main normally meet these conditions.

10.8.7 REFRESH LIMITATION

While running the application in the background mode, the automatic refresh functionality may not be able to keep up with all the debugging information produced by the running target. Typically, the collected information will be correctly displayed and automatically updated in the current open views and no information will be lost. You might lose the debugging information when scrolling these views during the background mode. The reason is that either CrossView Pro does not run fast enough or the communication with the target hardware is not handled fast enough by the operating system.

The information that cannot be processed by CrossView Pro within the specified update interval, is displayed as either '<unknown>' or dashes. The way the lost information is displayed depends on the internal communication level within CrossView Pro where the information is lost. Information lost during communication with the target hardware is displayed as '<unknown>'. Information lost by CrossView Pro while processing and interpreting this information, is displayed as dashes.

On the next automatic or manual update, all debugging information in the currently open views is automatically updated. All visible '<unknown>' values and dashes are replaced with their actual values as produced by the running target.

10.8.8 ASSERTIONS

CrossView Pro automatically suspends assertions with the **CB** command.

CHAPTER 11

DEBUGGING NOTES



TASKING



11

CHAPTER

Here are a few notes about debugging in special situations:

11.1 DEBUGGING ASSEMBLY LANGUAGE

You may debug assembly language programs or modules much as you do C source. The **s**, **S** and **si** commands single step through the assembly source. You may place code breakpoints on assembly language instructions with the **bi** command.

For additional information on debugging assembly code, see \$autosrc, \$mixedasm and \$symbols in the *Reserved Special Variables* table in section 3.4.

There is a restriction on debugging assembly language code:

- Assembly language subroutines cannot be called from the command line.

11.2 DEBUGGING MULTIPLE PROGRAMS

You probably have only one linked and located absolute object file that describes the whole system load. However, for various reasons, you may want to build your system load by linking and locating into several files. The debugger can handle the symbols from only one load module in one absolute object file at a time. Consequently, if there are several absolute files or several load modules within one absolute file, you will have to change the context from one to another explicitly. Use the **N** command or the Load Symbolic Debug Info dialog to load the appropriate symbols. This does not disturb the state of the target system.

You can also download the image part of another absolute object file (using the **dn** command), without leaving the debugger.

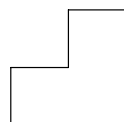
CHAPTER

COMMAND REFERENCE

12



TASKING



12

CHAPTER

This chapter contains a summary of all CrossView Pro commands, followed by a complete description of each command.

12.1 CONVENTIONS USED IN THIS CHAPTER

Each CrossView Pro command has a particular syntax, that is, the form it must take for CrossView Pro to recognize it. To help you learn the syntax of each command, this chapter uses a special notation to describe the syntax of each command. Consider the following example:

stream **sio** {**i** | **o**} {*file* | **screen**} [*/format*]

Command items in **bold** font are the actual command keywords typed from the keyboard. In the example above, **sio** is in bold font since you must type it exactly as shown.

Items in *italics* are names of the command part. Here *stream* is in italics, since you must substitute the appropriate value for stream. The Description section for each command describes what kinds of values should be substituted for italicized terms.

Expressions in [brackets] are optional items you may include in a particular command. In this example */format* is not necessary for the **sio** command to work. Usually if you omit an optional expression, CrossView Pro uses a default value.

The | symbol means *or*. For instance, {*file* | **screen**} means a filename or the word **screen** (but not both) may be used in the command.

12.2 COMMANDS: SUMMARY

12.2.1 STARTUP OPTIONS



From EDE, you can set the values of the **-a**, **-b** and **-c** options in the Miscellaneous tab of the EDE | CrossView Pro Options dialog.

- a *number*** Set the maximum number of assertions (the default is 100).
- b *number*** Set the maximum number of breakpoints (the default is 200).
- c *number*** Set the maximum number of instruction trace (the default is 32).
- C *cpu*** Force CPU type selection. This option also determines which register file (*regcpu.dat*) will be used. This option overrides the CPU type selection in both *xvw.ini* and a target configuration file.
- dsc *dsc*** Force locator description file selection. The default is *m16c.dsc*.
- D *device_type,opt1[opt2]***
Selects a device and specifies device specific options, such as communication port and baud rate. The allowed combinations for your execution environment are described in the manual addendum for that specific execution environment.

The following combinations are possible:

- D rs232,*port*,*speed***
Select RS-232 communication.

<i>port</i>	For PC this is COM1, COM2, COM3 or COM4. A colon should not be added. For UNIX this is the full path of the RS-232 device driver (e.g., <i>/dev/tty01</i>). By default CrossView Pro uses the first RS-232 port.
<i>speed</i>	This is the baud rate used for the specified <i>port</i> . The default is 9600.

-D parallel,*port*

Select parallel communication.

port For PC this is LPT1 or LPT2. Do not add a colon. For UNIX this is the full path of the parallel device driver. By default CrossView Pro uses the first parallel port.

-D tcp,*host,port*

Select TCP/IP communication. On UNIX the standard TCP/IP implementation is used. On MS-Windows the WINSOCK.DLL implementation is used.

host The name of the host to be accessed via TCP/IP.

port The port number on *host* to be accessed.

-D dev,*device-file*

Use a UNIX device driver as communication channel. For RS-232 devices use the **-D rs232** option, described above.

device-file The full path of the UNIX device file.

-D isa,*io-port,address*

Select communication channel to an (E)ISA interface card in the PC.

io-port PC I/O port number or I/O channel used for accessing the (E)ISA card.

address The memory address used to access the (E)ISA card.



From EDE, you can set communication parameters in the Communications tab of the EDE | CrossView Pro Options dialog.

-em *macro*[=*def*]

Add macro for pre-processing the description file. If *def* is not given ('=' is omitted), '1' is assumed.

-f *file* Read command line options from *file*.

-G *path* Specify startup directory for CrossView Pro.

-i Perform an initial download of the image of the absolute object file on startup.



From EDE, you can enable the Initial download of program check box in the Miscellaneous tab of the EDE | CrossView Pro Options dialog.

- l file** Make a log of CrossView Pro target communications in file.
- n address** Inform CrossView Pro that the program was loaded into memory at an address other than zero.
- p file** Begin by playing back commands from file.
- P file** Begin by playing back commands from file with command single step.
- r file** Begin recording commands in file.
- R file** Begin logging commands and screen output in *file*.



From EDE, you can enter record, log and playback filenames in the Logging tab of the EDE | CrossView Pro Options dialog.

- s number** Set the maximum number of special variables (variables independent of the program that CrossView Pro provides for your use). The default is 26.



From EDE, you can enter the maximum number of special variables in the Miscellaneous tab of the EDE | CrossView Pro Options dialog.

- sd directory ...** Specify the directories CrossView Pro should search for source files (separated by semicolons). Relative paths are allowed. When the **N** command is used to load a new symbol file, the current directory is set to the directory containing the symbol file and CrossView Pro now searches for source files relative to this directory.
- tcfg file** Specify a target configuration file. This overrides the filename specified in `xvw.ini`. See section *CrossView Pro Startup Settings* in the *Overview* chapter.
- T [file]** Start CrossView Pro in transparency mode; if *file* is specified, commands in *file* are sent to the execution environment. This option is not available for all execution environments.

12.2.2 VIEWING COMMANDS

- ^[*format*]** Display contents of preceding memory location.
- exp*** Print value of expression using /n format.
- exp @.format*** Print address of expression *exp* in format *format*.
- exp/format*** Print value of expression *exp* in format *format*.
- line*** Move viewing position to line *line*.
- number ct*** Display a source-level trace corresponding to the last number of machine instructions executed. This command is not available for all execution environments.
- number ct i*** Display a disassembled assembly-level trace corresponding to the last number of machine instructions executed. This command is not available for all execution environments.
- number ct r*** Display a raw trace corresponding to the last number of trace frames. This command is not available for all execution environments.
- e** [*func* | *file*]** Enter function *func* or file *file* or view current viewing position.
- stack e*** Enter function using stack address.
- [*addr*] **ei**** View current viewing position or view instruction at address *addr*.
- f** ["*printf-style-format*"]** Change default address display format.
- L**** Synchronize the viewing position with the execution position. Print current file, function and line number.
- l** {**a**|**b**|**d**|**f**|**g**|**k**|**l**|**L**|**m**|**p**|**r**|**s**|**S**} [*string*]** List **a**ssertions, **b**reakpoints, **d**irectories, **f**iles, **g**lobals, **k**ernel state data, **l**abels (on module scope), all **L**abels, **m**emory map (of application code sections), **p**rocedures, **r**egisters, **s**pecial variables, **S**ymbol tables. If given, only those starting with *string*.

l [<i>func</i>]	List all parameters and locals of function <i>func</i> . Without a function, this command lists all parameters and locals of the current function in view.
l <i>stack</i>	List all parameters and locals of function at depth <i>stack</i> .
nC	Move viewing position to next covered line.
nU	Move viewing position to next uncovered line.
opt [<i>option</i> [= <i>value</i>]]	List or set option value. Without an argument, list all option values.
[<i>line</i>] P [<i>exp</i>]	Print <i>exp</i> lines of source starting at line <i>line</i> , include machine addresses.
[<i>line</i>] p [<i>exp</i>]	Print <i>exp</i> lines of source starting at line <i>line</i> .
pC	Move viewing position to previous covered line.
pU	Move viewing position to previous uncovered line.
[<i>exp</i>] T	Trace the stack for <i>exp</i> number of levels, list local variables.
[<i>exp</i>] t	Trace the stack for <i>exp</i> number of levels, printing active functions and parameters passed.
td	Disable tracing.
te	Enable tracing.

12.2.3 DATA MONITORING

cd	Disable, turn off, gathering of coverage data.
ce	Enable, turn on, gathering of coverage data.
dis <i>address</i> [, { <i>address</i> # <i>count</i> } [, <i>i</i>]]	Disassemble a range of memory.
dump <i>address</i> [, { <i>address</i> # <i>count</i> } [, [<i>style</i> [<i>width</i>]] [, <i>filename</i> [<i>a</i>]]]	Dump a memory range.
M	Display list of monitored expressions in the Command window.

m <i>exp</i>	Monitor the expression <i>exp</i> .
<i>num</i> m d	Remove monitored expression labeled <i>num</i> .
<i>addr_start</i> mcp <i>addr_end, addr_dest</i>	Memory copy.
<i>addr</i> mF <i>exp[,exp]...</i>	Single fill memory address <i>addr</i> with expressions.
<i>addr_start</i> mf <i>addr_end, exp[,exp]...</i>	Fill memory address range with expressions and repeat the pattern until the end address of the memory region is reached.
<i>addr_start</i> ms <i>addr_end, exp[,exp]...</i>	Search memory address range for a given pattern.
pd	Disable, turn off, profiling.
pe	Enable, turn on, profiling.
proinfo	List profiling info.

12.2.4 EXECUTION CONTROL COMMANDS

A [a s]	Toggle state of assertion mechanism.
a <i>cmds</i>	Create a new assertion with the command list <i>cmds</i> .
<i>exp</i> a { a d s }	Activate, delete, suspend assertion <i>exp</i> .
B	List all breakpoints.
[<i>line</i>] b [<i>cmds</i>]	Set breakpoint at source line <i>line</i> , and associate command list <i>cmds</i> with breakpoint.
[<i>stack</i>] bb [<i>cmds</i>]	Set temporary breakpoint at beginning of function at stack level <i>stack</i> and associate command list <i>cmds</i> .
[<i>stack</i>] bb [<i>cmds</i>]	Set breakpoint at beginning of function at stack level <i>stack</i> and associate command list <i>cmds</i> .

- [number]* **bc** *[count]* *[reset_count]*
Set breakpoint *count* and *reset_count* for breakpoint with number *number*.
- count* **bcYC** *[cmds]*
Set temporary breakpoint after the specified cycle *count* and associate command list *cmds*.
- count* **bcyc** *[cmds]*
Set breakpoint after the specified cycle *count* and associate command list *cmds*.
- exp* **bd** {**r** | **w** | **b**} *exp2* *[cmds]*
Set a data range breakpoint (between addresses *exp* and *exp2*) read (**r**), write (**w**) or both read and write (**b**), and associate command list *cmds*. This command is not available for all execution environments.
- exp* **bd** {**r** | **w** | **b**} *[cmds]*
Set a data breakpoint, read (**r**), write (**w**) or both read and write (**b**) at address *exp*, and associate command list *cmds*. This command is not available for all execution environments.
- num* **bdis** Disable code breakpoint.
- num* **bena** Enable code breakpoint.
- [addr]* **bi** *[cmds]*
Set temporary breakpoint at machine instruction and associate command list *cmds*.
- [addr]* **bi** *[cmds]*
Set breakpoint at machine instruction and associate command list *cmds*.
- count* **binST** *[cmds]*
Set temporary breakpoint after *count* machine instructions and associate command list *cmds*.
- count* **binst** *[cmds]*
Set breakpoint after *count* machine instructions and associate command list *cmds*.

time **bTIM** [*cmds*]

Set temporary breakpoint after *time* number of seconds and associate command list *cmds*.

time **btim** [*cmds*]

Set breakpoint after *time* number of seconds and associate command list *cmds*.

[*stack*] **bU** [*cmds*]

Set a temporary up-level breakpoint at stack level *stack* and associate command list *cmds*.

[*stack*] **bu** [*cmds*]

Set up-level breakpoint at stack level *stack* and associate command list *cmds*.

[*exp*] **C** [*line*] Continue execution from current value of program counter. If *line* is specified, execution continues up to that line. Breakpoint's count is set to *exp*.

[*exp*] **CB** [*line*]

Continue execution in background from current value of program counter. If *line* is specified, execution continues up to that line. Breakpoint's count is set to *exp*. This command is not available for all execution environments.

D Delete all breakpoints.

Dy Delete all breakpoints without prompt for confirmation.

[*number*] **d** Delete breakpoint number.

cpu **eC** Start execution on the current CPU and switch to *cpu*.

[*cpu*] **ec** Select CPU or show current CPU number.

g *line* Go to the specified line in the current procedure.

address **gi** Go to the specified address.

if (*exp*) {*cmds*} [{*cmds*}]
Conditionally execute commands.

load [*file*] Load symbol table of *file* in CrossView Pro and download the image part to the target. This is a combination of **N** and **dn**.

N [<i>file</i>]	Load symbol table of <i>file</i> in CrossView Pro.
prst	Reset program counter.
Q	Report breakpoint quietly.
q [<i>y</i>]	Quit debugger (do not save desktop settings).
q s	Save current desktop settings and quit debugger.
R	Reset program counter and start execution.
rst	Reset target system to initial conditions.
[<i>exp</i>] S	Single step for <i>exp</i> lines, step over function calls.
[<i>exp</i>] s	Single step for <i>exp</i> lines, step into function calls.
[<i>exp</i>] Si	Single machine step for <i>exp</i> machine instructions, step over subroutine calls.
[<i>exp</i>] si	Single machine step for <i>exp</i> machine instructions, step into subroutine calls.
st	Stop the execution of the target immediately. This command is not available for all execution environments.
[<i>interval</i>] u [d k r cd ck cr s a mem t]	Toggle updating of the appropriate window when the target runs in the background. You can specify the update interval, in seconds. If <i>interval</i> is zero, never update automatically. This command is not available for all execution environments.
ubgw [s a k r d mem t all]	Refresh the appropriate window, or all open windows, when the target runs in the background. This command is not available for all execution environments.
use [<i>path</i>]...	Clear source directory search path or use the specified path to search for source files.
wt	Wait for the completion of the target. This command is not available for all execution environments.

[*exp*] x Force an exit from assertion mode. If *exp* is non-zero, finish executing command list of the current assertion.

12.2.5 RECORD & PLAYBACK

<*file* Play back commands from *file*.

<<*file* Play back commands with single step from *file*.

>*file* Record commands in *file*.

>{t**|**f**|**c**}** Set recording file status, true (**t**), false (**f**) or closed (**c**).

> Report status of command recording mechanism.

>>*file* Log commands and screen output in *file*.

>>{t**|**f**|**c**}** Set logging file status, true (**t**), false (**f**) or closed (**c**)

>> Report status of command and screen output logging mechanism.

>&*file* Log host-to-target communication in *file*. Not available for all execution environments.

>&{t**|**f**|**c**}** Turn target communication logging on (**t**), off (**f**) or close (**c**) log file. Not available for all execution environments.

>& Report status of target communication logging mechanism. Not available for all execution environments.

12.2.6 MACROS

echo *string* Display macro expansion of *string*.

save *file* Save current macros to *file*.

set Display all macros.

set *macro* "*cmds*"
Define macro *macro* as command list *cmds*.

unset Delete all macros.

unset macro!Delete definition of macro *macro*.**macro!**Prevent expansion of *macro*.**12.2.7 SIMULATED INPUT/OUTPUT****sio** List all simulated I/O streams.*stream* **sio {i|o} {file | screen} [/format]**Create simulated input (**i**) or output (**o**), numbered *stream*, directed from/to file or from/to the screen. Display data in *format*.*stream* **sio d** Delete I/O stream *stream*.*stream* **sio p** *prompt*Change input *stream*'s prompt to *prompt*.**12.2.8 TARGET SYSTEM CONTROL****dn** Download the image part of the current absolute file, specified when CrossView Pro was invoked or loaded with the **N** command.**dn file** Download the image part of the absolute file *file*.**n [addr]** Set code address bias (for overlays) to *addr*. If no address is given, then display the current bias.**o [cmd]** Enter transparency mode (exit with *ctrl-D*). If *cmd* is present, pass *cmd* to the execution environment. Not available for all execution environments.**! [command-line]**Execute shell command *command-line* or invoke new shell.**12.2.9 HELP COMMANDS****I** Print information about debugger state.

12.2.10 SEARCH COMMANDS

Z	Toggle case sensitivity in searches.
/[<i>string</i>]	Search forwards in source file for <i>string</i> . If <i>string</i> is not present, perform previous search again.
?[<i>string</i>]	Search backwards in source file for <i>string</i> . If <i>string</i> is not present, perform previous search again.
"<i>string</i>"	Print <i>string</i> .

12.3 COMMANDS: DETAILED DESCRIPTIONS

The rest of this chapter provides the detailed descriptions of the CrossView Pro commands.

expression

Function

Print the value or address of an expression.



Select the `Data | Evaluate Expression` menu item. Enter an expression and optionally select a format code. You may set up a monitor, which instructs the debugger to evaluate a particular expression each time the program stops, from the Source Window by selecting text there and by clicking on the `Watch selected source expression` accelerator button.



Enter the expression in the Command Window. You may specify in which format you want CrossView Pro to display the answer.

Description

In the Command Window, the syntax for this command is:

```
exp [/ format | @ format]
```

Print the value or address of *exp* with format *format*. A / (slash) is used to print the value of *exp* and a @ (commercial at) is used to print the address of *exp*. If *format* is not supplied, the natural (/n) format of the expression is used.

Formats have the syntax:

```
[count] style [size]
```

count is the number of times to apply the format *style* and defaults to 1. *style* may be one of:

```
a c D O U X d o u x E F G e f g i I n P p s t
```

See the *Accessing Code and Data* chapter and the section *Formatting Expressions* in the chapter *Command Language* for details on each of the format styles. *size* indicates the number of bytes to be formatted. Rather than a number for the integer type styles, *size* may also be: **c** for char, **s** for short, **i** for int, and **l** for long.

The default action, if no modifier is specified, is to print the value of *exp* using the /n (normal) format.

Be careful with one letter variable names, as they may be taken as an CrossView Pro command rather than as a variable. If an expression begins with a variable that might be mistaken for a command, then eliminate any white space between the variable and the first operator. For example: use `h=9` instead of `h = 9`.

To display the value of a variable that has the same name as an CrossView Pro command you must use the natural format modifier. For example: to print the value of the variable `C`, use `C/n`.

Variables may be altered as a side effect of evaluation of *exp*. See the example below.

Example

To set variable `aux` to `t` times 8, type:

```
aux = t++*8
```

As a side effect the variable `t` is post-incremented. If you type:

```
$s_aux = func(t,s)
```

CrossView Pro will set special variable `$s_aux` to the result of the function call to `func` with the variables `t` and `s` passed as parameters. If you type:

```
$s_aux/x4
```

Print the value of the special variable `$s_aux` as four hex bytes; you could also use: `$s_aux/x1`.



line

Function

Display the C source line numbered *line* in the current source file.



Select the Search | Find Line... menu item, enter the line number and click on the Find button. Alternately, you may click on the desired source line in the Source Window.



Enter the line number in the Command Window. The syntax is:

line

Description

The current viewing position becomes *line*.

Example

To display the twelfth line in the current source file, type:

12



e, p, P

string

Function

Echo a string to the terminal.



There is no mouse equivalent for this function. However, many distinct functions accept strings. See below.



Enter the string to the Command Window.

Description

A string may contain standard C escapes, such as `\n` for a newline. The syntax for a string in the Command Window is:

`"string"`

Example

This function can be useful for labelling breakpoints. For example, to insert a breakpoint at line 12 and have a message printed when that line is reached, enter:

```
12 b {"At the twelfth line\n"; C}
```

When CrossView Pro reached line 12, the message "At the twelfth line" will be printed and the program will continue. If you only type:

```
"Debug"
```

CrossView Pro will simply echo the word "Debug."



Q. *expression*



Function

Instruct CrossView Pro to interpret a command literally, ignoring any macro definitions of the same name. Also, enter a shell command.



There is no mouse equivalent for this command.



The syntax for this command is:

`[string] !`

or:

`! [string]`

Description

This command is useful whenever *string* should be treated literally and not as a potential macro invocation. It can be used, for example, in executing an CrossView Pro command whose name has been defined as a macro.

Example

To enter the host environment under a new shell, type:

`!`

To execute the host `date` command, type:

`!date`

To execute the CrossView Pro command **b** instead of the macro named `b`, type:

`b!`



set, unset, echo, save



Function

Search down (forward) for a string.



To search for a string in the Source Window, select the Search | Search String menu item or click on the Repeat search down for string accelerator button.



The command line syntax is:

`/ [string]`

Description

The search begins with the line after the current line. If the *string* is found the viewing position is changed to the line containing the string. The execution position is not affected. If you do not specify a string to search for, CrossView Pro will look for the most recent specified string.

Searches wrap around to the beginning of the file. Regular expressions are not recognized.

Example

To look for the next occurrence of Random in the current file, beginning with the line after the current line, type:

`/Random`



`?, Z`



Function

Search up (backward) for a string.



To search for a string in the Source Window, select the **Search | Search String** menu item or click on the **Repeat search up for string** accelerator button.



The command line syntax is:

? [*string*]

Description

The search begins with the line before the current line. If *string* is found, the current line is changed to point to the line containing the string. The execution position is not affected. If you do not specify *string*, CrossView Pro searches for the previously-specified string again.

Searches wrap around to the end of the file. Regular expressions are not recognized.

Example

To look for the previous occurrence of **Random** in the current file, beginning with the line above the current line, type:

?Random



/, Z



Function

Continuous command playback. Read commands continuously from a file.



To setup command playback, select the **Options | Playback** menu item. Enable the **Continuous playback** check box and click on the **Execute** button.



The command line syntax is:

< *file*

Description

All the commands in *file* will be read and executed. If a playback file contains either a **<** or **<<** command, playback switches to the newly specified file and does not return to the original file.

Record and playback options can also be specified via command line parameters.

If the execution of commands from the playback file is interrupted with the **Halt** button, CrossView Pro will begin reading the remainder of commands in file using single step playback (see the **<<** command.)

Example

To read and execute the commands found in the file `command.out`, type:

<command.out



<<, >, I



Function

Single-step command playback.



To setup command playback, select the Options | Playback menu item. Disable the Continuous playback check box and click on the Execute button.



The command line syntax is:

`<<file`

Description

Commands will be played back one at a time. Each command will be loaded sequentially into the entry field of the Command Window. The command can then be edited and executed.

The carriage return will execute the current command and stop at the next one.

If a playback file contains either a < or << command, playback switches to the newly specified file and does not return to the original file. Record and playback options can also be specified via command line parameters.

Example

To read and execute the commands found in the file `command.out`, type:

`<< command.out`



<, >, I



Function

Record CrossView Pro commands to a file.



To start recording or toggle the state of the command recording mechanism, select the **Options | Record** menu item. To start recording click on the **Start** button. To suspend recording click on the **Suspend** button. To resume recording click on the **Resume** button. To stop recording click on the **Stop** button.



The command line syntax is (note that the greater than sign must be typed as shown):

```
> [!] [file | t | f | c]
```

Description

CrossView Pro will start recording commands in a file if *file* is specified, otherwise, turn recording on (**t**), off (**f**), or close (**c**) the recording file. Specifying a different file while recording is on will cause the old output file to be closed and all successive commands will be sent to the new file. If no arguments are given, the state of the recording mechanism will be displayed.

The optional **!** forces flushing of the output after every write.

The commands recorded can be played back by using the **<** or **<<** command. It is possible to have a command recording file and a screen output recording file to be open concurrently. The file is also closed as a side effect of the **q** command.

Commands issued to the emulator under transparency mode are not recorded.

Files may not be named: **t**, **f** or **c**.

Example

To set (or change) the command recording file to `log.cmd` and turn command recording on, type:

```
>log.cmd
```

To suspend recording commands, type:

>f

To resume recording the commands to the recording file, type:

>t

To stop recording commands and close the file, type:

>c

To display the state of the recording mechanism, type:

>



>>, >&, <, <<, I, q



Function

Log Command Window screen output. All Command Window input and output will be saved to a file.



To create a log of Command Window screen output, select the Options | Log | Command Input/Output... menu item. To turn on logging click on the Start button. To suspend logging click on the Suspend button. To resume logging click on the Resume button. To turn off logging click on the Stop button.



The command line syntax is:

```
>> [!] [file | t | f | c]
```

Description

Start logging the commands typed and their output in a file if *file* is specified, otherwise, turn logging on (**t**), off (**f**), or close (**c**) the log file. Specifying a different file while logging is on will cause the old output file to be closed and all successive Command window output will be sent to the new file. If no arguments are given, the state of the recording and logging mechanism is displayed.

The optional **!** forces flushing of the output after every write.

Because output is logged as well as commands, files logged using >> cannot be played back like those recorded with the > command.

It is possible to have both a command recording file and a screen output logging file open concurrently. The log file is also closed as a side effect of the **q** command. Log files may not be named: **t**, **f** or **c**.

Example

To set (or change) screen output recording file to the file `log.out` and turn screen output recording on, type:

```
>>log.out
```

To suspend recording the screen output, type:

```
>>f
```


To resume recording the screen output in the recording file, type:

```
>>t
```

To stop recording the screen output and close the file, type:

```
>>c
```

To display the state of the recording mechanism, type:

```
>>
```



```
>, >&, l, q
```



Function

Log communications between debugger and emulator.



To save debugger/emulator communications, select the Options | Log | CrossView-Emulator I/O... menu item. To turn on logging click on the Start button. To suspend logging click on the Suspend button. To resume logging click on the Resume button. To turn off logging click on the Stop button.



The command line syntax is:

```
>& [!] [file | t | f | c]
```

Description

Start host-to-execution environment communication logging in a file if *file* is specified; otherwise, turn logging on (**t**), off (**f**), or close (**c**) the log file. This feature is most often used to diagnose problems with CrossView Pro itself.

The optional **!** forces flushing of the output after every write.

The commands captured cannot be played back the way commands recorded by the **>** command can. The log file is also closed as a side effect of the **q** command.



Not available for all execution environments.

Example

To open the file `log.out` and put the following host-to-emulator communications in this file, type:

```
>&log.out
```

To suspend logging communications in the log file, type:

```
>&f
```

To resume logging communications in the log file, type:

```
>&t
```

To stop logging communications and close the file, type:

>&c



>, >>, q



Function

Display contents of preceding memory location based on the size of the last data item displayed.



There is no direct mouse equivalent for this function.



