IAR Embedded Workbench®

C-SPY® Debugging Guide

for the Renesas

R32C/100 Microcomputer Family

UCSR32C-1

IAR SYSTEMS

EDITION NOTICE

First edition: October 2012

Part number: UCSR32C-1

This guide applies to version 1.x of IAR Embedded Workbench® for the Renesas R32C/100 microcomputer family.

The *C-SPY® Debugging Guide for R32C* replaces all debugging information in the *IAR Embedded Workbench IDE User Guide for R32C/100* and the hardware debugger guides for R32C/100 and E8a.

Internal reference: M12, Too6.4, IMAE.

Brief contents

Tables 11

Figures 13

Preface 17

The IAR C-SPY Debugger 23

Getting started using C-SPY 31

Executing your application 59

Variables and expressions 79

Breakpoints 101

Memory and registers 129

Trace 159

Profiling 183

Code coverage 191

Interrupts 195

C-SPY macros 215

The C-SPY Command Line Utility—cspybat 257

Debugger options 267

Additional information on C-SPY drivers 275

Index 289

Contents

Tables 11

Figures 13

Preface 17

Who should read this guide 17

Required knowledge 17

What this guide contains 17

Other documentation 18

User and reference guides 18

The online help system 19

Web sites 19

Document conventions 20

Typographic conventions 20

Naming conventions 21

The IAR C-SPY Debugger 23

Introduction to C-SPY 23

An integrated environment 23

General C-SPY debugger features 24

RTOS awareness 25

Debugger concepts 26

C-SPY and target systems 26

The debugger 26

The target system 26

The application 27

C-SPY debugger systems 27

The ROM-monitor program 27

Third-party debuggers 27

C-SPY plugin modules 28

C-SPY drivers overview 28

Differences between the C-SPY drivers 28

The IAR C-SPY Simulator 29
Simulator features 29
The C-SPY Hardware debugger drivers 29
Communication overview 30
Getting started using C-SPY 31
Setting up C-SPY 31
Setting up for debugging 31
Executing from reset 32
Using a setup macro file 32
Selecting a device description file 33
Loading plugin modules 33
Starting C-SPY 33
Starting a debug session 34
Loading executable files built outside of the IDE 34
Starting a debug session with source files missing 34
Loading multiple images 35
Adapting for target hardware 36
Modifying a device description file 36
Initializing target hardware before C-SPY starts 37
Downloading firmware 38
Running example projects 38
Running an example project 38
Connecting to the target board, E8a 39
Maintaining the connection to the hardware 42
Connecting to the target board, E30/E30A 43
Reference information on starting C-SPY 45
Executing your application 59
Introduction to application execution 59
Briefly about application execution 59
Source and disassembly mode debugging 59
Single stepping 60
Stepping speed 62
Running the application 63

Highlighting ... 64

Call stack information ... 64

Terminal input and output ... 65

Debug logging ... 65

Reference information on application execution ... 66

Variables and expressions ... 79

Introduction to working with variables and expressions ... 79

Briefly about working with variables and expressions ... 79

C-SPY expressions ... 80

Limitations on variable information ... 82

Working with variables and expressions ... 83

Using the windows related to variables and expressions ... 83

Viewing assembler variables ... 84

Getting started using data logging ... 84

Reference information on working with variables and expressions ... 85

Breakpoints ... 101

Introduction to setting and using breakpoints ... 101

Reasons for using breakpoints ... 101

Briefly about setting breakpoints ... 102

Breakpoint types ... 102

Breakpoint icons ... 104

Breakpoints in the C-SPY simulator ... 104

Breakpoints in the C-SPY hardware drivers ... 104

Breakpoint consumers ... 105

Setting breakpoints ... 106

Various ways to set a breakpoint ... 106

Toggling a simple code breakpoint ... 107

Setting breakpoints using the dialog box ... 107

Setting a data breakpoint in the Memory window ... 108

Setting breakpoints using system macros ... 109

Useful breakpoint hints ... 110

Reference information on breakpoints ... 111

Memory and registers 129

Introduction to monitoring memory and registers 129

Briefly about monitoring memory and registers 129

C-SPY memory zones 130

Stack display 131

Memory access checking 132

Monitoring memory and registers 133

Defining application-specific register groups 133

Reference information on memory and registers 134

Trace 159

Introduction to using trace 159

Reasons for using trace 159

Briefly about trace 159

Requirements for using trace 160

Collecting and using trace data 160

Getting started with trace 160

Trace data collection using breakpoints 161

Searching in trace data 161

Browsing through trace data 162

Reference information on trace 162

Profiling 183

Introduction to the profiler 183

Reasons for using the profiler 183

Briefly about the profiler 184

Requirements for using the profiler 184

Using the profiler 184

Getting started using the profiler on function level 185

Getting started using the profiler on instruction level 185

Reference information on the profiler 186

Code coverage 191

Introduction to code coverage 191

Reasons for using code coverage 191

Briefly about code coverage 191
Requirements for using code coverage 191
Reference information on code coverage 191
Interrupts 195
Introduction to interrupts 195
Briefly about interrupt logging 195
Briefly about the interrupt simulation system 196
Interrupt characteristics 197
Interrupt simulation states 197
C-SPY system macros for interrupt simulation 199
Target-adapting the interrupt simulation system 199
Using the interrupt system 200
Simulating a simple interrupt 200
Simulating an interrupt in a multi-task system 201
Getting started using interrupt logging 202
Reference information on interrupts 202
C-SPY macros 215
Introduction to C-SPY macros 215
Reasons for using C-SPY macros 215
Briefly about using C-SPY macros 216
Briefly about setup macro functions and files 216
Briefly about the macro language 217
Using C-SPY macros 217
Registering C-SPY macros—an overview 218
Executing C-SPY macros—an overview 218
Using the Macro Configuration dialog box 219
Registering and executing using setup macros and setup files 220
Executing macros using Quick Watch 221
Executing a macro by connecting it to a breakpoint 222
Reference information on the macro language 223
Macro functions 223
Macro variables 224
Macro strings 224

Macro statements 225
Formatted output 226
Reference information on reserved setup macro function names 228
Reference information on C-SPY system macros 229

The C-SPY Command Line Utility—cspybat 257

Using C-SPY in batch mode 257
Invocation syntax 257
Output 259
Using an automatically generated batch file 259
Summary of C-SPY command line options 259
General cspybat options 259
Options available for all C-SPY drivers 260
Options available for the simulator driver 260
Options available for the C-SPY hardware driver 260
Reference information on C-SPY command line options 260

Debugger options 267

Setting debugger options 267
Reference information on debugger options 268
Reference information on C-SPY hardware driver options 272

Additional information on C-SPY drivers 275

Reference information on the C-SPY simulator 275
Reference information on the C-SPY emulator drivers 277
Resolving problems 286
Write failure during load 286
No contact with the target hardware 287
Slow stepping speed 287

Index 289

Tables

1: Typographic conventions used in this guide 20
2: Naming conventions used in this guide 21
3: Driver differences 28
4: Emulator mode debugging functions 57
5: C-SPY assembler symbols expressions 81
6: Handling name conflicts between hardware registers and assembler labels 81
7: Available breakpoints in C-SPY hardware drivers 105
8: C-SPY macros for breakpoints 109
9: Supported graphs in the Timeline window 169
10: Project options for enabling the profiler 185
11: Project options for enabling code coverage 192
12: Timer interrupt settings 201
13: Examples of C-SPY macro variables 224
14: C-SPY setup macros 228
15: Summary of system macros 229
16: __cancelInterrupt return values 231
17: __disableInterrupts return values 232
18: __driverType return values 233
19: __enableInterrupts return values 233
20: __evaluate return values 234
21: __isBatchMode return values 234
22: __loadImage return values 235
23: __openFile return values 238
24: __readFile return values 240
25: __setCodeBreak return values 243
26: __setDataBreak return values 244
27: __setDataLogBreak return values 245
28: __setLogBreak return values 246
29: __setSimBreak return values 248
30: __setTraceStartBreak return values 249
31: __setTraceStopBreak return values 250

32: __sourcePosition return values 251
33: __unloadImage return values 254
34: cspybat parameters 257
35: Options specific to the C-SPY drivers you are using 267

Figures

1: C-SPY and target systems 26
2: C-SPY Hardware debugger communication overview 30
3: Get Alternative File dialog box 35
4: Example applications 38
5: The Emulator Setting dialog box for the E8a emulator 40
6: The Firmware Location page for the E8/E8a emulator 41
7: The Connecting dialog box 41
8: The Power Supply dialog box for the E8/E8a emulator 42
9: The ID Code verification dialog box for the E8/E8a emulator 42
10: Debug menu 47
11: Images window 50
12: Images window context menu 51
13: Get Alternative File dialog box 52
14: Emulator Hardware Setup dialog box 55
15: C-SPY highlighting source location 64
16: C-SPY Disassembly window 66
17: Disassembly window context menu 68
18: Call Stack window 71
19: Call Stack window context menu 71
20: Terminal I/O window 72
21: Ctrl codes menu 73
22: Input Mode dialog box 73
23: Terminal I/O Log File dialog box 74
24: Debug Log window (message window) 74
25: Debug Log window context menu 75
26: Log File dialog box 75
27: Report Assert dialog box 76
28: Autostep settings dialog box 77
29: Viewing assembler variables in the Watch window 84
30: Auto window 86
31: Locals window 86

32: Watch window ... 87
33: Watch window context menu ... 88
34: Live Watch window ... 89
35: Statics window ... 90
36: Statics window context menu ... 91
37: Quick Watch window ... 92
38: Symbols window ... 93
39: Symbols window context menu ... 94
40: Resolve Symbol Ambiguity dialog box ... 95
41: Data Log window ... 96
42: Data Log Summary window ... 98
43: Breakpoint icons ... 104
44: Modifying breakpoints via the context menu ... 108
45: Breakpoints window ... 112
46: Breakpoints window context menu ... 112
47: Breakpoint Usage window ... 114
48: Code breakpoints dialog box ... 115
49: Execution Address Breakpoint dialog box ... 116
50: Software Breakpoint dialog box ... 118
51: Log breakpoints dialog box ... 119
52: Data breakpoints dialog box ... 121
53: Data Break Event dialog box ... 123
54: Data Log breakpoints dialog box ... 124
55: Immediate breakpoints dialog box ... 125
56: Enter Location dialog box ... 126
57: Resolve Source Ambiguity dialog box ... 128
58: Zones in C-SPY ... 131
59: Register Filter options ... 133
60: Register Filter options ... 133
61: Memory window ... 135
62: Memory window context menu ... 137
63: Memory Save dialog box ... 139
64: Memory Restore dialog box ... 140
65: Fill dialog box ... 140

66: Symbolic Memory window 142
67: Symbolic Memory window context menu 143
68: Stack window 144
69: Stack window context menu 146
70: Register window 148
71: SFR Setup window 150
72: SFR Setup window context menu 152
73: Edit SFR dialog box 153
74: Memory Access Setup dialog box 154
75: Edit Memory Access dialog box 156
76: Trace Event dialog box 163
77: The Trace window 165
78: Function Trace window 168
79: Timeline window with Call Stack graph 170
80: Timeline window with Data Log graph 171
81: Timeline window with Interrupt Log graph 172
82: Timeline window context menu (for the Call Stack Graph context menu) 173
83: Viewing Range dialog box 175
84: Trace Start breakpoints dialog box 176
85: Trace Stop breakpoints dialog box 177
86: Trace Expressions window 178
87: Find in Trace dialog box 179
88: Find in Trace window 181
89: Instruction count in Disassembly window 186
90: Function Profiler window 186
91: Function Profiler window context menu 189
92: Code Coverage window 192
93: Code coverage window context menu 194
94: Simulated interrupt configuration 197
95: Simulation states - example 1 198
96: Simulation states - example 2 198
97: Interrupt Setup dialog box 203
98: Edit Interrupt dialog box 204
99: Forced Interrupt window 206

100: Forced Interrupt window context menu 206
101: Interrupt Status window 207
102: Interrupt Log window 209
103: Interrupt Log window context menu 211
104: Interrupt Log Summary window 212
105: Macro Configuration dialog box 219
106: Quick Watch window 221
107: Debugger setup options 268
108: Debugger extra options 269
109: Debugger images options 270
110: Debugger plugin options 271
111: Emulator communication options 272
112: Download page 273
113: Simulator menu 275
114: The Emulator menu 277
115: Events window 279
116: Time Measurement Event dialog box 282
117: Interval Time Measurement Event dialog box 283
118: Interval Time Measurement window 285

Preface

Welcome to the *C-SPY® Debugging Guide for R32C*. The purpose of this guide is to help you fully use the features in the IAR C-SPY® Debugger for debugging your application based on the R32C/100 microcomputer.

Who should read this guide

Read this guide if you plan to develop an application using IAR Embedded Workbench and want to get the most out of the features available in C-SPY.

REQUIRED KNOWLEDGE

To use the tools in IAR Embedded Workbench, you should have working knowledge of:

- The architecture and instruction set of the R32C/100 microcomputer (refer to the chip manufacturer's documentation)
- The C or C++ programming language
- Application development for embedded systems
- The operating system of your host computer.

For more information about the other development tools incorporated in the IDE, refer to their respective documentation, see *Other documentation*, page 18.

What this guide contains

This is a brief outline and summary of the chapters in this guide:

- *The IAR C-SPY Debugger* introduces you to the C-SPY debugger and to the concepts that are related to debugging in general and to C-SPY in particular. The chapter also introduces the various C-SPY drivers. The chapter briefly shows the difference in functionality that the various C-SPY drivers provide.
- *Getting started using C-SPY* helps you get started using C-SPY, which includes setting up, starting, and adapting C-SPY for target hardware.
- *Executing your application* describes the conceptual differences between source and disassembly mode debugging, the facilities for executing your application, and finally, how you can handle terminal input and output.
- *Variables and expressions* describes the syntax of the expressions and variables used in C-SPY, as well as the limitations on variable information. The chapter also demonstrates the various methods for monitoring variables and expressions.

- *Breakpoints* describes the breakpoint system and the various ways to set breakpoints.
- *Memory and registers* shows how you can examine memory and registers.
- *Collecting and using trace data* describes how you can inspect the program flow up to a specific state using trace data.
- *Using the profiler* describes how the profiler can help you find the functions in your application source code where the most time is spent during execution.
- *Code coverage* describes how the code coverage functionality can help you verify whether all parts of your code have been executed, thus identifying parts which have not been executed.
- *Interrupts* contains detailed information about the C-SPY interrupt simulation system and how to configure the simulated interrupts to make them reflect the interrupts of your target hardware.
- *Using C-SPY macros* describes the C-SPY macro system, its features, the purposes of these features, and how to use them.
- *The C-SPY Command Line Utility—cspybat* describes how to use C-SPY in batch mode.
- *Debugger options* describes the options you must set before you start the C-SPY debugger.
- *Additional information on C-SPY drivers* describes menus and features provided by the C-SPY drivers not described in any dedicated topics.

Other documentation

User documentation is available as hypertext PDFs and as a context-sensitive online help system in HTML format. You can access the documentation from the Information Center or from the **Help** menu in the IAR Embedded Workbench IDE. The online help system is also available via the F1 key.

USER AND REFERENCE GUIDES

The complete set of IAR Systems development tools is described in a series of guides. Information about:

- System requirements and information about how to install and register the IAR Systems products, is available in the booklet Quick Reference (available in the product box) and the *Installation and Licensing Guide*.
- Getting started using IAR Embedded Workbench and the tools it provides, is available in the guide *Getting Started with IAR Embedded Workbench®*.

- Using the IDE for project management and building, is available in the *IDE Project Management and Building Guide.*
- Using the IAR C-SPY® Debugger, is available in the *C-SPY® Debugging Guide for R32C.*
- Programming for the IAR C/C++ Compiler for R32C, is available in the *IAR C/C++ Compiler Reference Guide for R32C.*
- Using the IAR XLINK Linker, the IAR XAR Library Builder, and the IAR XLIB Librarian, is available in the IAR Linker and Library Tools Reference Guide.
- Programming for the IAR Assembler for R32C, is available in the *R32C IAR Assembler Reference Guide.*
- Using the IAR DLIB Library, is available in the *DLIB Library Reference information*, available in the online help system.
- Developing safety-critical applications using the MISRA C guidelines, is available in the *IAR Embedded Workbench® MISRA C:2004 Reference Guide* or the *IAR Embedded Workbench® MISRA C:1998 Reference Guide*.

Note: Additional documentation might be available depending on your product installation.

THE ONLINE HELP SYSTEM

The context-sensitive online help contains:

- Information about project management and building in the IDE
- Information about debugging using the IAR C-SPY® Debugger
- Information about using the editor
- Reference information about the menus, windows, and dialog boxes in the IDE
- Compiler reference information
- Keyword reference information for the DLIB library functions. To obtain reference information for a function, select the function name in the editor window and press F1.

WEB SITES

Recommended web sites:

- The Renesas web site, **www.renesas.com**, that contains information and news about the R32C/100 microcomputers.
- The IAR Systems web site, **www.iar.com**, that holds application notes and other product information.
- The web site of the C standardization working group, **www.open-std.org/jtc1/sc22/wg14**.

- The web site of the C++ Standards Committee, **www.open-std.org/jtc1/sc22/wg21**.
- Finally, the Embedded C++ Technical Committee web site, **www.caravan.net/ec2plus**, that contains information about the Embedded C++ standard.

Document conventions

When, in the IAR Systems documentation, we refer to the programming language C, the text also applies to C++, unless otherwise stated.

When referring to a directory in your product installation, for example `r32c\doc`, the full path to the location is assumed, for example `c:\Program Files\IAR Systems\Embedded Workbench 6.`*`n`*`\r32c\doc`.

TYPOGRAPHIC CONVENTIONS

The IAR Systems documentation set uses the following typographic conventions:

Style	Used for
`computer`	• Source code examples and file paths. • Text on the command line. • Binary, hexadecimal, and octal numbers.
`parameter`	A placeholder for an actual value used as a parameter, for example *`filename`*`.h` where *`filename`* represents the name of the file.
`[option]`	An optional part of a command.
`[a|b|c]`	An optional part of a command with alternatives.
`{a|b|c}`	A mandatory part of a command with alternatives.
bold	Names of menus, menu commands, buttons, and dialog boxes that appear on the screen.
italic	• A cross-reference within this guide or to another guide. • Emphasis.
…	An ellipsis indicates that the previous item can be repeated an arbitrary number of times.
🔧	Identifies instructions specific to the IAR Embedded Workbench® IDE interface.
🖥	Identifies instructions specific to the command line interface.
💡	Identifies helpful tips and programming hints.

Table 1: Typographic conventions used in this guide

Style	Used for
⚠	Identifies warnings.

Table 1: Typographic conventions used in this guide (Continued)

NAMING CONVENTIONS

The following naming conventions are used for the products and tools from IAR Systems®, when referred to in the documentation:

Brand name	Generic term
IAR Embedded Workbench® for R32C	IAR Embedded Workbench®
IAR Embedded Workbench® IDE for R32C	the IDE
IAR C-SPY® Debugger for R32C	C-SPY, the debugger
IAR C-SPY® Simulator	the simulator
IAR C/C++ Compiler™ for R32C	the compiler
IAR Assembler™ for R32C	the assembler
IAR XLINK Linker™	XLINK, the linker
IAR XAR Library Builder™	the library builder
IAR XLIB Librarian™	the librarian
IAR DLIB Library™	the DLIB library

Table 2: Naming conventions used in this guide

The IAR C-SPY Debugger

This chapter introduces you to the IAR C-SPY® Debugger and to the concepts that are related to debugging in general and to C-SPY in particular. The chapter also introduces the various C-SPY drivers. More specifically, this means:

- Introduction to C-SPY
- Debugger concepts
- C-SPY drivers overview
- The IAR C-SPY Simulator
- The C-SPY Hardware debugger drivers

Introduction to C-SPY

This section covers these topics:

- An integrated environment
- General C-SPY debugger features
- RTOS awareness.

AN INTEGRATED ENVIRONMENT

C-SPY is a high-level-language debugger for embedded applications. It is designed for use with the IAR Systems compilers and assemblers, and is completely integrated in the IDE, providing development and debugging within the same application. This will give you possibilities such as:

- Editing while debugging. During a debug session, you can make corrections directly in the same source code window that is used for controlling the debugging. Changes will be included in the next project rebuild.
- Setting breakpoints at any point during the development cycle. You can inspect and modify breakpoint definitions also when the debugger is not running, and breakpoint definitions flow with the text as you edit. Your debug settings, such as watch properties, window layouts, and register groups will be preserved between your debug sessions.

All windows that are open in the Embedded Workbench workspace will stay open when you start the C-SPY Debugger. In addition, a set of C-SPY-specific windows are opened.

GENERAL C-SPY DEBUGGER FEATURES

Because IAR Systems provides an entire toolchain, the output from the compiler and linker can include extensive debug information for the debugger, resulting in good debugging possibilities for you.

C-SPY offers these general features:

- Source and disassembly level debugging

  C-SPY allows you to switch between source and disassembly debugging as required, for both C or C++ and assembler source code.
- Single-stepping on a function call level

  Compared to traditional debuggers, where the finest granularity for source level stepping is line by line, C-SPY provides a finer level of control by identifying every statement and function call as a step point. This means that each function call—inside expressions, and function calls that are part of parameter lists to other functions—can be single-stepped. The latter is especially useful when debugging C++ code, where numerous extra function calls are made, for example to object constructors.
- Code and data breakpoints

  The C-SPY breakpoint system lets you set breakpoints of various kinds in the application being debugged, allowing you to stop at locations of particular interest. For example, you set breakpoints to investigate whether your program logic is correct or to investigate how and when the data changes.
- Monitoring variables and expressions

  For variables and expressions there is a wide choice of facilities. You can easily monitor values of a specified set of variables and expressions, continuously or on demand. You can also choose to monitor only local variables, static variables, etc.
- Container awareness

  When you run your application in C-SPY, you can view the elements of library data types such as STL lists and vectors. This gives you a very good overview and debugging opportunities when you work with C++ STL containers.
- Call stack information

  The compiler generates extensive call stack information. This allows the debugger to show, without any runtime penalty, the complete stack of function calls wherever the program counter is. You can select any function in the call stack, and for each function you get valid information for local variables and available registers.

- Powerful macro system

  C-SPY includes a powerful internal macro system, to allow you to define complex sets of actions to be performed. C-SPY macros can be used on their own or in conjunction with complex breakpoints and—if you are using the simulator—the interrupt simulation system to perform a wide variety of tasks.

Additional general C-SPY debugger features

This list shows some additional features:

- Threaded execution keeps the IDE responsive while running the target application
- Automatic stepping
- The source browser provides easy navigation to functions, types, and variables
- Extensive type recognition of variables
- Configurable registers (CPU and peripherals) and memory windows
- Graphical stack view with overflow detection
- Support for code coverage and function level profiling
- The target application can access files on the host PC using file I/O
- UBROF, Intel-extended, and Motorola input formats supported
- Optional terminal I/O emulation.

RTOS AWARENESS

C-SPY supports real-time OS aware debugging.

These operating systems are currently supported:

- SEGGER embOS
- Micrium uC/OS

RTOS plugin modules can be provided by IAR Systems, and by third-party suppliers. Contact your software distributor or IAR Systems representative, alternatively visit the IAR Systems web site, for information about supported RTOS modules.

A C-SPY RTOS awareness plugin module gives you a high level of control and visibility over an application built on top of an RTOS. It displays RTOS-specific items like task lists, queues, semaphores, mailboxes, and various RTOS system variables. Task-specific breakpoints and task-specific stepping make it easier to debug tasks.

A loaded plugin will add its own menu, set of windows, and buttons when a debug session is started (provided that the RTOS is linked with the application). For information about other RTOS awareness plugin modules, refer to the manufacturer of the plugin module.

Debugger concepts

This section introduces some of the concepts and terms that are related to debugging in general and to C-SPY in particular. This section does not contain specific information related to C-SPY features. Instead, you will find such information in the other chapters of this documentation. The IAR Systems user documentation uses the terms described in this section when referring to these concepts.

C-SPY AND TARGET SYSTEMS

You can use C-SPY to debug either a software target system or a hardware target system.

This figure gives an overview of C-SPY and possible target systems:

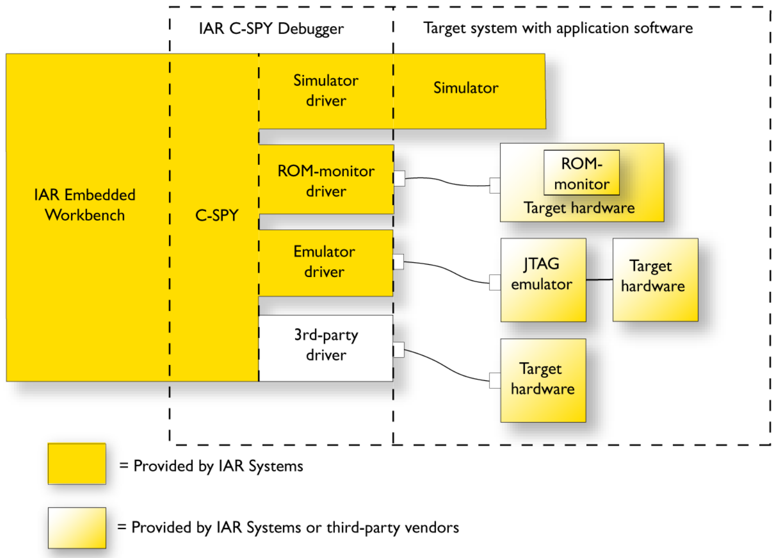


Figure 1: C-SPY and target systems

Note: In IAR Embedded Workbench for R32C, there are no ROM-monitor drivers.

THE DEBUGGER

The debugger, for instance C-SPY, is the program that you use for debugging your applications on a target system.

THE TARGET SYSTEM

The target system is the system on which you execute your application when you are debugging it. The target system can consist of hardware, either an evaluation board or your own hardware design. It can also be completely or partially simulated by software. Each type of target system needs a dedicated C-SPY driver.

THE APPLICATION

A user application is the software you have developed and which you want to debug using C-SPY.

C-SPY DEBUGGER SYSTEMS

C-SPY consists of both a general part which provides a basic set of debugger features, and a target-specific back end. The back end consists of two components: a processor module—one for every microcontroller, which defines the properties of the microcontroller, and a *C-SPY driver*. The C-SPY driver is the part that provides communication with and control of the target system. The driver also provides the user interface—menus, windows, and dialog boxes—to the functions provided by the target system, for instance, special breakpoints. Typically, there are three main types of C-SPY drivers:

- Simulator driver
- ROM-monitor driver
- Emulator driver.

C-SPY is available with a simulator driver, and depending on your product package, optional drivers for hardware debugger systems. For an overview of the available C-SPY drivers and the functionality provided by each driver, see *C-SPY drivers overview*, page 28.

THE ROM-MONITOR PROGRAM

The ROM-monitor program is a piece of firmware that is loaded to non-volatile memory on your target hardware; it runs in parallel with your application. The ROM-monitor communicates with the debugger and provides services needed for debugging the application, for instance stepping and breakpoints.

THIRD-PARTY DEBUGGERS

You can use a third-party debugger together with the IAR Systems toolchain as long as the third-party debugger can read any of the output formats provided by XLINK, such as UBROF, ELF/DWARF, COFF, Intel-extended, Motorola, or any other available format. For information about which format to use with a third-party debugger, see the user documentation supplied with that tool.

C-SPY PLUGIN MODULES

C-SPY is designed as a modular architecture with an open SDK that can be used for implementing additional functionality to the debugger in the form of plugin modules. These modules can be seamlessly integrated in the IDE.

Plugin modules are provided by IAR Systems, or can be supplied by third-party vendors. Examples of such modules are:

- Code Coverage, which is integrated in the IDE.
- The various C-SPY drivers for debugging using certain debug systems.
- RTOS plugin modules for support for real-time OS aware debugging.
- C-SPYLink that bridges IAR visualSTATE and IAR Embedded Workbench to make true high-level state machine debugging possible directly in C-SPY, in addition to the normal C level symbolic debugging. For more information, see the documentation provided with IAR visualSTATE.

For more information about the C-SPY SDK, contact IAR Systems.

C-SPY drivers overview

At the time of writing this guide, the IAR C-SPY Debugger for the R32C/100 microcomputers is available with drivers for these target systems and evaluation boards:

- Simulator
- E8a
- E30
- E30A.