The command line syntax is:

```
^ [format ]
```

Description

Use previous format or *format*, if supplied. Formats have the syntax:

```
[count] style [size]
```

count is the number of times to apply the format *style* and defaults to 1. *style* may be one of:

```
a c D O U X d o u x E F G e f g i I n P p s t
```

See the *Accessing Code and Data* chapter and the section *Formatting Expressions* in the chapter *Command Language* for details on each of the format styles. *size* indicates the number of bytes to be formatted. Rather than a number for the integer type styles, *size* may also be: **c** for char, **s** for short, **i** for int, and **l** for long.

This command is most often used in combination with *exp/format* to look at the value of some variable or memory location.

Example

To display the variable *aux* as two octal values of length two, type:

```
^ aux/2o2
```

To show the eight bytes before *aux* in hexadecimal format, next type:

```
^2x4
```



expression

A

Function

Toggle the state of the assertion mode.



To activate or suspend assertion mode, select the `Debug | Assertions...` menu item, and enable or disable the `Assertion Mode Active` check box.



The command line syntax is:

A [**a** | **s**]

Description

Activate (**A a**) or suspend (**A s**) overall state of the assertion mechanism. If no operand is given, toggle the state.

Example

To activate the assertion mechanism, type:

A a

To suspend the assertion mechanism, type:

A s

To toggle the state of the assertion mechanism, simply type:

A



a

a

Function

Define or modify an assertion.



Select the `Debug | Assertions...` menu item to view the Assertions dialog box. Select `New...` to define an assertion. Select an assertion and select `Edit...` to modify an assertion.



The command line syntax is:

```
exp a { a | d | s }
a cmds
```

Description

The **a** command is used to invoke two different commands. The syntax for each command is distinct. The first version allows modification of the state of the assertion specified by the expression *exp*. (The assertion can be activated (**a a**), deleted (**a d**) or suspended (**a s**.) The second version creates a new assertion with the given command list *cmds*. Using the mouse, you can create a new assertion or toggle the state of an existing one from the Assertions dialogue box.

Suspended assertions continue to exist, but are not active. Deleted assertions must be explicitly redefined in order to be made active again.

The commands for every active assertion are executed after every source statement is executed. The **x** command in an assertion command list forces an exit from assertion mode.

This command is not allowed when the target runs in the background.

Example

To suspend assertion 3, type:

```
3 a s
```

To delete assertion 1, type:

```
1 a d
```

To set an assertion to stop the program when global variable `myvar` exceeds 3, type:

```
a if (myvar > 3) {x}
```



A, l, x

B

Function

List all of the currently defined breakpoints.



Select the Debug | Breakpoints... menu item to view the Breakpoints dialog box.



The command line syntax is:

B

Description

Breakpoints are listed with numbers associated with them. These numbers can be used to delete individual breakpoints.



b, **bb**, **bB**, **bi**, **bI**, **bu**, **bU**, **R**, **C**, **D**, **l**

b

Function

Set a code breakpoint.



Select the `Debug | Breakpoints...` menu item to view the Breakpoints dialog box. Select the `New Code...` button to create a new code breakpoint, leave the `Task ID` field empty or enter the string "any".

Alternatively, you can set a code breakpoint directly in the source by double-clicking on unmarked text, the viewing position, or the execution position.



The command line syntax is:

`[line] b [commands]`

Description

You can attach a list of CrossView Pro commands with the breakpoint. If no line is given, set the breakpoint at the current viewing position.

When the breakpoint is hit execution is halted. By default the current execution position, function, line number, and source statement are displayed. Next, any commands associated with the breakpoint are executed. The **Q** command can be used to suppress the output from the **b** command.

Example

To set a breakpoint at the current line, type:

`b`

To set a breakpoint at line 10 that will list all global variables and halt execution, type:

`10 b {l g}`



`bd, bD, bb, bB, bi, bI, bt, bti, bti, bu, bU, Q`

bB

Function

Set a temporary breakpoint at the beginning of a function.



In the Stack Window, click on the desired function and select the Debug | Stack Breakpoint | At Function Entry menu item.



The command line syntax is:

```
[ stack ] bB [ cmds ]
```

Description

The function is designated by the stack level *stack*. If no function is specified, CrossView Pro uses the current function (stack level 0), and associates the list of CrossView Pro commands *cmds* with the breakpoint.

Breakpoints set in the Stack Window are always temporary, meaning they will be deleted after the first time you reach them. A breakpoint set in this manner will not be visible in the Source Window.

When the breakpoint is hit, execution is halted; the breakpoint is then removed. By default the current execution position, function, line number, and source statement are displayed. Next, any commands associated with the breakpoint are executed. The **Q** command can be used to suppress the output from the **bB** command.

This command is not allowed when the target runs in the background.

Example

To set a temporary breakpoint at the beginning of the current function which prints a stack trace, type:

```
bB {T}
```

To set a temporary breakpoint at the beginning of the function whose stack number is 2, type:

```
2 bB
```



b, **bb**, **bd**, **bD**, **bi**, **bI**, **bt**, **bti**, **btI**, **bu**, **bU**, **Q**

bb

Function

Set a permanent breakpoint at the beginning of a function.



In the Stack Window, click on the desired function and select the Debug | Stack Breakpoint | At Function Entry menu item. To make the stack breakpoint permanent, select the Debug | Breakpoints... menu item, select the desired stack breakpoint, click on the Edit... button and select the Permanent radio button.



The command line syntax is:

```
[ stack ] bb [ cmds ]
```

Description

Set a breakpoint at the beginning of the function designated by the stack level *stack*. Otherwise, use the current function (stack level 0), and associate the list of CrossView Pro commands *cmds* with the breakpoint.

When the breakpoint is hit, execution is halted. By default the current execution position, function, line number, and source statement are displayed. Next, any commands associated with the breakpoint are executed. The **Q** command can be used to suppress the output from the **bb** command.

This command is not allowed when the target runs in the background.

Example

To set a breakpoint at the beginning of the current function, which prints a stack trace, type:

```
bb {T}
```

To set a breakpoint at the beginning of a function whose stack number is 2, type:

```
2 bb
```



b, bB, bd, bD, bi, bI, bt, bti, bti, bu, bU, Q

bc

Function

Set a breakpoint's count and reset count.



Select the **Debug | Breakpoint...** menu item, select the breakpoint for which you want to set the count and reset count, click on the **Edit...** button, enter a breakpoint count and select the **Reset to 1** or **Reset to value** radio button.



The command line syntax is:

```
[ number ] bc [ count ] [ reset_count ]
```

Description

Set the *count* and *reset_count* for the breakpoint with breakpoint number *number*. When no arguments are given, the breakpoint at the current viewing position is set to a count of 1 and a reset count of 1. If no breakpoint is present at the current viewing position, the message "No such breakpoint" appears.

Each time a breakpoint is hit, CrossView Pro decrements the *count*. When the *count* reaches 0, execution is halted and the *count* is reset to the *reset_count*.

This command is not allowed when the target runs in the background.

Example

To set a breakpoint's count and reset count to 1 for the breakpoint at the current viewing position, type:

```
bc
```

To set the count to 3 and the reset count to 4 for the breakpoint whose breakpoint number is 2, type:

```
2 bc 3 4
```



C

bCYC

Function

Set a temporary cycle count breakpoint.



There is no mouse equivalent for this command.



The command line syntax is:

```
count bCYC [cmds]
```

Description

Set a temporary breakpoint after the specified cycle *count*. *count* can be any expression evaluating to a number. The list of CrossView Pro commands *cmds* are executed when the breakpoint is hit.

When the breakpoint is hit, execution is halted; the breakpoint is then removed. By default the current execution position, function, line number, and source statement are displayed. Next any commands associated with the breakpoint are executed. The **Q** command can be used to suppress the output from the **bCYC** command.

Example

To set a temporary breakpoint after 4 clock cycles and list all global variables, type:

```
4 bCYC {l g}
```



b, **bcyc**, **bINST**, **binst**, **bTIM**, **btim**, **D**

bcyc

Function

Set a permanent cycle count breakpoint.



There is no mouse equivalent for this command.



The command line syntax is:

```
count bcyc [cmds]
```

Description

Set a permanent breakpoint after the specified cycle *count*. *count* can be any expression evaluating to a number. The list of CrossView Pro commands *cmds* are executed when the breakpoint is hit.

When the breakpoint is hit, execution is halted. By default the current execution position, function, line number, and source statement are displayed. Next any commands associated with the breakpoint are executed. The **Q** command can be used to suppress the output from the **bcyc** command.

Example

To set a cycle count breakpoint after 4 clock cycles and list all global variables, type:

```
4 bcyc {l g}
```



b, **bCYC**, **bINST**, **binst**, **bTIM**, **btim**, **D**

bD

Function

Set a read and/or write data breakpoint over a range of addresses.



Select the **Debug | Breakpoints...** menu item to view the Breakpoints dialog box. Select the **New Data...** button to create a new Data breakpoint.



The command line syntax is:

```
exp1 bD { r | w | b } exp2 [cmds]
```

Description

Set a read, write, or both (read and write) data breakpoint in the address range *exp1* to *exp2* and associate the list of CrossView Pro commands *cmds* with the breakpoint.

When the breakpoint is hit, execution is halted. By default the current execution position, function, line number, and source statement are displayed. Next, any commands associated with the breakpoint are executed. The **Q** command can be used to suppress the output from the **bD** command.

If *exp1* is the address of a local (stack) variable, the function in which it was declared must be currently active on the stack. If the local variable corresponding to a data breakpoint goes out of scope due to a return from the function in which it is currently active, the data breakpoint will be removed and a message will be printed telling the user that the variable is no longer active.



Not available for all execution environments.

Example

To set a data breakpoint that includes the entire structure `rec1`, type:

```
&rec1 bD r (int)&rec1+sizeof(rec1)-1
```

This breakpoint will be hit only if any address in the range of addresses is read from.

To set a data breakpoint for the address range 10 to 10f hex (256 bytes) that will list all global variables, type:

```
0x10 bD b 0x10f {l g;}
```

This breakpoint will be hit if any memory locations within the range 10-10f hex are either read from or written to.



b, bb, bB, bd, bi, bI, bt, bti, btiI, bu, bU, Q

bd

Function

Set a read and/or write data breakpoint at an address.



Select the **Debug | Breakpoints...** menu item to view the Breakpoints dialog box. Select the **New Data...** button to create a new Data breakpoint.



The command line syntax is:

```
exp bd { r | w | b } [cmds]
```

Description

Set a read, write or both (read and write) data breakpoint at the address specified by *exp* and associate the list of CrossView Pro commands *cmds* with the breakpoint.

When the breakpoint is hit, execution is halted. By default the current execution position, function, line number, and source statement are displayed. Next any commands associated with the breakpoint are executed. The **Q** command can be used to suppress the output from the **bd** command.

If *exp* corresponds to a local (stack) variable, the function in which it was declared must be currently active on the stack. If the local variable corresponding to a data breakpoint goes out of scope due to a return from the function in which it is currently active, the data breakpoint will be removed and a message will be printed telling you that the variable is no longer active.



Not available for all execution environments.

Example

To set a breakpoint at the variable count which will all be hit only if the variable is read from memory, type:

```
&count bd r
```



Note that the breakpoint only acts on the lowest byte in memory of this variable.

To set a breakpoint at address 10 hex that will list all global variables, type:

```
0x10 bd b {1 g}
```

This breakpoint will be hit if address 10 hex is either read from or written to.



b, bb, bB, bD, bi, bI, bt, bti, bti, bu, bU, Q

bdis

Function

Disable code breakpoint.



Select the **Debug | Breakpoint...** menu item, select the breakpoint you want to disable, click on the **Edit...** button, disable the **Enabled** check box.



The command line syntax is:

number **bdis**

Description

Disable the code breakpoint associated with the given *number*.

This does not delete the code breakpoint. It disables the code breakpoint until you enable it again with the **bena** command.

This command does not work on data breakpoints, only on code breakpoints

Example

To disable code breakpoint number 3, type:

3 bdis



b, bena, D

bena

Function

Enable code breakpoint.



Select the **Debug | Breakpoint...** menu item, select the breakpoint you want to disable, click on the **Edit...** button, set the **Enabled** check box.



The command line syntax is:

number **bena**

Description

Enable the code breakpoint associated with the given *numbe.*, which was previously disabled by the **bdis** command.

This command does not work on data breakpoints, only on code breakpoints

Example

To enable code breakpoint number 3, type:

3 bena



b, bdis, D

bl

Function

Set a temporary low-level breakpoint at a machine instruction.



Select the **Debug | Breakpoints...** menu item to view the Breakpoints dialog box. Select the **New Code...** button to create a new code breakpoint. Select the **Temporary** radio button, edit the **Address** field, leave the **Task ID** field empty or enter the string "any" and select **Apply**.



The command line syntax is:

```
[addr] bl [cmds]
```

Description

Set a temporary breakpoint at the machine instruction at address *addr*, or the current viewing position's address if *addr* is not specified; the list of CrossView Pro commands *cmds* are executed when the breakpoint is hit.

Make sure that *addr* is the start address of a machine instruction, otherwise the results are unpredictable. When the breakpoint is hit execution is halted. By default the current execution position, function, line number, and source statement are displayed. Next any commands associated with the breakpoint are executed. The **Q** command can be used to suppress the output from the **bl** command.

Example

To set a temporary breakpoint at the current viewing position's address, type:

```
bl
```

To set a temporary breakpoint at address 100 that will print the addresses of the next five source statements, type:

```
100 bl {P 5}
```



b, **bb**, **bB**, **bd**, **bD**, **bi**, **bt**, **bti**, **btI**, **bu**, **bU**, **Q**

bi

Function

Set a permanent low-level breakpoint at a machine instruction.



Select the **Debug | Breakpoints...** menu item to view the Breakpoints dialog box. Select the **New Code...** button to create a new code breakpoint. Select the **Permanent** radio button, edit the **Address** field, leave the **Task ID** field empty or enter the string "any" and select **Apply**.

Alternatively, you can place a breakpoint in the intermixed window or assembly window by double clicking on the desired instruction.



The command line syntax is:

```
[addr] bi [cmds]
```

Description

Set a permanent breakpoint at the machine instruction at address *addr*, or the current viewing position's address if *addr* is not specified; the list of CrossView Pro commands *cmds* are executed when the breakpoint is hit.

Make sure that *addr* is the start address of a machine instruction, otherwise the results are unpredictable. When the breakpoint is hit execution is halted. By default the current execution position, function, line number, and source statement are displayed. Next any commands associated with the breakpoint are executed. The **Q** command can be used to suppress the output from the **bi** command.

Example

To set a breakpoint at the current viewing position's address, type:

```
bi
```

To set a breakpoint at address 100 that will print the addresses of the next five source statements, type:

```
100 bi {P 5}
```



b, **bb**, **bB**, **bd**, **bD**, **bl**, **bt**, **bti**, **btI**, **bu**, **bU**, **Q**

bINST

Function

Set a temporary instruction count breakpoint.



There is no mouse equivalent for this command.



The command line syntax is:

```
count bINST [cmds]
```

Description

Set a temporary breakpoint after the specified *count* number of machine instructions have been executed. *count* can be any expression evaluating to a number. The list of CrossView Pro commands *cmds* are executed when the breakpoint is hit.

When the breakpoint is hit, execution is halted; the breakpoint is then removed. By default the current execution position, function, line number, and source statement are displayed. Next any commands associated with the breakpoint are executed. The **Q** command can be used to suppress the output from the **bINST** command.

Example

To set a temporary breakpoint after execution of 5 instructions and list all global variables, type:

```
5 bINST {1 g}
```



b, **bCYC**, **bcyc**, **binst**, **bTIM**, **btim**, **D**

binst

Function

Set a permanent instruction count breakpoint.



There is no mouse equivalent for this command.



The command line syntax is:

```
count binst [cmds]
```

Description

Set a permanent breakpoint after the specified *count* number of machine instructions have been executed. *count* can be any expression evaluating to a number. The list of CrossView Pro commands *cmds* are executed when the breakpoint is hit.

When the breakpoint is hit, execution is halted. By default the current execution position, function, line number, and source statement are displayed. Next any commands associated with the breakpoint are executed. The **Q** command can be used to suppress the output from the **binst** command.

Example

To set a permanent breakpoint after execution of 5 instructions and list all global variables, type:

```
5 binst {1 g}
```



b, **bCYC**, **bccyc**, **binST**, **bTIM**, **btim**, **D**

bt

Function

Set a task aware code breakpoint.



Select the **Debug | Breakpoints...** menu item to view the Breakpoints dialog box. Select the **New Code...** button to create a new code breakpoint and fill in the **Task ID** field.



The command line syntax is:

```
[line] bt "TaskId" [cmds]
```

Description

Set a task aware code breakpoint at the specified source *line* and associate the list of CrossView Pro commands *cmds* with the breakpoint. If no line is given, set the breakpoint at the current viewing position. The *TaskId* is the identification of the task as displayed in the Tasks Window or specified by the **lk** command.

When the breakpoint is hit, execution is halted. By default the current execution position, function, line number, and source statement are displayed. Next, any commands associated with the breakpoint are executed. The **Q** command can be used to suppress the output from the **bt** command.

Example

To set a breakpoint for task 4 at the current viewing position, type:

```
bt "4"
```

To set a breakpoint for task 4 at line 10, which lists all global variables, type:

```
10 bt "4" {l g}
```



b, bb, bB, bd, bD, bi, bI, bti, btl, bu, bU, l, Q

btl

Function

Set a temporary low-level task aware breakpoint at a machine instruction.



Select the **Debug | Breakpoints...** menu item to view the Breakpoints dialog box. Select the **New Code...** button to create a new code breakpoint and fill in the **Task ID** field. Select the **Temporary** radio button, edit the **Address** field and select **Apply**.



The command line syntax is:

```
[addr] btl "TaskId" [cmds]
```

Description

Set a temporary task aware breakpoint at the machine instruction at address *addr*, or the current viewing position's address if *addr* is not specified; the list of CrossView Pro commands *cmds* are executed when the breakpoint is hit. The *TaskId* is the identification of the task as displayed in the **Tasks Window** or specified by the **Ik** command.

Make sure that *addr* is the start address of a machine instruction, otherwise the results are unpredictable. When the breakpoint is hit execution is halted. By default the current execution position, function, line number, and source statement are displayed. Next any commands associated with the breakpoint are executed. The **Q** command can be used to suppress the output from the **btl** command.

Example

To set a temporary breakpoint for task 4 at the current viewing position's address, type:

```
btl "4"
```

To set a temporary breakpoint for task 4 at address 0xF00 and print the message, type:

```
0xF00 btl "4" {"breakpoint triggered:  
address 0xF00, task 4"}
```



b, bb, bB, bd, bD, bi, bI, bt, bti, bu, bU, l, Q

bti

Function

Set a permanent low-level task aware breakpoint at a machine instruction.



Select the **Debug | Breakpoints...** menu item to view the Breakpoints dialog box. Select the **New Code...** button to create a new code breakpoint and fill in the **Task ID** field. Select the **Permanent** radio button, edit the **Address** field and select **Apply**.



The command line syntax is:

```
[addr] bti "TaskId" [cmds]
```

Description

Set a permanent task aware breakpoint at the machine instruction at address *addr*, or the current viewing position's address if *addr* is not specified; the list of CrossView Pro commands *cmds* are executed when the breakpoint is hit. The *TaskId* is the identification of the task as displayed in the Tasks Window or specified by the **Ik** command.

Make sure that *addr* is the start address of a machine instruction, otherwise the results are unpredictable. When the breakpoint is hit execution is halted. By default the current execution position, function, line number, and source statement are displayed. Next any commands associated with the breakpoint are executed. The **Q** command can be used to suppress the output from the **bti** command.

Example

To set a breakpoint for task 4 at the current viewing position's address, type:

```
bti "4"
```

To set a breakpoint for task 4 at address 0xF00 and print the message, type:

```
0xF00 bti "4" {"breakpoint triggered:  
address 0xF00, task 4"}
```



b, bb, bB, bd, bD, bi, bI, bt, bti, bu, bU, l, Q

bTIM

Function

Set a temporary time breakpoint.



There is no mouse equivalent for this command.



The command line syntax is:

time **bTIM** [*cmds*]

Description

Set a temporary breakpoint after the specified *time* (in seconds). *time* can be any expression evaluating to a number. The list of CrossView Pro commands *cmds* are executed when the breakpoint is hit.

When the breakpoint is hit, execution is halted; the breakpoint is then removed. By default the current execution position, function, line number, and source statement are displayed. Next any commands associated with the breakpoint are executed. The **Q** command can be used to suppress the output from the **bTIM** command.

Example

To set a temporary breakpoint after 0.5 seconds and list all global variables, type:

0.5 bTIM {1 g}



b, bCYC, bcyc, bINST, binst, btim, D

btim

Function

Set a permanent time breakpoint.



There is no mouse equivalent for this command.



The command line syntax is:

time **btim** [*cmds*]

Description

Set a permanent breakpoint after the specified *time* (in seconds). *time* can be any expression evaluating to a number. The list of CrossView Pro commands *cmds* are executed when the breakpoint is hit.

When the breakpoint is hit, execution is halted. By default the current execution position, function, line number, and source statement are displayed. Next any commands associated with the breakpoint are executed. The **Q** command can be used to suppress the output from the **btim** command.

Example

0.5 **btim** {1 **g**}



b, **bCYC**, **bcyc**, **bINST**, **binst**, **btim**, **D**

bU

Function

Set a temporary up-level breakpoint.



In the Stack Window, double-click on the desired function. Alternately, you can click on the desired function in the Stack Window and select the Debug | Stack Breakpoint | After Function Call menu item.



The command line syntax is:

```
[ stack ] bU [ commands ]
```

Description

This command sets a temporary up-level breakpoint immediately after the return to the function designated by the stack number *stack*, otherwise the currently viewed function is used. Associate the list of CrossView Pro commands *commands* with the breakpoint.

When the breakpoint is hit execution is halted. By default the current execution position, function, line number, and source statement are displayed. Next any commands associated with the breakpoint are executed. The **Q** command can be used to suppress the output from the **bU** command.

Breakpoints set in the Stack Window are always temporary, meaning they will be deleted after the first time you reach them. A breakpoint set in this manner will not be visible in the Source Window.

This command is not allowed when the target runs in the background.

Example

To set a temporary up-level breakpoint immediately after the return from the currently viewed function, type:

```
bU
```

To set a temporary up-level breakpoint immediately after the return from the function at stack level 2, type:

```
2 bU {1}
```

After stopping, this command will cause CrossView Pro to print out the function's local variables and arguments.



b, bb, bB, bd, bD, bi, bI, bt, bti, bti, bu, Q

bu

Function

Set a permanent up-level breakpoint.



In the Stack Window you can click on the desired function and select the Debug | Stack Breakpoint | After Function Call menu item. To make the stack breakpoint permanent, select the Debug | Breakpoints... menu item, select the desired stack breakpoint, click on the Edit... button and select the Permanent radio button.



The command line syntax is:

```
[ stack ] bu [ commands ]
```

Description

Set a permanent up-level breakpoint immediately after the return to the function designated by the stack number *stack*, otherwise the currently viewed function is used. Associate the list of CrossView Pro commands *commands* with the breakpoint.

When the breakpoint is hit execution is halted. By default the current execution position, function, line number, and source statement are displayed. Next any commands associated with the breakpoint are executed. The **Q** command can be used to suppress the output from the **bu** command.

This command is not allowed when the target runs in the background.

Example

To set an up-level breakpoint at the beginning of the currently viewed function, type:

```
bu
```

To set an up-level breakpoint at beginning of function whose stack number is 2 and, after stopping, print out the local variables and arguments of that function, type:

```
2 bu {l}
```



b, bb, bB, bd, bD, bi, bI, bt, bti, bti, bU, Q

C

Function

Continue using the current value of the program counter.



In the Source Window, click on the **Continue** execution accelerator button. You can also select the **Run | Run** menu item.



The command line syntax is:

```
[ exp ] C [ line ]
```

Description

If *exp* is specified and you are stopped at a breakpoint, then the breakpoint count is set to this value. If *line* is specified, a temporary breakpoint is set at that line number. Note that this temporary breakpoint will overwrite any existing breakpoint at that line.

The **C** command can be used in the command lists of breakpoints to resume execution automatically.

This command is not allowed when the target runs in the background.

Example

To continue execution from the current target program counter, type:

```
C
```

To set the breakpoint's count to 4 and continue, type:

```
4 C
```

To set a temporary breakpoint at line 52 and continue, type:

```
C 52
```



bc, g, R, CB

CB

Function

Continue execution in background using the current value of the target program counter.



There is no mouse equivalent for this command.



The command line syntax is:

[*exp*] **CB** [*line*]

Description

If *exp* is specified and you are stopped at a breakpoint, then the breakpoint count is set to this value. If *line* is specified, a temporary breakpoint is set at that line number. Note that this temporary breakpoint will overwrite any existing breakpoint at that line.

The **CB** command can be used in the command lists of breakpoints to resume execution automatically.

This command is not allowed when the target runs in the background.

Not available for all execution environments.

Example

To continue execution from the current target program counter, type:

CB

To set the breakpoint's count to 4 and continue, type:

4 CB

To set a temporary breakpoint at line 52 and continue, type:

CB 52



g, R, C, st, wt

cd

Function

Disable, turn off, gathering of coverage data.



Select the Run | Coverage menu item if this item was set.



The command line syntax is:

cd

Description

If coverage is supported by your version of CrossView Pro, this command disables the coverage system. Normally, you should disable coverage if you are not interested in the coverage results, as this will often improve the performance of the execution environment.

Example

To disable coverage, type:

cd



ce, nC, nU, pC, pU

ce

Function

Enable, turn on, gathering of coverage data.



Select the Run | Coverage menu item if this item was not set.



The command line syntax is:

ce

Description

If coverage is supported by your version of CrossView Pro, this command enables the coverage system. Normally, you should disable coverage if you are not interested in the coverage results, as this will often improve the performance of the execution environment.

Example

To enable coverage, type:

ce



cd, nC, nU, pC, pU

ct

Function

Display a C-execution trace.



Select the **View | Trace | Source Level** menu item. The Trace Window displays the most recently executed lines of code every time program execution is stopped. CrossView Pro automatically updates the Trace Window each time execution is halted, as long as the window is open.



The command line syntax is:

number **ct**

Description

Display a C-execution trace in the Command window, corresponding to the last *number* of machine instructions executed. Since the **ct** command relies on the emulator's trace buffer, the **ct** command will not be implemented on some emulators.

For each executed line of code, the Trace Window displays:

- The name of the source file
- The name of the function
- The line number and corresponding source code

The window shows all the code executed since the the last time the program halted.

This command is not allowed when the target runs in the background.



Not available for all execution environments.

Example

To display, in the Command window, the last C statements (corresponding to the last ten machine instructions) executed, type:

10 ct



ct i, ct r

ct i

Function

Display a disassembled trace.



Select the View | Trace | Instruction Level menu item. The Trace Window displays the most recently executed lines of code every time program execution is stopped. CrossView Pro automatically updates the Trace Window each time execution is halted, as long as the window is open.



The command line syntax is:

number **ct i**

Description

Display a disassembled trace in the Command window, corresponding to the last *number* of machine instructions executed.

Since the **ct i** command relies on the emulator's trace buffer, the **ct i** command will not be implemented on some emulators.

This command is not allowed when the target runs in the background.



Not available for all execution environments.

Example

To display in the Command window the last 20 disassembled instructions executed, type:

20 ct i



ct, ct r

ct r

Function

Display a raw trace.



Select the **View | Trace | Raw** menu item. The Trace Window displays the most recently executed lines of code every time program execution is stopped. CrossView Pro automatically updates the Trace Window each time execution is halted, as long as the window is open.



The command line syntax is:

number **ct r**

Description

Display a raw trace in the Command window, corresponding to the last *number* of trace frames. This command merely shows the contents of the emulator's trace buffer.

Since the **ct r** command relies on the emulator's trace buffer, the **ct r** command will not be implemented on some emulators.

This command is not allowed when the target runs in the background.



Not available for all execution environments.

Example

To display in the Command window the last 20 trace frames, type:

20 ct r



ct, ct i

D

Function

Delete all currently defined breakpoints.



Select the `Debug | Breakpoints...` menu item to view the Breakpoints dialog box. This box contains a delete function.



The command line syntax is:

D[y]

Description

D deletes all currently defined breakpoints. **Dy** does not ask for confirmation.



B, d

d

Function

Delete a specific breakpoint.



To delete a code breakpoint directly from the C source, simply double-click the mouse on the corresponding, highlighted source line in the Source Window. The code breakpoint will be deleted and the line will cease to be highlighted.

Otherwise, select the `Debug | Breakpoints...` menu item to view the Breakpoints dialog box. This box contains a delete function.



The command line syntax is:

[*number*] **d**

Description

Delete the breakpoint associated with the given *number*. If no number is given, delete the breakpoint at the current line. If there is no breakpoint at the current line, a **B** command will be executed to display all breakpoints.

Whenever a breakpoint is deleted the remaining breakpoints are renumbered starting at 0.

Example

To delete a breakpoint at the current line, type:

d

To delete breakpoint number 3, type:

3 d



b, bb, bB, bd, bD, bi, bI, bt, bti, btI, bu, bU, B, D

dis

Function

Disassemble a range of memory.



Select the View | Source | Disassembly or View | Source | Source and Disassembly menu item to open the Disassembly or Source and Disassembly window respectively.



The command line syntax is:

```
dis address [, {address | #count} [,i]]
```

Description

Disassemble a range of memory. The output is interleaved with source lines when **i** is specified. You can enter valid expressions as well for *address* and *count*.

Example

To disassemble 4 instructions starting at 3 bytes behind the start address of the function `main`., type:

```
dis main+3,#4
```

To disassemble memory for (`initval+1`) instructions, starting at the address of the function `main`., type:

```
dis main+3,#initval+1
```

To disassemble from `0x2000` up to and including the instruction at `0x2100` and also interleave C source lines of any function resident in that memory range, type:

```
dis 0x2000,0x2100,i
```



dump, *expression*

dn

Function

Download a file



Select the **File | Download Image** menu item to download the image part of the file to the execution environment.



The command line syntax is:

```
dn [file]
```

Description

Download the image part of the specified *file* to the execution environment. If no *file* is specified, use the file specified when CrossView Pro was invoked, and from which the symbolic information was read during startup, or the file specified in either the **N** command or the Load Symbolic Debug Info dialog.

Downloading a file only copies an image part into target memory. It will not cause CrossView Pro to re-read symbolic information.

This command is not allowed when the target runs in the background.

Example

To download the current file, type

```
dn
```

To download the IEEE file `demo.abs`, type:

```
dn demo.abs
```

To download the hex file `test.hex`, type:

```
dn test.hex
```



I, N

dump

Function

Dump a range of memory.



Select the View | Memory menu item to open the Memory Window.



The command line syntax is:

```
dump address [, [address | #count] [, [style [width ] ] [, filename [a]]]
```

Description

The **dump** command can dump memory as hexadecimal data or as C variables. You can enter valid C expressions as well for *address* and *count*. You can also dump Motorola S records or Intel hex records. Also, you can specify a *filename* in which the dump is to be written or appended.

style can be one of:

```
a c D O U X d o u x E F G e f g n P p R r s t I M
```

Style **I** dumps Intel hex and style **M** specifies Motorola S records output. See the *Accessing Code and Data* chapter and the section *Formatting Expressions* in the chapter *Command Language* for details on each of the other format styles. The **R** and **r** style are only available for targets that support the fractional type.



Mind the following:

- the commas are required
- the addresses can also be C expressions
- default width is MAU (usually byte) sized words
- additional style **M**: Motorola S records
- additional style **I**: Intel hex
- a semicolon is a command terminator
- the **dump** is end address INclusive

Example

To dump the first byte of the function `main.`, type:

```
dump main
```

To dump the first 10 bytes of the function main as Motorola S records in the file main.sre, type:

```
dump main,main+10,M,main.sre
```

To dump the first 5 bytes of the function main. as 1 string, type:

```
dump main,main+10,M,main.sre,a
```

To append the first 5 bytes of the function main. as 1 string, type:

```
dump main,,c5
```

To dump the resulting value bytes of 'the address of main binary anded with 3', type:

```
dump main+1,#main&3
```



dis, *expression*

e

Function

Establish viewing position



Select the **File | Open Source...** menu item to view a file. In the Source Window, click on the **Find Function** button to find a function, or select the **Search | Browse Function...** menu item.

In the Stack Window click once on the function to be examined.



The command line syntax is:

```
e [file | function ]  
stack e
```

Description

The **e** option invokes two distinct commands. The first version establishes the viewing position to be the first line of *file*, the first executable line of the function *function* or the current viewing position if no argument is given.

The second version establishes the viewing position to be the line at stack level *stack* in the stack trace. (See the **t** command.)

The *stack* **e** command is not allowed when the target runs in the background.

The **L** command is equivalent to **0 e**.

Example

To view the function `main`, type:

```
e main
```

To view the test file `test.c`, type:

```
e test.c
```

To view the call site of the current function, type:

```
0 e
```

To view the line at stack level 3, type:

3 e



?, /, ei, L, p, P, t

eC

Function

Start execution on current CPU and switch to another CPU.



The command line syntax is:

cpu_number **eC**

Description

Start execution on the current CPU and switch to CPU *cpu_number*.

This command can only be issued when the currently selected CPU is in debug mode.

Example

To start execution on the current CPU and select the CPU indicated by number 1, type:

1 eC



ec

ec

Function

Select a CPU or show current CPU number.



The command line syntax is:

```
[cpu_number] ec
```

Description

The **ec** command allows you to select a CPU in your current Execution Environment if your target has multi-CPU support.

This command can only be issued when the currently selected CPU is in debug mode.

Example

To view the current CPU selection, type:

```
ec
```

To select the CPU indicated by number 1, type:

```
1 ec
```



eC

echo

Function

Display the definition of a macro name without executing the macro.



You can view the definition of a macro by selecting the Options | Macro Definitions... menu item to view the Macro Definitions dialog box.



The command line syntax is:

echo *text*

Description

Perform macro expansion on *text* without executing. This allows you to see how a macro is expanded. It is particularly informative when macros call other macros.

Example

If you type:

echo macro(3)

CrossView Pro will display the expansion of `macro(3)`.



set, unset, save, !

ei

Function

Establish viewing position at a specified address.



Select the Search | Find Address... menu item.



The command line syntax is:

```
[addr] ei
```

Description

The **ei** command establishes the viewing position to be at the instruction specified.

This command is useful for viewing some code in the assembly window, without changing the program counter, since the execution position is not changed.

Example

To view the current viewing position, type:

```
ei
```

To view the instruction at address 0x100, type:

```
0x100 ei
```



?, /, e, L, p, P, t

et

Function

Select the specified task's context.



In the Tasks Window click once on the task to be examined.



The command line syntax is:

```
et "TaskId"
```

Description

Select the specified task's context. The *TaskId* is the identification of the task as displayed in the Tasks Window or specified by the **lk** command.

The current execution position, function, line number, and source statement are displayed. All other windows, except for the Kernel Windows, are updated accordingly.

Subsequent CrossView Pro commands use the context of the selected task. For example, the **t** command shows a stack trace of the selected task.

Example

To select task 4, type:

```
et "4"
```



1

f

Function

Set default address printing format



There is no mouse equivalent for this command.



The command line syntax is:

```
f [ " printf-style-format " ]
```

Description

Set the default address printing format, using a `printf` format specification.

If there is no argument, the format defaults to `%x`, which prints an address in hexadecimal.

This command is intended to allow users to see memory addresses in decimal, octal or a format of their choosing.

Example

To display addresses in octal, type:

```
f "%o"
```

To display addresses in hex, type:

```
f
```



expression

g

Function

Change the program counter to a new execution position.



Click on a source line and select the Run | Jump to Cursor menu item.



The command line syntax is:

g *line*

Description

This command changes the program counter so that *line* becomes the current execution position. *Line* must be a line in the current function.

This command changes only the program counter. It does not cause the target to begin execution.

Exercise caution when changing the execution position. Oftentimes, each line of C source code is compiled into several machine language instructions. Moving the program counter to a new address in the middle of a series of related assembly instructions is sometimes risky. Moreover, even though you change the program counter, registers and variables may not have the expected values if parts of the code are bypassed.

This command is not allowed when the target runs in the background.

Example

To change the program counter so that the next instruction to be executed corresponds to line 127, type:

g 127



C, gi, R

gi

Function

Change the program counter to a new execution position.



Click on a source line and select the Run | Jump to Cursor menu item.



The command line syntax is:

address **gi**

Description

This command changes the program counter so that *address* becomes the current execution position.

This command changes only the program counter. It does not cause the target to begin execution.

Exercise caution when changing the execution position. The Jump to Cursor menu item is not available in the source lines window mode to prevent problems by skipping pieces of C code which are required to be executed. Moving the program counter to a new address in the middle of a series of related assembly instructions is sometimes risky. Moreover, even though you change the program counter, registers and variables may not have the expected values if parts of the code are bypassed.

This command is not allowed when the target runs in the background.

Example

To change the program counter so that the next instruction to be executed corresponds to address 0x0800, type:

0x0800 gi



C, g, R



Function

Print out information about the state of CrossView Pro.



There is no mouse equivalent for this command.



The command line syntax is:

I

Description

Print out information about the state of CrossView Pro, including: the CrossView Pro version number, the execution environment version information, the name of the program being debugged (and the number of its files and functions), the state of the assertion mechanism, the state of output recording, the state of command recording, the state of target communication recording and the state of search case sensitivity.

The state of the assertion mechanism tells how many assertions have been defined and whether the overall assertion mechanism is active or suspended; it does not tell whether any individual assertions are active or suspended.



I, a, A, >, >>, >&, Z

if

Function

Conditional command execution.



There is no mouse equivalent for this function.



The command line syntax is:

```
if ( expression ) { cmds } [ { cmds } ]
```

Description

If *expression* evaluates to a non-zero value, execute the first group of commands. Otherwise, the second group of commands, if present, will be executed. This command is nestable.

Leave a space between **if** and *exp*. `if(a==b)` parses as a function call. The **if** statement is used primarily within breakpoint command lists.

Example

If you type:

```
if (a=b) {5t} {C}
```

CrossView Pro will trace back five levels on the stack if a is equal to b. Otherwise, CrossView Pro will continue.

The command line:

```
if (wait>1000) {wait;l r}
```

will print the value of `wait` and list all registers if the value of `wait` exceeds 1000.

L

Function

Synchronize the viewing and execution positions.



To synchronize the positions manually, click on the `Synchronize source` accelerator button in the Source Window or select the `Run | Synchronize Source` menu item.



The command line syntax is:

L

Description

This command synchronizes the viewing and execution positions. It also lists the current file, function and line number of the current program counter. The viewing position is always moved to match the execution position.

The **L** command is synonymous with a **0e** command and does not affect the execution position.



This command is not allowed when the target runs in the background.

Example

To synchronize the viewing and execution positions, then list current file, function, and line number, type:

L



e, l

Function

List.



There are multiple mouse equivalents for this command. Generally speaking, the dialog box in which you define a feature also maintains a list.



The command line syntax is:

```
l { a | b | d | f | g | k | l | L | m | p | r | s | S } [string]  
l [func]  
l stack
```

Description

In the first case above, list one of the following: **a**ssertions, **b**reakpoints, **d**irectories, **f**iles, **g**lobals, **k**ernel state data, **l**abels (on module scope), all **L**abels, **m**emory map (of application code sections), **p**rocedures, **r**egisters, **s**pecial variables, **S**ymbol tables. If *string* is present, then list only those items that start with *string*.

In the second case, list the values of all parameters and locals of the function *func*. Without a function, this command lists all parameters and locals of the current function in view.

In the third case, list all parameters and locals of the function at depth *stack*.

The **lf** and **lm** commands also show the address of the modules' first procedure. The **lm** command is identical to **lf**, list files, but the list of files is sorted on ascending segment addresses. *func* must be a function on the stack or the current function.

For configurations that support real-time kernels, the **lk** command can have one of the following arguments (**lk** is the same as specifying **lk t**):

- t** – Display tasks.
- m** – Display mailboxes.
- q** – Display queues.
- p** – Display pipes.
- s** – Display semaphores.
- e** – Display events.
- h** – Display HISRs (High-level Interrupt Service Routines)
- si** – Display signals.
- ti** – Display timers.
- pm** – Display partition memory.
- dm** – Display dynamic memory.
- r** – Display resources.
- misc** – Display miscellaneous information.

Example

To list defined assertions and the state of the assertion mechanism, type:

```
l a
```

To list all locals and parameters of the current function, type:

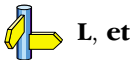
```
l p
```

Data is displayed using the normal (/n) format. To list all the parameters and locals of the function fc_n, type:

```
l fcn
```

To list queue information for the current tasks (only if your configuration supports it), type:

```
l k q
```



L, et

load

Function

Load a program's symbol file and download the image part.



Select the **File | Load Symbolic Debug Info...** menu item. This dialog allows you to specify the file.



The command syntax is:

load [*filename*]

Description

This command performs the **N** and **dn** commands sucessively.

Downloading a file only copies the image part into target memory (**dn**). It will not cause CrossView Pro to re-read symbolic information (**N**). The **load** command does both.



This command is not allowed when the target runs in the background.

Example

To load the symbol table of file `demo.abs` in CrossView Pro and to download the image part, type:

load demo.abs



dn, N

M

Function

List the data currently being monitored.



Refer to the Data Window. Each time the program stops, the debugger evaluates all monitored expressions and displays the results in the Data Window.



The command line syntax is:

M

Description

List all C expressions being monitored by CrossView Pro. The listing associates a unique number with each expression. This number is used to specify the deletion of monitored data.



m

m

Function

Monitor (watch) an expression. (Also delete a monitor.)



From the Source Window, double-click on an expression. A new monitor is created in the Data Window or the Expression Evaluation dialog is opened if the `Bypass Dialog` check box in the Data Display Setup dialog is not set. If the latter is the case, click on the `Watch` button to create a new monitor in the Data Window. To remove an existing monitor, select the monitor in the Data Window and click on the `Delete selected item` button.



The command syntax is:

```
m exp
number m d
```

Description

The **m** command has two distinct functions. The first monitors the given expression. The second deletes the monitoring of the expression specified by *number*.

Data monitoring takes place whenever the program stops execution, that is, for a breakpoint, assertion, single step, or user interrupt (*ctrl-C*). In window mode, the values of all currently monitored data are displayed in the Data window. Each piece of monitored data has a unique identifying number that is used when deleting it.

Example

To monitor the value of the variable `myvar`, type:

```
m myvar
```

To monitor the address of variable `myvar`, type:

```
m &myvar
```

To monitor the element `alpha+1` of `array`, type:

```
m array[alpha+1]
```

To delete expression number 2 of the monitored data, type:

2 m d



M, b, a, s, R, C

mcp

Function

Memory copy.



From the Memory Window, click on the Copy memory button to open the Memory Copy dialog. Enter the start address and end address (inclusive) of the memory region you want to copy. Enter the destination address and click on the OK button.



The command syntax is:

```
addr_start mcp addr_end, addr_dest
```

Description

The **mcp** command copies a block of target memory starting at address *addr_start* to destination address *addr_dest*. The size of the memory block is defined as: '*addr_end* - *addr_start* + 1'. The data item located at address *addr_end* is included in the copy.

If your target supports multiple memory spaces then it is legal to copy data between different memory spaces. Of course *addr_start* and *addr_end* must be located in the same memory space. This command does not have any effect on code breakpoints.

Example

To copy the contents of variable `buf` to address `0x200`, type:

```
&buf mcp &buf+sizeof(buf), 0x200
```



mF, mf

mF

Function

Memory single fill.



From the Memory Window, click on the **Single Fill** memory button to open the Memory Single Fill dialog. Enter the start address the memory region you want to fill. Enter one or more expressions separated by commas and click on the OK button.



The command syntax is:

addr **mF** *expr* [*expr*]...

Description

The **mF** command fills target memory with data. The value defined by *exp* is written to address *addr* in target memory. Multiple *exprs* separated by commas may be entered. Each *exp* is written to a subsequent MAU.

If your target supports multiple memory spaces then *addr* may refer to any memory space.

If the sizeof a given *exp* occupies more than one MAU, only the least significant MAU will be written to memory. This command does not have any effect on code breakpoints.

Example

To store value 0x12 at memory location 0x400 and value 0xAB at location 0x401, type:

0x400 mF 0x12, 0xAB



mcp, mf

mf

Function

Memory fill, repeating the specified pattern until the specified region is filled.



From the Memory Window, click on the **Fill** memory button to open the Memory Fill dialog. Enter the start address and end address (inclusive) of the memory region you want to fill. Enter one or more expressions separated by commas and click on the OK button.



The command syntax is:

```
addr_start mf addr_end, expr [,expr]...
```

Description

The **mf** command fills a block of target memory with a pattern. The memory region starting at address *addr_start* and ending at address *addr_end* is filled with the pattern defined by *exp* [*,exp*]. Multiple *exprs* separated by commas may be entered. Each *exp* is written to a subsequent MAU.

The specified pattern is repeated until the end address of memory region is reached.

If your target supports multiple memory spaces then *addr* may refer to any memory space.

If the sizeof a given *exp* occupies more than one MAU, only the least significant MAU will be written to memory. This command does not have any effect on code breakpoints.

Example

To store values 0x01 and 0x02 at succeeding memory locations in the range 0x400 to 0x404, type:

```
0x400 mf 0x404, 0x01, 0x02
```

The result of this command is:

address:	0x400	0x401	0x402	0x403	0x404
value:	1	2	1	2	1



ms

Function

Memory search.



From the Memory Window, click on the Search memory button to open the Memory Search dialog. Enter the start address and end address (inclusive) of the memory region you want to search. Enter one or more search patterns separated by commas and click on the OK button.



The command syntax is:

```
addr_start ms addr_end, expr [expr]...
```

Description

The **ms** command searches for a pattern within a block of target memory. The memory region starting at address *addr_start* and ending at address *addr_end* (inclusive) is searched for the pattern defined by *exp* [*exp*]. Multiple *exp*s separated by commas may be entered. Each *exp* corresponds to a subsequent MAU.

If your target supports multiple memory spaces then *addr* may refer to any memory space.

This command does not have any effect on code breakpoints.

Example

Suppose the memory range 0x400 to 0x4ff was filled using the following commands:


```
0x400 mf 0x4ff, 0
0x400 mf 0x404, 1, 2
```

To search for the values 0x01 and 0x02 at memory locations in the range 0x400 to 0x4ff, type:

```
0x400 ms 0x4ff, 0x01, 0x02
```

The result of this command is:

```
FOUND pattern at 0x400
FOUND pattern at 0x402
```

 **mcp, mF, mf**

N

Function

Load a program's symbol file.



Select the **File | Load Symbolic Debug Info...** menu item. This menu item allows you to specify the file.



The command syntax is:

N *[[path]filename[.abs]]*

Description

Load the symbol table of the specified file in CrossView Pro. If no filename is given, the file being debugged is reloaded. In this case only the breakpoints set by the user are removed. Monitors, simulated I/O streams, assertions and CrossView Pro local variables remain active.

If a new file (different filename) is loaded, all breakpoints, monitors, simulated I/O streams, assertions and CrossView Pro local variables are removed.

If a path is supplied, CrossView Pro changes its current directory according to the specified path. In case a relative search path to source files was provided at startup time, CrossView Pro will search relative to the new working directory.

This command is automatically executed during CrossView Pro startup when a filename was given on the command line. Use the **dn** command to send the associated executable code to the target.

Example

To load the symbol table of file `demo.abs` in CrossView Pro, type:

N demo.abs



dn

n

Function

Set address bias



Select the File | Load Symbolic Debug Info... menu item. In the Load Symbolic Debug Info dialog you can edit the Code address bias field.



The command syntax is:

```
n [ addr ]
```

Description

Set address bias of overlay files to *addr*. If no address is given, then display current bias.

If a program is to be loaded at a different address than that indicated in the linked and located (absolute object) file, then the address information in the debugger's symbol file will be incomplete, since it does not know where the program is actually going to be loaded. This command will normalize the addresses by adding the bias to every address.

Example

To add a bias of 1000 to every address in the code, type:

```
n 1000
```

To display the current bias, type:

```
n
```


nC

Function

Set the viewing position to the next covered block of statements.



Use the scroll bar and click on the desired line.



The command line syntax is:

nC

Description

If code coverage is supported by your version of CrossView Pro, this command enables you to skip to the next block of statements that have been executed while the program was running on the target.

Example

To move the cursor to the next executed block, type:

nC



nU, pC, pU

nU

Function

Set the viewing position to the next not covered block of statements.



Use the scroll bar and click on the desired line.



The command line syntax is:

nU

Description

If code coverage is supported by your version of CrossView Pro, this command enables you to skip to the next block of statements that have not been executed while the program was running on the target.

Example

To move the cursor to the next not executed block, type:

nU



nC, pC, pU

O

Function

Enter emulator mode.



Select the View | Command | Emulator menu item. If you know the emulator-level command language, you can communicate directly with the emulator from this window.



The command line syntax is:

- o *string*

Description

Pass *string* to emulator and show the emulator response.

The **o** command lets you communicate with the emulator directly via emulator commands.

Do not issue one-shot transparency emulator commands that result in large output (or otherwise require intervention other than a carriage return to terminate output). Instead, enter transparency mode first, then issue the command.

Example

To send the string `map` to the emulator, type:

- o **map**

opt

Function

Set or display specific options.



Option values can be changed in the corresponding dialogs and menus.



The command line syntax is:

```
opt [ option_name [= option_value]
```

Description

If no arguments are passed, all options with their current value are listed. By specifying an option's name, the current value of that option is displayed. By specifying an option name followed by a valid value, the option is set to that new value.

The options are a sub-set of CrossView's so-called "special variables". See chapter *Command Language* for a list of all special variables.

Example

To display all options, type:

```
opt
```

To disable mixing of disassembly code and source lines in the assembly window, type:

```
opt mixedasm=off
```



1

P

Function

Print source lines, including machine addresses.



In the Source Window, the machine address of the line at the current viewing position is displayed in the **Address** field in the upper left corner.



The command line syntax is:

[*line*] **P** [*exp*]

Description

Print *exp* lines of source starting at line *line*, including machine addresses. If *exp* is omitted, print one line. If *line* is omitted, start from the current viewing position.

Example

To print source lines 4, 5, 6, 7 and 8 (displaying machine addresses) of the current source file, type:

4 P 5



P

p

Function

Print source lines.



C source is displayed in the Source Window.



The command line syntax is:

[*line*] **p** [*exp*]

Description

Print *exp* lines of source starting at line *line*. If *exp* is omitted, print one line. If *line* is omitted, start from the current viewing position.

Example

To print source lines 4, 5, 6, 7 and 8 of the current source file, type:

4 **p** 5



P

pC

Function

Set the viewing position to the previous covered block of statements.



Use the scroll bar and click on the desired line.



The command line syntax is:

pC

Description

If code coverage is supported by your version of CrossView Pro, this command enables you to skip to the previous block of statements that have been executed while the program was running on the target.

Example

To move the cursor to the previous executed block, type:

pC



nC, nU, pU

pd

Function

Disable, turn off, profiling.



Select the Run | Profiling menu item if this item was set.



The command line syntax is:

pd

Description

If profiling is supported by your version of CrossView Pro, this command disables the profiling system. Normally, you should disable profiling if you are not interested in the profiling results, as this will often improve the performance of the execution environment.

Example

To disable profiling, type:

pd



pe

pe

Function

Enable, turn on, profiling.



Select the Run | Profiling menu item if this item was not set.



The command line syntax is:

pe

Description

If profiling is supported by your version of CrossView Pro, this command enables the profiling system. Normally, you should disable profiling if you are not interested in the profiling results, as this will often improve the performance of the execution environment.

Example

To enable profiling, type:

pe



pd

proinfo

Function

List profiling results.



Select the Debug | Profiling Report... menu item, make your changes and select the Update button..



The command line syntax is:

proinfo

Description

If profiling is supported by your version of CrossView Pro and profiling is enabled, this command lists the profiling results. Normally, you should disable profiling if you are not interested in the profiling results, as this will often improve the performance of the execution environment.

Example

To list the profiling results, type:

```
pe  
proinfo
```



pd, pe

prst

Function

Reset the application being debugged to initial conditions. That is, set the program counter to the start address of the application.



Select the Run | Program Reset menu item.



The command line syntax is:

prst

Description

The program counter is set to the start address of the application being debugged. This command does NOT perform a hardware reset of the target system. That is, no registers are modified except for the program counter.



This command is not allowed when the target runs in the background.



R, rst

pU

Function

Set the viewing position to the previous not covered block of statements.



Use the scroll bar and click on the desired line.



The command line syntax is:

pU

Description

If code coverage is supported by your version of CrossView Pro, this command enables you to skip to the previous block of statements that have not been executed while the program was running on the target.

Example

To move the cursor to the previous not executed block, type:

pU



nC, nU, pC



Function

Quiet breakpoint reporting.



There is no mouse equivalent for this command.



The command line syntax is:

Q

Description

If this appears as the first command in a breakpoint's command list, the debugger does not make the usual announcement of:

function: line number: source file

when the breakpoint is hit.

The purpose of this command is to allow quiet breakpoint reporting. For example, to check the value of a variable without cluttering the screen with text.

Example

If you type the following:

```
21 b {Q; var1}
```

CrossView Pro will set a breakpoint at line 21. When that breakpoint is hit, CrossView Pro will print the value of `var1`, but will not print the current function, line number, and source file.



b

q

Function

Quit a debugging session.



Select the **File** | **Exit** menu item.



The command line syntax is:

q [**s** | **y**]

Description

CrossView Pro will prompt you if you really want to quit if you do not specify anything. Note that the current desktop settings are NOT saved then!

Typing **q s** saves the current desktop settings and quits the debugger without confirmation.

Typing **q y** does not save the current desktop settings and quits the debugger without confirmation.

Inside a command line procedure call it will just quit from this.

When the target runs in the background CrossView Pro will first stop the target.

R

Function

Reset program and begin execution from initial conditions.



Select the Run | Program Reset menu item followed by the Run | Run menu item.



The command line syntax is:

R

Description

Reset the application being debugged and begin execution from initial conditions. The program counter is set to the start address of the application being debugged. This command does NOT perform a hardware reset of the target system. That is, no registers are modified except for the program counter.



This command is not allowed when the target runs in the background.



C, g, prst

rst

Function

Reset target system to initial conditions.



Select the Run | Target System Reset menu item.



The command line syntax is:

rst

Description

The target is initialized according to the power-up sequence for the processor. Almost all registers, including the system stack pointer and program counter are initialized.



A target system reset may have undesired side effects. To be sure that the application code is correct, a download must be performed after a target system reset.



This command is not allowed when the target runs in the background.



R, prst

S

Function

Single step C statements, stepping over function calls.



To step *over* a function, click on the Step Over accelerator button in the Source Window. You can also select the Run | Step Over menu item. Verify the Run | Step Mode menu item; Source Step must be selected.



The command line syntax is:

[*exp*] S

Description

If you try to step over a call to a function which contains a breakpoint (or which calls another function with a breakpoint) then the breakpoint will be hit.

Stepping over a function means that CrossView Pro treats function calls as a single statement and advances to the next line in the source. This is a useful operation if a function has already been debugged or if you do not want to take the time to step through a function line by line.

When multiple statements are present on one line, they are all executed by this single step.



This command is not allowed when the target runs in the background.

Example

To step one C statement, type:

S

To step five C statements, type:

5 S



C, S, si, Si

S

Function

Single step C statements, stepping into function calls



To single step *into* a function, click on the Step Into accelerator button in the Source Window. You can also select the Run | Step Into menu item. Verify the Run | Step Mode menu item; Source line step must be selected.