DIFFERENCES BETWEEN THE C-SPY DRIVERS

This table summarizes the key differences between the C-SPY drivers:

Feature	Simulator	E8a	E30	E30A
Code breakpoints	Unlimited	x	x	x
Data breakpoints	x	--	x	x
Execution in real time	--	x	x	x
Zero memory footprint	x	--	x	x
Simulated interrupts	x	--	--	--
Real interrupts	--	x	x	x
Interrupt logging	x	--	--	--

Table 3: Driver differences

Feature	Simulator	E8a	E30	E30A
Data logging	x	--	--	--
Live watch	--	x	x	x
Cycle counter	x	--	x[1]	x[1]
Code coverage	x	--	--	--
Data coverage	x	--	--	--
Function/instruction profiling	x	--	--	--
Trace	x	--	x[2]	x[2]

Table 3: Driver differences (Continued)

1 Not during single stepping.

2 Limited during branch source and destination information about data accesses, and a maximum of 512 recorded events.

The IAR C-SPY Simulator

The C-SPY Simulator simulates the functions of the target processor entirely in software, which means that you can debug the program logic long before any hardware is available. Because no hardware is required, it is also the most cost-effective solution for many applications.

SIMULATOR FEATURES

In addition to the general features in C-SPY, the simulator also provides:

- Instruction-level simulation
- Memory configuration and validation
- Interrupt simulation
- Peripheral simulation (using the C-SPY macro system in conjunction with immediate breakpoints).

The C-SPY Hardware debugger drivers

C-SPY can connect to a hardware debugger using a C-SPY Hardware debugger driver as an interface. The C-SPY Hardware debugger drivers are automatically installed during the installation of IAR Embedded Workbench.

IAR Embedded Workbench for R32C comes with several C-SPY Hardware debugger drivers and you use the driver that matches the hardware debugger you are using.

COMMUNICATION OVERVIEW

Most target systems have a debug probe or a debug adapter connected between the host computer and the evaluation board.

The C-SPY Hardware debugger driver uses USB to communicate with the hardware debugger. The hardware debugger communicates with the interface on the microcontroller.

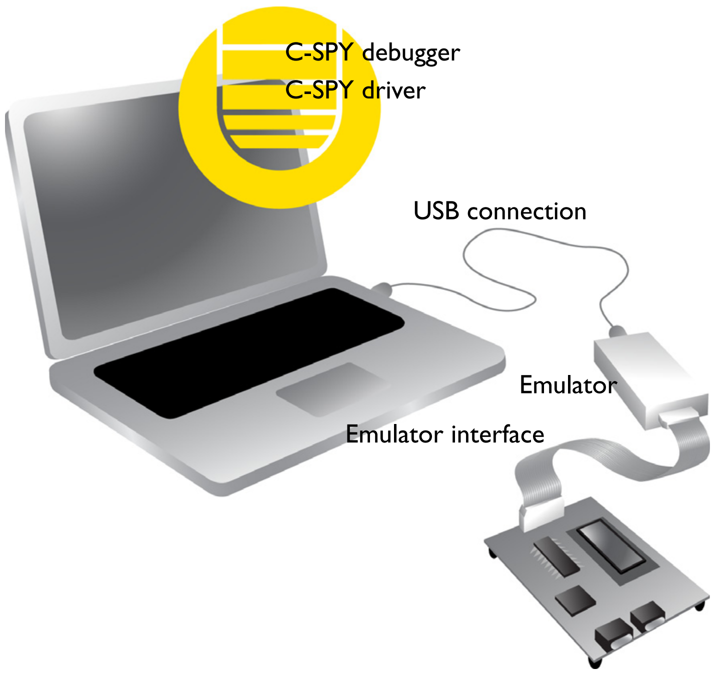


Figure 2: C-SPY Hardware debugger communication overview

For more information, refer to the documentation supplied with the hardware debugger.

When a debugging session is started, your application is automatically downloaded and programmed into flash memory. You can disable this feature, if necessary.

Hardware installation

USB drivers are automatically installed during the installation of IAR Embedded Workbench. If you need to re-install them, they are available both on the installation CD and in the `*install_dir*\drivers\Renesas` directory of your IAR Systems product installation.

Getting started using C-SPY

This chapter helps you get started using C-SPY®. More specifically, this means:

- Setting up C-SPY
- Starting C-SPY
- Adapting for target hardware
- Running example projects
- Reference information on starting C-SPY.

Setting up C-SPY

This section describes how to set up C-SPY.

More specifically, you will get information about:

- Setting up for debugging
- Executing from reset
- Using a setup macro file
- Selecting a device description file
- Loading plugin modules.

SETTING UP FOR DEBUGGING

1 Install a USB driver if your C-SPY driver requires it. For more information, see:

- *Hardware installation*, page 30

2 Before you start C-SPY, choose **Project>Options>Debugger>Setup** and select the C-SPY driver that matches your debugger system: simulator or a hardware debugger system.

Note: You can only choose a driver you have installed on your computer.

3 In the **Category** list, select the appropriate C-SPY driver and make your settings.

For information about these options, see *Debugger options*, page 267.

4 Click **OK**.

5 Choose **Tools>Options>Debugger** to configure:

- The debugger behavior
- The debugger's tracking of stack usage.

For more information about these options, see the *IDE Project Management and Building Guide*.

See also *Adapting for target hardware*, page 36.

EXECUTING FROM RESET

The **Run to** option—available on the **Debugger>Setup** page—specifies a location you want C-SPY to run to when you start a debug session as well as after each reset. C-SPY will place a temporary breakpoint at this location and all code up to this point is executed before stopping at the location.

The default location to run to is the `main` function. Type the name of the location if you want C-SPY to run to a different location. You can specify assembler labels or whatever can be evaluated to such, for instance function names.

If you leave the check box empty, the program counter will then contain the regular hardware reset address at each reset.

If no breakpoints are available when C-SPY starts, a warning message notifies you that single stepping will be required and that this is time-consuming. You can then continue execution in single-step mode or stop at the first instruction. If you choose to stop at the first instruction, the debugger starts executing with the `PC` (program counter) at the default reset location instead of the location you typed in the **Run to** box.

Note: This message will never be displayed in the C-SPY Simulator, where breakpoints are not limited.

USING A SETUP MACRO FILE

A setup macro file is a macro file that you choose to load automatically when C-SPY starts. You can define the setup macro file to perform actions according to your needs, using setup macro functions and system macros. Thus, if you load a setup macro file you can initialize C-SPY to perform actions automatically.

For more information about setup macro files and functions, see *Introduction to C-SPY macros, page 215*. For an example of how to use a setup macro file, see the chapter *Initializing target hardware before C-SPY starts*, page 37.

To register a setup macro file:

1 Before you start C-SPY, choose **Project>Options>Debugger>Setup**.

2 Select **Use macro file** and type the path and name of your setup macro file, for example `Setup.mac`. If you do not type a filename extension, the extension `mac` is assumed.

SELECTING A DEVICE DESCRIPTION FILE

C-SPY uses device description files to handle device-specific information.

A default device description file is automatically used based on your project settings. If you want to override the default file, you must select your device description file. Device description files are provided in the `R32C\config` directory and they have the filename extension `ddf`.

For more information about device description files, see *Adapting for target hardware*, page 36.

To override the default device description file:

1 Before you start C-SPY, choose **Project>Options>Debugger>Setup**.

2 Enable the use of a device description file and select a file using the **Device description file** browse button.

LOADING PLUGIN MODULES

On the **Plugins** page you can specify C-SPY plugin modules to load and make available during debug sessions. Plugin modules can be provided by IAR Systems, and by third-party suppliers. Contact your software distributor or IAR Systems representative, or visit the IAR Systems web site, for information about available modules.

For more information, see *Plugins*, page 271.

Starting C-SPY

When you have set up the debugger, you are ready to start a debug session; this section describes various ways to start C-SPY.

More specifically, you will get information about:

- Starting a debug session
- Loading executable files built outside of the IDE
- Starting a debug session with source files missing
- Loading multiple images.

STARTING A DEBUG SESSION

You can choose to start a debug session with or without loading the current project.

To start C-SPY and load the current project, click the **Download and Debug** button. Alternatively, choose **Project>Download and Debug**.

To start C-SPY without reloading the current project, click the **Debug without Downloading** button. Alternatively, choose **Project>Debug without Downloading**.

LOADING EXECUTABLE FILES BUILT OUTSIDE OF THE IDE

You can also load C-SPY with an application that was built outside the IDE, for example applications built on the command line. To load an externally built executable file and to set build options you must first create a project for it in your workspace.

To create a project for an externally built file:

1 Choose **Project>Create New Project**, and specify a project name.

2 To add the executable file to the project, choose **Project>Add Files** and make sure to choose **All Files** in the **Files of type** drop-down list. Locate the executable file.

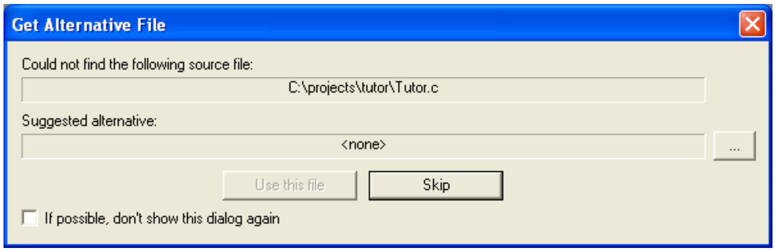
3 To start the executable file, click the **Download and Debug** button. The project can be reused whenever you rebuild your executable file.

The only project options that are meaningful to set for this kind of project are options in the **General Options** and **Debugger** categories. Make sure to set up the general project options in the same way as when the executable file was built.

STARTING A DEBUG SESSION WITH SOURCE FILES MISSING

Normally, when you use the IAR Embedded Workbench IDE to edit source files, build your project, and start the debug session, all required files are available and the process works as expected.

However, if C-SPY cannot automatically find the source files, for example if the application was built on another computer, the **Get Alternative File** dialog box is displayed:



Could not find the following source file:
C:\projects\tutor\Tutor.c
Suggested alternative:
<none>
If possible, don't show this dialog again

Figure 3: Get Alternative File dialog box

Typically, you can use the dialog box like this:

- The source files are not available: Click **If possible, don't show this dialog again** and then click **Skip**. C-SPY will assume that there simply is no source file available. The dialog box will not appear again, and the debug session will not try to display the source code.
- Alternative source files are available at another location: Specify an alternative source code file, click **If possible, don't show this dialog again**, and then click **Use this file**. C-SPY will assume that the alternative file should be used. The dialog box will not appear again, unless a file is needed for which there is no alternative file specified and which cannot be located automatically.

If you restart the IAR Embedded Workbench IDE, the **Get Alternative File** dialog box will be displayed again once even if you have clicked **If possible, don't show this dialog again**. This gives you an opportunity to modify your previous settings.

For more information, see *Get Alternative File dialog box*, page 52.

LOADING MULTIPLE IMAGES

Normally, a debuggable application consists of exactly one file that you debug. However, you can also load additional debug files (images). This means that the complete program consists of several images.

Typically, this is useful if you want to debug your application in combination with a prebuilt ROM image that contains an additional library for some platform-provided features. The ROM image and the application are built using separate projects in the IAR Embedded Workbench IDE and generate separate output files.

If more than one image has been loaded, you will have access to the combined debug information for all the loaded images. In the Images window you can choose whether you want to have access to debug information for one image or for all images.

To load additional images at C-SPY startup:

1. Choose **Project>Options>Debugger>Images** and specify up to three additional images to be loaded. For more information, see *Images*, page 270.
2. Start the debug session.

To load additional images at a specific moment:

Use the `__loadImage` system macro and execute it using either one of the methods described in *Using C-SPY macros*, page 217.

To display a list of loaded images:

Choose **Images** from the **View** menu. The Images window is displayed, see *Images window*, page 50.

Adapting for target hardware

This section provides information about how to describe the target hardware to C-SPY, and how you can make C-SPY initialize the target hardware before your application is downloaded to memory.

More specifically, you will get information about:

- Modifying a device description file
- Initializing target hardware before C-SPY starts

MODIFYING A DEVICE DESCRIPTION FILE

C-SPY uses device description files provided with the product to handle several of the target-specific adaptations, see *Selecting a device description file*, page 33. They contain device-specific information such as:

- Memory information for device-specific memory zones, see *C-SPY memory zones*, page 130.
- Definitions of memory-mapped peripheral units, device-specific CPU registers, and groups of these
- Definitions for device-specific interrupts, which makes it possible to simulate these interrupts in the C-SPY simulator; see *Interrupts*, page 195.
- Information used by the E8a, E30, and E30A emulators.

Normally, you do not need to modify the device description file. However, if the predefinitions are not sufficient for some reason, you can edit the file. Note, however, that the format of these descriptions might be updated in future upgrade versions of the product.

Make a copy of the device description file that best suits your needs, and modify it according to the description in the file.

For information about how to load a device description file, see *Selecting a device description file*, page 33.

INITIALIZING TARGET HARDWARE BEFORE C-SPY STARTS

You can use C-SPY macros to initialize target hardware before C-SPY starts. For example, if your hardware uses external memory that must be enabled before code can be downloaded to it, C-SPY needs a macro to perform this action before your application can be downloaded. For example:

1 Create a new text file and define your macro function. For example, a macro that enables external SDRAM might look like this:

```
/* Your macro function. */
enableExternalSDRAM()
{
  __message "Enabling external SDRAM\n";
  __writeMemory32( /* Place your code here. */ );
  /* And more code here, if needed. */
}

/* Setup macro determines time of execution. */
execUserPreload()
{
  enableExternalSDRAM();
}
```

Because the built-in `execUserPreload` setup macro function is used, your macro function will be executed directly after the communication with the target system is established but before C-SPY downloads your application.

2 Save the file with the filename extension `mac`.

3 Before you start C-SPY, choose **Project>Options>Debugger** and click the **Setup** tab.

4 Select the option **Use Setup file** and choose the macro file you just created.

Your setup macro will now be loaded during the C-SPY startup sequence.

DOWNLOADING FIRMWARE

If you need to download new firmware, for example if you are changing the processor configuration or if you need to upgrade the firmware, choose **Emulator>Download Firmware**. The emulator firmware files have the filename extension `.s` and are located in subdirectories of the `r32c\config\Renesas` directory of your product installation.

Running example projects

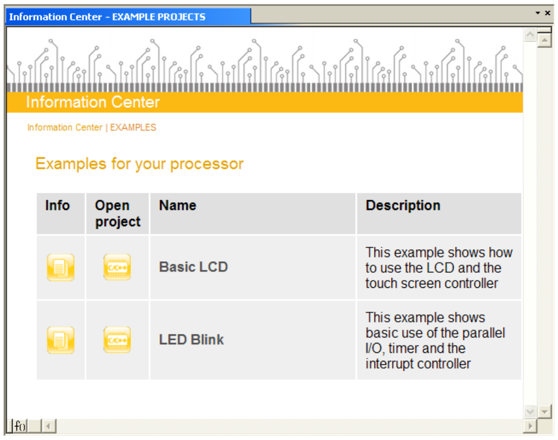
IAR Embedded Workbench comes with example applications. You can use these examples to get started using the development tools from IAR Systems or simply to verify that contact has been established with your target board. You can also use the examples as a starting point for your application project.

You can find the examples in the `R32C\examples` directory. The examples are ready to be used as is. They are supplied with ready-made workspace files, together with source code files and all other related files.

RUNNING AN EXAMPLE PROJECT

To run an example project:

1. Choose **Help>Information Center** and click **EXAMPLE PROJECTS**.
2. Browse to the example that matches the specific evaluation board or starter kit you are using.


Information Center - EXAMPLE PROJECTS
Information Center
Information Center | EXAMPLES
Examples for your processor

Info	Open project	Name	Description
		Basic LCD	This example shows how to use the LCD and the touch screen controller
		LED Blink	This example shows basic use of the parallel I/O, timer and the interrupt controller

Figure 4: Example applications

Click the **Open Project** button.

3 In the dialog box that appears, choose a destination folder for your project location. Click **Select** to confirm your choice.

4 The available example projects are displayed in the workspace window. Select one of the projects, and if it is not the active project (highlighted in bold), right-click it and choose **Set As Active** from the context menu.

5 To view the project settings, select the project and choose **Options** from the context menu. Verify the settings for **Device** and **Debugger>Setup>Driver**. As for other settings, the project is set up to suit the target system you selected.

For more information about the C-SPY options and how to configure C-SPY to interact with the target board, see *Debugger options*, page 267.

Click **OK** to close the project **Options** dialog box.

6 To compile and link the application, choose **Project>Make** or click the **Make** button.

7 To start C-SPY, choose **Project>Debug** or click the **Download and Debug** button. If C-SPY fails to establish contact with the target system, see *Resolving problems*, page 286.

8 Choose **Debug>Go** or click the **Go** button to start the application.

Click the **Stop** button to stop execution.

Connecting to the target board, E8a

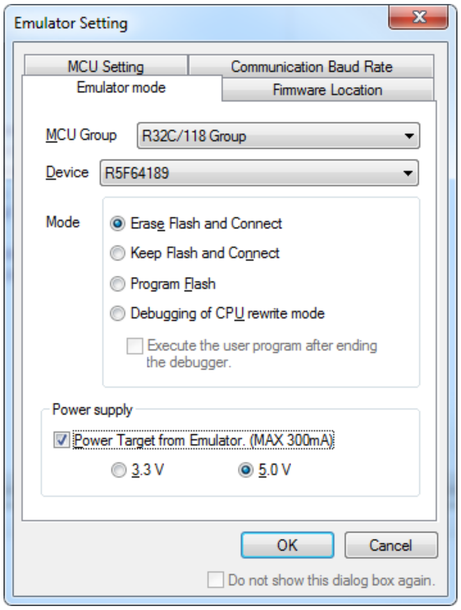
To establish a connection to the target board, you must follow this procedure:

1 To select the device that matches your target system, choose **Project>Options>General Options>Target**.

2 Select the E8a driver from the **Project>Options>Debugger>Setup>Driver** dropdown list.

3 Build the project if it has not been built and choose **Project>Debug** to start C-SPY.

4 The **Emulator Setting** dialog box is displayed.



MCU Group: R32C/118 Group

Device: R5F64189

Mode:
- [✓ Erase Flash and Connect]
- Keep Flash and Connect
- Program Flash
- Debugging of CPU rewrite mode
- ☐ Execute the user program after ending the debugger.

Power supply
- ☑ Power Target from Emulator. (MAX 300mA)
- 3.3 V [✓ 5.0 V]

☐ Do not show this dialog box again.

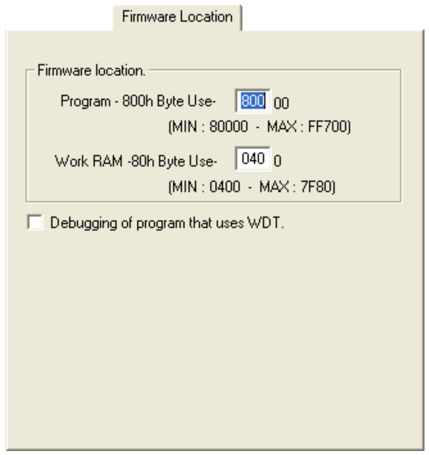
Figure 5: The Emulator Setting dialog box for the E8a emulator

Select your MCU group and device from the drop-down lists. On the **MCU Setting** and **Communication Baud Rate** pages, you do not need to make any settings at this point. See also *Emulator Setting*, page 53.

5 If the **Firmware Location & WDT** dialog box is displayed, you must confirm the placement of code and data on the target microcontroller. If your application uses a watchdog timer, select the **Debugging of program that uses WDT** option to cause the watchdog timer to be refreshed when your application is running.

If you use the default device-specific XLINK linker command file, you do not need to change any settings.

Note: Make sure that your application is not downloaded to the firmware location.


Firmware Location

Firmware location.

Program - 800h Byte Use- 800 00
(MIN : 80000 - MAX : FF700)

Work RAM -80h Byte Use- 040 0
(MIN : 0400 - MAX : 7F80)

☐ Debugging of program that uses WDT.

Figure 6: The Firmware Location page for the E8/E8a emulator

If this dialog box is not displayed, skip this step.

Click **OK** in the **Emulator Setting** dialog box to save your settings.

6 The **Connecting** dialog box is displayed and the emulator connection is started.

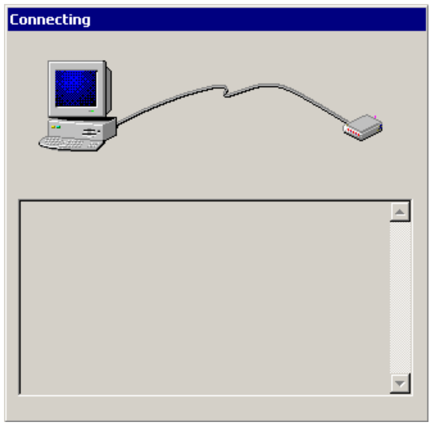
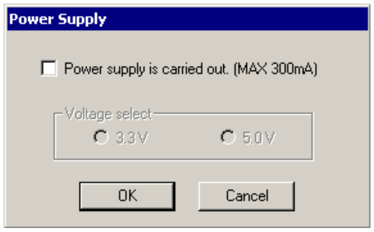


Figure 7: The Connecting dialog box

7 If you are prompted to confirm that the target board is supplied with power, select the check box and the correct voltage.



Power Supply

- [ ] Power supply is carried out. (MAX 300mA)

Voltage select

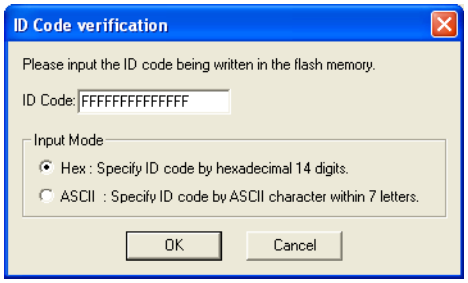
○ 3.3V ○ 5.0V

Figure 8: The Power Supply dialog box for the E8/E8a emulator

If this dialog box is not displayed, skip this step.

Note: Before you connect the target board to a power supply, check the power specifications and that there are no short circuits. Incorrect operation might damage the board and the emulator.

8 If the **ID Code verification** dialog box is displayed, enter the hexadecimal ID security code for the flash memory.



ID Code verification

Please input the ID code being written in the flash memory.

ID Code: FFFFFFFFFFFFFF

Input Mode

● Hex : Specify ID code by hexadecimal 14 digits.

○ ASCII : Specify ID code by ASCII character within 7 letters.

Figure 9: The ID Code verification dialog box for the E8/E8a emulator

When the emulator is ready to be used, `Connected` will be printed in the IAR Embedded Workbench Debug Log window together with the number of available hardware and software breakpoints.

MAINTAINING THE CONNECTION TO THE HARDWARE

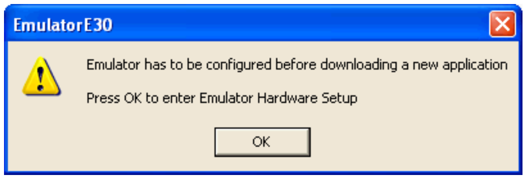
When a debug session is started, a connection to the hardware is established and an initialization routine is executed. The initialization sets the emulator options for your target system.

When you start a debug session with the same hardware as the previous session and without closing the IDE between the debug sessions, C-SPY can skip the initialization to save time. For information on how to set this behavior in C-SPY, see *Emulator Setting*, page 53, specifically the description of the option **Do not show this dialog box again**. If you set this option, the connection to the hardware is maintained and the initialization is not repeated. This option is mirrored in the menu command **Show Emulator Setting**, see the *Emulator menu*, page 277.

Connecting to the target board, E30/E30A

To establish a connection to the target board, you must follow this procedure:

1. To select the device that matches your target system, choose **Project>Options>General Options>Target**.
2. Select the driver that matches your emulator on the **Project>Options>Debugger>Setup** page. Click **OK** to close the **Options** dilog box.
3. To compile and link the application, choose **Project>Make** or click the **Make** button.
4. To start C-SPY, choose Project>Debug or click the **Download and Debug** button.
5. Before C-SPY is started for the first time in a new project, and when you change the device, the hardware must be set up. If you have not already set up the hardware by choosing **Emulator>Hardware Setup**, this dialog box will be displayed when you start the debug session:



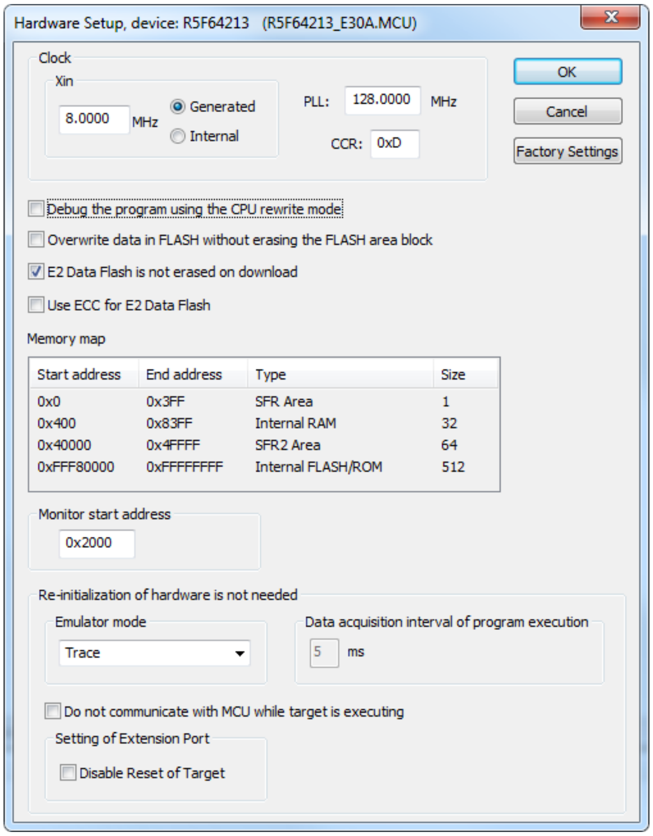
EmulatorE30

Emulator has to be configured before downloading a new application

Press OK to enter Emulator Hardware Setup

Click OK to enter the **Hardware Setup** dialog box.

6 The **Hardware Setup** dialog box is displayed.



Clock

Xin: 8.0000 MHz [✓ Generated] Internal

PLL: 128.0000 MHz

CCR: 0xD

- [ ] Debug the program using the CPU rewrite mode
- [ ] Overwrite data in FLASH without erasing the FLASH area block
- [x] E2 Data Flash is not erased on download
- [ ] Use ECC for E2 Data Flash

Memory map

Start address	End address	Type	Size
0x0	0x3FF	SFR Area	1
0x400	0x83FF	Internal RAM	32
0x40000	0x4FFFF	SFR2 Area	64
0xFFF80000	0xFFFFFFFF	Internal FLASH/ROM	512

Monitor start address: 0x2000

Re-initialization of hardware is not needed

Emulator mode: Trace

Data acquisition interval of program execution: 5 ms

- [ ] Do not communicate with MCU while target is executing

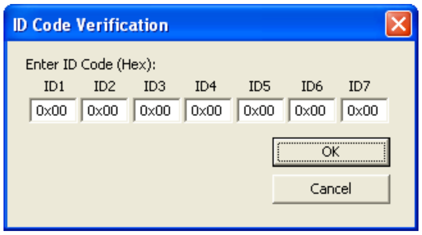
Setting of Extension Port

- [ ] Disable Reset of Target

Make sure that the setting of **Xin** matches your hardware. The settings of **PLL** and **CCR** must match the MCU initialization in your application code. See also *Hardware Setup*, page 55.

Note: This figure reflects the C-SPY E30A driver. Some of the options are not available when using the C-SPY E30 driver.

7 A device can be read-protected with an ID code. If this is the case, the **ID Code Verification** dialog box is displayed.



To debug a read-protected device, enter the correct seven bytes in hexadecimal notation. To protect the device, redefine the ID Code symbols in the extended linker configuration file. When the project is linked, this will reprogram the ID code in the internal flash ROM.

8 When you are asked to reset the MCU, reset it and then click **OK**. C-SPY will download your application to the target system.

Reference information on starting C-SPY

This section gives reference information about these windows and dialog boxes:

- *C-SPY Debugger main window*, page 45
- *Images window*, page 50
- *Get Alternative File dialog box*, page 52
- *Emulator Setting*, page 53
- *Hardware Setup*, page 55

See also:

- Tools options for the debugger in the *IDE Project Management and Building Guide*.

C-SPY Debugger main window

When you start a debug session, these debugger-specific items appear in the main IAR Embedded Workbench IDE window:

- A dedicated **Debug** menu with commands for executing and debugging your application

- Depending on the C-SPY driver you are using, a driver-specific menu, often referred to as the *Driver menu* in this documentation. Typically, this menu contains menu commands for opening driver-specific windows and dialog boxes.
- A special debug toolbar
- A special trace setup toolbar
- Several windows and dialog boxes specific to C-SPY.

The C-SPY main window might look different depending on which components of the product installation you are using.

Menu bar

These menus are available during a debug session:

Debug

Provides commands for executing and debugging the source application. Most of the commands are also available as icon buttons on the debug toolbar.

Simulator

Provides access to the dialog boxes for setting up interrupt simulation and memory access checking. This menu is only available when the C-SPY Simulator is used, see *Simulator menu*, page 275.