The command line syntax is:

[*exp*] **s**

Description

Single step *exp* (default is 1), C statements, stepping into function calls.

Stepping into a function means that CrossView Pro enters the function and executes its prologue machine instructions halting at the first C statement. When the end of the function is reached, CrossView Pro brings you back to the line after the function call. The debugger changes the source code file displayed in the Source Window, if necessary.



This command is not allowed when the target runs in the background.

Example

To step one source instruction, type:

s

To step five source instructions, type:

5 s



C, S, si, Si

save

Function

Save macros.



Select the `Options | Macro Definitions...` menu item to view the Macro Definitions dialog box. From this dialog box, you can save macros by clicking on the `Save` button. To save macro definitions in a file other than the current one, click on the `Save as...` button.



The command line syntax is:

save *file*

Description

Save all currently defined macros in the specified file. This file is in the format of a sequence of **set** commands, and thus can be loaded by reading it as a playback file. See the **<** and **<<** commands.

An existing save file with the same name will be overwritten.

Example

To save the definitions of the currently defined macros in the file `mac.sav`, type:

save mac.sav



set, unset, echo, !, <, <<

set

Function

Definition and display of macros.



To create a macro, select the Options | Macro Definitions... menu item to view the Macro Definitions dialog box. Click on the New... button.



The command line syntax is:

```
set [ name [ "cmds" ] ]
```

Description

The set command allows for definition and display of macros. If *name* and *cmds* are supplied, a macro entry is made associating the *name* with the commands. If only *name* is supplied, the body of the specified macro is displayed.

If no arguments are supplied the names of all currently defined macros are displayed. Macro definitions must contain the body of the macro in double quotation marks.

Macros may take arguments. In the body of a macro formal arguments are referred to as \$n, where n is the argument number starting from 1.

It is important to understand that macro expansion takes place for all names. Therefore, if you wish to pass the name of an existing macro to a command, such as **set**, you must escape it with '!', to keep CrossView Pro from expanding the name.

Example

To display the names of all currently defined macros, type:

```
set
```

To display the body of the macro named macro, type:

```
set macro!
```

To define `macro` to be a macro which lists the registers then enters the function given by its first argument, type:

```
set macro "l r; e $1"
```

To invoke this macro, you might type, for example:

```
macro(main)
```



```
unset, echo, save, !
```

Si

Function

Single step machine instructions, stepping over subroutine calls



Select the **Run | Step Mode | Machine step** menu item. Then click on the **Step Over** accelerator button in the Source Window, or select the **Run | Step Over** menu item.



The command line syntax is:

[*exp*] **Si**

Description

Single step *exp* (default is 1) machine instructions, stepping over subroutine calls.

If you try to step over a call to a subroutine which contains a breakpoint (or which calls another subroutine with a breakpoint) then the breakpoint will be hit.

The next instruction to be executed is shown as a disassembled instruction, not as a C statement.



This command is not allowed when the target runs in the background.

Example

To step one machine instruction, type:

si

To step five machine instructions, type:

5 si



C, s, S, si, R

si

Function

Single step machine instructions, stepping into subroutine calls



Select the Run | Step Mode | Machine step menu item. Then click on the Step Into accelerator button in the Source Window, or select the Run | Step Into menu item.



The command line syntax is:

[*exp*] **si**

Description

Single step *exp* (default is 1), machine instructions, stepping into subroutine calls.

The next instruction is shown as a disassembled instruction, not as a C statement.



This command is not allowed when the target runs in the background.

Example

To step one machine instruction, type:

si

To step five machine instructions, type:

5 si



C, s, S, Si, R

sio

Function

Associate a stream with a file or screen.



Use the Debug | Simulated I/O Setup menu item to set up simulated I/O streams. You can open a simulated I/O stream with one of the View | Simulated I/O | Stream *x* menu items or use the Debug | Simulated I/O Setup dialog to configure a number of streams. In this case, the associated window will be created when input is required from the window or when output is sent to the window.



The command line syntax is:

```
stream sio {i|o} {file | screen} [/format]
stream sio d
stream sio p prompt
sio
```

Description

Associate an I/O stream with a file *file* or screen. Valid *stream* numbers are 0 through 7. The stream may be input (**i**) or output (**o**). I/O is either done on the file *file* or via the screen/keyboard. The stream may be read from or written to using the format *format* which tells CrossView Pro how to interpret the data. The default format is character. Formats may also be **x** (hexadecimal) and **o** (octal).

Other options to the **sio** command are: delete (**d**) the specified stream, and change the input prompt string (*stream sio p prompt*). The **sio** command with no arguments lists all simulated I/O streams.

This command can only be used in conjunction with user I/O routines containing `_simi` or `_simo` calls. See the *Simulated Input/Output* chapter for further details.

Deleting a stream will close all communication with that stream. This is useful to stop output to the screen or file when enough output information has been seen and resumption of the program, running without I/O breaks from this simulated I/O stream, is desired.

Example

To start simulated input from the file `testdata.in`, type:

```
5 sio i testdata.in/x
```

This input will be associated with stream 5. Data in the `testdata.in` file are expected to be in hexadecimal format.

To list all simulated I/O streams, type:

```
sio
```

st

Function

Stop the execution of the target immediately.



There is no mouse equivalent for this command.



The command line syntax is:

st

Description

This command stops the running process immediately.



Not available for all execution environments.



CB, wt

T

Function

Stack trace with local variables



There is no mouse equivalent for this command.



The command line syntax is:

[*exp*] **T**

Description

Produce a trace of functions on the stack and show local variables. Only the first *exp* levels of the stack trace will be displayed. If *exp* is omitted, all of the levels of the stack trace (up to 20) will be printed.

This command works independently of the Stack Window.



This command is not allowed when the target runs in the background.

Example

To print out a stack trace of 20 levels with corresponding local variables, type:

T

To print out the top five levels of the stack trace with corresponding local variables, type:

5 T



e, l, t

t

Function

Stack trace.



Select the **View | Stack** menu item. The Stack Window shows the current situation in the stack after the program has been stopped. It displays the following information for each stack frame:

- The name of the function that was called
- The value of all input parameters to the function
- The line number in the source code from which the function was called



The command line syntax is:

`[exp] t`

Description

Produce a trace of functions on the stack.

exp specifies the number of levels of the stack trace to be displayed. If omitted, up to 20 levels of the stack trace will be printed.

Each stack level shown in the Stack Window is displayed with its level number first. The levels are numbered sequentially from zero. That is, the lowest/last level in the function call chain is always assigned zero.



This command is not allowed when the target runs in the background.

Example

To print out a stack trace of 20 levels, type:

`t`

To print out the top five levels of the stack trace, type:

`5 t`



`e, l, T`

td

Function

Disable, turn off, trace.



Select the Run | Trace menu item if this item was set.



The command line syntax is:

td

Description

If trace is supported by your version of CrossView Pro, this command disables tracing (both instruction level, high level and raw). Trace is automatically disabled when you close the Trace Window.

Example

To disable tracing, type:

td



tc

te

Function

Enable, turn on, trace.



Select the Run | Trace menu item if this item was not set.



The command line syntax is:

te

Description

If trace is supported by your version of CrossView Pro, this command enables tracing (both instruction level, high level and raw). Trace is automatically enabled when you open a Trace Window.

Example

To enable tracing, type:

te



td

u

Function

Toggle the updating of the appropriate window when the target runs in the background.



There is no mouse equivalent for this command.



The command line syntax is:

```
[interval] u [d|k|r|cd|ck|cr|s|a|mem|t]
```

Description

The following windows can be updated:

d (Data), **k** (Stack), **r** (Register),
cd (Data, composite), **ck** (Stack, composite), **cr** (Register, composite),
s (Source), **a** (Assembly), **mem** (Memory), **t** (Trace)

With *interval* you can specify the update interval (in seconds). If *interval* is zero, no window is automatically updated.

The updating of the Data Window is ON at startup, the others are OFF

If all windows are being updated and/or many monitor commands are active it will increase the load on the communication between CrossView Pro and the target.

The **u d** and the **u cd** commands toggle both the Data Window and the Composite Data Window. The same goes for the **u r**, **u k**, **u cr** and the **u ck** commands in respect to the Register Window, Stack Window, Composite Register Window and the Composite Stack Window.



This command is not available if the background mode is not supported (check the addendum).

Example

To toggle the updating of the Register Window, type:

```
u r
```

To toggle the updating of the Source Window, type:

u s

To disable period updating, type:

0 u



CB, ubgw

ubgw

Function

Update the appropriate window when the target runs in the background.



Select the View | Background Mode menu item and select one of the refresh options.



The command line syntax is:

```
ubgw [ s | a | k | r | d | mem | t | all ]
```

Description

The following windows can be updated:

s (Source), **a** (Assembly), **k** (Stack), **r** (Register), **d** (Data), **mem** (Memory), **t** (Trace), **all** (all open windows)

Without an argument, the **ubgw** command refreshes all windows selected by the background mode (**u** command).

The **ubgw all** command refreshes all open windows.

This command is not available if the background mode is not supported (check the addendum).

Example

To update the Source Window, type:

```
ubgw s
```

To update the Memory Window, type:

```
ubgw mem
```



u

unset

Function

Delete a macro definition.



Select the `Options | Macro Definitions...` menu item to view the Macro Definitions dialog box. Highlight the name of the macro and click on the Delete button.



The command line syntax is:

```
unset [ name !]
```

Description

The **unset** command deletes a macro. If *name* is supplied, the specified macro is deleted. If no arguments are supplied, all currently defined macros are deleted after CrossView Pro confirms your intent.

It is important to understand that macro expansion takes place for all names. Therefore if you wish to pass the name of a macro to a command, for example **unset**, you must escape it with '!', to keep from expanding the name.

Example

To delete all macros, type:

```
unset
```

CrossView Pro will first ask for confirmation. To delete all the macro definitions at the same time, click on the `Delete all` button in the Macro Definitions dialog box.

To delete the macro named `macro`, type:

```
unset macro!
```



set, echo, save, !

use

Function

Change source directories run-time.



Select the Options | Startup | CrossView menu item to view the CrossView Startup dialog box. Click on the Configure... button to specify the names of the directories containing your source files. Relative paths are allowed.



The command line syntax is:

```
use [path]...
```

Description

The **use** command changes the source directories. Without a *path* this command empties the search path, except for the path . (current directory). If one or more *paths* are supplied, this command adds the, semicolon separated, paths to the list of searched directories. Relative paths are allowed.

Example

To clear the source directory path, type:

```
use
```

To search for source files in the directory /project/src and in the src directory relative to your current directory, type:

```
use /project/src;../src
```



1 d

wt

Function

Wait for the completion of the target.



There is no mouse equivalent for this command.



The command line syntax is:

wt

Description

This command can only be used if the target runs in the background mode.

This command waits for the running process to stop.

Waiting can be interrupted by typing *ctrl*-C. The target continues to run without interruption. It could be that some informational messages from the target are displayed in the command window. They can be ignored.



Not available for all execution environments.



CB, st



Function

Force an exit from assertion mode.



There is no mouse equivalent for this command.



The command line syntax is:

[*exp*] **x**

Description

Normally this command stops execution immediately, but if *exp* is present and its value is non-zero, then CrossView Pro finishes executing the entire command list of the current assertion.

Example

To define an assertion to stop the program when the value of global variable `myvar` exceeds 10, type:

```
a if (myvar > 10) {x}
```

To define an assertion to suspend the assertion mechanism and continue program execution when global variable `myvar` exceeds 10, type:

```
a if (myvar > 10) { A s; 1 x; C }
```



a, A, 1

Z

Function

Toggle case sensitivity in searches



Select the `Search | Search String...` menu item to view the Search String dialog box. This dialog contains the `Case Sensitive` check box.



The command line syntax is:

Z

Description

Toggle case sensitivity in searches. The initial state of this toggle depends on information in the currently loaded absolute file. Use the **I** command to find out the state of the case sensitivity.

This command affects everything: file names, function names, variables and string searches.



`/, ?`

REFERENCE

CHAPTER

13

ERROR MESSAGES



TASKING



13

CHAPTER

13.1 WHAT THIS CHAPTER COVERS

The following is a list of common user error messages, and some suggested ways to solve the problem.

CrossView Pro is a complex program running on several hosts. From time to time, slight differences between the documentation and the program's operations do occur. The list of errors presented below and the suggested remedies may not be, therefore, entirely comprehensive.

If you get a message that begins with "XVW Error" or "XVW Fatal Error" please contact TASKING technical support for help.

13.2 ERROR MESSAGES

(in alphabetical order):

"member-name" is not defined for "enum enum"

You cannot assign or compare an enum type with a name that is not in the enumeration's members. Try casting the enum to a different type.

'save' must have a filename; type 'help save' for more information

The **save** command requires a file to be supplied. Note: if the supplied file name already exists, it will be overwritten.

***** Fatal XVW error**

CrossView Pro has detected a error which it can not handle. If information is displayed, you may be able to detect the source of the error and correct it. Otherwise, if the message persists, please contact TASKING Technical Support.

0xvalue is an invalid value. The register *register* is unchanged.

The *value* supplied is incorrect for the specified register. Verify that both the value and the register are correct and retry.

Adding 2 pointers not allowed

You cannot add two pointers together in an expression. If you intended to add to a pointer, make sure that the argument is a value, not another pointer.

Address not allowed for '! or ~ or % operator'

The "Not", "One's complement", and "Modulus" operators cannot be used with an address. If you intended to perform the operation on the *contents* of the address, please be sure to dereference the pointer.

Addresses not allowed in '* or / operator'

The multiply and divide operators cannot be used with address data. If you intended to perform the operation on the *contents* of the address, please be sure to dereference the pointer.

Addresses not allowed in 'bitwise logical or logical or shift operators'

Bitwise logical (&, ^, or |), logical (&& or ||), and shift (<< >>) operators only work on data, not addresses. If you intended to perform the operation on the *contents* of the address, please be sure to dereference the pointer.

Attempt to set breakpoint at invalid address

The memory location is not available. If the memory location is not out of the target chip's range, you may need to map the target system's memory to allow access to this location.

Bad argument to the *command* command

The argument you have given to the **sio** or **f** command is not allowed. Refer to the *Command Reference* chapter, for allowable arguments and their meanings.

Bad assertion number: *number*

The number *number* is not a valid assertion number. List assertions with the **la** (list assertions) command to determine which assertion numbers are valid.

both expressions must be addresses for 'relational operator'

If one of the expressions is an address type, both expressions for relational operators (<, <=, >, >=, ==, and !=) must be address types. Retry with both expressions as either addresses or arithmetic types.

Breakpoint is (or at the address of) an CrossView internal breakpoint. It can not be deleted.

You may not install a breakpoint *over* an CrossView Pro internal breakpoint. See *Breakpoints and Assertions* chapter for more information.

com* return code=*code

The MS-DOS version of CrossView Pro received a status condition from the monitor communication channel which it can not handle. If the condition persists, please contact your system administrator, or call the TASKING Technical Support staff for assistance.

***command* takes no arguments.**

The command *command* needs no arguments. Refer to the *Command Reference* chapter, for the command syntax.

Can not open file (*file*)

CrossView Pro could not open the file *file*. Check the spelling of *file* and check that the file is in the correct directory. You should also check the permission of *file*. With MS-DOS, check the CONFIG.SYS file for the maximum number of open files allowed. Increase the number and reboot if necessary.

Can not output to input stream

An attempt was made to output to an input stream. The most common case is incorrectly setting up your simulated i/o streams. Correct and retry.

Can not scroll that window

The window you have tried to scroll is not scrollable. Examine your choice of window and/or your choice of windowing commands.

Can't define macro: out of space

There is not enough host memory to add your macro. Eliminate one or more unused macros before adding a new one.

Can't expand macro: out of space

There is not enough host memory to expand your macro. Eliminate one or more unused macros before adding a new one.

Can't monitor data: out of space

CrossView Pro cannot add any more variables or expressions to monitor. You must delete one or more variables or expressions before adding any more.

Can't open *logfile-name* as log file

CrossView Pro could not open the specified host-to-target system communications logfile. Check the spelling of *logfile-name* and that *logfile-name* is in the correct directory. Check permissions of *logfile-name*. With MS-DOS, check the CONFIG.SYS file for the maximum number of open files allowed. Increase the number and reboot if necessary. Make sure the filename is valid for the host Operating System.

Can't open *output-file-name* as output file

CrossView Pro could not open the specified output file. Check the spelling of *output-file-name* and that *output-file-name* is in the correct directory. Check permissions of *output-file-name*. With MS-DOS, check the CONFIG.SYS file for the maximum number of open files allowed. Increase the number and reboot if necessary. Make sure the filename is valid for the host operating system.

Can't open *playback-file-name* as playback file

CrossView Pro could not open the specified playback file. Check the spelling of *playback-file-name* and that *playback-file-name* is in the correct directory. Check permissions of *playback-file-name*. With MS-DOS, check the CONFIG.SYS file for the maximum number of open files allowed. Increase the number and reboot if necessary. Make sure the filename is valid for the host operating system.

Can't open *record-file-name* as record file

CrossView Pro could not open the specified recording file. Check the spelling of *record-file-name* and that *record-file-name* is in the correct directory. Check permissions of *record-file-name*. With MS-DOS, check the CONFIG.SYS file for the maximum number of open files allowed. Increase the number and reboot if necessary. Make sure the filename is valid for the host operating system.

Can't open file '*file*'

CrossView Pro could not open the specified file. Check the spelling of *file* and that *file* is in the correct directory. Check permissions of *file*. With MS-DOS, check the CONFIG.SYS file for the maximum number of open files allowed. Increase the number and reboot if necessary. Make sure the filename is valid for the host operating system.

Can't perform trace, out of memory

There is not enough host memory to support tracing. Reduce memory demands and retry again. If the problem persists, please contact the TASKING Technical Support staff for assistance.

Can't set breakpoint; either the current file has no symbols, or line *line#* is not inside any procedure in the current file.

CrossView Pro was unable to set the breakpoint that you specified. First check the location of line *line#* and verify that it is in the current procedure being debugged. If it is within the current procedure, then you may need to compile/assemble/link/locate for debugging. Refer to chapter *Overview* for details.

Can't start a new process. Feature not implemented.

Your host system does not support shell commands. Any attempt to issue shell commands will cause this message to be displayed.

Can't write to a read-only SFR.

The SFR register is a read-only register. It can not be set or altered.

Cannot allocate memory for symbol table

Allocating memory for storing the symbol table failed. Remove some tasks from memory or add more memory to your computer system.

Cannot allocate symbol table memory buffers

The symbol table is too large for CrossView Pro. You may need to selectively compile with the **-g** switch only those files and procedures that most interest you.

Cannot allow that combination of operand(s) and operator

The operand(s) is/are incompatible for this type of operation. For example, you may not add two structures. Please verify the operation and data types you are using.

Character constant is missing ending '

Character constants must be delimited with single quotes. Example: 'a'.

Command '*command*' not allowed while emulator running in background

The target is running, this command is not allowed unless the target is stopped. See the **st** command.

couldn't *error-message*

VMS is reporting a condition that CrossView Pro can not handle. If the condition persists, please contact your system administrator, or call the TASKING Technical Support staff for assistance.

Data already being monitored "*task-id*":'*symbol*'

The variable or expression *symbol* is already being monitored by CrossView Pro. You do not need to enter it again.

Display format required

The display command expected an output format option that was not supplied. See chapter *Command Language* for valid format options and their meanings.

Double not allow in '*%* or ~ operator'

You may not use the one's complement or modulus operators on double floating point types.

Double not allow in '*bitwise operator*'

You may not use *bitwise* operators (&, ^ and |) on double floating point types.

ERROR: you must enter ?,i,r,d

CrossView Pro's line editor only supports the following commands: **?-help**, **i-insert**, **r-replace**, **d-delete**, and **<cr>** to execute command.

Establish a file context first.

The command executed requires an active file. Verify the file you specified to CrossView Pro on start up.

Establish a procedure context first

The command executed requires an active procedure. Either execute the command from within a procedure, or give a procedure name as an argument to the command.

Exiting procedure call state

An unknown system signal caused the end of a command line function call.

Expecting stream number

The following forms of the **sio** command expect a stream number:

stream sio {i | o} {file | screen}

stream sio d

stream sio p prompt

Expression garbaged

The symbol table contains a type that is unknown to CrossView Pro. Please verify that you are using the compiler and utilities supplied to you. If the condition persists, please contact the TASKING Technical Support staff for assistance.

file has already been edited, going to NEW file

The command executed requires that the file be edited only once. A new file has been created.

failed to allocate the SIO tables

Entries for recording simulated input/output information could not be allocated due to lack of host memory. Please contact your system administrator, or call the TASKING Technical Support staff for assistance.

Float not allowed in '% or ~ operator'

You may not use the modulus or one's complement operators on floating point types. Change the data type to an appropriate type, for example, integer.

Float not allowed in 'bitwise or shift operator'

You may not use the *bitwise* (&, ^, or |) or *shift* (>>, or <<) operators on floating point types. Change the data type to an appropriate type, for example, integer.

Framing Error on COM port number

The host computer detected a data frame communication error on COM port *number*. Check the host and target communication set up as well as line connections. If the problem persists, please contact your system administrator, or call the TASKING Technical Support staff for assistance.

I can't put something that big in the child process

The size of the expression exceeds the buffer size needed to spawn a child process. Be sure you have linked `end.c` in your application. This module implies space for CrossView Pro in your execution environment. Refer to section *Building Your Executable* in chapter *Overview*. If this condition persists, please contact the TASKING Technical Support staff for assistance.

I don't have symbols for this procedure

You will need to re-compile, assemble, link and locate with the proper debugging options before using this command. See section *Building Your Executable* in chapter *Overview* for details.

I have no source file for this address

The program counter holds an address which is outside all the address ranges that CrossView Pro knows about. This may happen if program execution has reached a file that was not compiled with the `-g` generate debug symbols switch.

I need a linenumber

The `go g` command requires a line number. Enter a line number and the command will be executed.

Illegal address for Emulator Hardware Breakpoint

The address specified is out of emulator hardware breakpoint memory range. Verify the address and retry.

Illegal argument ("0") to 'p' command

You must specify a number greater than 0 for the `p` command, which prints the specified number of lines.

Illegal argument to '*command*' command: '*argument*'

You have passed an illegal argument to the specified command. Refer to chapter *Command Reference* for legal arguments.

Illegal argument to `ct`: '*argument*'

You have passed an illegal argument to the C-trace command. Refer to chapter *Command Reference* for legal arguments.

Illegal data monitor command

You have passed an illegal argument to the **m** data monitor command.

Legal commands are:

- m exp** to set up monitoring
- id m d** to delete monitoring of a specific expressions
- m d** to delete monitoring of all expressions

Illegal third arg to set: 'argument'

The **set** command may have only two arguments: the name by which the macro is known and the command string to be executed when the macro is invoked. Enclose the command string in quotes, separating the individual commands with semicolons. Refer to chapter *Command Reference* for more information.

Improper floating point format length

You have specified a format length that is inconsistent with floating numbers. Legal lengths are 4 and 8 bytes.

Improper integer format length

You have specified a format length that is inconsistent with integer numbers. Legal length are 1, 2, and 4 bytes. You may also choose **b**, **s**, or **l** for 1, 2, and 4 byte integers.

Improper string format length

You have specified a format length that is inconsistent with character strings. Choose a positive number.

Input buffer overflow

CrossView Pro is over-running the input buffer. Contact your system administrator to either increase the input buffer or lower the communication line baudrate.

Input communications buffer overflow on COM port

CrossView Pro is over-running the input buffer. Contact your system administrator to either increase the input buffer or lower the communication line baudrate.

Input from stdin longer than *max-input-size* characters: *input-string* Command truncated

The input data is longer than the input buffer, therefore the data was truncated at *max-input-size*. Try to reduce the input data and/or commands.

Internal error while setting an instruction level breakpoint

If this error condition persists, please contact the TASKING Technical Support staff for assistance.

Invalid assertion maintenance command

You have entered an illegal assertion command. Valid commands are:

- a a** to activate assertions
- a d** to delete assertions
- a s** to suspend assertions

Invalid value for uplevel break.

You have entered an illegal value for an uplevel break. The form of the command is *exp* **bU** or *exp* **bU**, where *exp* determines how many returns from functions should occur before the break. Execute the **t** command to find out how many levels down in the stack you are, then choose an appropriate value for the uplevel break. See chapter *Command Reference* for more information.

Invoking procedure calls not allowed while emulator is running in the background

The target is running, this command is not allowed unless the target is stopped. See the **st** command.

Macro Expansion error: expansion looping

CrossView Pro looped 50 times while trying to expand this macro without completing the expansion. Check the logic of the macro arguments. It may need to be corrected or simplified.

Macro Expansion error: expansion too large

The macro expansion exceeds 200 commands. The macro must be simplified.

Macro Expansion error: missing '('

See the command reference page or use the help command to review macro command syntax.

Macro Expansion error: missing ')'

See the command reference page or use the help command to review macro command syntax.

Macro Expansion error: missing ','

See the command reference page or use the help command to review macro command syntax.

Macro Expansion error: not enough args

See the command reference page or use the help command to review macro command syntax.

Macro Expansion error: out of space

There is not enough memory to expand the macro. Eliminate one or more unused macros before adding a new one.

Maximum trace size is: *max-trace-size*

CrossView Pro can perform C tracing only up to *max-trace-size* source lines. Choose a number less than *max-trace-size* with the **ct** command.

Missing { after if command

The required format for the **if** command is: **if** *exp* {*commands*}

Missing file name or 'screen'

The **sio** command was missing a required parameter for setting up a simulated i/o stream. See chapter *Command Reference* for command definition and format.

Missing format character

You did not specify a display format type with your command. Either remove **'/'** from the command, or add a format character.

Missing prompt string

You did not specify a prompt string with the **sio** command. Either remove **p** from the **sio** command, or add a prompt string.

Must supply 'b' or 'f'

The color command requires a value of **f** for foreground or **b** for background to modify the screen color.

Must supply 'r','w' or 'b'

Both the data range (**bD**) and data (**bd**) breakpoint commands require the type of data modification to generate a break condition. Use **r** for read, **w** for write, and **b** for both read/write. Please see chapter *Command Reference* for more information.

Must supply data to be monitored

You did not specify a variable or expression to the **m** monitor command. Please provide a variable or expression to be monitored, for example, `m myvar`.

Must supply second address with bD command.

The **bD** command requires two addresses. Either specify an upper limit if you want to break anywhere in memory range, or use the **bd** command if you want to break on an individual address.

**Negative /baudrate value ignored. (VAX)
or****Negative baud rate (-S) value ignored.**

The baudrate specified was a negative value. Please specify a legal value or use the default.

**Negative /TIMEOUT value ignored. (VAX)
or****Negative timeout interval (-I) value ignored.**

The time out value specified was negative. Please specify a legal timeout value or use the default.

No child process

The CrossView Pro internal data structure containing user information about child processes is not as expected. Please contact the TASKING Technical Support staff for assistance.

No current file

Undefined special variable, `$file`; probably due to debugging where no symbols are present.

No current line number

Undefined special variable, `$line`; probably due to debugging where no symbols are present.

No current procedure

Undefined special variable, `$proc`; probably due to debugging where no symbols are present.

No host memory

There is not enough space in memory to execute this command. Check whether you have unnecessary processes running in the background or resident in memory.

No host memory for command

There is not enough space in memory to execute this command. Check whether you have unnecessary processes running in the background or resident in memory.

No macros to save; file not created

CrossView Pro found no macros to save, therefore the **save** command did not create a file.

No Match – *pattern*

CrossView Pro did not find the specified *pattern* in its search of this file. Check your spelling or case-sensitivity. Use the **Z** command to toggle case-sensitivity if necessary.

No memory space

There is not enough host memory to execute this command. Check whether you have unnecessary processes running in the background or resident in memory.

No more hardware breakpoints available

The target system uses hardware breakpoints to support the data breakpoint function. To continue, you must explicitly delete a data breakpoint before placing a new one.

No more room for directories (> *max-dir-size*)

You can reference no more than *max-dir-size* directories for source files.

No more SIO windows, I/O to command window.

Only four SIO streams can be displayed simultaneously in the SIO window. Subsequent SIO streams' output will be displayed in the command window.

No name of symbol file specified

CrossView Pro cannot deduce the name of a symbol file. No filename was given to the **N** command and no symbol file was currently loaded.

No playback name specified

Give the name of the playback file to be used for this session.

No process

CrossView Pro only allows one process to be debugged at the same time.

No such breakpoint

The breakpoint number was incorrect. List breakpoints with the **lb** command to find the correct breakpoint.

No such field name "*name*" for "<structure | union> *name*"

The field *name* is not recognized for the specified structure or union. Check the spelling of field name. The **/t** format will show you the names and types of a particular structure's or union's fields.

No Such Line

CrossView Pro can not find the specified line number in any of its known files. Please check the source window or a source listing for legal line numbers.

No such procedure, "*name*".

CrossView Pro does not recognize *name* as a procedure name. Check the spelling and whether the file was compiled/assembled/linked/located for debugging. Check that the file is in the appropriate directory.

No such procedure or file name: *procedure*

CrossView Pro does not recognize *procedure* as a procedure or file name. Check the spelling and whether the file was compiled/assembled/linked/located for debugging. Check that the file is in the appropriate directory.

No such PSW register state

Check register name and selected target.

No such register

The target processor does not have a register with that name.

No such sr reg state

Check register name and selected target.

No such stream

The stream you tried to delete does not exist. Check the stream number, correct, and retry.

No symbols – unable to determine end-of-procedure

CrossView Pro has no symbol information for this procedure. To facilitate debugging this procedure, you must compile, assemble, link and locate with the appropriate switches. Refer to the *Overview* chapter for details.

No symbols available in active procedures.

To get symbol information you must compile, assemble, link, and locate with the appropriate switches. Refer to the *Overview* chapter for details.

No symbols for that procedure

To get symbol information you must compile, assemble, link, and locate with the appropriate switches. Refer to the *Overview* chapter for details.

No User or System special variable matches prefix *name*

The *string* argument of the **ls** command did not match any user or system special defined variables. Check spelling and case-sensitivity and retry.
You may also enter **ls** to print out all the user and system special defined variables.

Not enough memory available to start up windows. Either use the **-nw (no window) option or remove resident programs from memory.**

CrossView Pro has detected that there is not enough host memory to execute the windowing software. You may need to use the **-nw** option to start up CrossView Pro in line mode. Check whether you have unnecessary processes running in the background or resident in memory.

Not enough memory to execute shell command.

The attempt to create a child process for the shell command failed due to the lack of host memory. Check whether you have unnecessary processes running in the background or resident in memory.

Not enough memory to start window mode

CrossView Pro has detected that there is not enough host memory to execute the windowing software. You may need to use the **-nw** option to start up CrossView Pro in line mode. Check whether you have unnecessary processes

Not enough space

CrossView Pro has detected a general error due to lack of host memory. Check whether you have unnecessary processes running in the background or resident in memory.

Not in known territory. Could not set breakpoint.

CrossView Pro's current location is not in a file or procedure that it knows about. The breakpoint request can not be performed.

Not in window mode

The command issued requires CrossView Pro windows to be active. Use the **WW** command and repeat the previous command.

Not that many levels active on the stack.

A stack level was specified that does not exist. Execute the **t** command to determine levels on the stack. See chapter *Command Reference* for more information.

Oops called with sig = *signal-number*

CrossView Pro has received a signal that it can not handle. Continuing from this point may result in a fatal process condition. If this condition persists, please contact your system administrator, or call the TASKING Technical Support staff for assistance.

Placement of the breakpoint handler must be in one of the restart vectors. Choose a value from 0 to 7.**Try again. (Hit <cr> to exit)?**

The specified placement for the breakpoint handler was not valid for this target. CrossView Pro is requesting a valid location.

Procedure "*name*" is not active on the stack.

The procedure *name* was not found on the current stack. Execute the **t** command to list functions which are active on the stack.

Procedure '*name*' is not at that stack depth

The procedure *name* was not found on the specified stack. Execute the **t** command to list functions which are active on the stack.

Procedure "*procedure*" is not active

The procedure *procedure* was not found on the current stack. Execute the **t** command to list functions which are active on the stack or **lp** for list of procedures known to CrossView Pro.

Program not completely loaded

An error occurred during loading a symbol file. Check what cause the problem (illegal filename or file format). You may retry to load a symbol file.

Prompt too long (> *max-number*)

Choose a prompt of no more than *max-number* characters.

Ran out of memory reading symbol file into memory

Reduce the size of the symbol file by re-compiling only the "interesting" files with the **-g** debug switch.

Read I/O request could not be queued

VMS detected an error for a read I/O queue which CrossView Pro can not handle. If the condition persists, please contact your system administrator, or call the TASKING Technical Support staff for assistance.

Readprompt I/O request could not be queued

VMS detected an error for a read I/O queue which CrossView Pro can not handle. If the condition persists, please contact your system administrator, or call the TASKING Technical Support staff for assistance.

Redo: line too large

Limit line length to fewer than 256 characters.

Result type too large for command line call.

A command line function call must pass the result back in a register. The specified function does not. You cannot call functions whose return value is greater than an integer, for instance floating point types and structures.

Result type undefined

Type casting from the expression or variable to the result type was not possible.

Second address smaller than first

When specifying a range of addresses for a data breakpoint, the second address must be higher than the first.

Sim I/O request too long (>*max-number* bytes)

The I/O request exceeds the maximum length.

Simulated I/O stream out of range

Choose a stream value between 0 and 7.

Sorry, the "v" command is not supported on this host

No visual editor is available on this host.

Stream already active

Either choose another stream, or deactivate this one before re-assigning it.

String constant is missing ending "

String constants must be delimited with double quotes: "

Subtracting 2 pointers not allowed

You cannot subtract two pointers in an expression. If you intended to subtract from a pointer, make sure that the argument is a value, not another pointer.

Symbol file is either unreadable or too short

The symbol file is not an absolute IEEE-695 file, or the file format is not correct, or the file is not an IEEE-695 file at all.

Symbol file is not formatted correctly

The symbol file is not intended for the type of microprocessor you are using.

Symbol not in current procedure

There is no symbol by this name in the current procedure. Check the spelling of the symbol name.

The '*command*' command accepts no args

The command *command* does not accept any arguments. See chapter *Command Reference* for more information on *command*.

The window would be too large; Total lines must not be greater than *max-size*

The window size options specified would create a window that would have exceeded the screen size. Retry with corrected window size options.

There is insufficient information to do a structure dump

CrossView Pro cannot uniquely identify the structure or part of the structure to be dumped.

There is no associated source.

The program counter holds an address which is outside all the address ranges that CrossView Pro knows about. This may happen if program execution has reached a file that was not compiled with the **-g** debug switch.

There is no available source line for the current address.***\$pc= address***

CrossView Pro is reporting that the current position has no associated source line. This may happen if you are trying to debug a routine that was not compiled with **-g** debug switch or are trying to debug a runtime library routine.

This does not appear to be a struct or a union

The data entered is not recognizable as a structure or union. Check the specified variable.

Timed read I/O request could not be queued

VMS reported a condition on a timed read i/o request that CrossView Pro could not handle. If the condition persists, please contact your system administrator, or call the TASKING Technical Support staff for assistance.

Too many args to unset: '*argument*'

You may specify only one macro at a time, for example, **unset** *name*, or you may remove all macros at once with **unset**.

Too many assertions (>*max-number*)

The maximum number of assertions allowed is *max-number* as shown in the error message. Remove a previous assertion before trying to add one, or reinvoke CrossView Pro with the **-a** option to increase the maximum number of assertions.

Too many breakpoints (> *max_number*)

The maximum number of breakpoints allowed is *max-number* as shown in the error message. You must explicitly delete a breakpoint before adding any new ones. Alternatively, you could re-invoke CrossView Pro with the **-b** option to increase the maximum number of breakpoints.

Too many locals (> *max-number*)

Eliminate some existing locals or reinvoke CrossView Pro with the **-s** switch to increase the number of locals allowed.

Too many modules

The symbol file describes an application that was constructed from more than 1818 modules.

Too many processes (> *max-number*)

CrossView Pro allows only one process to be debugged.

Too many streams (> *max-number*)

The maximum number of I/O streams, *max-number*, has been reached. You must eliminate an I/O stream before adding a new one.

Trace size is required

The required format of the command is *exp* **ct**, where *exp* is the number of statements to trace. Re-enter the command with a value for *exp*.

Type 'r', to run program from power-on conditions or 'c' to continue with current program pointer

This is to inform you that command *r* is not implemented and that you should use *r* or *c*.

Type of *command-line-expression* is too complex

The command line function returns a data type that CrossView Pro cannot handle. An example would be a function returning a structure.

Unexpected breakpoint type '*type*'

CrossView Pro has encountered a breakpoint with an unknown type attribute. Verify the previous break commands and re-try. If the condition persists, please contact the TASKING Technical Support staff for assistance.

Unknown command '*command*' (<*number*>)

CrossView Pro does not recognize *command*, and has echoed the command number for Technical Support purposes. Please check the spelling and retry. If the condition persists, please contact the TASKING Technical Support staff for assistance.

Unknown data monitor id '*number*'

The monitor number *number* that you tried to delete does not exist. Use the **M** command to list currently monitored variables.

Unknown data size

Valid data sizes are 1, 2, 4, or 8 bytes.

Unknown display mode

See chapter *Accessing Code and Data*, for a list of display mode options.

Unknown name '*name*'

Variable *name* is not in scope or is undefined.

Unknown procedure "*name*".

The function *name* does not exist in any file that CrossView Pro knows about. The file containing *name* may not have been compiled with the **-g** debug switch.

Unknown macro '*name*'

CrossView Pro does not recognize the macro name as given. Please check the spelling. You may list all current macros by using the **set** command with no arguments, or display the Macro window for currently defined macros.

Unknown window

CrossView Pro does not recognize the window name as given. See chapter *Command Reference* for valid window arguments.

Unsupported format type (*parameter*)

Supported types are **c** (character), **x** (hex), and **o** (octal).

Value *number* is not defined for this enum.

The member specified was not part of the enumerated set. Please check the spelling and verify that the correct enum was used.

Value exceeds depth of stack.

A stack level was specified that does not currently exist. Please check the value and retry. Check the stack window for valid stack levels, or execute a **t** command (trace stack) to determine the depth of the stack.

VMS error : cannot establish handler for signals

CrossView Pro on VMS could not establish proper error handlers. If the condition persists, please contact the TASKING Technical Support staff for assistance.

VMS error : cannot establish pasteboard

CrossView Pro on VMS can not establish the running environment. If the condition persists, please contact your system administrator, or call the TASKING Technical Support staff for assistance.

VMS error : cannot establish virtual keyboard

CrossView Pro on VMS can not establish the running environment. If the condition persists, please contact your system administrator, or call the TASKING Technical Support staff for assistance.

VMS error code = *number* \ Attempt to get message text fail.

CrossView Pro on VMS received an error while attempting to provide an error diagnostic message from the host error message library. If the condition persists, please contact your system administrator, or call the TASKING Technical Support staff for assistance.

Warning: NULL pointer dereference

The expression contained a null pointer dereference. Check the expression for possible errors, or verify that the pointer should in fact be null.

Warning: pointer dereference with invalid segment selector.

The pointer is addressing invalid memory and the dereference may report unexpected data results. Check the initialization of the pointer or verify that it has been set correctly.

Warning: too few parameters.

The command given was not invoked with the proper number of arguments. CrossView Pro will supply the command with defaults which may or may not produce the result you expected.

Warning: Using *file-b* instead of *file-a*

CrossView Pro could not find *file-a*, or *file-a*'s status was such that CrossView Pro could not use it. If *file-b* is not correct, check *file-a* spelling and its directory.

Warning: X=Y: X is *x-size* bytes and Y is *y-size* bytes

The assignment of two different size variables may cause unexpected results. Please correct the condition if possible. This condition is common when assigning string variables where string *y* is shorter than string *x*.

Warning: X=Y: X is *x-size* words and Y is *y-size* words

The assignment of two different size variables may cause unexpected results. Please correct the condition if possible. This condition is common when assigning string variables where string *y* is shorter than string *x*.

**Warning: CrossView comment terminated by end of command line
source-command-line**

The playback file has a comment that was not terminated. It is by default terminated, but if the next line was the continuation of the comment, then unexpected results may occur. Please terminate comment strings on each line to avoid this warning.

Windows not enabled; use WW to enable

The command issued can only be used when windows are enabled.

Write I/O request could not be queued

CrossView Pro received a condition that it could not handle. If the condition persists, please contact your system administrator, or call the TASKING Technical Support staff for assistance.

Write-only register. Value may not be valid.

CrossView Pro set a write-only register but has no way of verifying the correctness of the register contents.

Wrong storage class for data breakpoint

You may not set a data breakpoint at the address of a register variable or special variables.

CrossView could not disassemble the emulator's trace buffer because the address information in the buffer is incorrect.

The trace buffer may be corrupted. Re-check the commands leading to this condition, and retry. If the condition persists, please contact the TASKING Technical Support staff for assistance.

XVW error – *message*

or

XVW Fatal error – *message*

These messages are generated by internal conditions that should not normally occur. The *message* is usually encrypted and should be brought to the attention of the TASKING staff. Please contact the TASKING Technical Support staff for assistance.

XVW:main – Cannot continue, incomplete initialization.

CrossView Pro's initialization was interrupted and could not be completed. Please re-start CrossView Pro, and if the condition persists, contact the TASKING Technical Support staff for assistance.

You can't goto a line outside of the current procedure

The specified line number is outside the current procedure. Change the line number to one within the procedure or enter the correct procedure before executing this command.

You may not assign from a host system string/array

The expression given performs an assignment that CrossView Pro can not perform at this time.

You may not assign from a void function

The expression attempts to assign a variable from a void function. Please check the return value of the function and verify that you are referencing the correct function.

You may not assign more than *max-size* bytes to a special variable

An attempt was made to assign greater than the maximum number of bytes to a special variable. Check the expression for errors, and check the variable's spelling.

You may not assign to a constant

The value of a constant cannot be changed. Check the name that you have specified.

You may not mix assignment of a scalar and an aggregate

An attempt was made to assign incompatible types of data. Please check the data types and retry.

You need to supply a program name.

CrossView Pro requires a program name to debug on the invocation line.

ERRORS

CHAPTER 14

GLOSSARY



TASKING



14

CHAPTER

14.1 WHAT THIS CHAPTER COVERS

This chapter defines many terms common to CrossView Pro and source-level embedded systems debugging. Italicized items in definitions are also entries.

14.2 GLOSSARY TERMS

A

absolute file. The IEEE-695 file (*.abs*) that contains symbolic debug information and the executable code of the target system.

active window. A user-selected CrossView Pro window that commands operate on as a default. An active window's title appears in red (on color monitors) or inverse video (on monochrome).

Assembly window. An CrossView Pro window showing a part of the disassembled program space. It also indicates such information as the current execution position, viewing position and installed breakpoints.

assertion. A command or set of commands to be executed before every line of source code. Assertions are especially useful in tracking down hard to find bugs when other methods fail. Individual assertions may either be active or suspended. *See also* **assertion mode**.

assertion mode. A mode of CrossView Pro operation under which assertions will be executed. Before CrossView Pro executes a source line of code, it tests for an arbitrary condition specified by the user. Since CrossView Pro is single stepping breakpoints will not be effective. As long as there is at least one assertion active, CrossView Pro operates in assertion mode. A program running in assertion mode will stop when an asserted command executes an **X** (exit assertion mode) command.

B

background mode. A feature in CrossView Pro that lets the execution environment run while you can still enter CrossView Pro commands.

bias. The offset value from zero at which a target program is actually loaded into memory. The bias can be set with the **-n** startup option.

breakpoint. A mechanism for stopping target program execution at a particular line of code (*see* **code breakpoint**), when a memory address is accessed (*see* **data breakpoint**), or at a return from a function (*see* **uplevel breakpoint**). There are two general kinds of breakpoints, hardware which the emulator sets in its circuitry, and software, which are special instructions placed in user code. Since the number of simultaneous hardware breakpoints is limited in number, CrossView Pro uses software breakpoints. Using transparency mode, however, you may set your own hardware breakpoints.

Breakpoint window. An CrossView Pro pop-up window displaying code and data breakpoints, and any attached commands.

C

C-trace window. An CrossView Pro window keeping a record of the most recently executed C or machine statements.

cache. Some microprocessors keep in on-chip memory a copy of the most recently executed instructions to speed processing. Sometimes execution will not halt until the this private storage (the cache) is exhausted. *See also* **skidding**.

code breakpoint. A breakpoint that halts program execution when a particular line of code is reached. A code breakpoint can have a command list. A breakpoint can be set on a line of source code or at the address of a machine instruction. *See also* **count**.

Command window. The Command window contains all user input. Output may be directed to the Command window.

command list. A series of CrossView Pro and/or C commands attached to a code or data breakpoint, executed when the breakpoint is hit.

Composite window. An CrossView Pro window that is a combination of three separate windows: the Data, Register, and Stack windows.

count. The number of times a breakpoint must be hit before execution halts. Breakpoints are created with a count of 1. The **C** command may be used to change the count of a breakpoint.

current function. The function that execution is currently in. The current function is always at level 0 on the stack.

D

data breakpoint. A breakpoint that halts program execution when a particular memory address (or an address within a particular range) is written to, read from, or both. A data breakpoint may have a command list and a **count**.

data monitoring. CrossView Pro allows you to monitor expressions and variables in the Data window. CrossView Pro updates their values whenever execution stops.

Data window. An CrossView Pro window displaying the values of monitored expressions.

diagnostic output. Program output designed for debugging purposes. With CrossView Pro, breakpoints and data monitoring can be used for diagnostic output, eliminating the need for intrusive and annoying print calls compiled into code.

dot operand. The period character "." used in an expression to represent the last value CrossView Pro calculated. The dot operand is useful as shorthand.

E

embedded system. An application program built for a microprocessor running in real-time. An embedded system usually is part of a larger, non-computer system, hence the term "embedded." The TASKING product line is designed for embedded systems programming.

emulator. A device used to monitor various aspects of a microprocessor's operation. An emulator usually is built around two chips, the target microprocessor and a controlling chip. The controller chip can start and stop the target chip's program execution, and can examine and change registers and memory. An emulator can be connected via a probe to a hardware prototype to emulate fully the behavior of the target chip. Hence it is an ideal debugging device. *See* **ROM monitor**.

__end__. A run-time library routine used to implement command line function calls. It must be linked into the object code.

execution position. The source line corresponding to the value of the program counter. *See* **viewing position**.

F

format. The manner in which CrossView Pro displays data; for instance, hexadecimal, character and octal are different formats. You may include special format codes when specifying variables.

H

hardware breakpoint. *See* **breakpoint.**

Help window. One of two levels of windows summarizing the syntax and function of CrossView Pro commands.

history mechanism. The process by which CrossView Pro retains the last twenty CrossView Pro commands issued.

host system. The computer system on which CrossView Pro is run. The host system is connected to the target system, usually with an RS-232 cable.

I

image part. This is the downloadable part of the absolute file (.abs) that contains the executable code of the target program. *See also* **absolute file.**

interrupt key. The key that interrupts ongoing processes. On many systems this is *ctrl-C*.

L

Librarian. The utility which manages libraries of program modules at the prelink or postlink phase of development.

line mode. An CrossView Pro operating mode in which all screen output appears after the prompt. No windows are present. *See also* **window mode.**

Linker. The utility which combines separate object modules into a single module, resolving references.

local variable. A variable that can only be referenced from within its defining function.

Locator. The utility which assigns absolute target-memory locations to relocatable sections and resolves address references.

low-level breakpoint. A code breakpoint placed on an individual machine instruction. Low-level breakpoints can be set with the **bi** command.

M

macro. A user-created shorthand for an CrossView Pro command sequence. Macros can accept parameters and can be saved to a file.

Macro window. An CrossView Pro window that lists all currently defined macros. It automatically appears whenever you define a new macro in window mode.

main(). The function where a C program's execution begins. *See also* **system startup code.**

MAU. Abbreviation for **minimum addressable unit.**

memory map. The configuration of an emulator's memory that specifies which addresses are read-only, and which are read/write. With many emulators, you must first set up a memory map before using CrossView Pro.

minimum addressable unit. For a given processor, the amount of memory located between an address and the next address. It is not necessarily equivalent to a word or a byte. Abbreviated **MAU.**

O

object language. A representation for target machine instructions, with the ability to represent either relocatable or absolute address locations.

On-line Help. A complete summary of all CrossView Pro commands and individual descriptions available while CrossView Pro is running.

On-line Tutorial. A playback file supplied with CrossView Pro that demonstrates CrossView Pro's capabilities.

optimizer. A phase of the compiler which identifies sections of source code that can be made more efficient by the code emitter and directs the code emitter to implement those improvements.

output buffer. The location in memory where CrossView Pro directs simulated output. *See also* **simulated I/O**.

overlay. A module that may be loaded into memory at an address other than the address specified at locate time. CrossView Pro allows you to debug relocatable code by specifying an offset when invoking the debugger. CrossView Pro uses the information to correlate executable code with the symbol table.

P

patch. A technique to alter program flow (without recompiling the source code) with CrossView Pro commands and/or C expressions. With CrossView Pro, it is possible to use breakpoints to alter program flow by patching in new code or moving the execution position around existing code.

pop-up window. A window that appears in certain situations that overlaps the current display. Pop-up windows usually contain information (like a command definition) that need not be continuously displayed. The Breakpoint, Macro and Help windows are pop-up windows.

probe. A part of an emulator that can be inserted in place of the target chip in the actual embedded systems hardware.

pseudo-assembly. An optional output from the compiler that lists an assembly code representation of the compiled output. Since the compiler compiles directly to object code, there is no actual assembly output.

Q

quiet command. A **Q** instruction in the command list of a breakpoint suppressing the default display of *function: line number: source file*.

R

record & playback. The ability to save CrossView Pro commands (and, if desired, Command window output) to a file. CrossView Pro can play back simple text files consisting solely of CrossView Pro commands.

Register window. An CrossView Pro window showing the contents of the target microprocessor's registers.

reserved special variables. Special variables (\$LINE, \$PROCEDURE, \$FILE) whose values CrossView Pro maintains to reflect the current status of the debugging session. *See also* **special variables**.

ROM monitor. A program which supervises or controls, at an elementary level, the overall operation of an embedded system. There are ROM monitors, designed to TASKING's specifications that communicate with CrossView Pro. Because of the limited hardware features of most boards containing ROM monitors, some CrossView Pro features may not be supported. *See also* **emulator**.

RS-232 cable. A cable that transmits serial data between the host and target systems.

S

scope. The extent to which a variable can be referred to. Global variables are always in scope; local variables are only in scope when their defining function is the current function.

select. To make a window active.

simulated I/O. A technique to trap input and output calls for debugging purposes. Simulated I/O is often used for testing a program before the actual input and output hardware devices are present. *See also* **stream**.

Simulated I/O window. An CrossView Pro window containing all the input and output streams directed to the screen. CrossView Pro can display from one to four separate windows at a time.

single stepping. Executing a source statement or a machine instruction then halting. Single stepping lets you observe a program executing in stop-motion, to observe registers, variables and program flow.

skidding. When a microprocessor executes a few instructions after a data breakpoint halts execution. On some microprocessors, execution may not stop until all instructions in its cache have been executed. It is important to realize therefore that a target program may not halt at the precise instruction where the data breakpoint occurred.

software breakpoint. *See* **breakpoint.**

source level debugger. A debugger capable of correlating source code and variable names with object code. CrossView Pro is a source level debugger.

Source window. An CrossView Pro window containing the listing of the target program. It also indicates such information as the current execution position, viewing position and installed breakpoints.

special variable. A variable independent of the target program that CrossView Pro maintains for the user's benefit. Special variables start with a \$ and are defined when first mentioned. CrossView Pro also maintains **reserved special variables** that contain information about the state of the debugging session.

split screen mode. *See* **window mode.**

stack depth. The level that a particular return address from a function resides on the stack. The current function is always at stack depth zero.

stack traceback. An operation in which CrossView Pro reads the return addresses and passed parameters off the stack to reconstruct program flow.

Stack window. An CrossView Pro window showing the function calls on the stack, with the values of the parameters passed to them.

startup options. Special command line switches passed to CrossView Pro when the debugger is first loaded. These options control items such as the number of assertions allowed, or can perform various actions such as to start recording screen output to a file.

stream. A particular input or output data path for simulated I/O. Each stream has a unique number for identifying purposes.

switches. *See* **startup options.**

symbolic debugger. A type of debugger generally limited to dealing with global, non–dynamic variables. Symbolic debuggers know nothing of the data types; they translate global names and global subroutines into addresses. *See also* **source level debugger**.

symbol information. The necessary information for CrossView Pro to correlate object code with source code. The symbol information is part of the absolute file (.abs file). *See also* **absolute file**.

system startup code. A run–time library routine written in assembly language source that initializes the target environment before calling `main()`. *See also* **main()**.

T

target communication. The low–level communication between the host and the target system. For the most part, CrossView Pro handles target communications, allowing the programmer to concentrate on the high–level information.

target microprocessor. The chip on which the target program runs.

target system. The emulator or ROM monitor where the target microprocessor resides, and on which the target program runs.

trace buffer. A target–resident buffer that contains the most recent commands executed by the target microprocessor. CrossView Pro uses this buffer to perform a C–trace.

transparency mode. The mode in which CrossView Pro passes user input directly to the emulator. Transparency mode is often used when setting up memory maps.

U

uplevel breakpoint. A code breakpoint set at the return from a function at a specified stack depth.

V

viewing position. The line of source code currently being viewed. This line contains the cursor. Some commands operate as a default on the viewing position. The viewing position and the execution position are initially the same, but you may adjust each individually.

W

window mode. An CrossView Pro operating mode in which the screen is divided into several windows (which you may control) where various information is grouped by category. *See also* **line mode**.

Z

zoom. To expand the Data, Register or Stack window to the full width of the screen.

APPENDIX

A

FLEXIBLE LICENSE MANAGER (FLEXlm)



A

APPENDIX

1 INTRODUCTION

This appendix discusses Globetrotter Software's Flexible License Manager and how it is integrated into the TASKING toolchain. It also contains descriptions of the Flexible License Manager license administration tools that are included with the package, the daemon log file and its contents, and the use of daemon options files to customize your use of the TASKING toolchain.

2 LICENSE ADMINISTRATION

2.1 OVERVIEW

The Flexible License Manager (FLEXlm) is a set of utilities that, when incorporated into software such as the TASKING toolchain, provides for managing access to the software.

The following terms are used to describe FLEXlm concepts and software components:

feature	A feature could be any of the following: <ul style="list-style-type: none">• A TASKING software product.• A software product from another vendor.
license	The right to use a feature. FLEXlm restricts licenses for features by counting the number of licenses for features in use when new requests are made by the application software.
client	A TASKING application program.
daemon	A process that "serves" clients. Sometimes referred to as a <i>server</i> .
vendor daemon	The daemon that dispenses licenses for the requested features. This daemon is built by an application's vendor, and contains the vendor's personal encryption code. Tasking is the vendor daemon for the TASKING software.

license daemon

The daemon process that sends client processes to the correct vendor daemon on the correct machine. The same license daemon is used by all applications from all vendors, as this daemon neither performs encryption nor dispenses licenses. The license daemon processes no user requests on its own, but forwards these requests to other daemons (the vendor daemons).

server node A computer system that is running both the license and vendor daemon software. The server node will contain all the dynamic information regarding the usage of all the features.

license file An end-user specific file that contains descriptions of the server nodes that can run the license daemons, the various vendor daemons, and the restrictions for all the licensed features.