Emulator

Provides commands specific to the E8a, E30, and E30A emulator drivers. This menu is only available when one of the C-SPY emulator drivers is used, see *Emulator menu*, page 277.

Debug menu

The **Debug** menu is available during a debug session. The **Debug** menu provides commands for executing and debugging the source application. Most of the commands are also available as icon buttons on the debug toolbar.

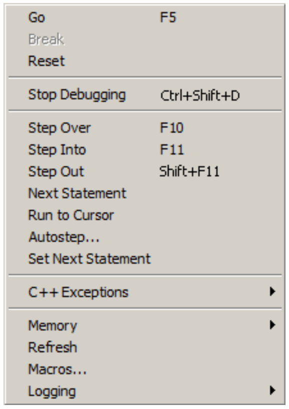


Figure 10: Debug menu

These commands are available:

Go F5

Executes from the current statement or instruction until a breakpoint or program exit is reached.

Break

Stops the application execution.

Reset

Resets the target processor.

Stop Debugging (Ctrl+Shift+D)

Stops the debugging session and returns you to the project manager.

Step Over (F10)

Executes the next statement, function call, or instruction, without entering C or C++ functions or assembler subroutines.

Step Into (F11)

Executes the next statement or instruction, or function call, entering C or C++ functions or assembler subroutines.

Step Out (Shift+F11)

Executes from the current statement up to the statement after the call to the current function.

Next Statement

Executes directly to the next statement without stopping at individual function calls.

Run to Cursor

Executes from the current statement or instruction up to a selected statement or instruction.

Autostep

Displays a dialog box where you can customize and perform autostepping, see *Autostep settings dialog box*, page 77.

Set Next Statement

Moves the program counter directly to where the cursor is, without executing any source code. Note, however, that this creates an anomaly in the program flow and might have unexpected effects.

C++ Exceptions>
Break on Throw

This menu command is not supported by your product package.

C++ Exceptions>
Break on Uncaught Exception

This menu command is not supported by your product package.

Memory>Save

Displays a dialog box where you can save the contents of a specified memory area to a file, see *Memory Save dialog box*, page 139.

Memory>Restore

Displays a dialog box where you can load the contents of a file in, for example Intel-extended or Motorola s-record format to a specified memory zone, see *Memory Restore dialog box*, page 140.

Refresh

Refreshes the contents of all debugger windows. Because window updates are automatic, this is needed only in unusual situations, such as when target memory is modified in ways C-SPY cannot detect. It is also useful if code that is displayed in the Disassembly window is changed.

Macros

Displays a dialog box where you can list, register, and edit your macro files and functions, see *Using the Macro Configuration dialog box*, page 219.

Logging>Set Log file

Displays a dialog box where you can choose to log the contents of the Debug Log window to a file. You can select the type and the location of the log file. You can choose what you want to log: errors, warnings, system information, user messages, or all of these. See *Log File dialog box*, page 75.

Logging>
Set Terminal I/O Log file

Displays a dialog box where you can choose to log simulated target access communication to a file. You can select the destination of the log file. See *Terminal I/O Log File dialog box*, page 74

C-SPY windows

Depending on the C-SPY driver you are using, these windows specific to C-SPY are available during a debug session:

- C-SPY Debugger main window
- Disassembly window
- Memory window
- Symbolic Memory window
- Register window
- Watch window
- Locals window
- Auto window
- Live Watch window
- Quick Watch window
- Statics window
- Call Stack window
- Trace window
- Function Trace window
- Timeline window
- Terminal I/O window
- Code Coverage window
- Function Profiler window
- Images window

- Stack window
- Symbols window.

Additional windows are available depending on which C-SPY driver you are using.

Editing in C-SPY windows

You can edit the contents of the Memory, Symbolic Memory, Register, Auto, Watch, Locals, Statics, Live Watch, and Quick Watch windows.

Use these keyboard keys to edit the contents of these windows:

Enter	Makes an item editable and saves the new value.
Esc	Cancels a new value.

In windows where you can edit the **Expression** field, you can specify the number of elements to be displayed in the field by adding a semicolon followed by an integer. For example, to display only the three first elements of an array named `myArray`, or three elements in sequence starting with the element pointed to by a pointer, write:

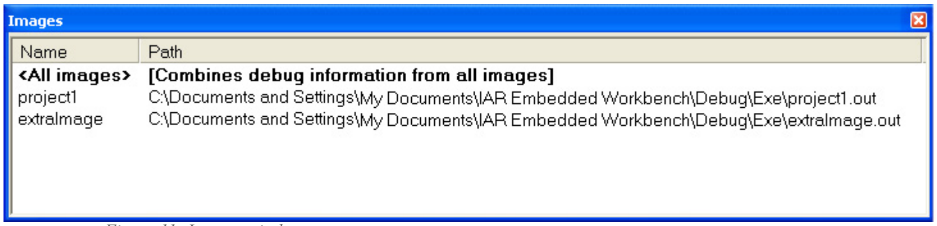
```
myArray;3
```

Optionally, add a comma and another integer that specifies which element to start with. For example, to display elements 10–14, write:

```
myArray;5,10
```

Images window

The Images window is available from the **View** menu.



Name	Path
<All images>	**[Combines debug information from all images]**
project1	C:\Documents and Settings\My Documents\IAR Embedded Workbench\Debug\Exe\project1.out
extraImage	C:\Documents and Settings\My Documents\IAR Embedded Workbench\Debug\Exe\extraImage.out

Figure 11: Images window

The Images window lists all currently loaded images (debug files).

Normally, a source application consists of exactly one image that you debug. However, you can also load additional images. This means that the complete debuggable unit consists of several images.

Requirements

None; this window is always available.

Display area

C-SPY can either use debug information from all of the loaded images simultaneously, or from one image at a time. Double-click on a row to show information only for that image. The current choice is highlighted.

This area lists the loaded images in these columns:

Name

The name of the loaded image.

Path

The path to the loaded image.

Context menu

This context menu is available:

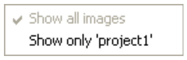


Figure 12: Images window context menu

These commands are available:

Show all images

Shows debug information for all loaded debug images.

Show only *image*

Shows debug information for the selected debug image.

Related information

For related information, see:

- *Loading multiple images*, page 35
- *Images*, page 270
- *__loadImage*, page 234.

Get Alternative File dialog box

The **Get Alternative File** dialog box is displayed if C-SPY cannot automatically find the source files to be loaded, for example if the application was built on another computer.

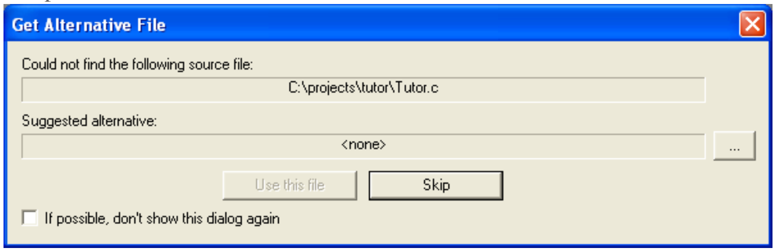

Get Alternative File
Could not find the following source file:
C:\projects\tutor\Tutor.c
Suggested alternative:
<none>
...
Use this file
Skip
If possible, don't show this dialog again

Figure 13: Get Alternative File dialog box

Could not find the following source file

The missing source file.

Suggested alternative

Specify an alternative file.

Use this file

After you have specified an alternative file, **Use this file** establishes that file as the alias for the requested file. Note that after you have chosen this action, C-SPY will automatically locate other source files if these files reside in a directory structure similar to the first selected alternative file.

The next time you start a debug session, the selected alternative file will be preloaded automatically.

Skip

C-SPY will assume that the source file is not available for this debug session.

If possible, don't show this dialog again

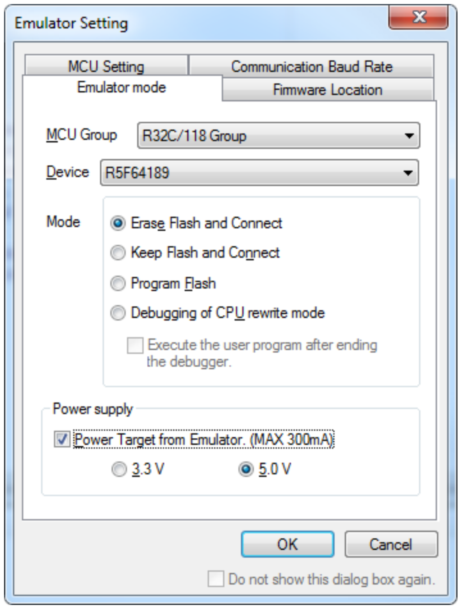
Instead of displaying the dialog box again for a missing source file, C-SPY will use the previously supplied response.

Related information

For related information, see *Starting a debug session with source files missing*, page 34.

Emulator Setting

The **Emulator Setting** dialog box is available from the **Emulator** menu.



MCU Group: R32C/118 Group
Device: R5F64189
Mode
[✓ Erase Flash and Connect]
Keep Flash and Connect
Program Flash
Debugging of CPU rewrite mode
Execute the user program after ending the debugger.
Power supply
[✓ Power Target from Emulator. (MAX 300mA)]
3.3 V [✓ 5.0 V]
Do not show this dialog box again.

Use this dialog box to configure the emulator debugger. For more information, see *Connecting to the target board, E8a*, page 39.

Requirements

A C-SPY E8a emulator.

MCU Group

Selects the MCU group to which your device belongs.

Device

Selects your device.

Erase Flash and Connect

Erases the flash memory data for the MCUs and simultaneously writes the E8a emulator firmware.

Keep Flash and Connect

Retains the flash memory data for the MCUs.

Note: The area for the E8a emulator firmware and the vector area used by the E8a emulator will change.

Program Flash

Writes your application to the flash memory. Debugging of the application is disabled. Select this mode when using the E8a emulator as a flash memory programmer.

Note: It is necessary to input the ID code of the flash memory to the target device.

Debugging of CPU rewrite mode

Erases the flash memory data for the MCUs and simultaneously writes the E8a emulator program, when starting the debugger. Select this setting when debugging a program that rewrites the CPU.

Note: In this mode, it is not possible to set PC breakpoints or to change the contents of the flash memory.

Execute the user program after ending the debugger

Executes your application when you stop the debugger. This option requires external power supply.

Note: This option is only available when **Program Flash** has been selected.

Power target from Emulator

Powers the target from the emulator. Select this option only if the target board is not connected to any power supply.

Do not show this dialog box again

If you select this option, the dialog box will not be displayed the next time the debugger is launched. This option is mirrored in the E8a Emulator menu command **Show Emulator Setting**. Use the menu command to display the dialog box again. See also *Maintaining the connection to the hardware*, page 42.

Use ECC for E2 Data Flash (MCU Setting)

Select this option if you want to use ECC in the E2 Data Flash. You can only make this setting when you start the debugger.

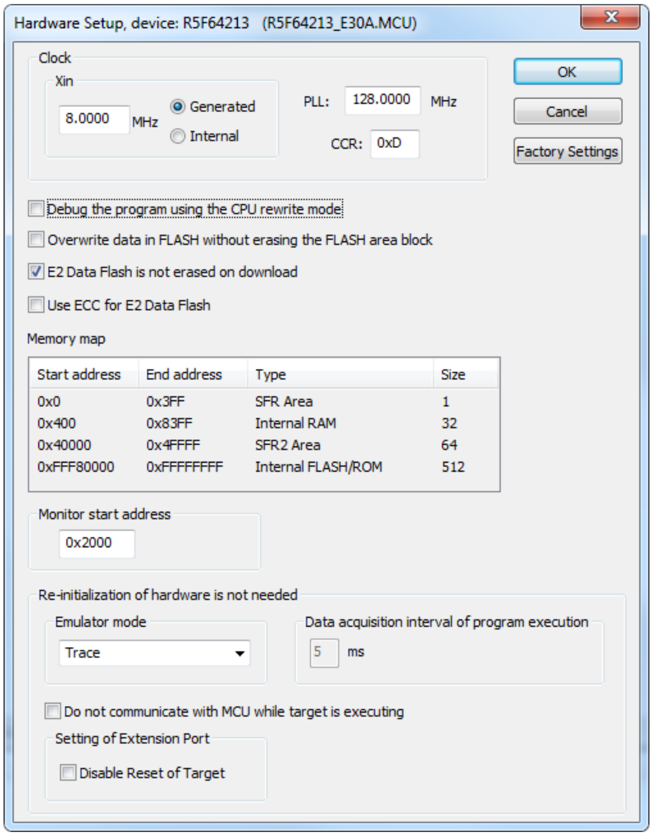
This option is invalid for an MCU that does not support the E2 Data Flash.

Communication Baud Rate

Selects the communication baud rate between the emulator and the MCU.

Hardware Setup

The **Hardware Setup** dialog box is available from the **Emulator** menu when you are using the C-SPY E30 emulator driver or the C-SPY E30A emulator driver.



Hardware Setup, device: R5F64213 (R5F64213_E30A.MCU)

Clock
Xin: 8.0000 MHz [✓ Generated] Internal
PLL: 128.0000 MHz
CCR: 0xD

☐ Debug the program using the CPU rewrite mode
☐ Overwrite data in FLASH without erasing the FLASH area block
☑ E2 Data Flash is not erased on download
☐ Use ECC for E2 Data Flash

Memory map

Start address	End address	Type	Size
0x0	0x3FF	SFR Area	1
0x400	0x83FF	Internal RAM	32
0x40000	0x4FFFF	SFR2 Area	64
0xFFF80000	0xFFFFFFFF	Internal FLASH/ROM	512

Monitor start address: 0x2000

Re-initialization of hardware is not needed
Emulator mode: Trace
Data acquisition interval of program execution: 5 ms
☐ Do not communicate with MCU while target is executing
Setting of Extension Port
☐ Disable Reset of Target

Figure 14: Emulator Hardware Setup dialog box

Note: This figure reflects the C-SPY E30A driver. Some of the options are not available when using the C-SPY E30 driver.

Use this dialog box to configure the emulator debugger.

The hardware setup is saved for each project and does not have to be set more than once. If you want to change the setup for a project, choose **Hardware Setup** from the **Emulator** menu.

Requirements

One of these alternatives:

- A C-SPY E30 emulator
- A C-SPY E30A emulator

Clock

Use the **Clock** options to set the CPU clock source:

Xin

Enter the frequency of the Xin clock and specify the clock source:
Generated uses a clock generated by the emulator
Internal uses the target oscillator circuit board

PLL

Enter the frequency of the internal phase-locked loop of the target microcomputer.

CCR

Enter the hexadecimal value of the internal clock control register of the target microcomputer.

Debug the program using the CPU rewrite mode

Select this option if you are debugging the target system using the CPU Rewrite Mode.

Note: When debugging in CPU Rewrite Mode, no software breakpoints can be set in the internal ROM area.

Overwrite data in FLASH without erasing the FLASH area block

If you select this option, writing to the flash memory area will merge the new data with the previous flash contents, and the addresses that are not being written to will keep their previous contents.

E2 Data Flash is not erased on download

Select this option if you want to preserve the contents of the E2 data flash memory.

Use ECC for E2 Data Flash

Select this option if you want to use ECC in the the E2 Data Flash. You can only make this setting when you start the debugger.

This option is invalid for an MCU that does not support the E2 Data Flash.

Memory map

The **Memory map** list shows the emulation memory areas. These areas are set up in the `*.mcu` files in the `..\r32c\config\Renesas\E30A\R32C\` directory, or for the E30 driver in the `..\r32c\config\Renesas\E30\R32C\` directory.

Monitor start address

Use this option to specify the start address of the internal RAM area that is used by the debug monitor.

Note: Approximately 1 Kbyte of the internal RAM will be used for other purposes, for instance to download your application. This means that you cannot specify a RAM area that overlaps the stack or an area that is accessed using DMA, but because the memory contents are saved before debugging starts it is not a problem otherwise.

Emulator mode

Use this option to specify debugging mode of the emulator debugger, choose between:

Trace	Enables the trace function.
RAM monitor	Enables the RAM-monitor function.
Time measurement	Enables the time measurement functions.

Note: The **Emulator mode** option is only available in the C-SPY E30A driver, not in the C-SPY E30 driver.

This table lists the main differences between the modes:

Debugging function	Trace mode: Trace priority	Trace mode: MCU Execution priority	RAM monitor mode	Time measurement mode
Break, execution PC	Yes	Yes	Yes	Yes
Break, data access	Yes	Yes	Yes	Yes
Break, address area	No	No	Yes	No
Break, data compare	No	Yes	No	No
Trace	Yes	Yes	No	No
Live watch	No	No	Yes	No

Table 4: Emulator mode debugging functions

Debugging function	Trace mode: Trace priority	Trace mode: MCU Execution priority	RAM monitor mode	Time measurement mode
Time measurement, execution time	No	No	No	Yes
Time measurement, interval time	No	No	No	Yes

Table 4: Emulator mode debugging functions (Continued)

Data acquisition interval of program execution

Use this option to specify the (byte or word) data acquisition interval by the RAM monitor function during execution. The interval can be 1–10 milliseconds.

This option can only be changed when the RAM monitor **Emulator mode** is selected.

Do not communicate with MCU while target is executing

When your application is executing, the emulator communicates with the MCU, for example, to check the operating status of the MCU or to collect trace data. The `PLL` clock divided by the base clock divider is used for communication between the emulator and the MCU. Therefore, when executing the `STOP` and `WAIT` instruction, or when temporarily switching to the `PLL` self-oscillation mode, use this option to avoid communication breakdown.

Disable Reset of Target

Disables the reset signal for the target system. This option is not available if the option board is not connected.

Setting Extension Port

Generates a high output to RSTMSK to disable reset of the target system.

Factory Settings

Click the **Factory Settings** button to restore the factory settings.

Executing your application

This chapter contains information about executing your application in C-SPY®. More specifically, this means:

- Introduction to application execution
- Reference information on application execution.

Introduction to application execution

This section covers these topics:

- Briefly about application execution
- Source and disassembly mode debugging
- Single stepping
- Stepping speed
- Running the application
- Highlighting
- Call stack information
- Terminal input and output
- Debug logging.

BRIEFLY ABOUT APPLICATION EXECUTION

C-SPY allows you to monitor and control the execution of your application. By single-stepping through it, and setting breakpoints, you can examine details about the application execution, for example the values of variables and registers. You can also use the call stack to step back and forth in the function call chain.

The terminal I/O and debug log features let you interact with your application.

You can find commands for execution on the **Debug** menu and on the toolbar.

SOURCE AND DISASSEMBLY MODE DEBUGGING

C-SPY allows you to switch between source mode and disassembly mode debugging as needed.

Source debugging provides the fastest and easiest way of developing your application, without having to worry about how the compiler or assembler has implemented the code. In the editor windows you can execute the application one statement at a time while monitoring the values of variables and data structures.

Disassembly mode debugging lets you focus on the critical sections of your application, and provides you with precise control of the application code. You can open a disassembly window which displays a mnemonic assembler listing of your application based on actual memory contents rather than source code, and lets you execute the application exactly one machine instruction at a time.

Regardless of which mode you are debugging in, you can display registers and memory, and change their contents.

SINGLE STEPPING

C-SPY allows more stepping precision than most other debuggers because it is not line-oriented but statement-oriented. The compiler generates detailed stepping information in the form of *step points* at each statement, and at each function call. That is, source code locations where you might consider whether to execute a step into or a step over command. Because the step points are located not only at each statement but also at each function call, the step functionality allows a finer granularity than just stepping on statements.

There are several factors that can slow down the stepping speed. If you find it too slow, see *Slow stepping speed*, page 287 for some tips.

The step commands

There are four step commands:

- **Step Into**
- **Step Over**
- **Next Statement**
- **Step Out**.

Using the **Autostep settings** dialog box, you can automate the single stepping. For more information, see *Autostep settings dialog box*, page 77.

Consider this example and assume that the previous step has taken you to the `f(i)` function call (highlighted):

```
extern int g(int);
int f(int n)
{
 value = g(n-1) + g(n-2) + g(n-3);
 return value;
}
int main()
{
  ...
  f(i);
  value ++;
}
```

Step Into

While stepping, you typically consider whether to step into a function and continue stepping inside the function or subroutine. The **Step Into** command takes you to the first step point within the subroutine `g(n-1)`:

```
extern int g(int);
int f(int n)
{
 value = g(n-1) + g(n-2) + g(n-3);
 return value;
}
```

The **Step Into** command executes to the next step point in the normal flow of control, regardless of whether it is in the same or another function.

Step Over

The **Step Over** command executes to the next step point in the same function, without stopping inside called functions. The command would take you to the `g(n-2)` function call, which is not a statement on its own but part of the same statement as `g(n-1)`. Thus, you can skip uninteresting calls which are parts of statements and instead focus on critical parts:

```
extern int g(int);
int f(int n)
{
 value = g(n-1) + g(n-2) + g(n-3);
 return value;
}
```

Next Statement

The **Next Statement** command executes directly to the next statement, in this case `return value`, allowing faster stepping:

```
extern int g(int);
int f(int n)
{
 value = g(n-1) + g(n-2) + g(n-3);
 return value;
}
```

Step Out

When inside the function, you can—if you wish—use the **Step Out** command to step out of it before it reaches the exit. This will take you directly to the statement immediately after the function call:

```
extern int g(int);
int f(int n)
{
 value = g(n-1) + g(n-2) g(n-3);
 return value;
}
int main()
{
  ...
  f(i);
  value ++;
}
```

The possibility of stepping into an individual function that is part of a more complex statement is particularly useful when you use C code containing many nested function calls. It is also very useful for C++, which tends to have many implicit function calls, such as constructors, destructors, assignment operators, and other user-defined operators.

This detailed stepping can in some circumstances be either invaluable or unnecessarily slow. For this reason, you can also step only on statements, which means faster stepping.

STEPPING SPEED

Stepping in C-SPY is normally performed using breakpoints. When performing a step command, a breakpoint is set on the next statement and the program executes until reaching this breakpoint. If you are debugging using a hardware debugger system, the number of hardware breakpoints—typically used for setting a stepping breakpoint, at least in code that is located in flash/ROM memory—is limited. If you for example, step into a C `switch` statement, breakpoints are set on each branch, and hence, this might consume several hardware breakpoints. If the number of available hardware breakpoints is exceeded, C-SPY switches into single stepping at assembly level, which can be very slow.

For this reason, it can be helpful to keep track of how many hardware breakpoints are used and make sure to some of them are left for stepping. For more information, see *Breakpoints in the C-SPY hardware drivers*, page 104 and *Breakpoint consumers*, page 105.

In addition to limited hardware breakpoints, these issues might also affect stepping speed:

- If Trace or Function profiling is enabled. This might slow down stepping because collected Trace data is processed after each step. Note that it is not sufficient to close the corresponding windows to disable Trace data collection. Instead, you must disable the **Enable/Disable** button in both the Trace and the Function profiling windows.
- If the Register window is open and displays SFR registers. This might slow down stepping because all registers in the selected register group must be read from the hardware after each step. To solve this, you can choose to view only a limited selection of SFR register; you can choose between two alternatives. Either type `#`*`SFR_name`* (where `#`*`SFR_name`* reflects the name of the SFR you want to monitor) in the Watch window, or create your own filter for displaying a limited group of SFRs in the Register window. See *Defining application-specific register groups*, page 133.
- If any of the Memory or Symbolic memory windows is open. This might slow down stepping because the visible memory must be read after each step.
- If any of the expression related windows such as Watch, Live Watch, Locals, Statics is open. This might slow down stepping speed because all these windows reads memory after each step.
- If the Stack window is open and especially if the option **Enable graphical stack display and stack usage tracking** option is enabled. To disable this option, choose **Toools>Options>Stack** and disable it.
- If a too slow communication speed has been set up between C-SPY and the target board/emulator you should consider to increase the speed, if possible.

RUNNING THE APPLICATION

Go

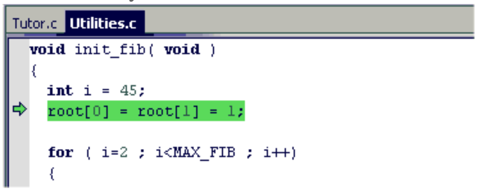
The **Go** command continues execution from the current position until a breakpoint or program exit is reached.

Run to Cursor

The **Run to Cursor** command executes to the position in the source code where you have placed the cursor. The **Run to Cursor** command also works in the Disassembly window and in the Call Stack window.

HIGHLIGHTING

At each stop, C-SPY highlights the corresponding C or C++ source or instruction with a green color, in the editor and the Disassembly window respectively. In addition, a green arrow appears in the editor window when you step on C or C++ source level, and in the Disassembly window when you step on disassembly level. This is determined by which of the windows is the active window. If none of the windows are active, it is determined by which of the windows was last active.



```
void init_fib( void )
{
  int i = 45;
  root[0] = root[1] = 1;

  for ( i=2 ; i<MAX_FIB ; i++)
  {
```

Figure 15: C-SPY highlighting source location

For simple statements without function calls, the whole statement is typically highlighted. When stopping at a statement with function calls, C-SPY highlights the first call because this illustrates more clearly what **Step Into** and **Step Over** would mean at that time.

Occasionally, you will notice that a statement in the source window is highlighted using a pale variant of the normal highlight color. This happens when the program counter is at an assembler instruction which is part of a source statement but not exactly at a step point. This is often the case when stepping in the Disassembly window. Only when the program counter is at the first instruction of the source statement, the ordinary highlight color is used.

CALL STACK INFORMATION

The compiler generates extensive backtrace information. This allows C-SPY to show, without any runtime penalty, the complete function call chain at any time.

Typically, this is useful for two purposes:

- Determining in what context the current function has been called
- Tracing the origin of incorrect values in variables and in parameters, thus locating the function in the call chain where the problem occurred.

The Call Stack window shows a list of function calls, with the current function at the top. When you inspect a function in the call chain, the contents of all affected windows are updated to display the state of that particular call frame. This includes the editor, Locals, Register, Watch and Disassembly windows. A function would normally not make use of all registers, so these registers might have undefined states and be displayed as dashes (---).

In the editor and Disassembly windows, a green highlight indicates the topmost, or current, call frame; a yellow highlight is used when inspecting other frames.

For your convenience, it is possible to select a function in the call stack and click the **Run to Cursor** command to execute to that function.

Assembler source code does not automatically contain any backtrace information. To see the call chain also for your assembler modules, you can add the appropriate `CFI` assembler directives to the assembler source code. For further information, see the *R32C IAR Assembler Reference Guide*.

TERMINAL INPUT AND OUTPUT

Sometimes you might have to debug constructions in your application that use `stdin` and `stdout` without an actual hardware device for input and output. The Terminal I/O window lets you enter input to your application, and display output from it. You can also direct terminal I/O to a file, using the **Terminal I/O Log Files** dialog box.

This facility is useful in two different contexts:

- If your application uses `stdin` and `stdout`
- For producing debug trace printouts.

For more information, see *Terminal I/O window*, page 72 and *Terminal I/O Log File dialog box*, page 74.

DEBUG LOGGING

The Debug Log window displays debugger output, such as diagnostic messages, macro-generated output, event log messages, and information about trace.

It can sometimes be convenient to log the information to a file where you can easily inspect it. The two main advantages are:

- The file can be opened in another tool, for instance an editor, so you can navigate and search within the file for particularly interesting parts
- The file provides history about how you have controlled the execution, for instance, which breakpoints that have been triggered etc.

Reference information on application execution

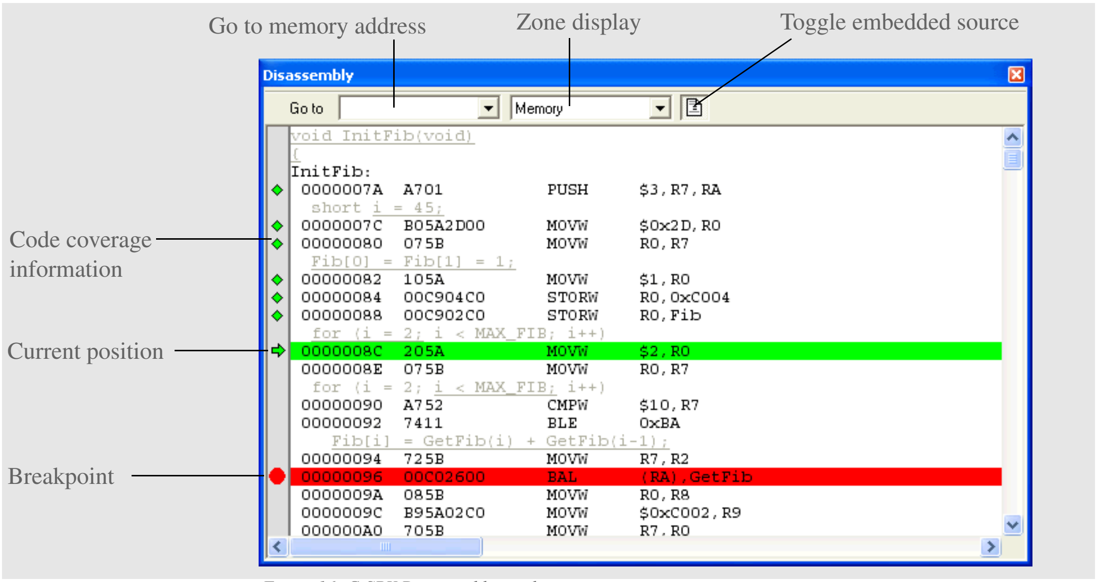
This section gives reference information about these windows and dialog boxes:

- *Disassembly window*, page 66
- *Call Stack window*, page 71
- *Terminal I/O window*, page 72
- *Terminal I/O Log File dialog box*, page 74
- *Debug Log window*, page 74
- *Log File dialog box*, page 75
- *Report Assert dialog box*, page 76
- *Autostep settings dialog box*, page 77

See also Terminal I/O options in *IDE Project Management and Building Guide*.