The TASKING software is granted permission to run by FLEXlm daemons; the daemons are started when the TASKING toolchain is installed and run continuously thereafter. Information needed by the FLEXlm daemons to perform access management is contained in a license data file that is created during the toolchain installation process. As part of their normal operation, the daemons log their actions in a daemon log file, which can be used to monitor usage of the TASKING toolchain.

The following sections discuss:

- Installation of the FLEXlm daemons to provide for access to the TASKING toolchain.
- Customizing your use of the toolchain through the use of a daemon options file.
- Utilities that are provided to assist you in performing license administration functions.
- The daemon log file and its contents.

For additional information regarding the use of FLEXlm, refer to the chapter *Software Installation*.

2.2 PROVIDING FOR UNINTERRUPTED FLEXLM OPERATION

TASKING products licensed through FLEXlm contain a number of utilities for managing licenses. These utilities are bundled in the form of an extra product under the name SW000098. TASKING products themselves contain two additional files for FLEXlm in a *flexlm* subdirectory:

Tasking	The Tasking daemon (vendor daemon).
license.dat	A template license file.

If you have already installed FLEXlm (e.g. as part of another product) then it is not needed to install the bundled SW000098. After installing SW000098 on UNIX the directory `/usr/local/flexlm` will contain two subdirectories, `bin` and `licenses`. After installing SW000098 on Windows the directory `c:\flexlm` will contain the subdirectory `bin`. The exact location may differ if FLEXlm has already been installed as part of a non-TASKING product but in general there will be a directory for executables such as `bin`. That directory must contain a copy of the **Tasking** daemon shipped with every TASKING product. It also contains the files:

lmgrd	The FLEXlm daemon (license daemon).
lm*	A group of FLEXlm license administration utilities.

Next to it, a license file must be present containing the information of all licenses. This file is usually called `license.dat`. The default location of the license file is in directory `c:\flexlm` for Windows and in `/usr/local/flexlm/licenses` for UNIX. If you did install SW000098 then the `licenses` directory on UNIX will be empty, and on Windows the file `license.dat` will be empty. In that case you can copy the `license.dat` file from the product to the `licenses` directory after filling in the data from your "License Information Form".



Be very careful not to overwrite an existing `license.dat` file because it contains valuable data.

Example `license.dat`:

```
SERVER HOSTNAME HOSTID PORT
DAEMON Tasking /usr/local/flexlm/bin/Tasking
FEATURE SW008002-32 Tasking 3.000 EXPDATE NUSERS PASSWORD SERIAL
```

After modifications from a license data sheet (example):

```
SERVER elliot 5100520c 7594
DAEMON Tasking /usr/local/flexlm/bin/Tasking
FEATURE SW008002-32 Tasking 3.000 1-jan-00 4 0B1810310210A6894 "123456"
```

If the `license.dat` file already exists then you should make sure that it contains the DAEMON and FEATURE lines from your license data sheet. An appropriate SERVER line should already be present in that case. You should only add a new SERVER line if no SERVER line is present. The third field of the DAEMON line is the pathname to the **Tasking** daemon and you may change it if necessary.

The default location for the license file on Windows is:

```
c:\flexlm\license.dat
```

On UNIX this is:

```
/usr/local/flexlm/licenses/license.dat
```

If the pathname of the resulting license file differs from this default location then you must set the environment variable **LM_LICENSE_FILE** to the correct pathname. If you have more than one product using the FLEXlm license manager you can specify multiple license files by separating each pathname (*lfp_{path}*) with a ';' (on UNIX also ':') :

Windows:

```
set LM_LICENSE_FILE=lfppath;lfppath...
```

UNIX:

```
setenv LM_LICENSE_FILE lfppath[:lfppath]...
```

If you are running the TASKING software on multiple nodes, you have three options for making your license file available on all the machines:

1. Place the license file in a partition which is available (via NFS on Unix systems) to all nodes in the network that need the license file.
2. Copy the license file to all of the nodes where it is needed.
3. Set LM_LICENSE_FILE to "*port@host*", where *host* and *port* come from the SERVER line in the license file.

When the main license daemon **lmgrd** already runs it is sufficient to type the command:

```
lmreread
```

for notifying the daemon that the `license.dat` file has been changed. Otherwise, you must type the command:

```
lmgrd >/usr/tmp/license.log &
```

Both commands reside in the flexlm bin directory mentioned before.

2.3 DAEMON OPTIONS FILE

It is possible to customize the use of TASKING software using a daemon options file. This options file allows you to reserve licenses for specified users or groups of users, to restrict access to the TASKING toolchain, and to set software timeouts. The following table lists the keywords that are recognized at the start of a line of a daemon options file.

Keywords	Function
RESERVE	Ensure that TASKING software will always be available to one or more users or on one or more host computer systems.
INCLUDE	Specify a list of users who are allowed exclusive access to the TASKING software.
EXCLUDE	Specify a list of users who are not allowed to use the TASKING software.
GROUP	Specify a group of users for use in the other commands.
TIMEOUT	Allow licenses that are idle for a specified time to be returned to the free pool, for use by someone else.
NOLOG	Causes messages of the specified type to be filtered out of the daemon's log output.

Table A-1: Daemon options file keywords

In order to use the daemon options capability, you must create a daemon options file and list its pathname as the fourth field on the **DAEMON** line for the **Tasking** daemon in the license file. For example, if the daemon options were in file `/usr/local/flexlm/Tasking.opt` (UNIX), then you would modify the license file **DAEMON** line as follows:

```
DAEMON Tasking /usr/local/Tasking /usr/local/flexlm/Tasking.opt
```

A daemon options file consists of lines in the following format:

```
RESERVE      number feature {USER | HOST | DISPLAY | GROUP} name
INCLUDE      feature {USER | HOST | DISPLAY | GROUP} name
EXCLUDE      feature {USER | HOST | DISPLAY | GROUP} name
GROUP        name <list_of_users>
TIMEOUT      feature timeout_in_seconds
NOLOG        {IN | OUT | DENIED | QUEUED}
REPORTLOG    file
```

Lines beginning with the sharp character (#) are ignored, and can be used as comments. For example, the following options file would reserve one copy of feature SWxxxxxx-xx for user “pat”, three copies for user “lee”, and one copy for anyone on a computer with the hostname of “terry”; and would cause QUEUED messages to be omitted from the log file. In addition, user “joe” and group “pinheads” would not be allowed to use the feature SWxxxxxx-xx:

```
GROUP        pinheads moe larry curley
RESERVE 1    SWxxxxxx-xx USER pat
RESERVE 3    SWxxxxxx-xx USER lee
RESERVE 1    SWxxxxxx-xx HOST terry
EXCLUDE     SWxxxxxx-xx USER joe
EXCLUDE     SWxxxxxx-xx GROUP pinheads
NOLOG       QUEUED
```

3 LICENSE ADMINISTRATION TOOLS

The following utilities are provided to facilitate license management by your system administrator. In certain cases, execution access to a utility is restricted to users with root privileges. Complete descriptions of these utilities are provided at the end of this section.

lmcksum

Prints license checksums.

lmdiag (Windows only)

Diagnoses license checkout problems.

lmdown

Gracefully shuts down all license daemons (both **lmgrd** all vendor daemons, such as **Tasking**) on the license server.

lmgrd

The main daemon program for FLEXlm.

lmbostid

Reports the hostid of a system.

lmremove

Removes a single user's license for a specified feature.

lmreread

Causes the license daemon to reread the license file and start any new vendor daemons.

lmstat

Helps you monitor the status of all network licensing activities.

lmswitchr

Switches the report log file.

lmver

Reports the FLEXlm version of a library or binary file.

lmtools (*Windows only*)

This is a graphical Windows version of the license administration tools.

3.1 LMCKSUM

Name

lmcksum – print license checksums

Synopsis

lmcksum [**-c** *license_file*] [**-k**]

Description

The **lmcksum** program will perform a checksum of a license file. This is useful to verify data entry errors at your location. **lmcksum** will print a line-by-line checksum for the file as well as an overall file checksum.

The following fields participate in the checksum:

- hostid on the SERVER lines
- daemon name on the DAEMON lines
- feature name, version, daemon name, expiration date, # of licenses, encryption code, vendor string and hostid on the FEATURE lines
- daemon name and encryption code on FEATURESET lines

Options

-c *license_file*

Use the specified *license_file*. If no **-c** option is specified, **lmcksum** looks for the environment variable LM_LICENSE_FILE in order to find the license file to use. If that environment variable is not set, **lmcksum** looks for the file c:\flexlm\license.dat (Windows), or /usr/local/flexlm/licenses/license.dat (UNIX).

-k

Case-sensitive checksum. If this option is specified, **lmcksum** will compute the checksum using the exact case of the FEATURE's and FEATURESET's encryption code.

3.2 LMDIAG (Windows only)

Name

lmdiag – diagnose license checkout problems

Synopsis

lmdiag [**-c** *license_file*] [**-n**] [*feature*]

Description

lmdiag (Windows only) allows you to diagnose problems when you cannot check out a license.

If no *feature* is specified, **lmdiag** will operate on all features in the license file(s) in your path. **lmdiag** will first print information about the license, then attempt to check out each license. If the checkout succeeds, **lmdiag** will indicate this. If the checkout fails, **lmdiag** will give you the reason for the failure. If the checkout fails because **lmdiag** cannot connect to the license server, then you have the option of running "extended connection diagnostics".

These extended diagnostics attempt to connect to each port on the license server node, and can detect if the port number in the license file is incorrect. **lmdiag** will indicate each port number that is listening, and if it is an **lmgrd** process, **lmdiag** will indicate this as well. If **lmdiag** finds the vendor daemon for the feature being tested, then it will indicate the correct port number for the license file to correct the problem.

Parameters

feature Diagnose this feature only.

Options

-c *license_file*

Diagnose the specified *license_file*. If no **-c** option is specified, **lmdiag** looks for the environment variable `LM_LICENSE_FILE` in order to find the license file to use. If that environment variable is not set, **lmdiag** looks for the file `c:\flexlm\license.dat` (Windows), or `/usr/local/flexlm/licenses/license.dat` (UNIX).

-n

Run in non-interactive mode; **lmdiag** will not prompt for any input in this mode. In this mode, extended connection diagnostics are not available.

3.3 LMDOWN

Name

lmdown – graceful shutdown of all license daemons

Synopsis

lmdown [**-c** *license_file*] [**-q**]

Description

The **lmdown** utility allows for the graceful shutdown of all license daemons (both **lmgrd** and all vendor daemons, such as **Tasking**) on all nodes. You may want to protect the execution of **lmdown**, since shutting down the servers causes users to lose their licenses. See the **-p** option in Section 3.4, **lmgrd**.

lmdown sends a message to every license daemon asking it to shut down. The license daemons write out their last messages to the log file, close the file, and exit. All licenses which have been given out by those daemons will be revoked, so that the next time a client program goes to verify his license, it will not be valid.

Options

-c *license_file*

Use the specified *license_file*. If no **-c** option is specified, **lmdown** looks for the environment variable **LM_LICENSE_FILE** in order to find the license file to use. If that environment variable is not set, **lmdown** looks for the file `c:\flexlm\license.dat` (Windows), or `/usr/local/flexlm/licenses/license.dat` (UNIX).

-q

Quiet mode. If this switch is not specified, **lmdown** asks for confirmation before asking the license daemons to shut down. If this switch is specified, **lmdown** will not ask for confirmation.



lmgrd, lmstat, lmread

3.4 LMGRD

Name

lmgrd – flexible license manager daemon

Synopsis

lmgrd [**-c** *license_file*] [**-l** *logfile*] [**-2 -p**] [**-t** *timeout*] [**-s** *interval*]

Description

lmgrd is the main daemon program for the FLEXlm distributed license management system. When invoked, it looks for a license file containing all required information about vendors and features. On UNIX systems, it is strongly recommended that **lmgrd** be run as a non-privileged user (not root).

Options

-c *license_file*

Use the specified *license_file*. If no **-c** option is specified, **lmgrd** looks for the environment variable `LM_LICENSE_FILE` in order to find the license file to use. If that environment variable is not set, **lmgrd** looks for the file `c:\flexlm\license.dat` (Windows), or `/usr/local/flexlm/licenses/license.dat` (UNIX).

-l *logfile*

Specifies the output log file to use. Instead of using the **-l** option you can use output redirection (**>** or **>>**) to specify the name of the output log file.

-2 -p

Restricts usage of **lmdown**, **lmreread**, and **lmremove** to a FLEXlm administrator who is by default root. If there is a UNIX group called "lmadmin" then use is restricted to only members of that group. If root is not a member of this group, then root does not have permission to use any of the above utilities.

-t *timeout*

Specifies the *timeout* interval, in seconds, during which the license daemon must complete its connection to other daemons if operating in multi-server mode. The default value is 10 seconds. A larger value may be desirable if the daemons are being run on busy systems or a very heavily loaded network.

-s *interval* Specifies the log file timestamp *interval*, in minutes. The default is 360 minutes. This means that every six hours **lmgrd** logs the time in the log file.



lmdown, lmstat

3.5 LMHOSTID

Name

lmhostid – report the hostid of a system

Synopsis

lmhostid

Description

lmhostid calls the FLEXlm version of `gethostid` and displays the results.

The output of **lmhostid** looks like this:

```
lmhostid - Copyright (C) 1989, 1999 Globetrotter Software, Inc.  
The FLEXlm host ID of this machine is "1200abcd"
```

Options

lmhostid has no command line options.

3.6 LMREMOVE

Name

lmremove – remove specific licenses and return them to license pool

Synopsis

lmremove [**-c** *license_file*] *feature user host* [*display*]

Description

The **lmremove** utility allows the system administrator to remove a single user's license for a specified feature. This could be required in the case where the licensed user was running the software on a node that subsequently crashed. This situation will sometimes cause the license to remain unusable. **lmremove** will allow the license to return to the pool of available licenses.

lmremove will remove all instances of “user” on node “host” on display “display” from usage of “feature”. If the optional **-c file** is specified, the indicated file will be used as the license file. Since removing a user's license can be disruptive, execution of **lmremove** is restricted to users with root privileges.

Options

-c *license_file*

Use the specified *license_file*. If no **-c** option is specified, **lmremove** looks for the environment variable `LM_LICENSE_FILE` in order to find the license file to use. If that environment variable is not set, **lmremove** looks for the file `c:\flexlm\license.dat` (Windows), or `/usr/local/flexlm/licenses/license.dat` (UNIX).



lmstat

3.7 LMREREAD

Name

lmreread – tells the license daemon to reread the license file

Synopsis

lmreread [**-c** *license_file*]

Description

lmreread allows the system administrator to tell the license daemon to reread the license file. This can be useful if the data in the license file has changed; the new data can be loaded into the license daemon without shutting down and restarting it.

The license administrator may want to protect the execution of **lmreread**. See the **-p** option in Section 3.4, *lmgrd* for details about securing access to **lmreread**.

lmreread uses the license file from the command line (or the default file, if none specified) only to find the license daemon to send it the command to reread the license file. The license daemon will always reread the file that it loaded from the original path. If you need to change the path to the license file read by the license daemon, then you must shut down the daemon and restart it with that new license file path.

You cannot use **lmreread** if the *SERVER* node names or port numbers have been changed in the license file. In this case, you must shut down the daemon and restart it in order for those changes to take effect.

lmreread does not change any option information specified in an options file. If the new license file specifies a different options file, that information is ignored. If you need to reread the options file, you must shut down (**lmdown**) the daemon and restart it.

Options

-c *license_file*

Use the specified *license_file*. If no **-c** option is specified, **lmreread** looks for the environment variable *LM_LICENSE_FILE* in order to find the license file to use. If that environment variable is not set, **lmreread** looks for the file *license.dat* in the default location.



lmdown

3.8 LMSTAT

Name

lmstat – report status on license manager daemons and feature usage

Synopsis

```
lmstat [ -a ] [ -A ] [ -c license_file ] [ -f feature ]
      [ -l regular_expression ] [ -s server ] [ -S daemon ] [ -t timeout ]
```

Description

License administration is simplified by the **lmstat** utility. **lmstat** allows you to instantly monitor the status of all network licensing activities.

lmstat allows a system administrator to monitor license management operations including:

- Which daemons are running
- Users of individual features
- Users of features served by a specific DAEMON

Options

-a Display all information.

-A List all active licenses.

-c *license_file*

Use the specified *license_file*. If no **-c** option is specified, **lmstat** looks for the environment variable LM_LICENSE_FILE in order to find the license file to use. If that environment variable is not set, **lmstat** looks for the file c:\flexlm\license.dat (Windows), or /usr/local/flexlm/licenses/license.dat (UNIX).

-f *feature* List all users of the specified *feature*(s).

-l *regular_expression*

List all users of the features matching the given *regular_expression*.

-s *server* Display the status of the specified *server* node(s).

-S *daemon* List all users of the specified *daemon*'s features.

- t *timeout*** Specifies the amount of time, in seconds, **lmstat** waits to establish contact with the servers. The default value is 10 seconds. A larger value may be desirable if the daemons are being run on busy systems or a very heavily loaded network.



lmgrd

3.9 LMSWITCHR (Windows only)

Name

lmswitchr – switch the report log file

Synopsis

lmswitchr [**-c** *license_file*] *feature new-file*

or:

lmswitchr [**-c** *license_file*] *vendor new-file*

Description

lmswitchr (Windows only) switches the report writer (REPORTLOG) log file. It will also start a new REPORTLOG file if one does not already exist.

Parameters

<i>feature</i>	Any feature this daemon supports.
<i>vendor</i>	The name of the vendor daemon (such as Tasking).
<i>new-file</i>	New file path.

Options

-c *license_file* Use the specified *license_file*. If no **-c** option is specified, **lmswitchr** looks for the environment variable LM_LICENSE_FILE in order to find the license file to use. If that environment variable is not set, **lmswitchr** looks for the file `c:\flexlm\license.dat` (Windows), or `/usr/local/flexlm/licenses/license.dat` (UNIX).

3.10 LMVER

Name

lmver – report the FLEXlm version of a library or binary file

Synopsis

lmver *filename*

Description

The **lmver** utility reports the FLEXlm version of a library or binary file.

Alternatively, on UNIX systems, you can use the following commands to get the FLEXlm version of a binary:

strings *file* | grep Copy

Parameters

filename Name of the executable of the product.

3.11 LICENSE ADMINISTRATION TOOLS FOR WINDOWS

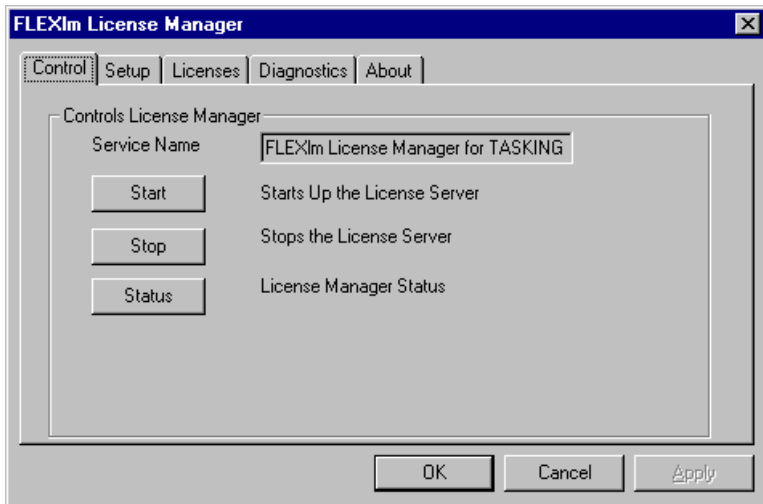
3.11.1 LMTOOLS FOR WINDOWS

For the 32 Bit Windows Platforms, an **lmtools.exe** Windows program is provided. It has the same functionality as listed in the previous sections but is graphically-oriented. Simply run the program (Start | Programs | TASKING FLEXlm | FLEXlm Tools) and choose a button for the functionality required. Refer to the previous sections for information about the options of each feature. The command line interface is replaced by pop-up dialogs that can be filled out. The central EDIT field is where the license file path is placed. This will be used for all other functions and replaces the "**-c** *license_file*" argument in the other utilities.

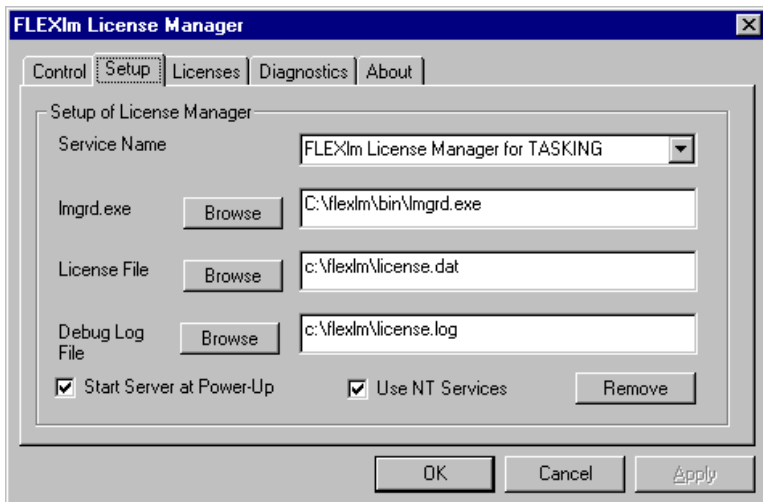
The HOSTID button displays the hostid's for the computer on which the program is running. The TIME button prints out the system's internal time settings, intended to diagnose any time zone problems. The TCP Settings button is intended to fix a bug in the Microsoft TCP protocol stack which has a symptom of very slow connections to computers. After pressing this button, the system will need to be rebooted for the settings to become effective.

3.11.2 FLEXLM LICENSE MANAGER FOR WINDOWS

lmgrd.exe can be run manually or using the graphical Windows tool. You can start this tool from the FLEXlm program folder. Click on Start | Programs | TASKING FLEXlm | FLEXlm Tools



From the Control tab you can start, stop, and check the status of your license server. Select the Setup tab to enter information about your license server.



Select the **Control** tab and click the **Start** button to start your license server. **lmgrd.exe** will be launched as a background application with the license file and debug log file locations passed as parameters.

If you want **lmgrd.exe** to start automatically on NT, select the **Use NT Services** check box and **lmgrd.exe** will be installed as an NT service. Next, select the **Start Server at Power-UP** check box.

The **Licenses** tab provides information about the license file and the **Advanced** tab allows you to perform diagnostics and check versions.

4 THE DAEMON LOG FILE

The FLEXlm daemons all generate log files containing messages in the following format:

mm/dd hh:mm (DAEMON name) message

Where:

mm/dd hh:mm Is the month/day hour:minute that the message was logged.

DAEMON name Either “license daemon” or the string from the DAEMON line that describes your daemon.

In the case where a single copy of the daemon cannot handle all of the requested licenses, an optional “_” followed by a number indicates that this message comes from a forked daemon.

message The text of the message.

The log files can be used to:

- Inform you when it may be necessary to update your application software licensing arrangement.
- Diagnose configuration problems.
- Diagnose daemon software errors.

The messages are grouped below into the above three categories, with each message followed by a brief description of its meaning.

4.1 INFORMATIONAL MESSAGES

Connected to node

This daemon is connected to its peer on node *node*.

CONNECTED, master is name

The license daemons log this message when a quorum is up and everyone has selected a master.

DEMO mode supports only one SERVER host!

An attempt was made to configure a demo version of the software for more than one server host.

DENIED: N feature to user (mm/dd/yy hh:mm)

user was denied access to *N* licenses of *feature*. This message may indicate a need to purchase more licenses.

EXITING DUE TO SIGNAL mm

EXITING with code mm

All daemons list the reason that the daemon has exited.

EXPIRED: feature

feature has passed its expiration date.

IN: feature by user (N licenses) (used: d:hh:mm:ss) (mm/dd/yy hh:mm)

user has checked back in *N* licenses of *feature* at *mm/dd/yy hh:mm*.

IN server died: feature by user (number licenses) (used: d:hh:mm:ss) (mm/dd/yy hh:mm)

user has checked in *N* licenses by virtue of the fact that his server died.

License Manager server started

The license daemon was started.

Lost connection to host

A daemon can no longer communicate with its peer on node *host*, which can cause the clients to have to reconnect, or cause the number of daemons to go below the minimum number, in which case clients may start exiting. If the license daemons lose the connection to the master, they will kill all the vendor daemons; vendor daemons will shut themselves down.

Lost quorum

The daemon lost quorum, so will process only connection requests from other daemons.

MASTER SERVER died due to signal mm

The license daemon received fatal signal *mm*.

MULTIPLE xxx servers running. Please kill, and restart license daemon

The license daemon has detected that multiple copies of vendor daemon *xxx* are running. The user should kill all *xxx* daemon processes and re-start the license daemon.

OUT: feature by user (N licenses) (mm/dd/yy hh:mm)

user has checked out *N* licenses of *feature* at *mm/dd/yy hh:mm*

Removing clients of children

The top-level daemon logs this message when one of the child daemons dies.

RESERVE feature for HOST name***RESERVE feature for USER name***

A license of *feature* is reserved for either user *name* or host *name*.

REStarted xxx (internet port mm)

Vendor daemon *xxx* was restarted at internet port *mm*.

Retrying socket bind (address in use)

The license servers try to bind their sockets for approximately 6 minutes if they detect *address in use* errors.

Selected (EXISTING) master node

This license daemon has selected an existing master (node) as the master.

SERVER shutdown requested

A daemon was requested to shut down via a user-generated kill command.

[NEW] Server started for: feature-list

A (possibly new) server was started for the features listed.

Shutting down xxx

The license daemon is shutting down the vendor daemon *xxx*.

SIGCHLD received. Killing child servers

A vendor daemon logs this message when a shutdown was requested by the license daemon.

Started name

The license daemon logs this message whenever it starts a new vendor daemon.

Trying connection to node

The daemon is attempting a connection to *node*.

4.2 CONFIGURATION PROBLEM MESSAGES

hostname: Not a valid server host, exiting

This daemon was run on an invalid hostname.

hostname: Wrong hostid, exiting

The hostid is wrong for *hostname*.

BAD CODE for feature-name

The specified feature name has a bad encryption code.

CANNOT OPEN options file “file”

The options file specified in the license file could not be opened.

Couldn't find a master

The daemons could not agree on a master.

license daemon: lost all connections

This message is logged when all the connections to a server are lost, which often indicates a network problem.

lost lock, exiting

Error closing lock file

Unable to re-open lock file

The vendor daemon has a problem with its lock file, usually because of an attempt to run more than one copy of the daemon on a single node. Locate the other daemon that is running via a **ps** command, and kill it with **kill -9**.

NO DAEMON line for daemon

The license file does not contain a DAEMON line for *daemon*.

No “license” service found

The TCP *license* service did not exist in `/etc/services`.

No license data for “feat”, feature unsupported

There is no feature line for *feat* in the license file.

No features to serve!

A vendor daemon found no features to serve. This could be caused by bad data in the license file.

UNSUPPORTED FEATURE request: feature by user

The *user* has requested a feature that this vendor daemon does not support. This can happen for a number of reasons: the license file is bad, the feature has expired, or the daemon is accessing the wrong license file.

Unknown host: hostname

The hostname specified on a `SERVER` line in the license file does not exist in the network database (probably `/etc/hosts`).

lm_server: lost all connections

This message is logged when all the connections to a server are lost. This probably indicates a network problem.

NO DAEMON lines, exiting

The license daemon logs this message if there are no `DAEMON` lines in the license file. Since there are no vendor daemons to start, there is nothing to do.

NO DAEMON line for name

A vendor daemon logs this error if it cannot find its own `DAEMON` name in the license file.

4.3 DAEMON SOFTWARE ERROR MESSAGES

accept: message

An error was detected in the accept system call.

ATTEMPT TO START VENDOR DAEMON xxx with NO MASTER

A vendor daemon was started with no master selected. This is an internal consistency error in the daemons.

BAD PID message from mm: pid: xxx (msg)

A top-level vendor daemon received an invalid PID message from one of its children (daemon number xxx).