Disassembly window

The C-SPY Disassembly window is available from the **View** menu.



```
void InitFib(void)
{
InitFib:
 0000007A  A701          PUSH     $3,R7,RA
  short i = 45;
 0000007C  B05A2D00      MOVW     $0x2D,R0
 00000080  075B          MOVW     R0,R7
  Fib[0] = Fib[1] = 1;
 00000082  105A          MOVW     $1,R0
 00000084  00C904C0      STORW    R0,0xC004
 00000088  00C902C0      STORW    R0,Fib
  for (i = 2; i < MAX_FIB; i++)
 0000008C  205A          MOVW     $2,R0
 0000008E  075B          MOVW     R0,R7
  for (i = 2; i < MAX_FIB; i++)
 00000090  A752          CMPW     $10,R7
 00000092  7411          BLE      0xBA
    Fib[i] = GetFib(i) + GetFib(i-1);
 00000094  725B          MOVW     R7,R2
 00000096  00C02600      BAL      (RA),GetFib
 0000009A  085B          MOVW     R0,R8
 0000009C  B95A02C0      MOVW     $0xC002,R9
 000000A0  705B          MOVW     R7,R0
```

Figure 16: C-SPY Disassembly window

This window shows the application being debugged as disassembled application code.

To change the default color of the source code in the Disassembly window:

1. Choose **Tools>Options>Debugger**.
2. Set the default color using the **Source code coloring in disassembly window** option.

To view the corresponding assembler code for a function, you can select it in the editor window and drag it to the Disassembly window.

Requirements

None; this window is always available.

Toolbar

The toolbar contains:

Go to

The memory location or symbol you want to view.

Zone

Selects a memory zone, see *C-SPY memory zones*, page 130.

Toggle Mixed-Mode

Toggles between displaying only disassembled code or disassembled code together with the corresponding source code. Source code requires that the corresponding source file has been compiled with debug information

Display area

The display area shows the disassembled application code.

This area contains these graphic elements:

Green highlight	Indicates the current position, that is the next assembler instruction to be executed. To move the cursor to any line in the Disassembly window, click the line. Alternatively, move the cursor using the navigation keys.
Yellow highlight	Indicates a position other than the current position, such as when navigating between frames in the Call Stack window or between items in the Trace window.
Red dot	Indicates a breakpoint. Double-click in the gray left-side margin of the window to set a breakpoint. For more information, see *Breakpoints*, page 101.
Green diamond	Indicates code that has been executed—that is, code coverage.

If instruction profiling has been enabled from the context menu, an extra column in the left-side margin appears with information about how many times each instruction has been executed.

Context menu

This context menu is available:

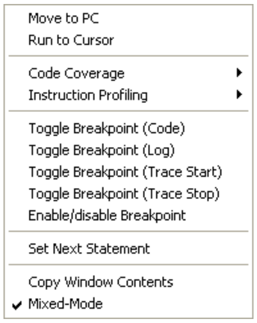


Figure 17: Disassembly window context menu

Note: The contents of this menu are dynamic, which means it might look different depending on your product package.

These commands are available:

Move to PC

Displays code at the current program counter location.

Run to Cursor

Executes the application from the current position up to the line containing the cursor.

Code Coverage

Displays a submenu that provides commands for controlling code coverage. This command is only enabled if the driver you are using supports it.

Enable	Toggles code coverage on or off.
Show	Toggles the display of code coverage on or off. Executed code is indicated by a green diamond.
Clear	Clears all code coverage information.

Instruction Profiling

Displays a submenu that provides commands for controlling instruction profiling. This command is only enabled if the driver you are using supports it.

Enable	Toggles instruction profiling on or off.
Show	Toggles the display of instruction profiling on or off. For each instruction, the left-side margin displays how many times the instruction has been executed.
Clear	Clears all instruction profiling information.

Toggle Breakpoint (Code)

Toggles a code breakpoint. Assembler instructions and any corresponding label at which code breakpoints have been set are highlighted in red. For more information, see *Code breakpoints dialog box*, page 115.

Toggle Breakpoint (Address)

Toggles an execution address breakpoint. For more information, see *Execution Address Breakpoint dialog box*, page 116.

Toggle Breakpoint (Software)

Toggles a software breakpoint. For more information, see *Software Breakpoint dialog box*, page 118.

Toggle Breakpoint (Log)

Toggles a log breakpoint for trace printouts. Assembler instructions at which log breakpoints have been set are highlighted in red. For more information, see *Log breakpoints dialog box*, page 119.

Toggle Breakpoint (Trace Start)

Toggles a Trace Start breakpoint. When the breakpoint is triggered, the trace data collection starts. Note that this menu command is only available if the C-SPY driver you are using supports trace. For more information, see *Trace Start breakpoints dialog box*, page 176.

Toggle Breakpoint (Trace Stop)

Toggles a Trace Stop breakpoint. When the breakpoint is triggered, the trace data collection stops. Note that this menu command is only available if the C-SPY driver you are using supports trace. For more information, see *Trace Stop breakpoints dialog box*, page 177.

Enable/Disable Breakpoint

Enables and Disables a breakpoint. If there is more than one breakpoint at a specific line, all those breakpoints are affected by the **Enable/Disable** command.

Edit Breakpoint

Displays the breakpoint dialog box to let you edit the currently selected breakpoint. If there is more than one breakpoint on the selected line, a submenu is displayed that lists all available breakpoints on that line.

Set Next Statement

Sets the program counter to the address of the instruction at the insertion point.

Copy Window Contents

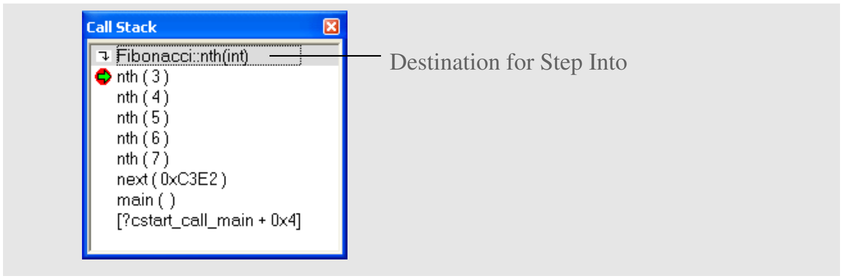
Copies the selected contents of the Disassembly window to the clipboard.

Mixed-Mode

Toggles between showing only disassembled code or disassembled code together with the corresponding source code. Source code requires that the corresponding source file has been compiled with debug information.

Call Stack window

The Call stack window is available from the **View** menu.



Call Stack

Fibonacci::nth(int)
nth (3)
nth (4)
nth (5)
nth (6)
nth (7)
next (0xC3E2)
main ( )
[?cstart_call_main + 0x4]

Figure 18: Call Stack window

This window displays the C function call stack with the current function at the top. To inspect a function call, double-click it. C-SPY now focuses on that call frame instead.

If the next **Step Into** command would step to a function call, the name of the function is displayed in the grey bar at the top of the window. This is especially useful for implicit function calls, such as C++ constructors, destructors, and operators.

Requirements

None; this window is always available.

Display area

Provided that the command **Show Arguments** is enabled, each entry in the display area has the format:

`function(values)`

where `(values)` is a list of the current value of the parameters, or empty if the function does not take any parameters.

Context menu

This context menu is available:

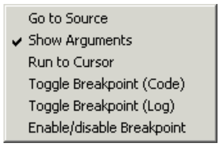


Figure 19: Call Stack window context menu

These commands are available:

Go to Source

Displays the selected function in the Disassembly or editor windows.

Show Arguments

Shows function arguments.

Run to Cursor

Executes until return to the function selected in the call stack.

Toggle Breakpoint (Code)

Toggles a code breakpoint.

Toggle Breakpoint (Log)

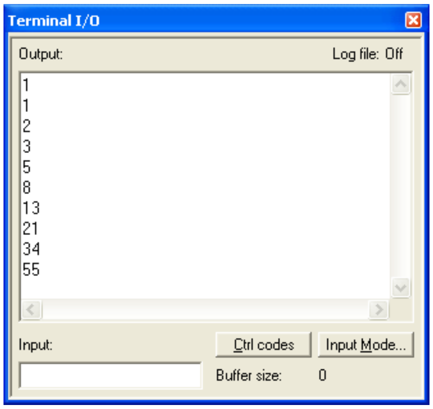
Toggles a log breakpoint.

Enable/Disable Breakpoint

Enables or disables the selected breakpoint

Terminal I/O window

The Terminal I/O window is available from the **View** menu.



```
1
1
2
3
5
8
13
21
34
55
```

Figure 20: Terminal I/O window

Use this window to enter input to your application, and display output from it.

To use this window, you must:

- Link your application with the option **With I/O emulation modules**.

C-SPY will then direct `stdin`, `stdout` and `stderr` to this window. If the Terminal I/O window is closed, C-SPY will open it automatically when input is required, but not for output.

Requirements

None; this window is always available.

Input

Type the text that you want to input to your application.

Ctrl codes

Opens a menu for input of special characters, such as `EOF` (end of file) and `NUL`.

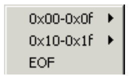
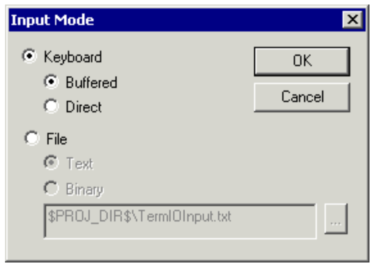


Figure 21: Ctrl codes menu

Input Mode

Opens the **Input Mode** dialog box where you choose whether to input data from the keyboard or from a file.



$PROJ_DIR$\TermIOInput.txt

Figure 22: Input Mode dialog box

For reference information about the options available in this dialog box, see Terminal I/O options in *IDE Project Management and Building Guide*.

Terminal I/O Log File dialog box

The **Terminal I/O Log File** dialog box is available by choosing **Debug>Logging>Set Terminal I/O Log File**.

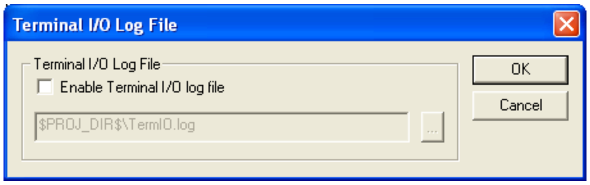


Figure 23: Terminal I/O Log File dialog box

Use this dialog box to select a destination log file for terminal I/O from C-SPY.

Requirements

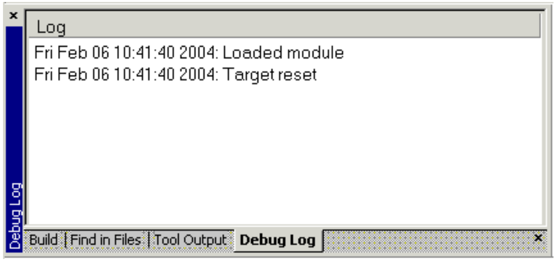
None; this dialog box is always available.

Terminal IO Log Files

Controls the logging of terminal I/O. To enable logging of terminal I/O to a file, select **Enable Terminal IO log file** and specify a filename. The default filename extension is `log`. A browse button is available for your convenience.

Debug Log window

The Debug Log window is available by choosing **View>Messages**.



```
Fri Feb 06 10:41:40 2004: Loaded module
Fri Feb 06 10:41:40 2004: Target reset
```

Figure 24: Debug Log window (message window)

This window displays debugger output, such as diagnostic messages, macro-generated output, event log messages, and information about trace. This output is only available during a debug session. When opened, this window is, by default, grouped together with the other message windows, see *IDE Project Management and Building Guide*.

Double-click any rows in one of the following formats to display the corresponding source code in the editor window:

```
<path> (<row>):<message>
<path> (<row>,<column>):<message>
```

Requirements

None; this window is always available.

Context menu

This context menu is available:

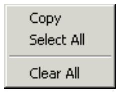


Figure 25: Debug Log window context menu

These commands are available:

Copy

Copies the contents of the window.

Select All

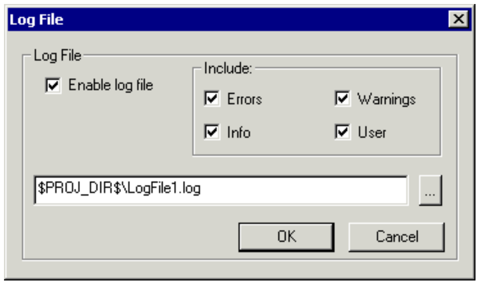
Selects the contents of the window.

Clear All

Clears the contents of the window.

Log File dialog box

The **Log File** dialog box is available by choosing **Debug>Logging>Set Log File**.



$PROJ_DIR$\LogFile1.log

Figure 26: Log File dialog box

Use this dialog box to log output from C-SPY to a file.

Requirements

None; this dialog box is always available.

Enable Log file

Enables or disables logging to the file.

Include

The information printed in the file is, by default, the same as the information listed in the Log window. Use the browse button, to override the default file and location of the log file (the default filename extension is `log`). To change the information logged, choose between:

Errors

C-SPY has failed to perform an operation.

Warnings

An error or omission of concern.

Info

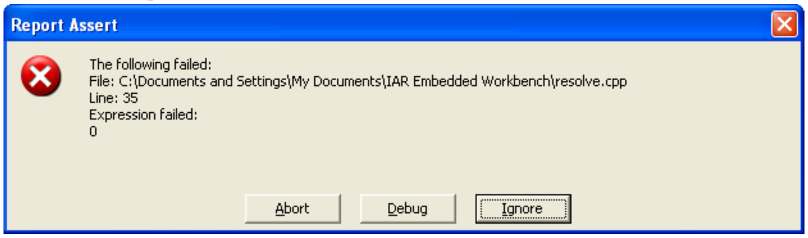
Progress information about actions C-SPY has performed.

User

Messages from C-SPY macros, that is, your messages using the `__message` statement.

Report Assert dialog box

The **Report Assert dialog box** appears if you have a call to the `assert` function in your application source code, and the assert condition is false. In this dialog box you can choose how to proceed.



```
The following failed:
File: C:\Documents and Settings\My Documents\IAR Embedded Workbench\resolve.cpp
Line: 35
Expression failed:
0
```

Figure 27: Report Assert dialog box

Abort

The application stops executing and the runtime library function `abort`, which is part of your application on the target system, will be called. This means that the application itself terminates its execution.

Debug

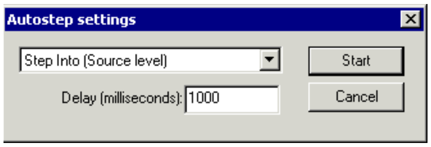
C-SPY stops the execution of the application and returns control to you.

Ignore

The assertion is ignored and the application continues to execute.

Autostep settings dialog box

The **Autostep settings** dialog box is available from the **Debug** menu.



Step Into (Source level)

Delay (milliseconds): 1000

Figure 28: Autostep settings dialog box

Use this dialog box to customize autostepping.

The drop-down menu lists the available step commands.

Requirements

None; this dialog box is always available.

Delay

Specify the delay between each step in milliseconds.

Variables and expressions

This chapter describes how variables and expressions can be used in C-SPY®. More specifically, this means:

- Introduction to working with variables and expressions
- Working with variables and expressions
- Reference information on working with variables and expressions.

Introduction to working with variables and expressions

This section covers these topics:

- Briefly about working with variables and expressions
- C-SPY expressions
- Limitations on variable information.

BRIEFLY ABOUT WORKING WITH VARIABLES AND EXPRESSIONS

There are several methods for looking at variables and calculating their values:

- Tooltip watch—in the editor window—provides the simplest way of viewing the value of a variable or more complex expressions. Just point at the variable with the mouse pointer. The value is displayed next to the variable.
- The Auto window displays a useful selection of variables and expressions in, or near, the current statement. The window is automatically updated when execution stops.
- The Locals window displays the local variables, that is, auto variables and function parameters for the active function. The window is automatically updated when execution stops.
- The Watch window allows you to monitor the values of C-SPY expressions and variables. The window is automatically updated when execution stops.
- The Live Watch window repeatedly samples and displays the values of expressions while your application is executing. Variables in the expressions must be statically located, such as global variables.
- The Statics window displays the values of variables with static storage duration. The window is automatically updated when execution stops.

- The Quick Watch window gives you precise control over when to evaluate an expression.
- The Symbols window displays all symbols with a static location, that is, C/C++ functions, assembler labels, and variables with static storage duration, including symbols from the runtime library.
- The Data Log window and the Data Log Summary window display logs of accesses up to four different memory locations or areas you choose by setting Data Log breakpoints. Data logging can help you locate frequently accessed data. You can then consider whether you should place that data in more efficient memory.
- The Trace-related windows let you inspect the program flow up to a specific state. For more information, see *Trace*, page 159.

C-SPY EXPRESSIONS

C-SPY expressions can include any type of C expression, except for calls to functions. The following types of symbols can be used in expressions:

- C/C++ symbols
- Assembler symbols (register names and assembler labels)
- C-SPY macro functions
- C-SPY macro variables.

Expressions that are built with these types of symbols are called C-SPY expressions and there are several methods for monitoring these in C-SPY. Examples of valid C-SPY expressions are:

```
i + j
i = 42
myVar = cVar
cVar = myVar + 2
#asm_label
#R2
#PC
my_macro_func(19)
```

If you have a static variable with the same name declared in several different functions, use the notation `*function*::*variable*` to specify which variable to monitor.

C/C++ symbols

C symbols are symbols that you have defined in the C source code of your application, for instance variables, constants, and functions (functions can be used as symbols but cannot be executed). C symbols can be referenced by their names. Note that C++ symbols might implicitly contain function calls which are not allowed in C-SPY symbols and expressions.

Assembler symbols

Assembler symbols can be assembler labels or registers, for example the program counter, the stack pointer, or other CPU registers. If a device description file is used, all memory-mapped peripheral units, such as I/O ports, can also be used as assembler symbols in the same way as the CPU registers. See *Modifying a device description file*, page 36.

Assembler symbols can be used in C-SPY expressions if they are prefixed by `#`.

Example	What it does
`#PC++`	Increments the value of the program counter.
`myVar = #SP`	Assigns the current value of the stack pointer register to your C-SPY variable.
`myVar = #label`	Sets `myVar` to the value of an integer at the address of `label`.
`myptr = #label7`	Sets `myptr` to an `int *` pointer pointing at `label7`.

Table 5: C-SPY assembler symbols expressions

In case of a name conflict between a hardware register and an assembler label, hardware registers have a higher precedence. To refer to an assembler label in such a case, you must enclose the label in back quotes ` (ASCII character 0x60). For example:

Example	What it does
`#PC`	Refers to the program counter.
``#`PC` ``	Refers to the assembler label `PC`.

Table 6: Handling name conflicts between hardware registers and assembler labels

Which processor-specific symbols are available by default can be seen in the Register window, using the `CPU Registers` register group. See *Register window*, page 148.

C-SPY macro functions

Macro functions consist of C-SPY macro variable definitions and macro statements which are executed when the macro is called.

For information about C-SPY macro functions and how to use them, see *Briefly about the macro language*, page 217.

C-SPY macro variables

Macro variables are defined and allocated outside your application, and can be used in a C-SPY expression. In case of a name conflict between a C symbol and a C-SPY macro variable, the C-SPY macro variable will have a higher precedence than the C variable. Assignments to a macro variable assign both its value and type.

For information about C-SPY macro variables and how to use them, see *Reference information on the macro language*, page 223.

Using sizeof

According to standard C, there are two syntactical forms of `sizeof`:

```
sizeof(type)
sizeof expr
```

The former is for types and the latter for expressions.

Note: In C-SPY, do not use parentheses around an expression when you use the `sizeof` operator. For example, use `sizeof x+2` instead of `sizeof (x+2)`.

LIMITATIONS ON VARIABLE INFORMATION

The value of a C variable is valid only on step points, that is, the first instruction of a statement and on function calls. This is indicated in the editor window with a bright green highlight color. In practice, the value of the variable is accessible and correct more often than that.

When the program counter is inside a statement, but not at a step point, the statement or part of the statement is highlighted with a pale variant of the ordinary highlight color.

Effects of optimizations

The compiler is free to optimize the application software as much as possible, as long as the expected behavior remains. The optimization can affect the code so that debugging might be more difficult because it will be less clear how the generated code relates to the source code. Typically, using a high optimization level can affect the code in a way that will not allow you to view a value of a variable as expected.

Consider this example:

```
myFunction()
{
 int i = 42;
 ...
 x = computer(i); /* Here, the value of i is known to C-SPY */
 ...
}
```

From the point where the variable `i` is declared until it is actually used, the compiler does not need to waste stack or register space on it. The compiler can optimize the code, which means that C-SPY will not be able to display the value until it is actually used. If you try to view the value of a variable that is temporarily unavailable, C-SPY will display the text:

```
Unavailable
```

If you need full information about values of variables during your debugging session, you should make sure to use the lowest optimization level during compilation, that is, **None**.

Working with variables and expressions

This section describes various tasks related to working with variables and expressions.

More specifically, you will get information about:

- Using the windows related to variables and expressions
- Viewing assembler variables
- Getting started using data logging

USING THE WINDOWS RELATED TO VARIABLES AND EXPRESSIONS

Where applicable, you can add, modify, and remove expressions, and change the display format in the windows related to variables and expressions.

To add a value you can also click in the dotted rectangle and type the expression you want to examine. To modify the value of an expression, click the **Value** field and modify its content. To remove an expression, select it and press the Delete key.

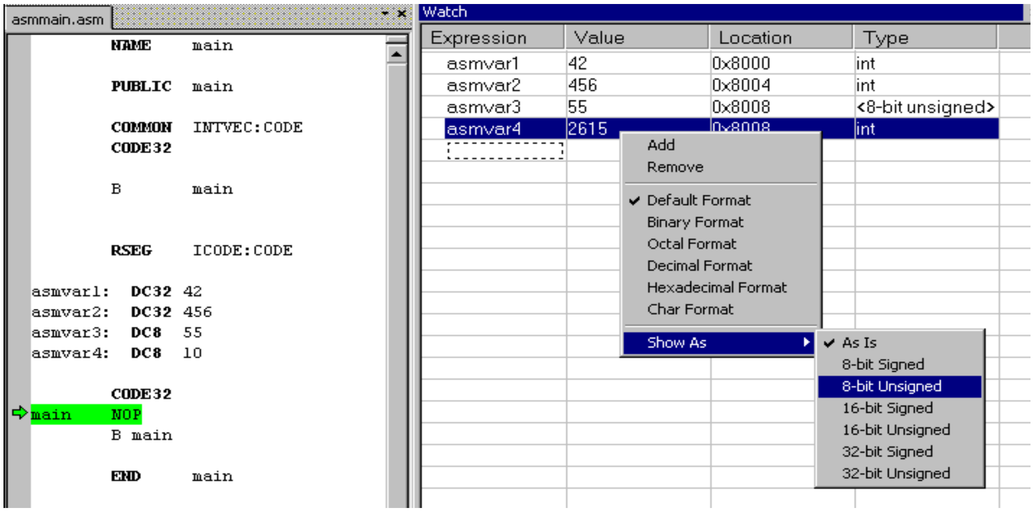
For text that is too wide to fit in a column—in any of the these windows, except the Trace window—and thus is truncated, just point at the text with the mouse pointer and tooltip information is displayed.

Right-click in any of the windows to access the context menu which contains additional commands. Convenient drag-and-drop between windows is supported, except for in the Locals window, Data logging windows, and the Quick Watch window where it is not relevant.

VIEWING ASSEMBLER VARIABLES

An assembler label does not convey any type information at all, which means C-SPY cannot easily display data located at that label without getting extra information. To view data conveniently, C-SPY by default treats all data located at assembler labels as variables of type `int`. However, in the Watch, Quick Watch, and Live Watch windows, you can select a different interpretation to better suit the declaration of the variables.

In this figure, you can see four variables in the Watch window and their corresponding declarations in the assembler source file to the left:



```
        NAME    main

        PUBLIC  main

        COMMON  INTVEC:CODE
        CODE32

        B       main


        RSEG    ICODE:CODE

asmvar1:  DC32  42
asmvar2:  DC32  456
asmvar3:  DC8   55
asmvar4:  DC8   10

        CODE32
main    NOP
        B main

        END     main
```

Expression	Value	Location	Type
asmvar1	42	0x8000	int
asmvar2	456	0x8004	int
asmvar3	55	0x8008	<8-bit unsigned>
asmvar4	2615	0x8008	int

Figure 29: Viewing assembler variables in the Watch window

Note that `asmvar4` is displayed as an `int`, although the original assembler declaration probably intended for it to be a single byte quantity. From the context menu you can make C-SPY display the variable as, for example, an 8-bit unsigned variable. This has already been specified for the `asmvar3` variable.

GETTING STARTED USING DATA LOGGING

1 In the Breakpoints or Memory window, right-click and choose **New Breakpoints>Data Log** to open the breakpoints dialog box. Set a Data Log breakpoint on the data you want to collect log information for. You can set up to four Data Log breakpoints.

2 Choose *C-SPY driver*>**Data Log** to open the Data Log window. Optionally, you can also choose:

- *C-SPY driver*>**Data Log Summary** to open the Data Log Summary window

- *C-SPY driver*>**Timeline** to open the Timeline window to view the Data Log graph.

3 From the context menu, available in the Data Log window, choose **Enable** to enable the logging.

4 Start executing your application program to collect the log information.

5 To view the data log information, look in any of the Data Log, Data Log Summary, or the Data graph in the Timeline window.

6 If you want to save the log or summary to a file, choose **Save to log file** from the context menu in the window in question.

7 To disable data and interrupt logging, choose **Disable** from the context menu in each window where you have enabled it.

Reference information on working with variables and expressions

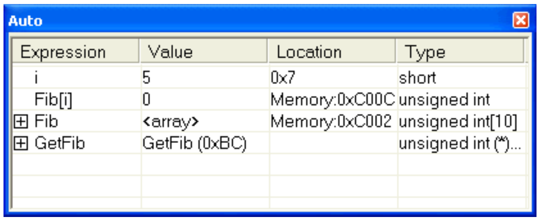
This section gives reference information about these windows and dialog boxes:

- *Auto window*, page 86
- *Locals window*, page 86
- *Watch window*, page 87
- *Live Watch window*, page 89
- *Statics window*, page 90
- *Quick Watch window*, page 92
- *Symbols window*, page 93
- *Resolve Symbol Ambiguity dialog box*, page 95
- *Data Log window*, page 96
- *Data Log Summary window*, page 98

For trace-related reference information, see *Reference information on trace*, page 162.

Auto window

The Auto window is available from the **View** menu.



Expression	Value	Location	Type
i	5	0x7	short
Fib[i]	0	Memory:0xC00C	unsigned int
⊞ Fib	<array>	Memory:0xC002	unsigned int[10]
⊞ GetFib	GetFib (0xBC)		unsigned int (*)...

Figure 30: Auto window

This window displays a useful selection of variables and expressions in, or near, the current statement. Every time execution in C-SPY stops, the values in the Auto window are recalculated. Values that have changed since the last stop are highlighted in red.

Requirements

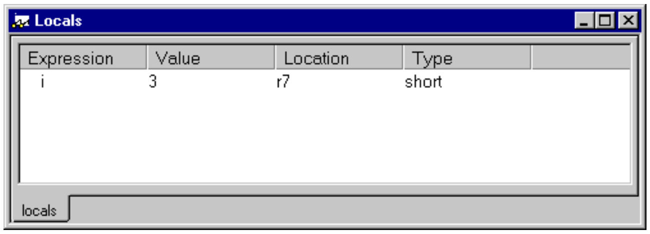
None; this window is always available.

Context menu

For more information about the context menu, see *Watch window*, page 87.

Locals window

The Locals window is available from the **View** menu.



Expression	Value	Location	Type
i	3	r7	short

Figure 31: Locals window

This window displays the local variables and parameters for the current function. Every time execution in C-SPY stops, the values in the Locals window are recalculated. Values that have changed since the last stop are highlighted in red.

Requirements

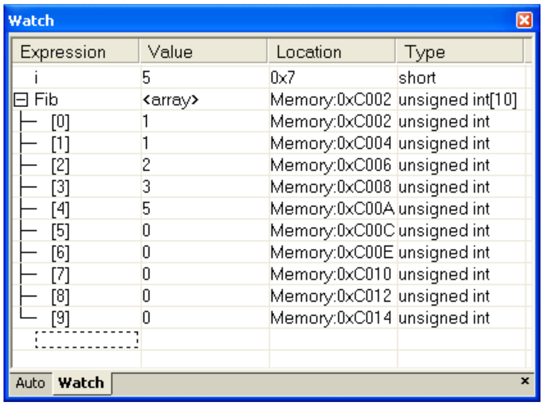
None; this window is always available.

Context menu

For more information about the context menu, see *Watch window*, page 87.

Watch window

The Watch window is available from the **View** menu.



Expression	Value	Location	Type
i	5	0x7	short
⊟ Fib	<array>	Memory:0xC002	unsigned int[10]
[0]	1	Memory:0xC002	unsigned int
[1]	1	Memory:0xC004	unsigned int
[2]	2	Memory:0xC006	unsigned int
[3]	3	Memory:0xC008	unsigned int
[4]	5	Memory:0xC00A	unsigned int
[5]	0	Memory:0xC00C	unsigned int
[6]	0	Memory:0xC00E	unsigned int
[7]	0	Memory:0xC010	unsigned int
[8]	0	Memory:0xC012	unsigned int
[9]	0	Memory:0xC014	unsigned int

Figure 32: Watch window

Use this window to monitor the values of C-SPY expressions or variables. You can open up to four instances of this window, where you can view, add, modify, and remove expressions. Tree structures of arrays, structs, and unions are expandable, which means that you can study each item of these.

Every time execution in C-SPY stops, the values in the Watch window are recalculated. Values that have changed since the last stop are highlighted in red.

Requirements

None; this window is always available.

Context menu

This context menu is available:

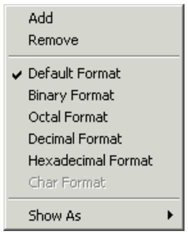


Figure 33: Watch window context menu

These commands are available:

Add

Adds an expression.

Remove

Removes the selected expression.

Default Format,
Binary Format,
Octal Format,
Decimal Format,
Hexadecimal Format,
Char Format

Changes the display format of expressions. The display format setting affects different types of expressions in different ways. Your selection of display format is saved between debug sessions. These commands are available if a selected line in the window contains a variable.

The display format setting affects different types of expressions in these ways:

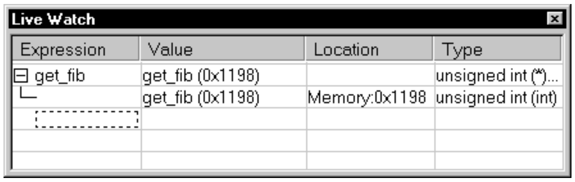
Variables	The display setting affects only the selected variable, not other variables.
Array elements	The display setting affects the complete array, that is, the same display format is used for each array element.
Structure fields	All elements with the same definition—the same field name and C declaration type—are affected by the display setting.

Show As

Displays a submenu that provides commands for changing the default type interpretation of variables. The commands on this submenu are mainly useful for assembler variables—data at assembler labels—because these are, by default, displayed as integers. For more information, see *Viewing assembler variables*, page 84.

Live Watch window

The Live Watch window is available from the **View** menu.



Expression	Value	Location	Type
⊟ get_fib	get_fib (0x1198)		unsigned int (*)...
└	get_fib (0x1198)	Memory:0x1198	unsigned int (int)

Figure 34: Live Watch window

This window repeatedly samples and displays the value of expressions while your application is executing. Variables in the expressions must be statically located, such as global variables.

This window can only be used for hardware target systems supporting this feature.

Requirements

None; this window is always available.

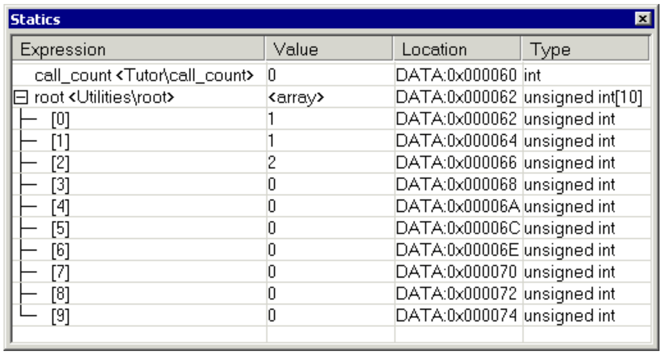
Context menu

For more information about the context menu, see *Watch window*, page 87.

In addition, the menu contains the **Options** command, which opens the **Debugger** dialog box where you can set the **Update interval** option. The default value of this option is 1000 milliseconds, which means the **Live Watch** window will be updated once every second during program execution.

Statics window

The Statics window is available from the **View** menu.



Expression	Value	Location	Type
call_count <Tutor\call_count>	0	DATA:0x000060	int
⊟ root <Utilities\root>	<array>	DATA:0x000062	unsigned int[10]
├─ [0]	1	DATA:0x000062	unsigned int
├─ [1]	1	DATA:0x000064	unsigned int
├─ [2]	2	DATA:0x000066	unsigned int
├─ [3]	0	DATA:0x000068	unsigned int
├─ [4]	0	DATA:0x00006A	unsigned int
├─ [5]	0	DATA:0x00006C	unsigned int
├─ [6]	0	DATA:0x00006E	unsigned int
├─ [7]	0	DATA:0x000070	unsigned int
├─ [8]	0	DATA:0x000072	unsigned int
└─ [9]	0	DATA:0x000074	unsigned int

Figure 35: Statics window

This window displays the values of variables with static storage duration that you have selected. Typically, that is variables with file scope but it can also be static variables in functions and classes. Note that `volatile` declared variables with static storage duration will not be displayed.

Every time execution in C-SPY stops, the values in the Statics window are recalculated. Values that have changed since the last stop are highlighted in red.

To select variables to monitor:

1. In the window, right-click and choose **Select statics** from the context menu. The window now lists all variables with static storage duration.
2. Either individually select the variables you want to be displayed, or choose **Select All** or **Deselect All** from the context menu.
3. When you have made your selections, choose **Select statics** from the context menu to toggle back to the normal display mode.

Requirements

None; this window is always available.

Display area

This area contains these columns:

Expression

The name of the variable. The base name of the variable is followed by the full name, which includes module, class, or function scope. This column is not editable.

Value

The value of the variable. Values that have changed are highlighted in red.

Dragging text or a variable from another window and dropping it on the **Value** column will assign a new value to the variable in that row.

This column is editable.

Location

The location in memory where this variable is stored.

Type

The data type of the variable.

Context menu

This context menu is available:

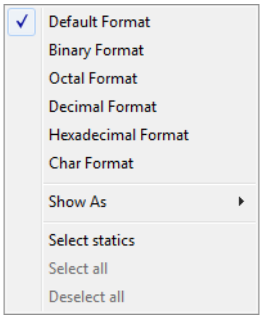


Figure 36: Statics window context menu

These commands are available:

Default Format,
Binary Format,
Octal Format,
Decimal Format,
Hexadecimal Format,
Char Format

Changes the display format of expressions. The display format setting affects different types of expressions in different ways. Your selection of display format is saved between debug sessions. These commands are available if a selected line in the window contains a variable.

The display format setting affects different types of expressions in these ways:

Variables	The display setting affects only the selected variable, not other variables.
Array elements	The display setting affects the complete array, that is, the same display format is used for each array element.
Structure fields	All elements with the same definition—the same field name and C declaration type—are affected by the display setting.

Select Statics

Lists all variables with static storage duration. Select the variables you want to be monitored. When you have made your selections, select this menu command again to toggle back to normal display mode.

Select all

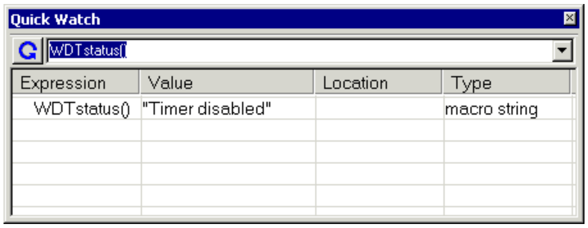
Selects all variables.

Deselect all

Deselects all variables.

Quick Watch window

The Quick Watch window is available from the **View** menu and from the context menu in the editor window.



Expression	Value	Location	Type
WDTstatus()	"Timer disabled"		macro string

Figure 37: Quick Watch window

Use this window to watch the value of a variable or expression and evaluate expressions at a specific point in time.

In contrast to the Watch window, the Quick Watch window gives you precise control over when to evaluate the expression. For single variables this might not be necessary, but for expressions with possible side effects, such as assignments and C-SPY macro functions, it allows you to perform evaluations under controlled conditions.

To evaluate an expression:

1. In the editor window, right-click on the expression you want to examine and choose Quick Watch from the context menu that appears.
2. The expression will automatically appear in the Quick Watch window.

   Alternatively:
3. In the Quick Watch window, type the expression you want to examine in the **Expressions** text box.
4. Click the **Recalculate** button to calculate the value of the expression.

   For an example, see *Executing macros using Quick Watch*, page 221.

Requirements

None; this window is always available.

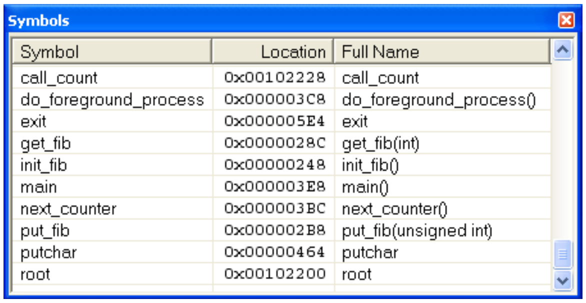
Context menu

For more information about the context menu, see *Watch window*, page 87.

In addition, the menu contains the **Add to Watch window** command, which adds the selected expression to the Watch window.

Symbols window

The Symbols window is available from the **View** menu.



Symbol	Location	Full Name
call_count	0x00102228	call_count
do_foreground_process	0x000003C8	do_foreground_process()
exit	0x000005E4	exit
get_fib	0x0000028C	get_fib(int)
init_fib	0x00000248	init_fib()
main	0x000003E8	main()
next_counter	0x000003BC	next_counter()
put_fib	0x000002B8	put_fib(unsigned int)
putchar	0x00000464	putchar
root	0x00102200	root

Figure 38: Symbols window

This window displays all symbols with a static location, that is, C/C++ functions, assembler labels, and variables with static storage duration, including symbols from the runtime library.

Requirements

None; this window is always available.

Display area

This area contains these columns:

Symbol

The symbol name.

Location

The memory address.

Full name

The symbol name; often the same as the contents of the Symbol column but differs for example for C++ member functions.

Click the column headers to sort the list by symbol name, location, or full name.

Context menu

This context menu is available:

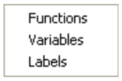


Figure 39: Symbols window context menu

These commands are available:

Functions

Toggles the display of function symbols on or off in the list.

Variables

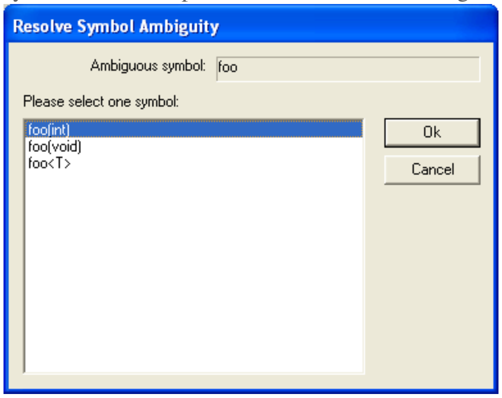
Toggles the display of variables on or off in the list.

Labels

Toggles the display of labels on or off in the list.

Resolve Symbol Ambiguity dialog box

The **Resolve Symbol Ambiguity** dialog box appears, for example, when you specify a symbol in the Disassembly window to go to, and there are several instances of the same symbol due to templates or function overloading.



Ambiguous symbol: foo

Please select one symbol:

foo(int)
foo(void)
foo<T>

Figure 40: Resolve Symbol Ambiguity dialog box

Requirements

None; this window is always available.

Ambiguous symbol

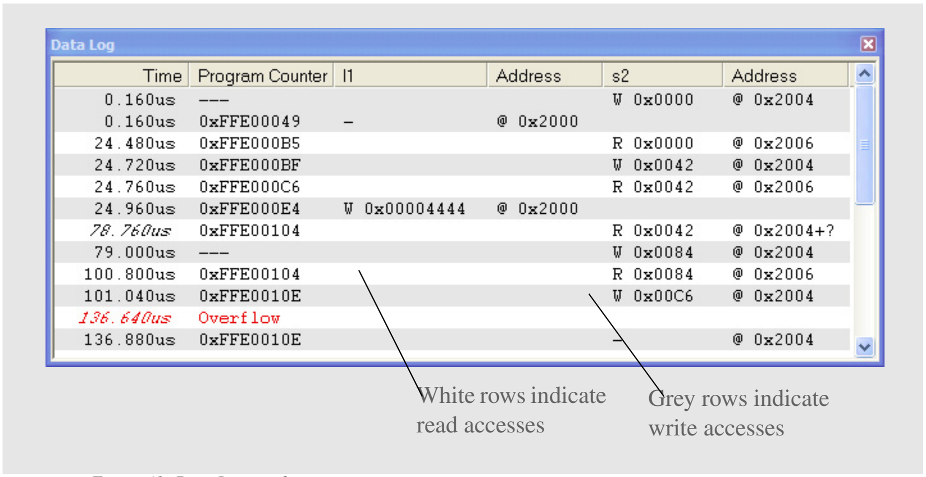
Indicates which symbol that is ambiguous.

Please select one symbol

A list of possible matches for the ambiguous symbol. Select the one you want to use.

Data Log window

The Data Log window is available from the C-SPY driver menu.



Data Log

Time	Program Counter	I1	Address	s2	Address
0.160us	---			W 0x0000	@ 0x2004
0.160us	0xFFE00049	–	@ 0x2000		
24.480us	0xFFE000B5			R 0x0000	@ 0x2006
24.720us	0xFFE000BF			W 0x0042	@ 0x2004
24.760us	0xFFE000C6			R 0x0042	@ 0x2006
24.960us	0xFFE000E4	W 0x00004444	@ 0x2000		
78.760us	0xFFE00104			R 0x0042	@ 0x2004+?
79.000us	---			W 0x0084	@ 0x2004
100.800us	0xFFE00104			R 0x0084	@ 0x2006
101.040us	0xFFE0010E			W 0x00C6	@ 0x2004
136.640us	Overflow				
136.880us	0xFFE0010E			–	@ 0x2004

White rows indicate read accesses
Grey rows indicate write accesses

Figure 41: Data Log window

Use this window to log accesses to up to four different memory locations or areas.

See also *Getting started using data logging*, page 84.

Requirements

The C-SPY simulator.

Display area

Each row in the display area shows the time, the program counter, and, for every tracked data object, its value and address in these columns:

Time

The time for the data access, based on the clock frequency.

If the time is displayed in italics, the target system has not been able to collect a correct time, but instead had to approximate it.

This column is available when you have selected **Show cycles** from the context menu.

Cycles

The number of cycles from the start of the execution until the event. This information is cleared at reset.

If a cycle is displayed in italics, the target system has not been able to collect a correct time, but instead had to approximate it.

This column is available when you have selected **Show cycles** from the context menu.

Program Counter*

Displays one of these:

An address, which is the content of the `PC`, that is, the address of the instruction that performed the memory access.

`---`, the target system failed to provide the debugger with any information.

`Overflow` in red, the communication channel failed to transmit all data from the target system.

Value

Displays the access type and the value (using the access size) for the location or area you want to log accesses to. For example, if zero is read using a byte access it will be displayed as `0x00`, and for a long access it will be displayed as `0x00000000`.

To specify what data you want to log accesses to, use the **Data Log** breakpoint dialog box. See *Data Log breakpoints*, page 103.

Address

The actual memory address that is accessed. For example, if only a byte of a word is accessed, only the address of the byte is displayed. The address is calculated as base address + offset, where the base address is retrieved from the **Data Log** breakpoint dialog box and the offset is retrieved from the logs. If the log from the target system does not provide the debugger with an offset, the offset contains + ?.

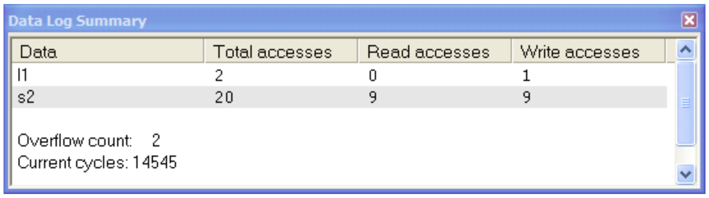
* You can double-click a line in the display area. If the value of the `PC` for that line is available in the source code, the editor window displays the corresponding source code (this does not include library source code).

Context menu

Identical to the context menu of the Interrupt Log window, see *Interrupt Log window*, page 209.

Data Log Summary window

The Data Log Summary window is available from the C-SPY driver menu.



Data Log Summary

Data	Total accesses	Read accesses	Write accesses
I1	2	0	1
s2	20	9	9

Overflow count: 2
Current cycles: 14545

Figure 42: Data Log Summary window

This window displays a summary of data accesses to specific memory location or areas.

See also *Getting started using data logging*, page 84.

Requirements

The C-SPY simulator.

Display area

Each row in this area displays the type and the number of accesses to each memory location or area in these columns:

Data

The name of the data object you have selected to log accesses to. To specify what data object you want to log accesses to, use the **Data Log** breakpoint dialog box. See *Data Log breakpoints*, page 103.

At the bottom of the column, the current time or cycles is displayed—execution time since the start of execution or the number of cycles. Overflow count displays the number of overflows.

Total accesses

The number of total accesses.

If the sum of read accesses and write accesses is less than the total accesses, there have been a number of access logs for which the target system for some reason did not provide valid access type information.

Read accesses

The number of total read accesses.

Write accesses

The number of total write accesses.

Context menu

Identical to the context menu of the Interrupt Log window, see *Interrupt Log window*, page 209.

Breakpoints

This chapter describes breakpoints and the various ways to define and monitor them. More specifically, this means:

- Introduction to setting and using breakpoints
- Setting breakpoints
- Reference information on breakpoints.

Introduction to setting and using breakpoints

This section introduces breakpoints.

These topics are covered:

- Reasons for using breakpoints
- Briefly about setting breakpoints
- Breakpoint types
- Breakpoint icons
- Breakpoints in the C-SPY simulator
- Breakpoints in the C-SPY hardware drivers
- Breakpoint consumers.

REASONS FOR USING BREAKPOINTS

C-SPY® lets you set various types of breakpoints in the application you are debugging, allowing you to stop at locations of particular interest. You can set a breakpoint at a *code* location to investigate whether your program logic is correct, or to get trace printouts. In addition to code breakpoints, and depending on what C-SPY driver you are using, additional breakpoint types might be available. For example, you might be able to set a *data* breakpoint, to investigate how and when the data changes.

You can let the execution stop under certain *conditions*, which you specify. You can also let the breakpoint trigger a *side effect*, for instance executing a C-SPY macro function, by transparently stopping the execution and then resuming. The macro function can be defined to perform a wide variety of actions, for instance, simulating hardware behavior.

All these possibilities provide you with a flexible tool for investigating the status of your application.

BRIEFLY ABOUT SETTING BREAKPOINTS

You can set breakpoints in many various ways, allowing for different levels of interaction, precision, timing, and automation. All the breakpoints you define will appear in the Breakpoints window. From this window you can conveniently view all breakpoints, enable and disable breakpoints, and open a dialog box for defining new breakpoints. The Breakpoint Usage window also lists all internally used breakpoints, see *Breakpoint consumers*, page 105.

Breakpoints are set with a higher precision than single lines, using the same mechanism as when stepping; for more information about the precision, see *Single stepping*, page 60.

You can set breakpoints while you edit your code even if no debug session is active. The breakpoints will then be validated when the debug session starts. Breakpoints are preserved between debug sessions.

Note: For most hardware debugger systems it is only possible to set breakpoints when the application is not executing.

BREAKPOINT TYPES

Depending on the C-SPY driver you are using, C-SPY supports different types of breakpoints.

Code breakpoints

Code breakpoints are used for code locations to investigate whether your program logic is correct or to get trace printouts. Code breakpoints are triggered when an instruction is fetched from the specified location. If you have set the breakpoint on a specific machine instruction, the breakpoint will be triggered and the execution will stop, before the instruction is executed.

If you are using any of the C-SPY hardware drivers, code breakpoints are implemented as two different types of breakpoints. If you set a code breakpoint by right-clicking in the editor window, the default breakpoint type will be used, depending on if it is in ROM or RAM memory:

- *Execution address breakpoints* are the default code breakpoints for ROM memory if there are any free events to use. If there are no free events, a software code breakpoint will be set. These software breakpoints are considerably slower than the execution address breakpoints. The Breakpoint Usage window displays which breakpoint types are actually used.
- *Software breakpoints* are the default code breakpoints for RAM memory.

Log breakpoints

Log breakpoints provide a convenient way to add trace printouts without having to add any code to your application source code. Log breakpoints are triggered when an instruction is fetched from the specified location. If you have set the breakpoint on a specific machine instruction, the breakpoint will be triggered and the execution will temporarily stop and print the specified message in the C-SPY Debug Log window.

Trace breakpoints

Trace Start and Stop breakpoints start and stop trace data collection—a convenient way to analyze instructions between two execution points.

When you are using the C-SPY E30 or E30A emulator driver, you can set trace events, See *Trace Event dialog box*, page 163.

Data breakpoints

Data breakpoints are primarily useful for variables that have a fixed address in memory. If you set a breakpoint on an accessible local variable, the breakpoint is set on the corresponding memory location. The validity of this location is only guaranteed for small parts of the code. Data breakpoints are triggered when data is accessed at the specified location. The execution will usually stop directly after the instruction that accessed the data has been executed.

When you are using the C-SPY E30 or E30A emulator driver, the data breakpoints are represented as *data break events*, see *Data Break Event dialog box*, page 123.

Data Log breakpoints

Data Log breakpoints are triggered when data is accessed at the specified location. If you have set a breakpoint on a specific address or a range, a log message is displayed in the Data Log window for each access to that location. Data logs can also be displayed on the Data Log graph in the Timeline window, if that window is enabled.

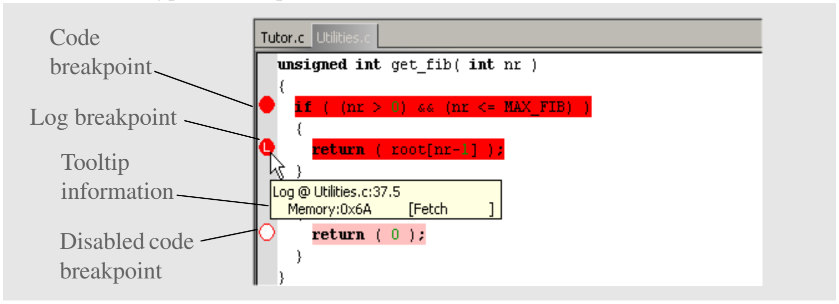
Immediate breakpoints

The C-SPY Simulator lets you set *immediate* breakpoints, which will halt instruction execution only temporarily. This allows a C-SPY macro function to be called when the simulated processor is about to read data from a location or immediately after it has written data. Instruction execution will resume after the action.

This type of breakpoint is useful for simulating memory-mapped devices of various kinds (for instance serial ports and timers). When the simulated processor reads from a memory-mapped location, a C-SPY macro function can intervene and supply appropriate data. Conversely, when the simulated processor writes to a memory-mapped location, a C-SPY macro function can act on the value that was written.

BREAKPOINT ICONS

A breakpoint is marked with an icon in the left margin of the editor window, and the icon varies with the type of breakpoint:



```
unsigned int get_fib( int nr )
{
  if ( (nr > 0) && (nr <= MAX_FIB) )
  {
    return ( root[nr-1] );
  }
    return ( 0 );
  }
}
```

```
Log @ Utilities.c:37.5
  Memory:0x6A        [Fetch        ]
```

Figure 43: Breakpoint icons

If the breakpoint icon does not appear, make sure the option **Show bookmarks** is selected, see Editor options in the *IDE Project Management and Building Guide*.

Just point at the breakpoint icon with the mouse pointer to get detailed tooltip information about all breakpoints set on the same location. The first row gives user breakpoint information, the following rows describe the physical breakpoints used for implementing the user breakpoint. The latter information can also be seen in the Breakpoint Usage window.

Note: The breakpoint icons might look different for the C-SPY driver you are using.

BREAKPOINTS IN THE C-SPY SIMULATOR

The C-SPY simulator supports all breakpoint types and you can set an unlimited amount of breakpoints.

BREAKPOINTS IN THE C-SPY HARDWARE DRIVERS

Using the C-SPY drivers for hardware debugger systems you can set various breakpoint types. The amount of breakpoints you can set depends on the number of *hardware breakpoints* available on the target system.

This table summarizes the characteristics of breakpoints for the different target systems:

C-SPY hardware driver	Code and Log breakpoints*	Trace breakpoints*	Data breakpoints*
E8a			
using 8 hardware breakpoints	8	--	--
using software breakpoints	255	--	--
E30			
using 8 hardware breakpoints	8	4	8
using software breakpoints	64	--	--
E30A			
using 8 hardware breakpoints	8	8	8
using software breakpoints	256	--	--

Table 7: Available breakpoints in C-SPY hardware drivers

* These breakpoint types share the same breakpoint resources.

The debugger will first use any available hardware breakpoints before using software breakpoints. Exceeding the number of available hardware breakpoints causes the debugger to single step. This will significantly reduce the execution speed. For this reason you must be aware of the different breakpoint consumers.

BREAKPOINT CONSUMERS

A debugger system includes several consumers of breakpoints.

User breakpoints

The breakpoints you define in the breakpoint dialog box or by toggling breakpoints in the editor window often consume one physical breakpoint each, but this can vary greatly. Some user breakpoints consume several physical breakpoints and conversely, several user breakpoints can share one physical breakpoint. User breakpoints are displayed in the same way both in the Breakpoint Usage window and in the Breakpoints window, for example `Data @[R] callCount`.

C-SPY itself

C-SPY itself also consumes breakpoints. C-SPY will set a breakpoint if:

- The debugger option **Run to** has been selected, and any step command is used. These are temporary breakpoints which are only set during a debug session. This means that they are not visible in the Breakpoints window.
- The linker option **With I/O emulation modules** has been selected.

In the DLIB runtime environment, C-SPY will set a system breakpoint on the `__DebugBreak` label.

These types of breakpoint consumers are displayed in the Breakpoint Usage window, for example, `C-SPY Terminal I/O & libsupport module`.

C-SPY plugin modules

For example, modules for real-time operating systems can consume additional breakpoints. Specifically, by default, the Stack window consumes one physical breakpoint.

To disable the breakpoint used by the Stack window:

1. Choose **Tools>Options>Stack**.
2. Deselect the **Stack pointer(s) not valid until program reaches:** *label* option.

To disable the Stack window entirely, choose **Project>Options>Debugger>Plugins** and deselect the Stack plugin.

Setting breakpoints

This section describes various tasks related to setting and using breakpoints.

More specifically, you will get information about:

- Various ways to set a breakpoint
- Toggling a simple code breakpoint
- Setting breakpoints using the dialog box
- Setting a data breakpoint in the Memory window
- Setting breakpoints using system macros
- Useful breakpoint hints.

VARIOUS WAYS TO SET A BREAKPOINT

You can set a breakpoint in various ways:

- Toggling a simple code breakpoint.
- Using the **New Breakpoints** dialog box and the **Edit Breakpoints** dialog box available from the context menus in the editor window, Breakpoints window, and in the Disassembly window. The dialog boxes give you access to all breakpoint options.
- Setting a data breakpoint on a memory area directly in the Memory window.
- Using predefined system macros for setting breakpoints, which allows automation.

The different methods offer different levels of simplicity, complexity, and automation.

TOGGLING A SIMPLE CODE BREAKPOINT

Toggling a code breakpoint is a quick method of setting a breakpoint. If you are using a C-SPY hardware driver, the default breakpoint type will be used, depending on RAM or ROM memory. The following methods are available both in the editor window and in the Disassembly window:

- Click in the gray left-side margin of the window
- Place the insertion point in the C source statement or assembler instruction where you want the breakpoint, and click the **Toggle Breakpoint** button in the toolbar
- Choose **Edit>Toggle Breakpoint**
- Right-click and choose **Toggle Breakpoint** from the context menu.

SETTING BREAKPOINTS USING THE DIALOG BOX

The advantage of using a breakpoint dialog box is that it provides you with a graphical interface where you can interactively fine-tune the characteristics of the breakpoints. You can set the options and quickly test whether the breakpoint works according to your intentions.