BAD SCONNECT message: (message)

An invalid “server connect” message was received.

Cannot create pipes for server communication

The pipe call failed.

Can't allocate server table space

A malloc error. Check swap space.

Connection to node TIMED OUT

The daemon could not connect to *node*.

Error sending PID to master server

The vendor server could not send its PID to the top-level server in the hierarchy.

Illegal connection request to DAEMON

A connection request was made to DAEMON, but this vendor daemon is not DAEMON.

Illegal server connection request

A connection request came in from another server without a DAEMON name.

KILL of child failed, errno = mm

A daemon could not kill its child.

No internet port number specified

A vendor daemon was started without an internet port.

Not enough descriptors to re-create pipes

The “top-level” daemon detected one of its sub-daemon’s death. In trying to restart the chain of sub-daemons, it was unable to get the file descriptors to set up the pipes to communicate. This is a fatal error, and the daemons must be re-started.

read: error message

An error in a read system call was detected.

recycle_control BUT WE DIDN'T HAVE CONTROL

The hierarchy of vendor daemons has become confused over who holds the control token. This is an internal error.

return_reserved: can't find feature listhead

When a daemon is returning a reservation to the “free reservation” list, it could not find the listhead of features.

select: message

An error in a select system call was detected.

Server exiting

The server is exiting. This is normally due to an error.

SHELLO for wrong DAEMON

This vendor daemon was sent a “server hello” message that was destined for a different DAEMON.

Unsolicited msg from parent!

Normally, the top-level vendor daemon sends no unsolicited messages. If one arrives, this message is logged. This is a bug.

WARNING: CORRUPTED options list (o->next == 0)***Options list TERMINATED at bad entry***

An internal inconsistency was detected in the daemon’s option list.

5 FLEXLM LICENSE ERRORS

FLEXlm license error, encryption code in license file is inconsistent

Check the contents of the license file using the license data sheet for the product. Correct the license file and run the **lmreread** command.

However, do not change the last (fourth) field of a SERVER line in the license file. This cannot have any effect on the error message but changing it will cause other problems.

license file does not support this version

If this is a first time install then follow the procedure for the error message:

```
FLEXlm license error, encryption code in license file is
inconsistent
```

because there may be a typo in the fourth field of a FEATURE line of your license file. In all other cases you need a new license because the current license is for an older version of the product.

Replace the FEATURE line for the old version of the product with a FEATURE line for the new version (it can be found on the new license data sheet). Run the **lmreread** command afterwards. You can have only one version of a feature (previous versions of the product will continue to work).

FLEXlm license error, cannot find license file

Make sure the license file exists. If the pathname printed on the line after the error message is incorrect, correct this by setting the `LM_LICENSE_FILE` environment variable to the full pathname of the license file.

FLEXlm license error, cannot read license file

Every user needs to have read access on the license file and at least execute access on every directory component in the pathname of the license file. Write access is never needed. Read access on directories is recommended.

FLEXlm license error, no such feature exists

Check the license file. There should be a line starting with:

```
FEATURE SWiiiiii-jj
```


where "iiiiii" is a six digit software code and "jj" is a two digit host code for identifying a compatible host architecture. During product installations the product code is shown, e.g. SW008002, SW019002. The number in the software code is the same as the number in the product code except that the first number may contain an extra leading zero (it must be six digits long).

The line after the license error message describes the expected feature format and includes the host code.

Correct the license file using the license data sheet for the product and run the **lmreread** command. There is one catch: do not add extra SERVER lines or change existing SERVER lines in the license file.

FLEXlm license error, license server does not support this feature

If the LM_LICENSE_FILE variable has been set to the format *number@host* then see first the solution for the message:

```
FLEXlm license error, no such feature exists
```

Run the **lmreread** program to inform the license server about a changed license data file. If **lmreread** succeeds informing the license server but the error message persists, there are basically three possibilities:

1. The license key is incorrect. If this is the case then there must be an error message in the log file of **lmgrd**. Correct the key using the license data sheet for the product. Finally rerun **lmreread**. The log file of **lmgrd** is usually specified to **lmgrd** at startup with the **-l** option or with **>**.
2. Your network has more than one FLEXlm license server daemon and the default license file location for **lmreread** differs from the default assumed by the program. Also, there must be more than one license file. Try one of the following solutions on the same host which produced the error message:

- type:

```
lmreread -c /usr/local/flexlm/licenses/license.dat
```

- set LM_LICENSE_FILE to the license file location and retry the **lmreread** command.
- use the **lmreread** program supplied with the product SW000098, Flexible License Manager. SW000098 is bundled with all TASKING products.

3. There is a protocol version mismatch between **lmgrd** and the daemon with the name "Tasking" (the vendor daemon according to FLEXlm terminology) or there is some other internal error. These errors are always written to the log file of **lmgrd**. The solution is to upgrade the **lmgrd** daemon to the one supplied in SW000098, the bundled Flexible License Manager product.

On the other hand, if **lmreread** complains about not being able to connect to the license server then follow the procedure described in the next section for the error message "Cannot read license file data from server". The only difference with the current situation is that not the product but a license management utility shows a connect problem.

FLEXlm license error, Cannot read license file data from server

This indicates that the program could not connect to the license server daemon. This can have a number of causes. If the program did not immediately print the error message but waited for about 30 seconds (this can vary) then probably the license server host is down or unreachable. If the program responded immediately with the error message then check the following if the LM_LICENSE_FILE variable has been set to the format *number@host*:

- is the number correct? It should match the fourth field of a SERVER line in the license file on the license server host. Also, the host name on that SERVER line should be the same as the host name set in the LM_LICENSE_FILE variable. Correct LM_LICENSE_FILE if necessary.

In any case one should verify if the license server daemon is running. Type the following command on the host where the license server daemon (**lmgrd**) is supposed to run.

On SunOS 4.x:

```
ps wwax | grep lmgrd | grep -v grep
```

On HP-UX or SunOS 5.x (Solaris 2.x):

```
ps -ef | grep lmgrd | grep -v grep
```

If the command does not produce any output then the license server daemon is not running. See below for an example how to start **lmgrd**.

Make sure that both license server daemon (**lmgrd**) and the program are using the same license data. All TASKING products use the license file `/usr/local/flexlm/licenses/license.dat` unless overruled by the environment variable `LM_LICENSE_FILE`. However, not all existing **lmgrd** daemons may use the same default. In case of doubt, specify the license file pathname with the `-c` option when starting the license server daemon. For example:

```
lmgrd -c /usr/local/flexlm/licenses/license.dat \  
-l /usr/local/flexlm/licenses/license.log &
```

and set the `LM_LICENSE_FILE` environment variable to the `license.dat` pathname mentioned with the `-c` option of **lmgrd** before running any license based program (including **lmreread**, **lmstat**, **lmdown**). If **lmgrd** and the program run on different hosts, transparent access to the license file is assumed in the situation described above (e.g. NFS). If this is not the case, make a local copy of the license file (not recommended) or set `LM_LICENSE_FILE` to the form *number@host*, as described earlier.

If none of the above seems to apply (i.e. **lmgrd** was already running and `LM_LICENSE_FILE` has been set correctly) then it is very likely that there is a TCP port mismatch. The fourth field of a `SERVER` line in the license file specifies a TCP port number. That number can be changed without affecting any license. However, it must never be changed while the license server daemon is running. If it has been changed, change it back to the original value. If you do not know the original number anymore, restart the license server daemon after typing the following command on the license server host:

```
kill PID
```

where `PID` is the process id of **lmgrd**.

6 FREQUENTLY ASKED QUESTIONS (FAQS)

6.1 LICENSE FILE QUESTIONS

I've received FLEXlm license files from 2 different companies. Do I have to combine them?

You don't have to combine license files. Each license file that has any 'counted' lines (the 'number of licenses' field is >0) requires a server. It's perfectly OK to have any number of separate license files, with different **lmgrd** server processes supporting each file. Moreover, since **lmgrd** is a lightweight process, for sites without system administrators, this is often the simplest (and therefore recommended) way to proceed. With v6+ **lmgrd/lmdown/lmreread**, you can stop/reread/restart a single vendor daemon (of any FLEXlm version). This makes combining licenses more attractive than previously. Also, if the application is v6+, using 'dir/*.lic' for license file management behaves like combining licenses without physically combining them.

When is it recommended to combine license files?

Many system administrators, especially for larger sites, prefer to combine license files to ease administration of FLEXlm licenses. It's purely a matter of preference.

Does FLEXlm handle dates in the year 2000 and beyond?

Yes. The FLEXlm date format uses a 4-digit year. Dates in the 20th century (19xx) can be abbreviated to the last 2 digits of the year (xx), and use of this feature is quite widespread. Dates in the year 2000 and beyond must specify all 4 year digits.

6.2 FLEXLM VERSION

Which FLEXlm versions does TASKING deliver?

For Windows we deliver FLEXlm v6.1 and for UNIX we deliver v2.4.

I have products from several companies at various FLEXlm version levels. Do I have to worry about how these versions work together?

If you're not combining license files from different vendors, the simplest thing to do is make sure you use the tools (especially **lmgrd**) that are shipped by each vendor.

lmgrd will always correctly support older versions of vendor daemons and applications, so it's **always** safe to use the latest version of **lmgrd** and the other FLEXlm utilities. If you've combined license files from 2 vendors, you **must** use the latest version of **lmgrd**.

If you've received 2 versions of a product from the same vendor, you must use the latest vendor daemon they sent you. An older vendor daemon with a newer client will cause communication errors.

Please ignore letters appended to FLEXlm versions, i.e., v2.4d. The appended letter indicates a patch, and does NOT indicate any compatibility differences. In particular, some elements of FLEXlm didn't require certain patches, so a 2.4 **lmgrd** will work successfully with a 2.4b vendor daemon.

I've received a new copy of a product from a vendor, and it uses a new version of FLEXlm. Is my old license file still valid?

Yes. Older FLEXlm license files are always valid with newer versions of FLEXlm.

6.3 WINDOWS QUESTIONS

What Windows Host Platforms can be used as a server for Floating Licenses?

The system being used as the server (where the FLEXlm License Manager is running) for Floating licenses, must be Windows NT. The FLEXlm License Manager does not run properly with Windows 95/98.

Why do I need to include NWlink IPX/SPX on NT?

This is necessary for either obtaining the Ethernet card address, or to provide connectivity with a Netware License server.

6.4 TASKING QUESTIONS

How will the TASKING licensing/pricing model change with License Management (FLEXlm)?

TASKING will now offer the following types of licenses so you can purchase licenses based upon usage:

License	Description	Pricing
Node Locked	This license can only be used on a specific system. It cannot be moved to another system.	The pricing for this license will be the current product pricing.
Floating	This license requires a network (license server and a TCP/IP (or IPX/SPX) connection between clients and server) and can be used on any host system (using the same operating system) in the network.	The pricing for this license will be 50% higher than the node locked license.

How does FLEXlm affect future product ordering?

For all licenses, node locked or floating, you must provide information that is used to create a license key. For node locked licenses we must have the HOST ID. Floating licenses require the HOST ID and HOST NAME. The HOST ID is a unique identification of the machine, which is based upon different hardware depending upon host platform. The HOST NAME is the network name of the machine.



TASKING Logistics CANNOT ship ANY orders that do not include the HOST ID and/or HOST NAME information.

What if I do not know the information needed for the license key?

We have a software utility (**tkhostid.exe**) which will obtain and display the HOST ID so a customer can easily obtain this information. This utility is available from our web site, placed on all product CDs (which support FLEXlm), and from technical support. If you have already installed FLEXlm, you can also use **lmhostid**.

- In the case of a *Node locked license*, it is important that the customer runs this utility on the exact machine he intends to run the TASKING tools on.

- In the case of a *Floating License*, the **tkhostid.exe** (or **lmhostid**) utility should be run on the machine on which the FLEXlm license manager will be installed, e.g. the server. The HOST NAME information can be obtained from within the Windows Control Panel. Select "Network", click on "Identification", look for "Computer name".

How will the "locking" mechanism work?

- For node locked licenses, FLEXlm will first search for an ethernet card. If one exists, it will lock onto the number of the ethernet card. If an ethernet card does not exist, FLEXlm will lock onto the hard disk serial number.
- For floating licenses, the ethernet card number will be used.

What happens if I try to move my node locked license to another system?

The software will not run.

What does linger-time for floating licenses mean?

When the TASKING product starts to run, it will try to obtain a license from the license server. The license server keeps track of the number of licenses already issued, and grants or denies the request. When the software has finished running, the license is kept by the license server for a period of time known as the "linger-time". If the same user requests the TASKING product again within the linger-time, he is granted the license again. If another user requests a license during the linger-time, his request is denied until the linger-time has finished.

What is the length of the linger-time for floating licenses?

The length of the linger-time for both the PC and UNIX floating licenses is 5 minutes.

Can the linger-time be changed?

Yes. A customer can change the linger-time to be larger (but not shorter) than the time specified by TASKING.

What happens if my system crashes or I upgrade to a new system?

You will need to contact Technical Support for temporary license keys due to a system crash or to move from one system to another system. You will then need to work with your local sales representative to obtain a permanent new license key.

6.5 USING FLEXLM FOR FLOATING LICENSES

Does FLEXlm work across the internet?

Yes. A server on the internet will serve licenses to anyone else on the internet. This can be limited with the 'INTERNET=' attribute on the FEATURE line, which limits access to a range of internet addresses. You can also use the INCLUDE and EXCLUDE options in the daemon option file to allow (or deny) access to clients running on a range of internet addresses.

Does FLEXlm work with Internet firewalls?

Many firewalls require that port numbers be specified to the firewall. FLEXlm v5 **lmgrd** supports this.

If my client dies, does the server free the license?

Yes, unless the client's whole system crashes. Assuming communications is TCP, the license is automatically freed immediately. If communications are UDP, then the license is freed after the UDP timeout, which is set by each vendor, but defaults to 45 minutes. UDP communications is normally only set by the end-user, so TCP should be assumed. If the whole system crashes, then the license is not freed, and you should use '**lmremove**' to free the license.

What happens when the license server dies?

FLEXlm applications send periodic heartbeats to the server to discover if it has died. What happens when the server dies is then up to the application. Some will simply continue periodically attempting to re-checkout the license when the server comes back up. Some will attempt to re-checkout a license a few times, and then, presumably with some warning, exit. Some GUI applications will present pop-ups to the user periodically letting them know the server is down and needs to be re-started.

How do you tell if a port is already in use?

99.44% of the time, if it's in use, it's because **lmgrd** is already running on the port – or was recently killed, and the port isn't freed yet. Assuming this is not the case, then use '**telnet host port**' – if it says "*can't connect*", it's a free port.

Does FLEXlm require root permissions?

No. There is no part of FLEXlm, **lmgrd**, vendor daemon or application, that requires root permissions. In fact, it is strongly recommended that you do not run the license server (**lmgrd**) as root, since root processes can introduce security risks.

If **lmgrd** must be started from the root user (for example, in a system boot script), we recommend that you use the '**su**' command to run **lmgrd** as a non-privileged user:

```
su username -c"/path/lmgrd -c /path/license.dat \  
-l /path/log"
```

where *username* is a non-privileged user, and *path* is the correct paths to **lmgrd**, *license.dat* and debug log file. You will have to ensure that the vendor daemons listed in */path-to-license/license.dat* have execute permissions for *username*. The paths to all the vendor daemons in the license file are listed on each DAEMON line.

Is it ok to run lmgrd as 'root' (UNIX only)?

It is not prudent to run any command, particularly a daemon, as root on UNIX, as it may pose a security risk to the Operating System. Therefore, we recommend that **lmgrd** be run as a non-privileged user (not 'root'). If you are starting **lmgrd** from a boot script, we recommend that you use

```
su username -c"umask 022; /path/lmgrd \  
-c /path/license.dat -l /path/log"
```

to run **lmgrd** as a non-privileged user.

Does FLEXlm licensing impose a heavy load on the network?

No, but partly this depends on the application, and end-user's use. A typical checkout request requires 5 messages and responses between client and server, and each message is < 150 bytes.

When a server is not receiving requests, it requires virtually no CPU time. When an application, or **lmstat**, requests the list of current users, this can significantly increase the amount of networking FLEXlm uses, depending on the number of current users. Also, prior to FLEXlm v5, use of 'port@host' can increase network load, since the license file is down-loaded from the server to the client. 'port@host' should be, if possible, limited to small license files (say < 50 features). In v5, 'port@host' actually improves performance.

Does FLEXlm work with NFS?

Yes. FLEXlm has no direct interaction with NFS. FLEXlm uses an NFS-mounted file like any other application.

Does FLEXlm work with ATM, ISDN, Token-Ring, etc.?

In general, these have no impact on FLEXlm. FLEXlm requires TCP/IP or SPX (Novell Netware). So long as TCP/IP works, FLEXlm will work.

Does FLEXlm work with subnets, fully-qualified names, multiple domains, etc.?

Yes, although this behavior was improved in v3.0, and v6.0. When a license server and a client are located in different domains, fully-qualified host names have to be used. A fully-qualified hostname is of the form:

node.domain

where *node* is the local hostname (usually returned by the '**hostname**' command or '**uname -n**') *domain* is the internet domain name, e.g. 'globes.com'.

To ensure success with FLEXlm across domains, do the following:

1. Make the sure the fully-qualified hostname is the name on the SERVER line of the license file.
2. Make sure ALL client nodes, as well as the server node, are able to 'telnet' to that fully-qualified hostname. For example, if the host is locally called 'speedy', and the domain name is 'corp.com', local systems will be able to logon to speedy via 'telnet speedy'. But very often, 'telnet speedy.corp.com' will fail, locally.
Note that this telnet command will always succeed on hosts in other domains (assuming everything is configured correctly), since the network will resolve speedy.corp.com automatically.
3. Finally, there must be an 'alias' for speedy so it's also known locally as speedy.corp.com. This alias is added to the `/etc/hosts` file, or if NIS/Yellow Pages are being used, then it will have to be added to the NIS database. This requirement goes away in version 3.0 of FLEXlm.

If all components (application, **lmgrd** and vendor daemon) are v6.0 or higher, no aliases are required; the only requirement is that the fully-qualified domain name, or IP-address, is used as a hostname on the SERVER, or as a hostname in `LM_LICENSE_FILE port@host, or @host`.

Does FLEXlm work with NIS and DNS?

Yes. However, some sites have broken NIS or DNS, which will cause FLEXlm to fail. In v5 of FLEXlm, NIS and DNS can be avoided to solve this problem. In particular, sometimes DNS is configured for a server that's not current available (e.g., a dial-up connection from a PC). Again, if DNS is configured, but the server is not available, FLEXlm will fail.

In addition, some systems, particularly Sun, SGI, HP, require that applications be linked dynamically to support NIS or DNS. If a vendor links statically, this can cause the application to fail at a site that uses NIS or DNS. In these situations, the vendor will have to relink, or recompile with v5 FLEXlm. Vendors are strongly encouraged to use dynamic libraries for libc and networking libraries, since this tends to improve quality in general, as well as making NIS/DNS work.

On PCs, if a checkout seems to take 3 minutes and then fails, this is usually because the system is configured for a dial-up DNS server which is not currently available. The solution here is to turn off DNS.

Finally, hostnames must NOT have periods in the name. These are not legal hostnames, although PCs will allow you to enter them, and they will not work with DNS.

We're using FLEXlm over a wide-area network. What can we do to improve performance?

FLEXlm network traffic should be minimized. With the most common uses of FLEXlm, traffic is negligible. In particular, checkout, checkin and heartbeats use very little networking traffic. There are two items, however, which can send considerably more data and should be avoided or used sparingly:

- **'lmstat -a'** should be used sparingly. **'lmstat -a'** should not be used more than, say, once every 15 minutes, and should be particularly avoided when there's a lot of features, or concurrent users, and therefore a lot of data to transmit; say, more than 20 concurrent users or features.
- Prior to FLEXlm v5, the 'port@host' mode of the LM_LICENSE_FILE environment variable should be avoided, especially when the license file has many features, or there are a lot of license files included in LM_LICENSE_FILE. The license file information is sent via the network, and can place a heavy load. Failures due to 'port@host' will generate the error LM_SERVNOREADLIC (-61).

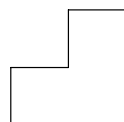
APPENDIX

B

SOUND SUPPORT (MS–Windows)



TASKING



B

APPENDIX

You can have sound effects being played when a predefined event in CrossView Pro occurs. You can configure the sound in the Sound settings of the Control Panel of MS–Windows. Similar to assigning a sound to a system event, you can assign a sound to a CrossView Pro event.

Currently the following events are supported:

- Breakpoint hit
- File has been downloaded
- CrossView Pro has started execution
- CrossView Pro is exiting
- Run command/button
- Step command/button
- StepOver command/button
- Halt command/button
- Symbols Loaded
- Fatal (system) error occurred
- Non-fatal error

How to add sound support

1. Firstly all events must be specified to MS–Windows. You can do this by adding the following lines to the Registry under:

```
My Computer\HKEY_CURRENT_USER\AppData\Local\Microsoft\Windows\CurrentVersion\EventLabels\
```

Use **regedit** to start the registry editor.

snd_xvw_bphit	"XVW Breakpoint Hit"
snd_xvw_download	"XVW Program Download"
snd_xvw_start	"XVW Start"
snd_xvw_exit	"XVW Exit"
snd_xvw_run	"XVW Run"
snd_xvw_step	"XVW Step Into"
snd_xvw_stepover	"XVW Step Over"
snd_xvw_stop	"XVW Stop"
snd_xvw_syms_load	"XVW Load Symbols"
snd_xvw_syserror	"XVW syserror"
snd_xvw_uerror	"XVW uerror"

2. You must also add the same list of keys (without values) to

```
My Computer\HKEY_CURRENT_USER\AppData\Local\Microsoft\Windows\CurrentVersion\EventLabels\
```

3. Now go and start the Sound settings in your Control Panel. Here you can assign a sound to each event. You can also assign None to an event, which prevents CrossView Pro from playing a sound if that specific event occurs.

4. For the sound effects to become operational, you also have to edit the `xvw.ini` file. You can do this using any editor, e.g. the Windows **notepad** command. Add the following line at an arbitrary line to your `xvw.ini` file:

```
sound_effects: TRUE
```

It is also possible to disable the sound effects by changing this line into:

```
sound_effects: FALSE
```

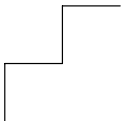
Now all sound effects are disabled.

ADDENDUM

SIMULATOR



TASKING



ADDENDUM

1 INTRODUCTION

This addendum contains information specific to the simulator version of CrossView Pro for the M16C.

2 EXECUTABLE NAME

The following CrossView Pro executable is delivered with the package:

xfwm16.exe CrossView Pro Debugger Simulator.

The simulator is delivered as a separate DLL within the package:

simm16.dll CrossView Pro Debugger Simulator.

3 SUPPORTED FEATURES

Except for the restrictions mentioned in the next section, the simulator version of the debugger cleanly supports all the standard features of CrossView Pro, including single stepping, code breakpoints, data breakpoints, trace support, C expression evaluation, code and data coverage and record/playback capability. With respect to setting breakpoints the simulator version of the debugger is capable of supporting all breakpoint types, including separate data-read and data-write breakpoints. Each of these breakpoints can be placed on any of the memory addresses.

All instructions listed in the *M16C/60 Series Software Manual* are supported. The granularity for timing measurements is one instruction state.

Because this is a simulator version, you do not have to setup a serial communication at startup, as with an emulator.

3.1 MAPPING MEMORY

The simulator version of the debugger uses the locator description file to determine how much memory must be allocated from the system and how logical addresses are mapped to physical addresses. The default file used is `etc\m16c.dsc` for all memory models. The CPU description file is `etc\m16c.cpu` and the memory description file is `etc\m16c.mem`.

When you use EDE, the memory settings are saved and automatically transferred to the debugger. All memory mappings of your applications are automatically done by the debugger.

You can specify a different locator description file in the CrossView Startup dialog. Make sure, however, that the memory configuration as defined by this description file does not differ from the one used for locating.

3.2 STATE COUNTER

In addition to the standard features of CrossView Pro, the simulator version executes the M16C instruction set and can perform state counting. For this purpose the simulator has a state counter, which can be monitored. The first one is the regular 'state counter', this counter can be monitored in the register window where it is shown as 'CCNT' and it can also be accessed through the register `$ccnt`.

3.3 COVERAGE

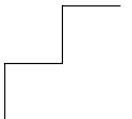
The simulator traces all memory access performed during program execution. This feature can be switched on or off. If you are not interested in coverage information you should turn off this feature because gathering coverage data will slow down the execution speed of the simulator. The simulator differentiates between data read, data write and instruction fetch. CrossView Pro shows the gathered coverage data through various windows and dialogs.

4 RESTRICTIONS

Facilities for background mode are absent in the simulator version of CrossView Pro. As a consequence, the CrossView Pro commands **CB**, **st**, **u**, and **wt** for background mode, are not available. Because the debugger and the simulator have been integrated into one executable program, the **>&** command to record target communication and the **o** command for transparency mode are not available. Also, the simulator version of the debugger does not support command line function calling.

ADDENDUM

ROM MONITOR



ADDENDUM

1 WHAT IS A ROM MONITOR?

CrossView Pro ROM is a ROM monitor-based source level debugger for debugging real-time embedded C and assembly programs. CrossView Pro ROM integrates two separate debugging components. The figure below shows these two components and how they communicate with each other.

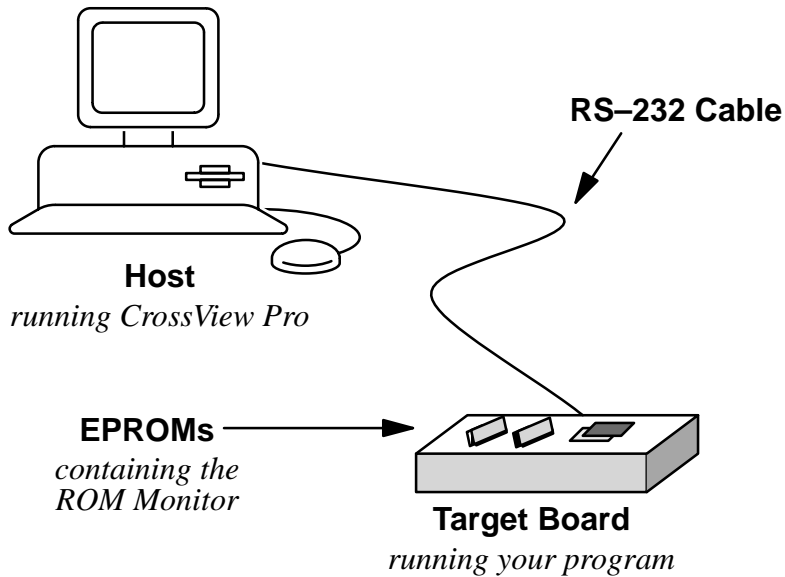


Figure Rom-1: CrossView Pro ROM hardware configuration

The first component is CrossView Pro, the source level debugger that runs on the host development system. CrossView Pro translates the low level target information obtained from the embedded ROM monitor to the C and assembly language source level. CrossView Pro has both a powerful command language to control the target's execution and a multi-window user interface to display target and debugging information.

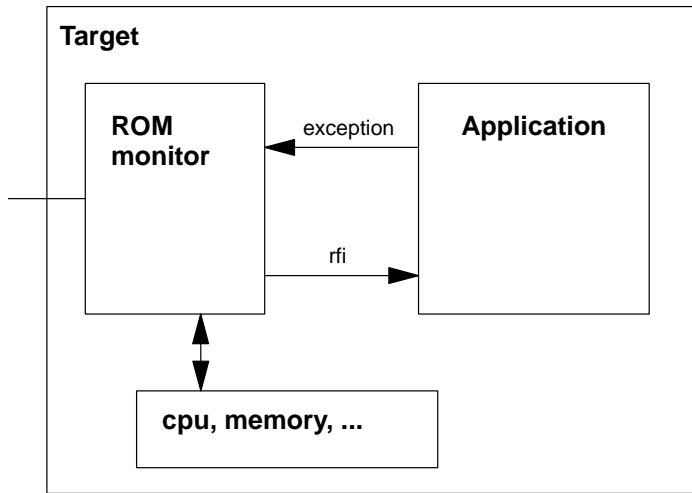


Figure Rom-2: Target functional blocks

The second component of CrossView Pro ROM is the ROM monitor, also referred to as the target monitor. The ROM monitor is a low level debugging program that normally resides in ROM on the target board and runs on the target microprocessor. The ROM monitor includes a serial communications interface that lets it accept and interpret commands from CrossView Pro.