All breakpoints you define using a breakpoint dialog box are preserved between debug sessions.

You can open the dialog box from the context menu available in the editor window, Breakpoints window, and in the Disassembly window.

To set a new breakpoint:

1 Choose **View>Breakpoints** to open the Breakpoints window.

2 In the Breakpoints window, right-click, and choose **New Breakpoint** from the context menu.

3 On the submenu, choose the breakpoint type you want to set.

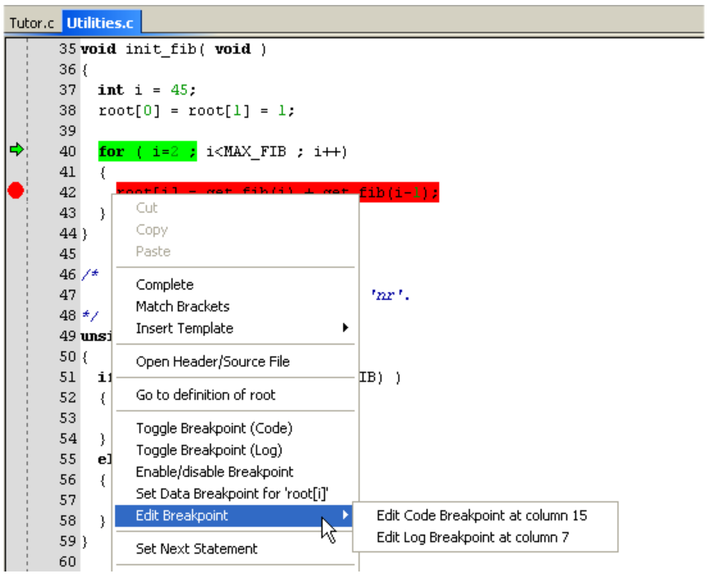
   Depending on the C-SPY driver you are using, different breakpoint types are available.

4 In the breakpoint dialog box that appears, specify the breakpoint settings and click **OK**.

   The breakpoint is displayed in the Breakpoints window.

To modify an existing breakpoint:

5 In the Breakpoints window, editor window, or in the Disassembly window, select the breakpoint you want to modify and right-click to open the context menu.



```
void init_fib( void )
{
  int i = 45;
  root[0] = root[1] = 1;

  for ( i=2 ; i<MAX_FIB ; i++)
  {
    root[i] = get_fib(i) + get_fib(i-1);
  }
}

/*
                                          'nr'.
*/
unsi
{
  i                                      IB) )
  {

  }
  el
  {

  }
}

```

Figure 44: Modifying breakpoints via the context menu

If there are several breakpoints on the same source code line, the breakpoints will be listed on a submenu.

6 On the context menu, choose the appropriate command.

7 In the breakpoint dialog box that appears, specify the breakpoint settings and click **OK**.

The breakpoint is displayed in the Breakpoints window.

SETTING A DATA BREAKPOINT IN THE MEMORY WINDOW

You can set breakpoints directly on a memory location in the Memory window. Right-click in the window and choose the breakpoint command from the context menu that appears. To set the breakpoint on a range, select a portion of the memory contents.

The breakpoint is not highlighted in the Memory window; instead, you can see, edit, and remove it using the Breakpoints window, which is available from the **View** menu. The breakpoints you set in the Memory window will be triggered for both read and write accesses. All breakpoints defined in this window are preserved between debug sessions.

Note: Setting breakpoints directly in the Memory window is only possible if the driver you use supports this.

SETTING BREAKPOINTS USING SYSTEM MACROS

You can set breakpoints not only in the breakpoint dialog box but also by using built-in C-SPY system macros. When you use system macros for setting breakpoints, the breakpoint characteristics are specified as macro parameters.

Macros are useful when you have already specified your breakpoints so that they fully meet your requirements. You can define your breakpoints in a macro file, using built-in system macros, and execute the file at C-SPY startup. The breakpoints will then be set automatically each time you start C-SPY. Another advantage is that the debug session will be documented, and that several engineers involved in the development project can share the macro files.

Note: If you use system macros for setting breakpoints, you can still view and modify them in the Breakpoints window. In contrast to using the dialog box for defining breakpoints, all breakpoints that are defined using system macros are removed when you exit the debug session.

These breakpoint macros are available:

C-SPY macro for breakpoints	Simulator	E8a	E30	E30A
`__setCodeBreak`	Yes	Yes	Yes	Yes
`__setDataBreak`	Yes	--	--	--
`__setLogBreak`	Yes	Yes	Yes	Yes
`__setDataLogBreak`	Yes	--	--	--
`__setSimBreak`	Yes	--	--	--
`__setTraceStartBreak`	Yes	--	--	--
`__setTraceStopBreak`	Yes	--	--	--
`__clearBreak`	Yes	Yes	Yes	Yes

Table 8: C-SPY macros for breakpoints

For information about each breakpoint macro, see *Reference information on C-SPY system macros*, page 229.

Setting breakpoints at C-SPY startup using a setup macro file

You can use a setup macro file to define breakpoints at C-SPY startup. Follow the procedure described in *Registering and executing using setup macros and setup files*, page 220.

USEFUL BREAKPOINT HINTS

Below are some useful hints related to setting breakpoints.

Tracing incorrect function arguments

If a function with a pointer argument is sometimes incorrectly called with a `NULL` argument, you might want to debug that behavior. These methods can be useful:

- Set a breakpoint on the first line of the function with a condition that is true only when the parameter is `0`. The breakpoint will then not be triggered until the problematic situation actually occurs. The advantage of this method is that no extra source code is needed. The drawback is that the execution speed might become unacceptably low.
- You can use the `assert` macro in your problematic function, for example:

  ```
  int MyFunction(int * MyPtr)
  {
    assert(MyPtr != 0); /* Assert macro added to your source
                           code. */
    /* Here comes the rest of your function. */
  }
  ```

  The execution will break whenever the condition is true. The advantage is that the execution speed is only very slightly affected, but the drawback is that you will get a small extra footprint in your source code. In addition, the only way to get rid of the execution stop is to remove the macro and rebuild your source code.
- Instead of using the `assert` macro, you can modify your function like this:

  ```
  int MyFunction(int * MyPtr)
  {
    if(MyPtr == 0)
       MyDummyStatement; /* Dummy statement where you set a
                            breakpoint. */
    /* Here comes the rest of your function. */
  }
  ```

  You must also set a breakpoint on the extra dummy statement, so that the execution will break whenever the condition is true. The advantage is that the execution speed is only very slightly affected, but the drawback is that you will still get a small extra footprint in your source code. However, in this way you can get rid of the execution stop by just removing the breakpoint.

Performing a task and continuing execution

You can perform a task when a breakpoint is triggered and then automatically continue execution.

You can use the **Action** text box to associate an action with the breakpoint, for instance a C-SPY macro function. When the breakpoint is triggered and the execution of your application has stopped, the macro function will be executed. In this case, the execution will not continue automatically.

Instead, you can set a condition which returns `0` (false). When the breakpoint is triggered, the condition—which can be a call to a C-SPY macro that performs a task—is evaluated and because it is not true, execution continues.

Consider this example where the C-SPY macro function performs a simple task:

```
__var my_counter;

count()
{
  my_counter += 1;
  return 0;
}
```

To use this function as a condition for the breakpoint, type `count()` in the **Expression** text box under **Conditions**. The task will then be performed when the breakpoint is triggered. Because the macro function `count` returns `0`, the condition is false and the execution of the program will resume automatically, without any stop.

Reference information on breakpoints

This section gives reference information about these windows and dialog boxes:

- *Breakpoints window*, page 112
- *Breakpoint Usage window*, page 114
- *Code breakpoints dialog box*, page 115
- *Execution Address Breakpoint dialog box*, page 116
- *Software Breakpoint dialog box*, page 118
- *Log breakpoints dialog box*, page 119
- *Data breakpoints dialog box*, page 121
- *Data Log breakpoints dialog box*, page 124
- *Immediate breakpoints dialog box*, page 125
- *Enter Location dialog box*, page 126
- *Resolve Source Ambiguity dialog box*, page 128.

See also:

- *Reference information on C-SPY system macros*, page 229

- *Reference information on trace*, page 162.

Breakpoints window

The Breakpoints window is available from the **View** menu.

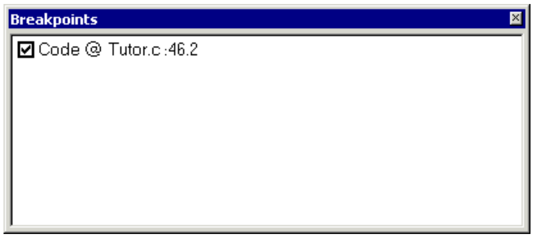


Figure 45: Breakpoints window

The Breakpoints window lists all breakpoints you define.

Use this window to conveniently monitor, enable, and disable breakpoints; you can also define new breakpoints and modify existing breakpoints.

Requirements

None; this window is always available.

Display area

This area lists all breakpoints you define. For each breakpoint, information about the breakpoint type, source file, source line, and source column is provided.

Context menu

This context menu is available:

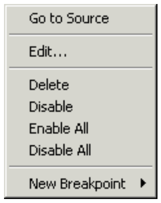


Figure 46: Breakpoints window context menu

These commands are available:

Go to Source

Moves the insertion point to the location of the breakpoint, if the breakpoint has a source location. Double-click a breakpoint in the Breakpoints window to perform the same command.

Edit

Opens the breakpoint dialog box for the breakpoint you selected.

Delete

Deletes the breakpoint. Press the Delete key to perform the same command.

Enable

Enables the breakpoint. The check box at the beginning of the line will be selected. You can also perform the command by manually selecting the check box. This command is only available if the breakpoint is disabled.

Disable

Disables the breakpoint. The check box at the beginning of the line will be deselected. You can also perform this command by manually deselecting the check box. This command is only available if the breakpoint is enabled.

Enable All

Enables all defined breakpoints.

Disable All

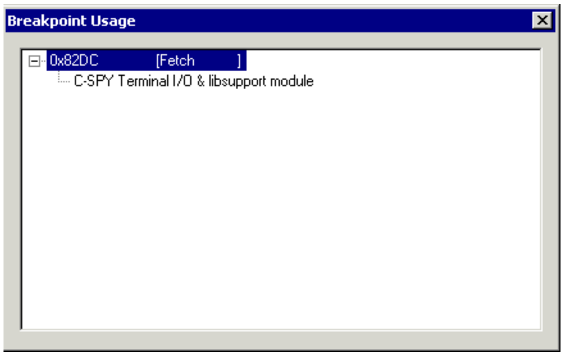
Disables all defined breakpoints.

New Breakpoint

Displays a submenu where you can open the breakpoint dialog box for the available breakpoint types. All breakpoints you define using this dialog box are preserved between debug sessions.

Breakpoint Usage window

The **Breakpoint Usage** window is available from the menu specific to the C-SPY driver you are using.



```
⊟ 0x82DC          [Fetch        ]
    C-SPY Terminal I/O & libsupport module
```

Figure 47: Breakpoint Usage window

The Breakpoint Usage window lists all breakpoints currently set in the target system, both the ones you have defined and the ones used internally by C-SPY. The format of the items in this dialog box depends on the C-SPY driver you are using.

The window gives a low-level view of all breakpoints, related but not identical to the list of breakpoints displayed in the Breakpoints window.

C-SPY uses breakpoints when stepping. If your target system has a limited number of hardware breakpoints exceeding the number of available hardware breakpoints will cause the debugger to single step. This will significantly reduce the execution speed. Therefore, in a debugger system with a limited amount of hardware breakpoints, you can use the Breakpoint Usage dialog box for:

- Identifying all breakpoint consumers
- Checking that the number of active breakpoints is supported by the target system
- Configuring the debugger to use the available breakpoints in a better way, if possible.

Requirements

None; this window is always available.

Display area

For each breakpoint in the list, the address and access type are displayed. Each breakpoint in the list can also be expanded to show its originator.

Code breakpoints dialog box

The **Code** breakpoints dialog box is available from the context menu in the editor window, Breakpoints window, and in the Disassembly window.

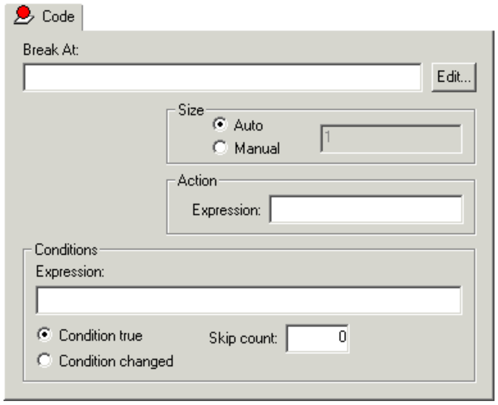


Figure 48: Code breakpoints dialog box

Use the **Code** breakpoints dialog box to set a code breakpoint.

Note: The **Code** breakpoints dialog box depends on the C-SPY driver you are using.

Requirements

None; this dialog box is always available.

Break At

Specify the code location of the breakpoint in the text box. Alternatively, click the **Edit** button to open the **Enter Location** dialog box, see *Enter Location dialog box*, page 126.

Size

Determines whether there should be a size—in practice, a range—of locations where the breakpoint will trigger. Each fetch access to the specified memory range will trigger the breakpoint. Select how to specify the size:

Auto

The size will be set automatically, typically to 1.

Manual

Specify the size of the breakpoint range in the text box.

Note: The **Size** option is only available for the C-SPY Simulator.

Action

Specify a valid C-SPY expression, which is evaluated when the breakpoint is triggered and the condition is true. For more information, see *Useful breakpoint hints*, page 110.

Conditions

Specify simple or complex conditions:

Expression

Specify a valid C-SPY expression, see *Expressions*, page 225.

Condition true

The breakpoint is triggered if the value of the expression is true.

Condition changed

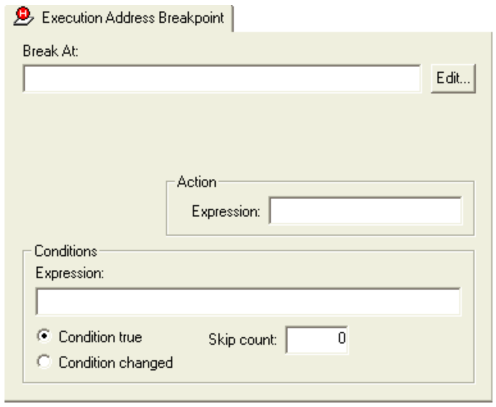
The breakpoint is triggered if the value of the expression has changed since it was last evaluated.

Skip count

The number of times that the breakpoint condition must be fulfilled before the breakpoint starts triggering. After that, the breakpoint will trigger every time the condition is fulfilled.

Execution Address Breakpoint dialog box

The **Execution Address Breakpoint** dialog box is available from the context menu in the Breakpoints window when you choose **New Breakpoint>Address**.



(●) Condition true
( ) Condition changed
Skip count: 0

Figure 49: Execution Address Breakpoint dialog box

Use this dialog box to set an execution address breakpoint. To modify an existing breakpoint, select it in the Breakpoints window and choose **Edit** on the context menu.

When an execution address breakpoint is triggered, there is no need for instruction rewrite/write back processing, which makes it fast. Up to eight address breakpoints can be set. They share events with data breaks, trace, the RAM monitor, and time measurement.

Requirements

One of these alternatives:

- A C-SPY E8a emulator
- A C-SPY E30 emulator
- A C-SPY E30A emulator

Break At

Specify the data location of the breakpoint in the text box. Alternatively, click the **Edit** button to open the **Enter Location** dialog box, see *Enter Location dialog box*, page 126.

Action

Specify a valid C-SPY expression, which is evaluated when the breakpoint is triggered and the condition is true. For more information, see *Useful breakpoint hints*, page 110.

Conditions

Specify simple or complex conditions:

Expression

Specify a valid C-SPY expression, see *Expressions*, page 225.

Condition true

The breakpoint is triggered if the value of the expression is true.

Condition changed

The breakpoint is triggered if the value of the expression has changed since it was last evaluated.

Skip count

The number of times that the breakpoint condition must be fulfilled before the breakpoint starts triggering. After that, the breakpoint will trigger every time the condition is fulfilled.

Software Breakpoint dialog box

The Software Breakpoint dialog box is available from the context menu in the Breakpoints window when you choose **New Breakpoint>Software**.

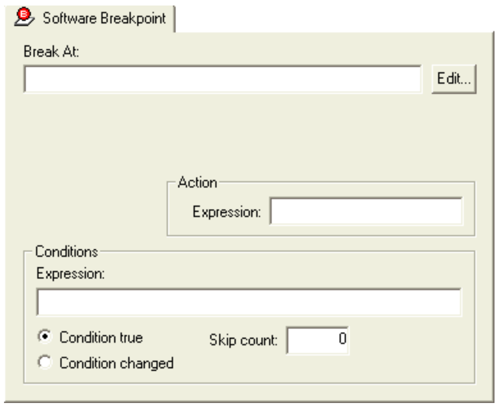


Figure 50: Software Breakpoint dialog box

Use this dialog box to set a software breakpoint.

To modify an existing breakpoint, select it in the Breakpoints window and choose **Edit** on the context menu.

When a software breakpoint is set in the internal flash ROM of the target microcomputer, a number of instructions must be rewritten and processed every time it is triggered, which makes software code breakpoints slow.

Requirements

One of these alternatives:

- A C-SPY E8a emulator
- A C-SPY E30 emulator
- A C-SPY E30A emulator

Break At

Specify the code location of the breakpoint in the text box. Alternatively, click the **Edit** button to open the **Enter Location** dialog box, see *Enter Location dialog box*, page 126.

Action

Specify a valid C-SPY expression, which is evaluated when the breakpoint is triggered and the condition is true. For more information, see *Useful breakpoint hints*, page 110.

Conditions

Specify simple or complex conditions:

Expression

Specify a valid C-SPY expression, see *Expressions*, page 225.

Condition true

The breakpoint is triggered if the value of the expression is true.

Condition changed

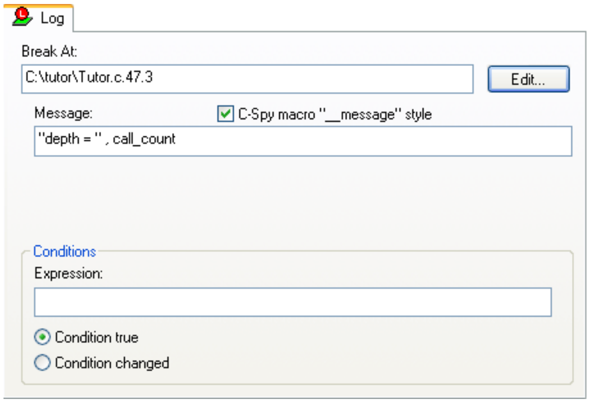
The breakpoint is triggered if the value of the expression has changed since it was last evaluated.

Skip count

The number of times that the breakpoint condition must be fulfilled before the breakpoint starts triggering. After that, the breakpoint will trigger every time the condition is fulfilled.

Log breakpoints dialog box

The **Log** breakpoints dialog box is available from the context menu in the editor window, Breakpoints window, and in the Disassembly window.



Break At: C:\tutor\Tutor.c.47.3

Message: "depth = " , call_count

[✓] C-Spy macro "__message" style

Expression:

(●) Condition true
( ) Condition changed

Figure 51: Log breakpoints dialog box

Use the **Log** breakpoints dialog box to set a log breakpoint.

Note: The **Log** breakpoints dialog box depends on the C-SPY driver you are using.

Requirements

None; this dialog box is always available.

Trigger at

Specify the code location of the breakpoint. Alternatively, click the **Edit** button to open the **Enter Location** dialog box, see *Enter Location dialog box*, page 126.

Message

Specify the message you want to be displayed in the C-SPY Debug Log window. The message can either be plain text, or—if you also select the option **C-SPY macro "__message" style**—a comma-separated list of arguments.

C-SPY macro "__message" style

Select this option to make a comma-separated list of arguments specified in the Message text box be treated exactly as the arguments to the C-SPY macro language statement `__message`, see *Formatted output*, page 226.

Conditions

Specify simple or complex conditions:

Expression

Specify a valid C-SPY expression, see *Expressions*, page 225.

Condition true

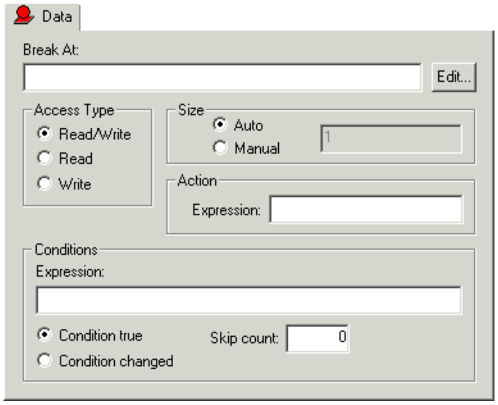
The breakpoint is triggered if the value of the expression is true.

Condition changed

The breakpoint is triggered if the value of the expression has changed since it was last evaluated.

Data breakpoints dialog box

The **Data** breakpoints dialog box is available from the context menu in the editor window, Breakpoints window, the Memory window, and in the Disassembly window.



Access Type: (●) Read/Write ( ) Read ( ) Write

Size: (●) Auto ( ) Manual 1

Condition true (●) Condition changed ( )

Skip count: 0

Figure 52: Data breakpoints dialog box

Use the **Data** breakpoints dialog box to set a data breakpoint. Data breakpoints never stop execution within a single instruction. They are recorded and reported after the instruction is executed.

Requirements

The C-SPY simulator.

Break At

Specify the data location of the breakpoint in the text box. Alternatively, click the **Edit** button to open the **Enter Location** dialog box, see *Enter Location dialog box*, page 126.

Access Type

Selects the type of memory access that triggers the breakpoint:

Read/Write

Reads from or writes to location.

Read

Reads from location.

Write

Writes to location.

Size

Determines whether there should be a size—in practice, a range—of locations where the breakpoint will trigger. Each fetch access to the specified memory range will trigger the breakpoint. Select how to specify the size:

Auto

The size will automatically be based on the type of expression the breakpoint is set on. For example, if you set the breakpoint on a 12-byte structure, the size of the breakpoint will be 12 bytes.

Manual

Specify the size of the breakpoint range in the text box.

For data breakpoints, this can be useful if you want the breakpoint to be triggered on accesses to data structures, such as arrays, structs, and unions.

Action

Specify a valid C-SPY expression, which is evaluated when the breakpoint is triggered and the condition is true. For more information, see *Useful breakpoint hints*, page 110.

Conditions

Specify simple or complex conditions:

Expression

Specify a valid C-SPY expression, see *Expressions*, page 225.

Condition true

The breakpoint is triggered if the value of the expression is true.

Condition changed

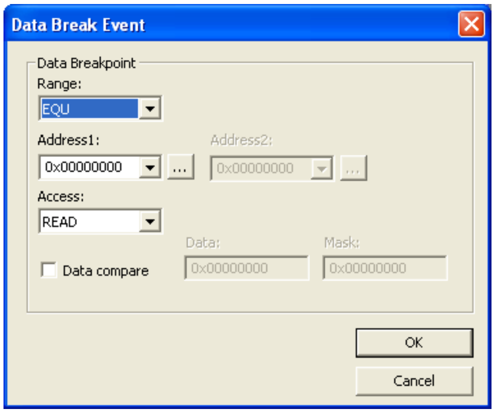
The breakpoint is triggered if the value of the expression has changed since it was last evaluated.

Skip count

The number of times that the breakpoint condition must be fulfilled before the breakpoint starts triggering. After that, the breakpoint will trigger every time the condition is fulfilled.

Data Break Event dialog box

In the **Data Break Event** dialog box is available from the Events window context menu.



Data Break Event

Data Breakpoint

Range: EQU

Address1: 0x00000000

Address2: 0x00000000

Access: READ

☐ Data compare

Data: 0x00000000

Mask: 0x00000000

Figure 53: Data Break Event dialog box

Use this dialog box to set up and modify data break events.

When you specify a break event for a memory access, you can specify the data to be compared with the data written to/read from a specified address. You can also specify mask bits for the comparison data.

Requirements

One of these alternatives:

- A C-SPY E30 emulator
- A C-SPY E30A emulator

Range

Selects a range condition. `EQU` means address match. `IN` means address range.

Address 1

Selects the start address of the address range or the address used for the address match.

Address 2

Selects the type of access that will trigger the event. The event is triggered when memory is accessed in this way at the specified address or under conditions set for the specified address range.

Access

Selects the end address of an address range.

Data compare

Creates a data compare break.

Data

The data that should match to trigger the event.

Mask

Mask for the data that compare break.

Data Log breakpoints dialog box

The **Data Log** breakpoints dialog box is available from the context menu in the editor window, Breakpoints window, the Memory window, and in the Disassembly window.

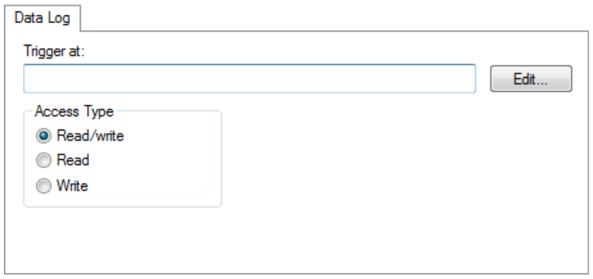


Figure 54: Data Log breakpoints dialog box

Use the Data Log breakpoints dialog box to set a maximum of four data log breakpoints.

To get started using data logging, see *Getting started using data logging*, page 84

Requirements

The C-SPY simulator

Trigger at

Specify the data location of the breakpoint. Alternatively, click the **Edit** button to open the **Enter Location** dialog box, see *Enter Location dialog box*, page 126.

Access Type

Selects the type of memory access that triggers the breakpoint:

Read/Write

Reads from or writes to location.

Read

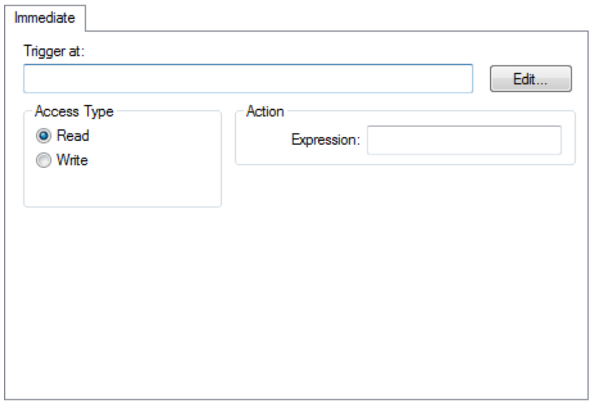
Reads from location.

Write

Writes to location.

Immediate breakpoints dialog box

The **Immediate** breakpoints dialog box is available from the context menu in the editor window, Breakpoints window, the Memory window, and in the Disassembly window.



Access Type: [✓ Read] Write

Figure 55: Immediate breakpoints dialog box

In the C-SPY simulator, use the **Immediate** breakpoints dialog box to set an immediate breakpoint. Immediate breakpoints do not stop execution at all; they only suspend it temporarily.

Requirements

The C-SPY simulator.

Trigger at

Specify the data location of the breakpoint. Alternatively, click the **Edit** button to open the **Enter Location** dialog box, see *Enter Location dialog box*, page 126.

Access Type

Selects the type of memory access that triggers the breakpoint:

Read

Reads from location.

Write

Writes to location.

Action

Specify a valid C-SPY expression, which is evaluated when the breakpoint is triggered and the condition is true. For more information, see *Useful breakpoint hints*, page 110.

Enter Location dialog box

The **Enter Location** dialog box is available from the breakpoints dialog box, either when you set a new breakpoint or when you edit a breakpoint.

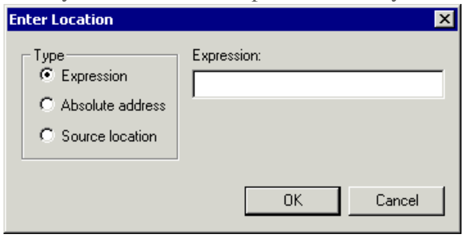


Figure 56: Enter Location dialog box

Use the **Enter Location** dialog box to specify the location of the breakpoint.

Note: This dialog box looks different depending on the **Type** you select.

Type

Selects the type of location to be used for the breakpoint, choose between:

Expression

A C-SPY expression, whose value evaluates to a valid code or data location.

A code location, for example the function `main`, is typically used for code breakpoints.

A data location is the name of a variable and is typically used for data breakpoints. For example, `my_var` refers to the location of the variable `my_var`, and `arr[3]` refers to the location of the fourth element of the array `arr`. For static variables declared with the same name in several functions, use the syntax `my_func::my_static_variable` to refer to a specific variable.

For more information about C-SPY expressions, see *Expressions*, page 225.