The ROM monitor has the following general capabilities:

- It gives you complete access to the target machine via the serial port. The access includes reading and writing memory and registers (both general purpose and special purpose).
- It can take over from any application going wrong. For example, executing an illegal or non-existent instruction or accessing non-existent memory.



M16C cannot detect non-existent memory, therefore it will not be a trap as in other monitors.

- It can interrupt a running application.

In the rest of this Addendum we use the following terms:

Host: the computer running CrossView Pro.

Application: a program written by you and to be tested and debugged by CrossView Pro. Also called 'user program' or 'user code'.

Target: the embedded controller where an application is running on.

2 SETTING UP THE TARGET ENVIRONMENT

The Mitsubishi ROM monitor must be present and running on the target board before CrossView Pro is able to communicate properly with the execution environment.

To download/debug your application you must connect the board to a serial communications port of the computer.

From EDE:

1. Select `CrossView Pro Options...` from the EDE menu.
2. In the `Settings` tab select `Mitsubishi ROM monitor`.
3. In the `Communications` tab select the correct serial port and serial baud rate.

3 RESTRICTIONS

The Mitsubishi ROM monitor will download the application in Flash memory. Breakpoint support is implemented through the Address Match interrupts. Since only 2 ranges are supported the number of breakpoints is also limited to 2. Besides these limits Crossview Pro cleanly supports the standard features including single stepping, code breakpoints, C expression evaluation and record/playback capability.

Facilities for hardware breakpoints, trace and background mode are absent on the ROM monitor. As a consequence, the CrossView Pro commands **bd** and **bd** for data breakpoints, the command **ct** for C-level tracing, and the commands **CB**, **st**, **u** and **wt** for background mode, are not available.

4 RESOURCES USED BY THE ROM MONITOR

Every ROM monitor uses some resources, first it will use some code space in which it is running, then it will use some data space for keeping state information. Finally, it will require some kind of communication channel with the debugger. Currently only serial connections are supported by the Crossview Pro debugger. This means that one of the serial ports on the target is reserved by the ROM monitor and cannot be used by the application anymore.

Since the ROM monitor communication is interrupt based the corresponding interrupt vectors are also reserved. This is very important when creating an application to be run using the ROM monitor. The M16C supports a variable vector table that will mostly be initialized in the startup code of your application. If this vector table does not contain the serial interrupt vectors used by the ROM monitor, then the connection will certainly hang after initialization of the interrupt vector table register! To add the serial interrupt vectors to the applications vector table you can use the following assembly file:

```
DEFSECT ".vecttab", FDATA, ROMDATA, MAX
SECT    ".vecttab", RESET

OFFSET <serial-vector-number> * 4
DL      <serial-interrupt-transmit-vector>
        ; used by the ROM monitor
DL      <serial-interrupt-receive-vector>
        ; used by the ROM monitor

END
```

for example, in case of the Mitsubishi MDECE0222 evaluation board the ROM monitor uses serial port 0, with corresponding interrupt vectors 17 (transmit) and 18 (receive). The interrupt handler for both interrupts is located on address 0xF8A00. This results in the following assembly definition:

```
DEFSECT ".vecttab", FDATA, ROMDATA, MAX
SECT    ".vecttab", RESET

OFFSET 17 * 4
DL      0xF8A00    ; used by the MDECE0222 ROM monitor
DL      0xF8A00    ; used by the MDECE0222 ROM monitor

END
```

For exact numbers on the code area, the data area and the interrupts being used by the ROM monitor please check the documentation on your specific target board.

5 SUPPORTED TARGETS

In principle all targets using the Mitsubishi ROM monitor interface are supported. Currently the following target boards have been tested:

- Mitsubishi MDECE022 evaluation board
- Glyn EVBM16C/62 evaluation board
- Glyn EVBM16C/6N evaluation board

EDE contains several example projects for use with these target boards.



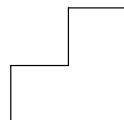
ROM MONITOR

INDEX

INDEX



TASKING



INDEX

Symbols

. (period) operand, 3-18
 ! command, 12-20
 ? command, 5-15, 12-22
 & operator, 3-18
 @format code, 3-13
 --timeout, 9-10
 / command, 5-15, 12-21
 /format code, 3-13
 ^ command, 12-31
 < command, 12-23
 << command, 12-24
 > command, 12-25
 >& command, 12-29
 >> command, 12-27
 _simi, 10-14
 _simo, 10-14

A

A command, 12-32
 a command, 12-33
 absolute file, 14-3
 accelerator bar, 4-24
 accelerator button, 4-11, 4-24, 4-38
 accessing code and data, 6-1
 adding files to a project, 1-32
 address bias, set, 12-99
 addresses
 in expressions, 3-18
 specifying format of, 6-16
 application
 debugging, 1-22
 executing, 1-20
 argument of a function, 3-9
 arm16, 1-11
 arrays
 display address of, 6-6
 display character, 3-15, 6-6
 displaying two-dimensional, 6-15

viewing contents of, 3-16, 6-15
 asm16, 1-11
 assembler, 1-11
 assembly
 pseudo-assembly listings, 14-8
 window, 14-3
 assembly window
 hexadecimal display, 3-10
 intermixed assembly, 3-10
 pipeline, 3-10
 source merge limit, 3-11
 assertion mode, 14-3
 assertions, 1-4, 7-25, 14-3
 activating and suspending, 7-28
 active, 7-25
 debugging with, 7-30
 define or modify assertion, 12-33
 defining, 7-26
 deleting, 7-29
 editing, 7-28
 global assertion mode, 7-25
 quit assertion mode, 12-136
 statistics, 7-32
 toggle mode, 12-32
 autosrc, 6-18

B

B command, 12-35
 b command, 12-36
 background color, 2-11
 background mode, 10-21, 14-3
 assertions, 10-26
 leaving, 10-24
 local and global variables, 10-25
 manual refresh, 10-22
 refresh limitations, 10-25
 running a program, 10-23
 stack, 10-25
 starting, 10-23
 stopping a program, 10-24

updating windows, 10-21
waiting, 10-24
 batch mode, 9-10
 batch processing, 9-10
 bB command, 12-37
 bb command, 12-38
 bc command, 12-39
 bCYC command, 12-40
 bcyc command, 12-41
 bD command, 12-42
 bd command, 12-44
 bdis command, 12-46
 bena command, 12-47
 bI command, 12-48
 bi command, 12-49
 bias, 14-3
 binary constants, 3-5
 binary notation, 3-4
 bINST command, 12-50
 binst command, 12-51
 breakpoint toggle, 4-23, 7-3
 breakpoints, 7-1, 14-4
 and diagnostic output, 7-24
 and multi-line statements, 7-5
 and multiple statements, 7-4
 and statistical information, 7-24
 attaching macros to, 7-17
 code, 1-4, 7-3
 commands associated with, 7-15
 conditionals, 7-16
 count, 14-4
 count of, 7-3
 cycle count, 12-40, 12-41
 data, 1-4, 7-6
 data breakpoints over a range of
 addresses, 7-10
 delete, 12-68
 delete all, 12-67
 deleting, 7-13
 disable, 7-14, 12-46
 emulator mode, 7-6
 enable, 7-14, 12-47

for loops, 7-5
function, permanent, 12-38
instruction count, 12-50, 12-51
list, 12-35
listing, 7-7
low-level, 7-19, 14-7
patching code with, 7-22
permanent, 7-4
permanent low-level, 12-49
 task aware, 12-54
permanent up-level, 12-59
quiet reporting of, 7-18
reset count, 7-3, 7-12
set at beginning of function, 12-37
set count, 12-39
setting, 1-20, 7-8
 from command window, 7-8
 from menu, 7-8
 from source window, 7-8
 from stack window, 7-8
setting the count of, 7-12
strings, 7-17
system startup code, 7-6
task aware
 code, 12-52
 permanent low-level, 12-54
 temporary low-level, 12-53
temporary, 7-4, 7-11
temporary low-level, 12-48
 task aware, 12-53
temporary up-level, 12-57
time, 12-55, 12-56
up-level, 7-20
while loops, 7-5
 bt command, 12-52
 btI command, 12-53
 bti command, 12-54
 bTIM command, 12-55, 12-56
 bU command, 12-57
 bu command, 12-59
 buttons, 4-37

C

C

- character constants*, 3-6
- compiler*, 1-11
- C command, 5-12, 12-60
- C trace, 1-4, 12-64
- cache, debugging with, 14-4
- case sensitivity, 3-21, 12-137
- casting values, 3-16, 6-15
- CB command, 12-61
- ccm16, 1-11
- cd command, 12-62
- ce command, 12-63
- character codes, 6-13
- character codes table, 3-6
- character constants, 3-6
- check box, 4-37
- cm16, 1-11
- code breakpoints, 1-4
 - See also breakpoints*
 - set breakpoint*, 12-36
 - task aware*, 12-52
- code coverage, 1-5
- color, windows, 2-11
- color settings, 2-11
- command history, displaying recent
 - commands, 9-18
- command language, 3-1
- command line, batch processing, 9-10
- command line options, 4-5
- Command Window, 4-21
 - displaying data in*, 6-8
 - opening*, 1-24
- commands
 - multiple*, 3-17
 - syntax*, 4-3
- comments, 3-17
- communication setup, 1-16
- compiler, 1-11
- conditional command execution, 12-84
- conditional keywords, 3-19
- configure CrossView Pro, 1-16
- constants, 3-4
 - binary*, 3-5
 - character*, 3-6
 - character constants in C*, 3-6
 - floating point*, 3-5
 - hexadecimal*, 3-4
 - long integer*, 3-5
 - octal*, 3-5
 - strings*, 3-6
- continue execution, 5-9
- control operations, 4-37
- control program, 1-11
- coverage, 1-5, 10-5, Sim-4
 - disable*, 10-5, 12-62
 - enable*, 10-5, 12-63
 - marker*, 4-23, 7-3
 - memory window*, 4-33
 - next covered block*, 12-100
 - next not covered block*, 12-101
 - previous covered block*, 12-106
 - previous not covered block*, 12-111
 - source window*, 4-24
- cpu, 3-10, 12-4
- cpu selection, 12-75, 12-76
- creating a makefile, 1-33
- CrossView
 - and command line options*, 4-5
 - command files*, 4-6
 - command language*, 3-1
 - command line batch processing*, 9-10
 - command reference*, 12-1
 - commands summary*, 12-4, 12-15
 - customizing*, 4-17
 - desktop*, 4-11
 - executable name*, Sim-3
 - features of the execution environment*, Sim-3
 - invoking*, 4-4
 - restrictions of execution environment*, Sim-4

sound support, B-1
special features, 10-1
starting, 4-4
startup options, 12-4
state of, 12-83
using, 4-1

CrossView Pro

before starting, 1-13
debugger, 1-11
debugging environment, 1-7
documentation, 1-7
exiting, 1-25
features, 1-3
how it works, 1-9
invoking, 1-14
output, 1-24
source level debugging, 1-7
startup settings, 1-15
using windows, 1-3
windows, 1-3

ct command, 12-64

ct i command, 12-65

ct r command, 12-66

cursor, 5-3

D

D command, 12-67

d command, 12-68

data

displaying, 6-1

enumerated, 6-5

list data monitors, 12-89

data breakpoints, 1-4

set at an address, 12-44

set over range of addresses, 12-42

data coverage, 4-33

data monitoring, 1-5, 14-5

removing expressions, 6-12

Data Window, 1-5, 4-29, 6-11

toolbar, 4-30

debugger, starting, 1-32

debugging

and optimized code, 3-7

assembly language, 11-3

code without symbols, 5-14

environment, 1-7

multiple programs, 11-3

notes about, 11-1

source-level, 1-7

viewing source while, 1-20

debugging an application, 1-22

description file, 3-10, 4-7, 12-4, Sim-3

desktop, 4-11

development flow, 1-12

diagnostic output, and breakpoints,
7-24

diagnostics, 14-5

dialog boxes, 4-16

dis command, 12-69

disassemble memory, 12-69

disassembly, 6-17

display, customizing, 4-17

display formats, set default, 12-80

dn command, 12-70

documentation, 1-7

dot operand, 6-11

download a file, 12-70

download image, 12-88

downloading, files to the execution
environment, 1-17

dsc, 3-10

dump, 3-16, 6-15

dump command, 12-71

Dy command, 12-67

E

e command, 5-14, 12-73

eC command, 12-75

ec command, 12-76

echo command, 12-77

echo string to terminal, 12-19
 EDE, 1-26
 build an application, 1-28
 load files, 1-28
 open a project, 1-27
 select a toolchain, 1-27
 start a new project, 1-32
 starting, 1-26
 edit source, 4-26
 ei command, 12-78
 embedded development environment.
 See EDE
 embedded system, 14-5
 emulator communication setup, 1-16
 emulator mode, 1-8
 environment variable
 LD_LIBRARY_PATH, 2-9
 LM_LICENSE_FILE, 2-19, A-6
 PATH, 2-4, 2-5, 2-6, 2-8
 UIDPATH, 2-9
 error messages, alphabetical listing of,
 13-1
 errors, FLEXlm license, A-33
 Esc key, 4-21
 et command, 12-79
 evaluate expression, 12-16
 example
 starting EDE, 1-26
 using EDE, 1-26
 using the control program, 1-33
 using the makefile, 1-35
 executable, building for CrossView,
 1-26
 executing an application, 1-20
 execution control commands,
 summary of, 12-9
 execution environment, Sim-1, Rom-1
 connecting to CrossView, 4-6
 downloading files to, 1-17
 setting up, Rom-5
 execution position, 5-3
 changing the, 5-5
 definition of, 14-5

sync with viewing position, 5-7
 exit, 4-18
 exponential notation, 3-5
 expression evaluator, 1-4
 expressions, 3-3
 C character codes, 3-6
 character constants, 3-6
 evaluating, 6-10
 evaluation precision, 3-4
 floating point constants, 3-5
 format of, 3-13
 monitoring, 6-11
 removing monitored, 6-12
 show, 4-29
 special expressions, 3-18
 specifying variables in, 3-8
 strings, 3-6
 watch, 4-29

F

f command, 12-80
 FAQ, FLEXlm, A-37
 filenames, 2-3
 Flexible License Manager, A-1
 FLEXlm, A-1
 daemon log file, A-25
 daemon options file, A-7
 FAQ, A-37
 frequently asked questions, A-37
 license administration tools, A-8
 for Windows, A-22
 license errors, A-33
 floating license, 2-13
 floating point constants, 3-5
 format codes, 3-14
 formats, for variables, 6-13
 formatting expressions, 3-13
 frame pointer, 3-10
 functions, 3-20
 listing all, 6-8

listing local variables and parameters of, 6-22

G

g command, 5-5, 12-81
GDI, 9-12, 9-14
 logging, 9-13, 9-14, 9-16
getting started, 1-13
gi command, 5-6, 12-82
global variables, 3-8
glossary, 14-1

H

halt execution, 5-9
help
 on-line, 1-6, 4-39
 summary of help commands, 12-14
hexadecimal disassembly, 3-10
hexadecimal notation, 3-4
history mechanism, 14-6
hostid, determining, 2-21
hostname, determining, 2-21

I

I command, 12-83
if command, 12-84
image part, 14-6
in-situ editing, 6-7, 6-27
installation
 licensing, 2-13
 Linux, 2-4
 RPM, 2-5
 tar.gz, 2-6
integers, 3-4
 binary, 3-5
 hexadecimal, 3-4

integral promotion, 3-5
long, 3-5
negative, 3-4
octal, 3-5

integral promotion, 3-5
intermixed source and disassembly,
 6-18
interrupt key, 14-6

J

jump to cursor, 5-5

K

kernel support, 1-6, 10-4
keywords, conditional, 3-19-3-22

L

L command, 12-85
l command, 12-86
label, in disassembly, 6-17
language, 3-1
lcm16, 1-11
LD_LIBRARY_PATH, 2-9
librarian, 1-11, 14-6
license
 floating, 2-13
 node-locked, 2-13
 obtaining, 2-13
license file
 default location, A-6
 location, 2-19
licensing, 2-13
line command, 12-18
line mode, 14-6
line numbers, 3-11
linker, 1-11, 14-6

listing, 12-86
 lkm16, 1-11
 LM_LICENSE_FILE, 2-19, A-6
 lmcksum, A-10
 lmdiag, A-11
 lmdown, A-12
 lmgrd, A-13
 lmhostid, A-15
 lmremove, A-16
 lmread, A-17
 lmstat, A-18
 lmswitchr, A-20
 lmver, A-21
 load command, 12-88
 load symbol file, 12-88, 12-98
 local variables, 3-7
 and the stack, 3-7
 auto-watch, 4-31
 locator, 1-11, 14-7
 logging, 9-12
 command window output, 12-27
 commands and screen output, 9-14
 debugger-emulator I/O, 12-29
 example, 9-14
 resume, 9-14
 setting up, 9-13
 start, 9-13
 startup options, 9-17
 stop, 9-16
 summary of commands, 12-13
 suspend, 9-14
 long integer constants, 3-5

M

M command, 12-89
 m command, 12-90
 M16C program development, 1-11
 macros, 1-6, 8-1, 14-7
 calling other macros, 8-4
 define, 12-119

defining, 8-3
 delete definition, 12-133
 deleting, 8-8
 echo command, 12-77
 expanding, 8-5
 listing, 8-5
 parameters of, 8-9
 reading from a file, 8-7
 redefining, 8-5, 8-10
 save, 12-118
 saving to a file, 8-6
 summary of commands, 12-13
 using the toolbox, 8-11
 main() function, 14-7
 make utility, 1-11
 makefile
 automatic creation of, 1-33
 updating, 1-33
 MAU (minimum addressable unit),
 14-7
 mcp command, 12-92
 memory
 copy, 12-92
 disassembly, 12-69
 displaying, 6-14
 dump, 12-71
 fill, 12-94
 mapping, Sim-3
 search, 12-96
 single fill, 12-93
 memory access, tracing, 1-5
 memory dump, 3-16, 6-15
 memory map, 4-6, 14-7
 Memory Window, 4-31
 setup, 4-32
 toolbar, 4-33
 menu, 4-13
 local popup, 4-14
 menu bar, 4-11
 mF command, 12-93
 mf command, 12-94
 minimum addressable unit, 14-7

mkm16, 1-11
 monitor. *See* ROM Monitor
 monitor data, 12-89
 monitors, 12-90
 more, 3-10
 ms command, 12-96
 MS-Windows
 installing CrossView, 2-3
 requirements, 2-4

N

N command, 12-98
 n command, 12-99
 nC command, 12-100
 node-locked license, 2-13
 nU command, 12-101

O

o command, 12-102
 object reader, 1-11
 octal constants, 3-5
 octal notation, 3-4
 operators, 3-17
 order of precedence, 3-17
 using addresses, 3-18
 opt command, 12-103
 optimization, and debugging, 3-7
 optimizer, 14-8
 options, display or set, 12-103
 output paging mechanism, 3-10
 overview, 1-1

P

P command, 12-104
 p command, 12-105

patches, 14-8
 and breakpoints, 7-22
 PATH, 2-4, 2-5, 2-6, 2-8
 pC command, 12-106
 pd command, 12-107
 pe command, 12-108
 performing timing analysis, 1-5
 pipeline, 3-10
 playback, 9-7
 calling other playback files, 9-9
 quitting, 9-9
 setting the type of, 9-8
 startup options, 9-17
 summary of commands, 12-13
 playback mode, 1-6
 continuous, 12-23
 single step, 12-24
 pointer, 3-16
 display character, 3-15, 6-6
 precision, evaluating expressions, 3-4
 print source lines, 12-104, 12-105
 prm16, 1-11
 problems
 common, 1-25
 communicating with CrossView, 4-9
 profiling, 1-5, 10-7
 code range, 1-5, 10-8
 cumulative, 10-7
 disable, 10-8, 12-107
 enable, 10-8, 12-108
 function, 10-7
 functions, 1-5
 information, 12-109
 program builder, 1-11
 program counter, 3-10, 12-60
 g command (change), 12-81
 gi command (change), 12-82
 inside function, 3-9
 program execution
 controlling, 5-1
 notes about, 5-14
 program reset, 12-110

proinfo command, 12-109
 project files, adding files, 1-32
 prst command, 12-110
 pseudo-assembly, 6-18
 pU command, 12-111
 push button, 4-38

Q

Q command, 12-112
 q command, 12-113
 quiet breakpoint recording, 12-112
 quit debugger, 12-113

R

R command, 5-8, 12-114
 radio button, 4-37
 record, commands only, 12-25
 record and playback, 9-1
 definition of, 14-9
 record mode, 1-6
 recording
 checking status, 9-5
 close file for, 9-6
 entering comments, 9-4
 example, 9-6
 resume, 9-5
 start, 9-3
 startup options, 9-17
 stop, 9-6
 summary of commands, 12-13
 suspend, 9-4
 refresh windows, 12-132
 Register Window, 4-28, 6-26
 setup, 6-26
 registers, 3-11
 displaying the contents of, 6-8
 special variable, 3-10
 reset program, 5-8, 12-110
 reset target system, 12-114, 12-115

resource file, 2-9
 return address, 6-19
 ROM Monitor, Rom-1
 capabilities of, Rom-4
 restrictions of, Rom-5
 rst command, 12-115
 RTOS aware debugging, 10-4

S

S command, 5-11, 12-116
 s command, 12-117
 save command, 12-118
 save on exit, 4-18
 scoping rules and variables, 3-9
 scroll bar, 4-11
 search
 backward for string, 12-22
 forward for string, 12-21
 summary of commands, 12-15
 searching, 5-14-5-16
 for a function, 5-14
 for a source line, 5-16
 for a string, 5-15
 serial ports, 4-6
 set command, 12-119
 Si command, 5-12, 12-121
 si command, 5-12, 12-122
 Simulated I/O. *See simulated input/output*
 simulated I/O, 10-14
 Simulated I/O Windows, 4-36, 12-123
 simulated input/output, 1-4
 buffers, 10-18
 changing stream properties, 10-17
 changing the prompt, 10-19
 defined, 14-9
 deleting a stream, 10-18
 directing I/O to a file, 10-20
 installing calls in code, 10-15
 setting up streams, 10-15
 summary of commands, 12-14

- viewing active streams, 10-17*
- windows, 4-36*
- simulator, Sim-1
- single stepping, 1-5, 5-9-5-10
 - at machine level, 5-12-5-16*
 - defined, 14-9*
 - into, 5-10*
 - into function calls, 12-117*
 - into functions, 5-10*
 - machine level into functions, 12-122*
 - machine level over functions, 12-121*
 - over, 5-11*
 - over function calls, 12-116*
 - over functions, 5-11*
- sio command, 12-123
- sizeof() function, 6-7
- skidding, 14-10
- software installation
 - MS-Windows, 2-3*
 - UNIX, 2-7*
 - Windows 95/NT, 2-3*
- sound support, B-1
- source directory, change, 12-134
- source level debugging, 1-7
- source line, jump to, 5-16
- source merge limit, 3-11
- source positioning, 5-3
- Source Window, 4-23
 - change execution position, 5-5*
 - change viewing position, 5-4*
 - controlling program execution, 5-8-5-16*
 - edit source, 4-26*
 - searching in, 5-14-5-16*
 - single stepping, 5-9*
 - sync execution and viewing positions, 5-7*
 - toolbar, 4-24*
- source window, line numbers, 3-11
- special variables, 3-9, 14-10
 - reserved, 14-9*
 - user-defined, 3-12*
- st command, 12-125
- stack, 6-19
 - beginning of, 6-20*
 - end of, 6-20*
 - local variables, 3-7*
 - organization of, 6-19*
- stack pointer, 3-10
- stack trace, 12-126, 12-127
- Stack Window, 4-27, 6-20
 - toolbar, 4-27*
- startup options, 4-5, 12-4
 - defined, 14-10*
 - list of, 4-7*
- startup settings, 1-15
- state counter, Sim-4
- static variables, 3-7
- status bar, 4-11
- stop target execution, 12-125
- storage classes, 3-7
- string command, 3-18
- strings, 3-6
- structures
 - assignment, 6-8*
 - viewing, 6-5*
- style codes, 3-14
- symbol information, 14-11
- symbolic disassembly, 6-17
- symbols, in disassembly, 3-10
- synchronize execution and viewing
 - positions, 5-7, 12-85*
- system startup code, 14-11

T

- T command, 12-126
- t command, 12-127
- Tab key, 4-21
- target communication, 14-11
- target configuration file, 1-16
- target program counter, 12-61
- target system, 1-7
- task selection, 12-79

td command, 12-128
 te command, 12-129
 toolbar, 4-11, 4-38
 data window, 4-30
 memory window, 4-33
 source window, 4-24
 stack window, 4-27
 toolbox, 8-11
 trace
 C, 12-64
 disable, 12-128
 disassembled, 12-65
 enable, 12-129
 instruction level, 6-25
 raw, 6-25, 12-66
 source level, 6-24
 trace buffer, 14-11
 Trace Window, 4-26, 6-24
 instruction level, 12-65
 raw, 12-66
 source level, 12-64
 traceback mode, 1-4
 transparency mode, 1-8, 10-3, 12-102
 and CrossView startup, 4-6
 defined, 14-11
 entering, 10-3
 one-shot commands, 10-3
 startup options, 10-3
 trigraph sequence, 3-7
 troubleshooting, 1-25, 4-9

U

u command, 12-130
 ubgw command, 12-132
 UIDPATH, 2-9
 UNIX, installing CrossView on, 2-7
 unset command, 12-133
 update windows, 12-130, 12-132
 updating makefile, 1-33
 use command, 12-134
 user defined functions, 1-6

using EDE, 1-26
 USR_*, 10-9

V

variables, 3-7
 and case sensitivity, 3-21
 and scoping rules, 3-9
 casting, 3-7
 changing, 6-7
 determining the size of, 6-7
 formats of, 6-13
 global, 6-8
 global variables, 3-8
 local, 14-7
 local variables, 3-7
 scope, 14-9
 special, 14-10
 special variables, *Pages*, 3-9
 specifying in expressions, 3-8
 static variables, 3-7
 user-defined special variables, 3-12
 viewing position, 3-10, 5-3
 changing the, 5-4-5-7
 defined, 14-12
 establish, 12-73
 establish at address, 12-78
 sync with execution position, 5-7
 virtual I/O channels, 10-9
 keyboard mappings, 10-10
 ROM monitor, 10-9
 Virtual I/O Windows, 4-35
 virtual input/output, windows, 4-35

W

wait for target completion, 12-135
 waiting, 10-24
 window mode, 14-12

windows, 4-20
 active, 4-15, 14-3
 *automatic switching between source
 and assembly*, 3-10
 closing, 4-15
 command window, 4-21
 customizing, 4-17
 data window, 4-29
 help window, 4-36
 memory window, 4-31
 opening, 4-15
 pop-up, 4-36
 register window, 4-28
 selecting, 4-15
 simulated I/O windows, 4-36
 source positioning, 5-3
 source window, 4-23
 stack window, 4-27
 toolbox, 4-36
 trace window, 4-26
 virtual I/O windows, 4-35

Windows 95/NT, installing CrossView,
 2-3
 wt command, 12-135

X

x command, 12-136
 X Resources, 2-10
 X Widgets, CrossView Motif, 2-10
 X Windows
 Motif environment, 2-9
 resources, 2-10
 xfwm16, 1-11
 xvwedit, 4-26

Z

Z command, 12-137