Absolute address

An absolute location on the form `zone:hexaddress` or simply `hexaddress` (for example `Memory:0x42`). `zone` refers to C-SPY memory zones and specifies in which memory the address belongs, see *C-SPY memory zones*, page 130.

Source location

A location in your C source code using the syntax:
`{filename}.row.column.`

`filename` specifies the filename and full path.

`row` specifies the row in which you want the breakpoint.

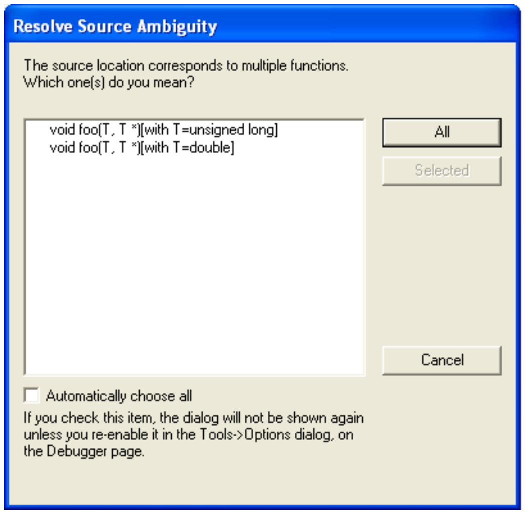
`column` specifies the column in which you want the breakpoint.

For example, `{C:\src\prog.c}.22.3`
sets a breakpoint on the third character position on row 22 in the source file `prog.c`. Note that in quoted form, for example in a C-SPY macro, you must instead write `{C:\\src\\prog.c}.22.3`.

Note that the Source location type is usually meaningful only for code locations in code breakpoints.

Resolve Source Ambiguity dialog box

The **Resolve Source Ambiguity** dialog box appears, for example, when you try to set a breakpoint on templates and the source location corresponds to more than one function.


Resolve Source Ambiguity

The source location corresponds to multiple functions.
Which one(s) do you mean?

```
void foo(T, T *)[with T=unsigned long]
void foo(T, T *)[with T=double]
```

If you check this item, the dialog will not be shown again unless you re-enable it in the Tools->Options dialog, on the Debugger page.

Figure 57: Resolve Source Ambiguity dialog box

To resolve a source ambiguity, perform one of these actions:

- In the text box, select one or several of the listed locations and click **Selected**.
- Click **All**.

All

The breakpoint will be set on all listed locations.

Selected

The breakpoint will be set on the source locations that you have selected in the text box.

Cancel

No location will be used.

Automatically choose all

Determines that whenever a specified source location corresponds to more than one function, all locations will be used.

Note that this option can also be specified in the **IDE Options** dialog box, see Debugger options in the *IDE Project Management and Building Guide*.

Memory and registers

This chapter describes how to use the features available in C-SPY® for examining memory and registers. More specifically, this means information about:

- Introduction to monitoring memory and registers
- Monitoring memory and registers
- Reference information on memory and registers.

Introduction to monitoring memory and registers

This section covers these topics:

- Briefly about monitoring memory and registers
- C-SPY memory zones
- Stack display
- Memory access checking.

BRIEFLY ABOUT MONITORING MEMORY AND REGISTERS

C-SPY provides many windows for monitoring memory and registers, each of them available from the **View** menu:

- The Memory window

  Gives an up-to-date display of a specified area of memory—a memory zone—and allows you to edit it. Different colors are used for indicating data coverage along with execution of your application. You can fill specified areas with specific values and you can set breakpoints directly on a memory location or range. You can open several instances of this window, to monitor different memory areas. The content of the window can be regularly updated while your application is executing.
- The Symbolic memory window

  Displays how variables with static storage duration are laid out in memory. This can be useful for better understanding memory usage or for investigating problems caused by variables being overwritten, for example by buffer overruns.

- The Stack window

  Displays the contents of the stack, including how stack variables are laid out in memory. In addition, some integrity checks of the stack can be performed to detect and warn about problems with stack overflow. For example, the Stack window is useful for determining the optimal size of the stack. You can open up to two instances of this window, each showing different stacks or different display modes of the same stack.
- The Register window

  Gives an up-to-date display of the contents of the processor registers and SFRs, and allows you to edit them. Due to the large amount of registers—memory-mapped peripheral unit registers and CPU registers—it is inconvenient to show all registers concurrently in the Register window. Instead you can divide registers into *register groups*. You can choose to load either predefined register groups or define your own application-specific groups. You can open several instances of this window, each showing a different register group.
- The SFR Setup window

  Displays the currently defined SFRs that C-SPY has information about. If required, you can use this window to customize aspects of the SFRs.

To view the memory contents for a specific variable, simply drag the variable to the Memory window or the Symbolic memory window. The memory area where the variable is located will appear.

Reading the value of some registers might influence the runtime behavior of your application. For example, reading the value of a UART status register might reset a pending bit, which leads to the lack of an interrupt that would have processed a received byte. To prevent this from happening, make sure that the Register window containing any such registers is closed when debugging a running application.

C-SPY MEMORY ZONES

In C-SPY, the term *zone* is used for a named memory area. A memory address, or *location*, is a combination of a zone and a numerical offset into that zone. By default, the R32C/100 architecture has five zones—`SFR1`, `SFR2`, `RAM`, `ROM`, and `DATA_FLASH`—that cover the whole R32C/100 memory range.

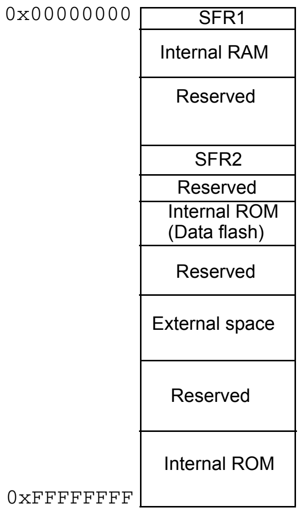


Figure 58: Zones in C-SPY

Memory zones are used in several contexts, most importantly in the Memory and Disassembly windows, and in C-SPY macros. In the windows, use the **Zone** box to choose which memory zone to display.

Device-specific zones

Memory information for device-specific zones is defined in the *device description files*. When you load a device description file, additional zones that adhere to the specific memory layout become available.

See the device description file for information about available memory zones.

For more information, see *Selecting a device description file*, page 33 and *Modifying a device description file*, page 36.

STACK DISPLAY

The Stack window displays the contents of the stack, overflow warnings, and it has a graphical stack bar. These can be useful in many contexts. Some examples are:

- Investigating the stack usage when assembler modules are called from C modules and vice versa
- Investigating whether the correct elements are located on the stack

- Investigating whether the stack is restored properly
- Determining the optimal stack size
- Detecting stack overflows.

For microcomputers with multiple stacks, you can select which stack to view.

Stack usage

When your application is first loaded, and upon each reset, the memory for the stack area is filled with the dedicated byte value `0xCD` before the application starts executing. Whenever execution stops, the stack memory is searched from the end of the stack until a byte with a value different from `0xCD` is found, which is assumed to be how far the stack has been used. Although this is a reasonably reliable way to track stack usage, there is no guarantee that a stack overflow is detected. For example, a stack *can* incorrectly grow outside its bounds, and even modify memory outside the stack area, without actually modifying any of the bytes near the stack range. Likewise, your application might modify memory within the stack area by mistake.

The Stack window cannot detect a stack overflow when it happens, but can only detect the signs it leaves behind. However, when the graphical stack bar is enabled, the functionality needed to detect and warn about stack overflows is also enabled.

Note: The size and location of the stack is retrieved from the definition of the segment holding the stack, made in the linker configuration file. If you, for some reason, modify the stack initialization made in the system startup code, `cstartup`, you should also change the segment definition in the linker configuration file accordingly; otherwise the Stack window cannot track the stack usage. For more information about this, see the *IAR C/C++ Compiler Reference Guide for R32C*.

MEMORY ACCESS CHECKING

The C-SPY simulator can simulate various memory access types of the target hardware and detect illegal accesses, for example a read access to write-only memory. If a memory access occurs that does not agree with the access type specified for the specific memory area, C-SPY will regard this as an illegal access. Also, a memory access to memory which is not defined is regarded as an illegal access. The purpose of memory access checking is to help you to identify any memory access violations.

The memory areas can either be the zones predefined in the device description file, or memory areas based on the segment information available in the debug file. In addition to these, you can define your own memory areas. The access type can be read and write, read-only, or write-only. You cannot map two different access types to the same memory area. You can check for access type violation and accesses to unspecified ranges. Any violations are logged in the Debug Log window. You can also choose to have the execution halted.

Monitoring memory and registers

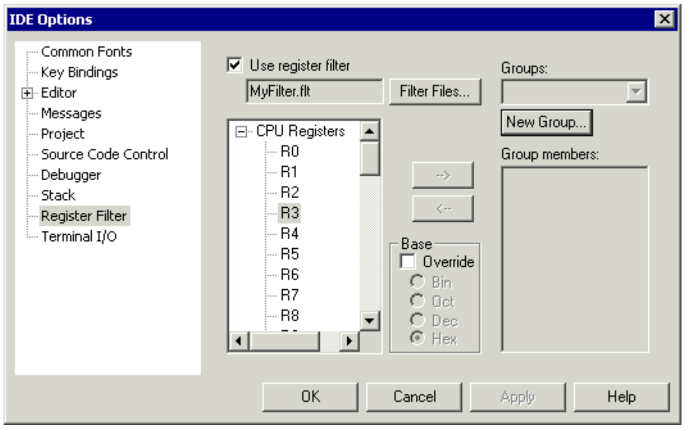
This section describes various tasks related to monitoring memory and registers.

- *Defining application-specific register groups*, page 133.

DEFINING APPLICATION-SPECIFIC REGISTER GROUPS

Defining application-specific register groups minimizes the amount of registers displayed in the Register window and speeds up the debugging.

1 Choose **Tools>Options>Register Filter**.



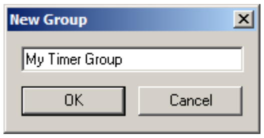
Use register filter [✓]
MyFilter.flt
Override [ ]

Figure 59: Register Filter options

For information about the register filter options, see the *IDE Project Management and Building Guide*.

2 Select **Use register filter** and specify the filename and destination of the filter file for your new group in the dialog box that appears.

3 Click **New Group** and specify the name of your group, for example My Timer Group.



My Timer Group

Figure 60: Register Filter options

4 In the register tree view on the Register Filter page, select a register and click the arrow button to add it to your group. Repeat this process for all registers that you want to add to your group.

5 Optionally, select any registers for which you want to change the integer base, and choose a suitable base.

6 When you are done, click **OK**. Your new group is now available in the Register window.

If you want to add more groups to your filter file, repeat this procedure for each group you want to add.

Note: The registers that appear in the list of registers are retrieved from the ddf file that is currently used. If a certain SFR that you need does not appear, you can register your own SFRs. For more information, see *SFR Setup window*, page 150.

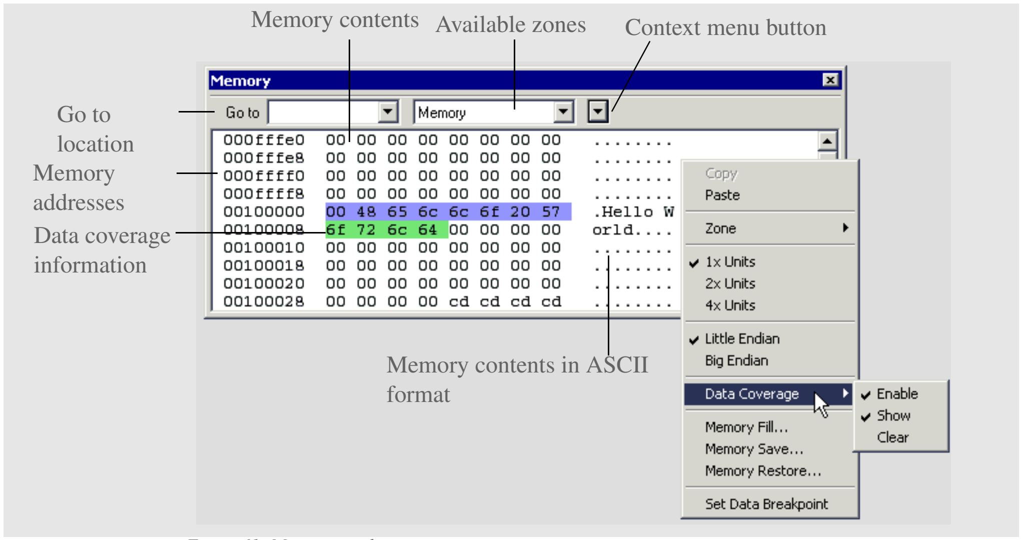
Reference information on memory and registers

This section gives reference information about these windows and dialog boxes:

- *Memory window*, page 135
- *Memory Save dialog box*, page 139
- *Memory Restore dialog box*, page 140
- *Fill dialog box*, page 140
- *Symbolic Memory window*, page 142
- *Stack window*, page 144
- *Register window*, page 148
- *SFR Setup window*, page 150
- *Edit SFR dialog box*, page 153
- *Memory Access Setup dialog box*, page 154
- *Edit Memory Access dialog box*, page 156.

Memory window

The Memory window is available from the **View** menu.



```
000fffe0  00 00 00 00 00 00 00 00   ........
000fffe8  00 00 00 00 00 00 00 00   ........
000ffff0  00 00 00 00 00 00 00 00   ........
000ffff8  00 00 00 00 00 00 00 00   ........
00100000  00 48 65 6c 6c 6f 20 57   .Hello W
00100008  6f 72 6c 64 00 00 00 00   orld....
00100010  00 00 00 00 00 00 00 00   ........
00100018  00 00 00 00 00 00 00 00   ........
00100020  00 00 00 00 00 00 00 00   ........
00100028  00 00 00 00 cd cd cd cd   ........
```

Figure 61: Memory window

This window gives an up-to-date display of a specified area of memory—a memory zone—and allows you to edit it. You can open several instances of this window, which is very convenient if you want to keep track of several memory or register zones, or monitor different parts of the memory.

To view the memory corresponding to a variable, you can select it in the editor window and drag it to the Memory window.

Requirements

None; this window is always available.

Toolbar

The toolbar contains:

Go to

The memory location or symbol you want to view.

Zone

Selects a memory zone, see *C-SPY memory zones*, page 130.

Context menu button

Displays the context menu.

Update Now

Updates the content of the Memory window while your application is executing. This button is only enabled if the C-SPY driver you are using has access to the target system memory while your application is executing.

Live Update

Updates the contents of the Memory window regularly while your application is executing. This button is only enabled if the C-SPY driver you are using has access to the target system memory while your application is executing. To set the update frequency, specify an appropriate frequency in the **IDE Options>Debugger** dialog box.

Display area

The display area shows the addresses currently being viewed, the memory contents in the format you have chosen, and—provided that the display mode is set to **1x Units**—the memory contents in ASCII format. You can edit the contents of the display area, both in the hexadecimal part and the ASCII part of the area.

Data coverage is displayed with these colors:

Yellow	Indicates data that has been read.
Blue	Indicates data that has been written
Green	Indicates data that has been both read and written.

Note: Data coverage is not supported by all C-SPY drivers. Data coverage is supported by the C-SPY Simulator.

Context menu

This context menu is available:

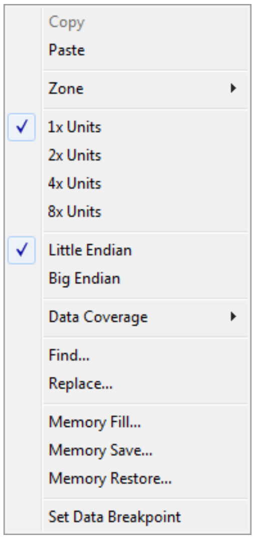


Figure 62: Memory window context menu

These commands are available:

Copy, Paste

Standard editing commands.

Zone

Selects a memory zone, see *C-SPY memory zones*, page 130.

1x Units

Displays the memory contents as single bytes.

2x Units

Displays the memory contents as 2-byte groups.

4x Units

Displays the memory contents as 4-byte groups.

8x Units

Displays the memory contents as 8-byte groups.

Little Endian

Displays the contents in little-endian byte order.

Big Endian

Displays the contents in big-endian byte order.

Data Coverage

Choose between:

Enable toggles data coverage on or off.

Show toggles between showing or hiding data coverage.

Clear clears all data coverage information.

These commands are only available if your C-SPY driver supports data coverage.

Find

Displays a dialog box where you can search for text within the Memory window; read about the **Find** dialog box in the *IDE Project Management and Building Guide*.

Replace

Displays a dialog box where you can search for a specified string and replace each occurrence with another string; read about the **Replace** dialog box in the *IDE Project Management and Building Guide*.

Memory Fill

Displays a dialog box, where you can fill a specified area with a value, see *Fill dialog box*, page 140.

Memory Save

Displays a dialog box, where you can save the contents of a specified memory area to a file, see *Memory Save dialog box*, page 139.

Memory Restore

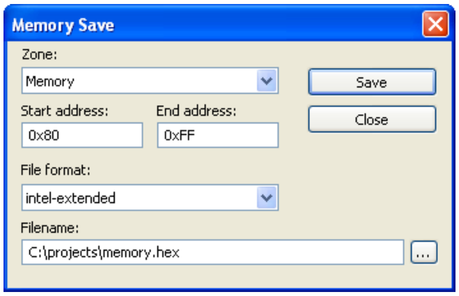
Displays a dialog box, where you can load the contents of a file in Intel-hex or Motorola s-record format to a specified memory zone, see *Memory Restore dialog box*, page 140.

Set Data Breakpoint

Sets breakpoints directly in the Memory window. The breakpoint is not highlighted; you can see, edit, and remove it in the Breakpoints dialog box. The breakpoints you set in this window will be triggered for both read and write access. For more information, see *Setting a data breakpoint in the Memory window*, page 108.

Memory Save dialog box

The **Memory Save** dialog box is available by choosing **Debug>Memory>Save** or from the context menu in the Memory window.



Zone: Memory
Start address: 0x80
End address: 0xFF
File format: intel-extended
Filename: C:\projects\memory.hex

Figure 63: Memory Save dialog box

Use this dialog box to save the contents of a specified memory area to a file.

Requirements

None; this dialog box is always available.

Zone

Selects a memory zone, see *C-SPY memory zones*, page 130.

Start address

Specify the start address of the memory range to be saved.

End address

Specify the end address of the memory range to be saved.

File format

Selects the file format to be used, which is Intel-extended by default.

Filename

Specify the destination file to be used; a browse button is available for your convenience.

Save

Saves the selected range of the memory zone to the specified file.

Memory Restore dialog box

The **Memory Restore** dialog box is available by choosing **Debug>Memory>Restore** or from the context menu in the Memory window.

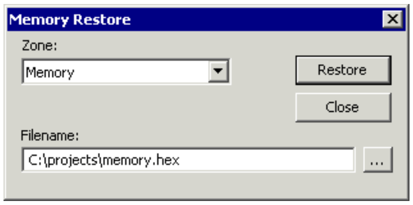


Figure 64: Memory Restore dialog box

Use this dialog box to load the contents of a file in Intel-extended or Motorola S-record format to a specified memory zone.

Requirements

None; this dialog box is always available.

Zone

Selects a memory zone, see *C-SPY memory zones*, page 130.

Filename

Specify the file to be read; a browse button is available for your convenience.

Restore

Loads the contents of the specified file to the selected memory zone.

Fill dialog box

The **Fill** dialog box is available from the context menu in the Memory window.

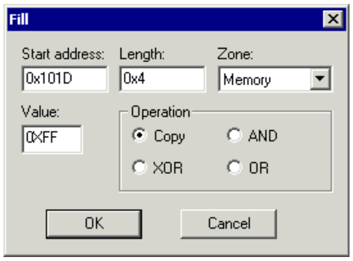


Figure 65: Fill dialog box

Use this dialog box to fill a specified area of memory with a value.

Requirements

None; this dialog box is always available.

Start address

Type the start address—in binary, octal, decimal, or hexadecimal notation.

Length

Type the length—in binary, octal, decimal, or hexadecimal notation.

Zone

Selects a memory zone, see *C-SPY memory zones*, page 130.

Value

Type the 8-bit value to be used for filling each memory location.

Operation

These are the available memory fill operations:

Copy

Value will be copied to the specified memory area.

AND

An `AND` operation will be performed between Value and the existing contents of memory before writing the result to memory.

XOR

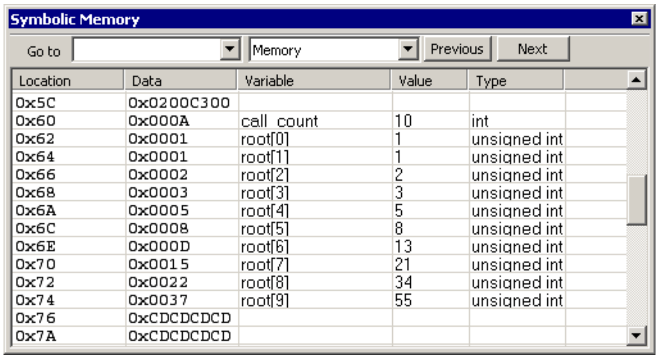
An `XOR` operation will be performed between Value and the existing contents of memory before writing the result to memory.

OR

An `OR` operation will be performed between Value and the existing contents of memory before writing the result to memory.

Symbolic Memory window

The Symbolic Memory window is available from the **View** menu during a debug session.



Location	Data	Variable	Value	Type
0x5C	0x0200C300			
0x60	0x000A	call count	10	int
0x62	0x0001	root[0]	1	unsigned int
0x64	0x0001	root[1]	1	unsigned int
0x66	0x0002	root[2]	2	unsigned int
0x68	0x0003	root[3]	3	unsigned int
0x6A	0x0005	root[4]	5	unsigned int
0x6C	0x0008	root[5]	8	unsigned int
0x6E	0x000D	root[6]	13	unsigned int
0x70	0x0015	root[7]	21	unsigned int
0x72	0x0022	root[8]	34	unsigned int
0x74	0x0037	root[9]	55	unsigned int
0x76	0xCDCDCDCD			
0x7A	0xCDCDCDCD			

Figure 66: Symbolic Memory window

This window displays how variables with static storage duration, typically variables with file scope but also static variables in functions and classes, are laid out in memory. This can be useful for better understanding memory usage or for investigating problems caused by variables being overwritten, for example buffer overruns. Other areas of use are spotting alignment holes or for understanding problems caused by buffers being overwritten.

To view the memory corresponding to a variable, you can select it in the editor window and drag it to the Symbolic Memory window.

Requirements

None; this window is always available.

Toolbar

The toolbar contains:

Go to

The memory location or symbol you want to view.

Zone

Selects a memory zone, see *C-SPY memory zones*, page 130.

Previous

Highlights the previous symbol in the display area.

Next

Highlights the next symbol in the display area.

Display area

This area contains these columns:

Location

The memory address.

Data

The memory contents in hexadecimal format. The data is grouped according to the size of the symbol. This column is editable.

Variable

The variable name; requires that the variable has a fixed memory location. Local variables are not displayed.

Value

The value of the variable. This column is editable.

Type

The type of the variable.

There are several different ways to navigate within the memory space:

- Text that is dropped in the window is interpreted as symbols
- The scroll bar at the right-side of the window
- The toolbar buttons **Next** and **Previous**
- The toolbar list box **Go to** can be used for locating specific locations or symbols.

Note: Rows are marked in red when the corresponding value has changed.

Context menu

This context menu is available:

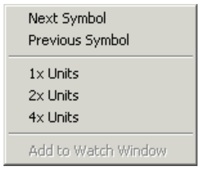


Figure 67: Symbolic Memory window context menu

These commands are available:

Next Symbol

Highlights the next symbol in the display area.

Previous Symbol

Highlights the previous symbol in the display area.

1x Units

Displays the memory contents as single bytes. This applies only to rows which do not contain a variable.

2x Units

Displays the memory contents as 2-byte groups.

4x Units

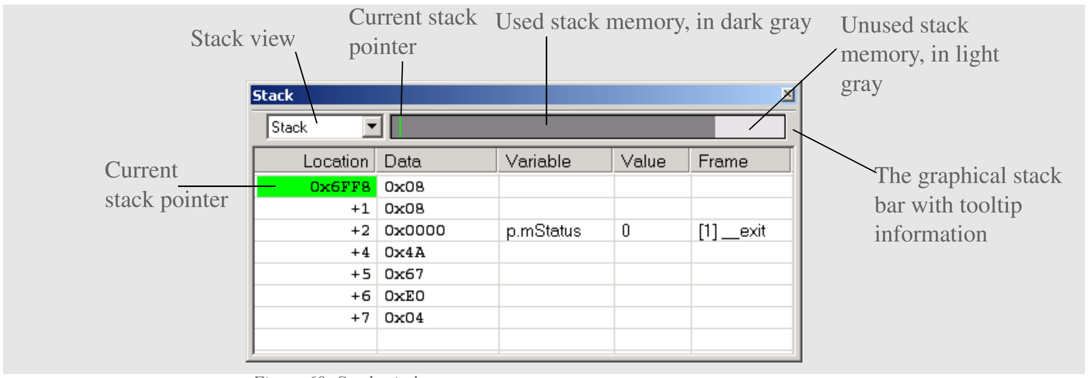
Displays the memory contents as 4-byte groups.

Add to Watch Window

Adds the selected symbol to the Watch window.

Stack window

The Stack window is available from the **View** menu.


Stack view
Current stack pointer
Used stack memory, in dark gray
Unused stack memory, in light gray
Current stack pointer
The graphical stack bar with tooltip information
Stack
Stack

Location	Data	Variable	Value	Frame
0x6FF8	0x08			
+1	0x08			
+2	0x0000	p.mStatus	0	[1] __exit
+4	0x4A			
+5	0x67			
+6	0xE0			
+7	0x04			

Figure 68: Stack window

This window is a memory window that displays the contents of the stack. In addition, some integrity checks of the stack can be performed to detect and warn about problems with stack overflow. For example, the Stack window is useful for determining the optimal size of the stack.

To view the graphical stack bar:

1 Choose **Tools>Options>Stack**.

2 Select the option **Enable graphical stack display and stack usage**.

You can open up to two Stack windows, each showing a different stack—if several stacks are available—or the same stack with different display settings.

Note: By default, this window uses one physical breakpoint. For more information, see *Breakpoint consumers*, page 105.

For information about options specific to the Stack window, see the *IDE Project Management and Building Guide.*

Requirements

None; this window is always available.

Toolbar

The toolbar contains:

Stack

Selects which stack to view. This applies to microcomputers with multiple stacks.

The graphical stack bar

Displays the state of the stack graphically.

The left end of the stack bar represents the bottom of the stack, in other words, the position of the stack pointer when the stack is empty. The right end represents the end of the memory space reserved for the stack. The graphical stack bar turns red when the stack usage exceeds a threshold that you can specify.

When the stack bar is enabled, the functionality needed to detect and warn about stack overflows is also enabled.

Place the mouse pointer over the stack bar to get tooltip information about stack usage.

Display area

This area contains these columns:

Location

Displays the location in memory. The addresses are displayed in increasing order. The address referenced by the stack pointer, in other words the top of the stack, is highlighted in a green color.

Data

Displays the contents of the memory unit at the given location. From the Stack window context menu, you can select how the data should be displayed; as a 1-, 2-, or 4-byte group of data.

Variable

Displays the name of a variable, if there is a local variable at the given location. Variables are only displayed if they are declared locally in a function, and located on the stack and not in registers.

Value

Displays the value of the variable that is displayed in the **Variable** column.

Frame

Displays the name of the function that the call frame corresponds to.

Context menu

This context menu is available:

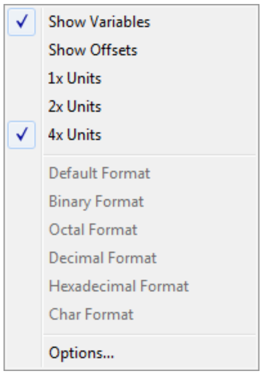


Figure 69: Stack window context menu

These commands are available:

Show variables

Displays separate columns named **Variables**, **Value**, and **Frame** in the Stack window. Variables located at memory addresses listed in the Stack window are displayed in these columns.

Show offsets

Displays locations in the **Location** column as offsets from the stack pointer. When deselected, locations are displayed as absolute addresses.

1x Units

Displays the memory contents as single bytes.

2x Units

Displays the memory contents as 2-byte groups.

4x Units

Displays the memory contents as 4-byte groups.

Default Format,
Binary Format,
Octal Format,
Decimal Format,
Hexadecimal Format,
Char Format

Changes the display format of expressions. The display format setting affects different types of expressions in different ways. Your selection of display format is saved between debug sessions. These commands are available if a selected line in the window contains a variable.

The display format setting affects different types of expressions in these ways:

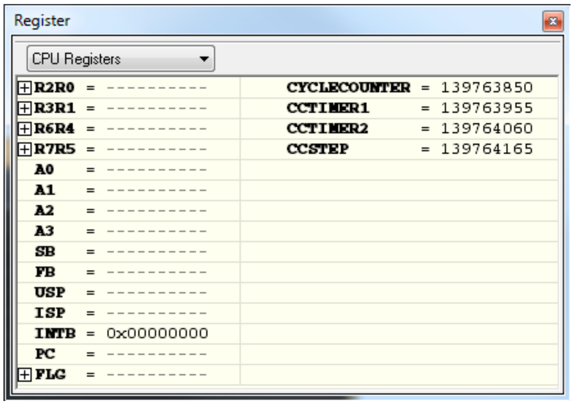
Variables	The display setting affects only the selected variable, not other variables.
Array elements	The display setting affects the complete array, that is, the same display format is used for each array element.
Structure fields	All elements with the same definition—the same field name and C declaration type—are affected by the display setting.

Options

Opens the **IDE Options** dialog box where you can set options specific to the Stack window, see the *IDE Project Management and Building Guide*.

Register window

The Register window is available from the **View** menu.



```
⊞R2R0 = ----------        CYCLECOUNTER = 139763850
⊞R3R1 = ----------        CCTIMER1     = 139763955
⊞R6R4 = ----------        CCTIMER2     = 139764060
⊞R7R5 = ----------        CCSTEP       = 139764165
 A0   = ----------
 A1   = ----------
 A2   = ----------
 A3   = ----------
 SB   = ----------
 FB   = ----------
 USP  = ----------
 ISP  = ----------
 INTB = 0x00000000
 PC   = ----------
⊞FLG  = ----------
```

Figure 70: Register window

This window gives an up-to-date display of the contents of the processor registers and special function registers, and allows you to edit their contents. Optionally, you can choose to load either predefined register groups or to define your own application-specific groups.

You can open several instances of this window, which is very convenient if you want to keep track of different register groups.

To enable predefined register groups:

1 Select a device description file that suits your device, see *Selecting a device description file*, page 33.

2 The register groups appear in the Register window, provided that they are defined in the device description file. Note that the available register groups are also listed on the **Register Filter** page.

To define application-specific register groups:

See *Defining application-specific register groups*, page 133.

Requirements

None; this window is always available.

Toolbar

The toolbar contains:

CPU Registers

Selects which register group to display, by default CPU Registers. Additional register groups are predefined in the device description files that make SFR registers available in the register window. The device description file contains a section that defines the special function registers and their groups. If some of your SFRs are missing, you can register your own SFRs in a Custom group, see *SFR Setup window*, page 150.

Display area

Displays registers and their values. Every time C-SPY stops, a value that has changed since the last stop is highlighted. To edit the contents of a register, click it, and modify the value.

Some registers are expandable, which means that the register contains interesting bits or subgroups of bits.

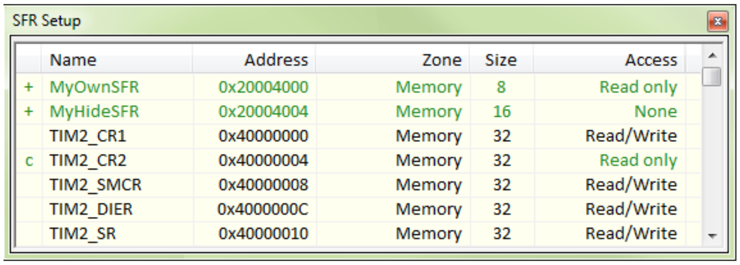
To change the display format, change the **Base** setting on the **Register Filter** page—available by choosing **Tools>Options**.

If you are using the simulator, these additional support registers are available in the CPU Registers group:

CYCLECOUNTER	Cleared when an application is started or reset and is incremented with the number of used cycles during instruction simulation. This register is also available if you are using the E30/E30A emulator driver.
CCSTEP	Shows the number of used cycles during the last performed C/C++ source or assembler step.
CCTIMER 1 and **CCTIMER2**	Two *trip counts* that can be cleared manually at any given time. They are incremented with the number of used cycles during instruction simulation.

SFR Setup window

The SFR Setup window is available from the context menu from the **Project** menu.



	Name	Address	Zone	Size	Access
+	MyOwnSFR	0x20004000	Memory	8	Read only
+	MyHideSFR	0x20004004	Memory	16	None
	TIM2_CR1	0x40000000	Memory	32	Read/Write
c	TIM2_CR2	0x40000004	Memory	32	Read only
	TIM2_SMCR	0x40000008	Memory	32	Read/Write
	TIM2_DIER	0x4000000C	Memory	32	Read/Write
	TIM2_SR	0x40000010	Memory	32	Read/Write

Figure 71: SFR Setup window

This window displays the currently defined SFRs that C-SPY has information about. You can choose to display only factory-defined or custom-defined SFRs, or both. If required, you can use this window to customize the aspects of the SFRs. For factory-defined SFRs (that is, retrieved from the `ddf` file that is currently used), you can only customize the access type.

Any custom-defined SFRs are added to a dedicated register group called Custom, which you can choose to display in the Register window. Your custom-defined SFRs are saved in *`project`*`CustomSFR.sfr`.

You can only add or modify SFRs when the C-SPY debugger is not running.

Requirements

None; this window is always available.

Display area

This area contains these columns:

Status

A character that signals the status of the SFR, which can be one of:

blank, a factory-defined SFR.

`C`, a factory-defined SFR that has been modified.

`+`, a custom-defined SFR.

`?`, an SFR that is ignored for some reason. An SFR can be ignored when a factory-defined SFR has been modified, but the SFR is no longer available, or it is located somewhere else or with a different size. Typically, this might happen if you change to another device.

Name

A unique name of the SFR.

Address

The memory address of the SFR.

Zone

Selects a memory zone, see *C-SPY memory zones*, page 130.

Size

The size of the register, which can be any of `8`, `16`, `32`, or `64`.

Access

The access type of the register, which can be one of `Read/Write`, `Read only`, `Write only`, or `None`.

You can click a name or an address to change the value. The hexadecimal `0x` prefix for the address can be omitted, the value you enter will still be interpreted as hexadecimal. For example, if you enter `4567`, you will get `0x4567`.

You can click a column header to sort the SFRs according to the column property.

Color coding used in the display area:

- Green, which indicates that the corresponding value has changed
- Red, which indicates an ignored SFR.

Context menu

This context menu is available:

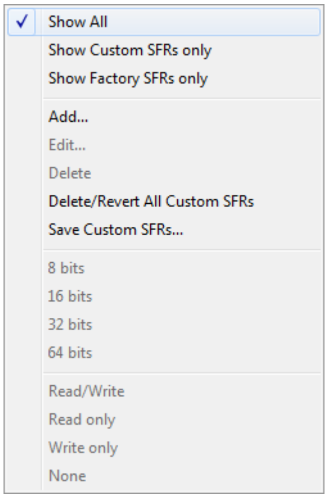


Figure 72: SFR Setup window context menu

These commands are available:

Show All

Shows all SFR.

Show Custom SFRs only

Shows all custom-defined SFRs.

Show Factory SFRs only

Shows all factory-defined SFRs retrieved from the ddf file.

Add

Displays the **Edit SFR** dialog box where you can add a new SFR, see *Edit SFR dialog box*, page 153.

Edit

Displays the **Edit SFR** dialog box where you can edit an SFR, see *Edit SFR dialog box*, page 153.

Delete

Deletes an SFR. This command only works on custom-defined SFRs.

Delete/revert All Custom SFRs

Deletes all custom-defined SFRs and reverts all modified factory-defined SFRs to their factory settings.

Save Custom SFRs

Opens a standard save dialog box to save all custom-defined SFRs.

8|16|32|64 bits

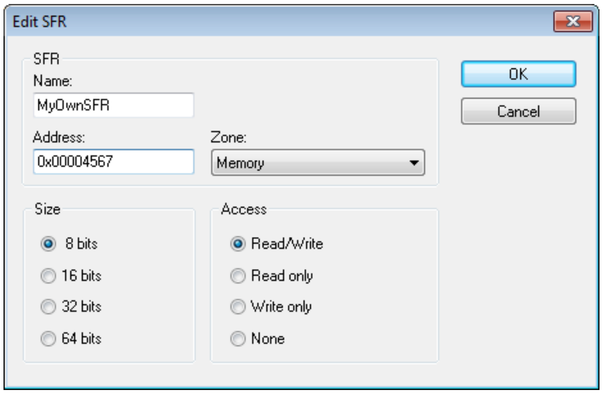
Selects display format for the selected SFR, which can be 8, 16, 32, or 64 bits. Note that the display format can only be changed for custom-defined SFRs.

Read/Write|Read only|Write only|None

Selects the access type of the selected SFR, which can be **Read/Write**, **Read only**, **Write only**, or **None**. Note that for factory-defined SFRs, the default access type is indicated.

Edit SFR dialog box

The **Edit SFR** dialog box is available from the SFR Setup window.



Name: MyOwnSFR

Address: 0x00004567

Zone: Memory

Size: [✓ 8 bits] 16 bits 32 bits 64 bits

Access: [✓ Read/Write] Read only Write only None

Figure 73: Edit SFR dialog box

Use this dialog box to define the SFRs.

Requirements

None; this window is always available.

Name

Specify the name of the SFR that you want to add or edit.

Address

Specify the address of the SFR that you want to add or edit. The hexadecimal `0x` prefix for the address can be omitted, the value you enter will still be interpreted as hexadecimal. For example, if you enter `4567`, you will get `0x4567`.

Zone

Selects the memory zone for the SFR you want to add or edit. The list of zones is retrieved from the `ddf` file that is currently used.

Size

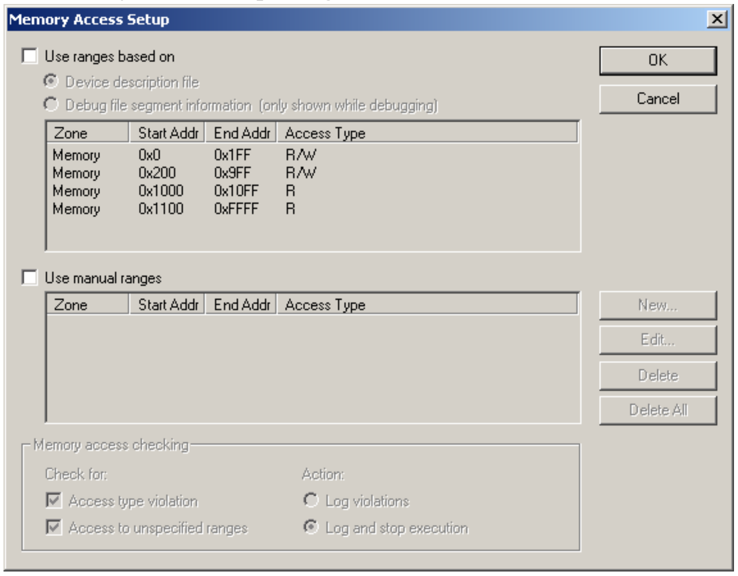
Selects the size of the SFR. Choose between 8, 16, 32, or 64 bits. Note that the display format can only be changed for custom-defined SFRs.

Access

Selects the access type of the SFR. Choose between **Read/Write**, **Read only**, **Write only**, or **None**. Note that for factory-defined SFRs, the default access type is indicated.

Memory Access Setup dialog box

The **Memory Access Setup** dialog box is available from the C-SPY driver menu.



Memory Access Setup

Use ranges based on

[✓ Device description file] Debug file segment information (only shown while debugging)

Zone	Start Addr	End Addr	Access Type
Memory	0x0	0x1FF	R/W
Memory	0x200	0x9FF	R/W
Memory	0x1000	0x10FF	R
Memory	0x1100	0xFFFF	R

Use manual ranges

Zone	Start Addr	End Addr	Access Type

Memory access checking

Check for:

[✓] Access type violation

[✓] Access to unspecified ranges

Action:

Log violations [✓ Log and stop execution]

Figure 74: Memory Access Setup dialog box

This dialog box lists all defined memory areas, where each column in the list specifies the properties of the area. In other words, the dialog box displays the memory access setup that will be used during the simulation.

Note: If you enable both the **Use ranges based on** and the **Use manual ranges** option, memory accesses are checked for all defined ranges.

For information about the columns and the properties displayed, see *Edit Memory Access dialog box*, page 156.

Requirements

The C-SPY simulator.

Use ranges based on

Selects any of the predefined alternatives for the memory access setup. Choose between:

Device description file
Loads properties from the device description file.

Debug file segment information
Properties are based on the segment information available in the debug file. This information is only available while debugging. The advantage of using this option, is that the simulator can catch memory accesses outside the linked application.

Use manual ranges

Specify your own ranges manually via the **Edit Memory Access** dialog box. To open this dialog box, choose **New** to specify a new memory range, or select a memory zone and choose **Edit** to modify it. For more information, see *Edit Memory Access dialog box*, page 156.

The ranges you define manually are saved between debug sessions.

Memory access checking

Check for determines what to check for;

- Access type violation
- Access to unspecified ranges.

Action selects the action to be performed if an access violation occurs; choose between:

- Log violations
- Log and stop execution.

Any violations are logged in the Debug Log window.

Buttons

These buttons are available:

New

Opens the **Edit Memory Access** dialog box, where you can specify a new memory range and attach an access type to it, see *Edit Memory Access dialog box*, page 156.

Edit

Opens the **Edit Memory Access** dialog box, where you can edit the selected memory area. See *Edit Memory Access dialog box*, page 156.

Delete

Deletes the selected memory area definition.

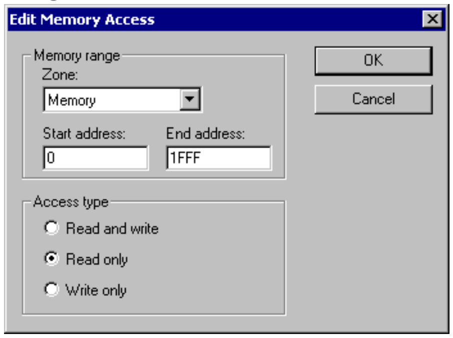
Delete All

Deletes all defined memory area definitions.

Note that except for the OK and Cancel buttons, buttons are only available when the option **Use manual ranges** is selected.

Edit Memory Access dialog box

The **Edit Memory Access** dialog box is available from the **Memory Access Setup** dialog box.



Zone: Memory

Start address: 0

End address: 1FFF

( ) Read and write
(●) Read only
( ) Write only

Figure 75: Edit Memory Access dialog box

Requirements

The C-SPY simulator.

Memory range

Defines the memory area specific to your device:

Zone

Selects a memory zone, see *C-SPY memory zones*, page 130.

Start address

Specify the start address for the memory area, in hexadecimal notation.

End address

Specify the end address for the memory area, in hexadecimal notation.

Access type

Selects an access type to the memory range; choose between:

- **Read and write**
- **Read only**
- **Write only**.

Trace

This chapter gives you information about collecting and using trace data in C-SPY®. More specifically, this means:

- Introduction to using trace
- Collecting and using trace data
- Reference information on trace.

Introduction to using trace

This section introduces trace.

These topics are covered:

- Reasons for using trace
- Briefly about trace
- Requirements for using trace.

See also:

- *Getting started using data logging*, page 84
- *Getting started using interrupt logging*, page 202
- *Profiling*, page 183.

REASONS FOR USING TRACE

By using trace, you can inspect the program flow up to a specific state, for instance an application crash, and use the trace data to locate the origin of the problem. Trace data can be useful for locating programming errors that have irregular symptoms and occur sporadically.

BRIEFLY ABOUT TRACE

Your target system must be able to generate trace data. Once generated, C-SPY can collect it and you can visualize and analyze the data in various windows and dialog boxes.

Depending on your target system, different types of trace data can be generated.

Trace data is a continuously collected sequence of every executed instruction for a selected portion of the execution.

Trace features in C-SPY

In C-SPY, you can use the trace-related windows Trace, Function Trace, Timeline, and Find in Trace. In the C-SPY simulator, you can also use the Trace Expressions window. Depending on your C-SPY driver, you can set various types of trace breakpoints and triggers to control the collection of trace data.

If you use the C-SPY Simulator driver, you have access to windows such as the Interrupt Log, Interrupt Log Summary, Data Log, and Data Log Summary windows.

In addition, several other features in C-SPY also use trace data, features such as the Profiler, Code coverage, and Instruction profiling.

REQUIREMENTS FOR USING TRACE

The C-SPY simulator supports trace-related functionality, and there are no specific requirements.

To use trace in your hardware debugger system, you need one of these alternatives:

- An E30 emulator for limited trace.
- An E30A emulator for assembler trace.

Note: The specific set of debug components you are using (hardware, a debug probe, and a C-SPY driver) determine which trace features in C-SPY that are supported.

Collecting and using trace data

This section describes various tasks related to collecting and using trace data.

More specifically, you will get information about:

- Getting started with trace
- Trace data collection using breakpoints
- Searching in trace data
- Browsing through trace data.

GETTING STARTED WITH TRACE

1 Before you start C-SPY:

- For the C-SPY simulator, no specific build settings are required before starting C-SPY.

- For the C-SPY E30A emulator, you must choose **E30A Emulator>Hardware Setup** and select **Trace** as the **Emulator mode**. If you are using the E30 emulator, you do not need to perform this.

  Note that the pins used on the hardware for the trace signals cannot be used by your application.

2 Start C-SPY and choose **Events** from the C-SPY driver menu. In the **Events** window, right-click and choose **New Trace Event**. In the **Trace Event** dialog box that appears, create a new trace event.

If you are using the C-SPY simulator you can ignore this step.

3 Open the Trace window—available from the driver-specific menu—and click the **Activate** button to enable collecting trace data. For the E30 emulator driver, it is always activated.

4 Start the execution. When the execution stops, for instance because a breakpoint is triggered, trace data is displayed in the Trace window. For more information about the window, see *Trace window*, page 165.

TRACE DATA COLLECTION USING BREAKPOINTS

A convenient way to collect trace data between two execution points is to start and stop the data collection using dedicated breakpoints. Choose between these alternatives:

- In the editor or Disassembly window, position your insertion point, right-click, and toggle a **Trace Start** or **Trace Stop** breakpoint from the context menu.
- In the Breakpoints window, choose **Trace Start** or **Trace Stop**.
- The C-SPY system macros `__setTraceStartBreak` and `__setTraceStopBreak` can also be used.

For more information about these breakpoints, see *Trace Start breakpoints dialog box*, page 176 and *Trace Stop breakpoints dialog box*, page 177, respectively.

SEARCHING IN TRACE DATA

When you have collected trace data, you can perform searches in the collected data to locate the parts of your code or data that you are interested in, for example, a specific interrupt or accesses of a specific variable.

You specify the search criteria in the **Find in Trace** dialog box and view the result in the Find in Trace window.

Note: The Find in Trace window is very similar to the Trace window, showing the same columns and data, but *only* those rows that match the specified search criteria. Double-clicking an item in the Find in Trace window brings up the same item in the Trace window.

To search in your trace data:

1. On the Trace window toolbar, click the **Find** button.
2. In the **Find in Trace** dialog box, specify your search criteria.

   Typically, you can choose to search for:

   - A specific piece of text, for which you can apply further search criteria
   - An address range
   - A combination of these, like a specific piece of text within a specific address range.

   For more information about the various options, see *Find in Trace dialog box*, page 179.
3. When you have specified your search criteria, click **Find**. The Find in Trace window is displayed, which means you can start analyzing the trace data. For more information, see *Find in Trace window*, page 181.

BROWSING THROUGH TRACE DATA

To follow the execution history, simply look and scroll in the Trace window. Alternatively, you can enter *browse mode*.

To enter browse mode, double-click an item in the Trace window, or click the **Browse** toolbar button.

The selected item turns yellow and the source and disassembly windows will highlight the corresponding location. You can now move around in the trace data using the up and down arrow keys, or by scrolling and clicking; the source and Disassembly windows will be updated to show the corresponding location. This is like stepping backward and forward through the execution history.

Double-click again to leave browse mode.

Reference information on trace

This section gives reference information about these windows and dialog boxes:

- *Trace Event dialog box*, page 163
- *Trace window*, page 165
- *Function Trace window*, page 168
- *Trace Expressions window*, page 178
- *Timeline window*, page 168
- *Viewing Range dialog box*, page 175
- *Trace Start breakpoints dialog box*, page 176

- *Trace Stop breakpoints dialog box*, page 177
- *Trace Expressions window*, page 178
- *Find in Trace dialog box*, page 179
- *Find in Trace window*, page 181.

Trace Event dialog box

The **Trace Event** dialog box is available from the Events window context menu.

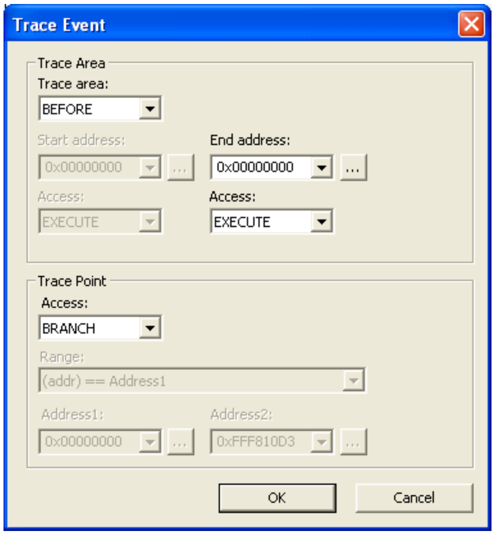


Figure 76: Trace Event dialog box

In this dialog box, you can set up and modify trace events.

A trace event can be either a branch or a data access, or a combination thereof.

Requirements

One of these alternatives:

- A C-SPY E30 emulator
- A C-SPY E30A emulator

Trace Area

Use these options to specify the trace area modes:

Trace area

Selects the execution history that will be listed in the Trace window. Up to 512 cycles (branches or data accesses) can be listed with MCU Execution and up to 8,000,000 cycles with Trace priority.

BREAK	Lists trace cycles before the target application stops.
BEFORE	Lists trace cycles before the trace point.
AFTER	Lists trace cycles after the trace point.
FULL	Lists from the start until the trace memory is full.

Start address

The address for the trace start condition when using the AFTER trace area.

End address

The address for the trace end condition when using the BEFORE trace area.

Access

The access condition for the event.

Trace Point

The following **Trace Point** options are available:

Access

The access condition for the event.

Range

The trace point range condition.

`(addr) == Address1` Address match. The address matches Address1 below.

`Address1 <= (addr) <= Address2` Address range. The address is located between Address1 and Address2 below.

Address 1

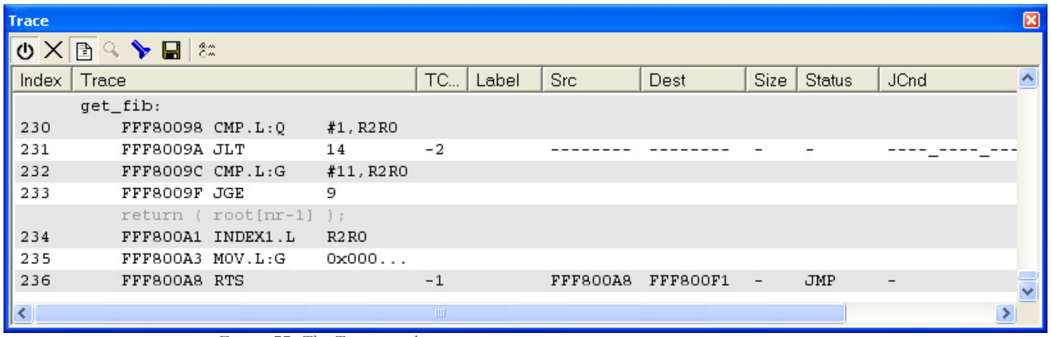
The start address of the trace point address range or the address used for the trace point address match. Use the browse button to specify a predefined symbol instead of a hardwired address. Note that the browse button is not available in the C-SPY E30 driver.

Address 2

The end address of a trace point address range. Use the browse button to specify a predefined symbol instead of a hardwired address. Note that the browse button is not available in the C-SPY E30 driver.

Trace window

The Trace window is available from the C-SPY driver menu.



Index	Trace	TC...	Label	Src	Dest	Size	Status	JCnd
	get_fib:							
230	FFF80098 CMP.L:Q #1,R2R0							
231	FFF8009A JLT 14	-2		--------	--------	-	-	----_----_---
232	FFF8009C CMP.L:G #11,R2R0							
233	FFF8009F JGE 9							
	return (root[nr-1]);							
234	FFF800A1 INDEX1.L R2R0							
235	FFF800A3 MOV.L:G 0x000...							
236	FFF800A8 RTS	-1		FFF800A8	FFF800F1	-	JMP	-

Figure 77: The Trace window

This window displays the collected trace data.

The Trace window depends on the C-SPY driver you are using.

Requirements

One of these alternatives:

- The C-SPY Simulator
- A C-SPY E30 emulator
- A C-SPY E30A emulator

Trace toolbar

The toolbar in the Trace window and in the Function trace window contains:

Enable/Disable

Enables and disables collecting and viewing trace data in this window. This button is not available in the Function trace window.

Clear trace data

Clears the trace buffer. Both the Trace window and the Function trace window are cleared.

Toggle source

Toggles the Trace column between showing only disassembly or disassembly together with the corresponding source code.

Browse

Toggles browse mode on or off for a selected item in the Trace window, see *Browsing through trace data*, page 162.

Find

Displays a dialog box where you can perform a search, see *Find in Trace dialog box*, page 179.

Save

Displays a standard **Save As** dialog box where you can save the collected trace data to a text file, with tab-separated columns.

Edit Settings

In the C-SPY simulator, this button is not enabled.

For C-SPY emulators, this button opens the **Events** window.

Edit Expressions (C-SPY simulator only)

Opens the Trace Expressions window, see *Trace Expressions window*, page 178.

Display area

This area displays a collected sequence of executed machine instructions. In addition, the window can display trace data for expressions.

This area contains these columns for the C-SPY simulator:

#

A serial number for each row in the trace buffer. Simplifies the navigation within the buffer.

Cycles

The number of cycles elapsed to this point.

Trace

The collected sequence of executed machine instructions. Optionally, the corresponding source code can also be displayed.

Expression

Each expression you have defined to be displayed appears in a separate column. Each entry in the expression column displays the value *after* executing the instruction on the same row. You specify the expressions for which you want to collect trace data in the Trace Expressions window, see *Trace Expressions window*, page 178.

Display area (in the C-SPY E30A emulator)

Index

A number that corresponds to each packet. Examples of packets are instructions, synchronization points, and exception markers.

Trace

The collected sequence of executed machine instructions. Optionally, the corresponding source code can also be displayed.

TCNT

Trace cycles, up to either 512 or 8,000,000 depending on used trace mode.

Label

Shows labels corresponding to address bus information.

Src

The state of the data bus.

Dest

The state of the address bus.

Size

The size of the data access:

`B`	8-bit
`W`	16-bit
`L`	32-bit
`Q`	64-bit

Status

Shows the instruction status between the memory and I/O:

`JMP` (jump) is Branch information (also for `RTS`). The address in the Address column is the jump address. The address in the Data column is the branch instruction address.

`RD` (read data) is Data access information.

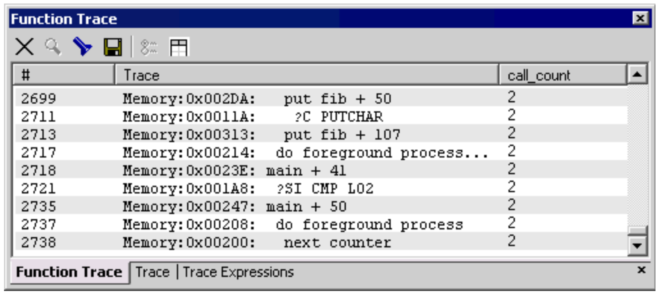
`WD` (write data) is Data access information.

JCnd

For conditional branches; `0` for jumps and `1` for non-jumps.

Function Trace window

The Function Trace window is available from the C-SPY driver menu during a debug session.



#	Trace	call_count
2699	Memory:0x002DA: put fib + 50	2
2711	Memory:0x0011A: ?C PUTCHAR	2
2713	Memory:0x00313: put fib + 107	2
2717	Memory:0x00214: do foreground process...	2
2718	Memory:0x0023E: main + 41	2
2721	Memory:0x001A8: ?SI CMP L02	2
2735	Memory:0x00247: main + 50	2
2737	Memory:0x00208: do foreground process	2
2738	Memory:0x00200: next counter	2

Figure 78: Function Trace window

This window displays a subset of the trace data displayed in the Trace window. Instead of displaying all rows, the Function Trace window only shows trace data corresponding to calls to and returns from functions.

Requirements

The C-SPY simulator

Toolbar

For information about the toolbar, see *Trace window*, page 165.

Display area

For information about the columns in the display area, see *Trace window*, page 165

Timeline window

The Timeline window is available from the C-SPY driver menu during a debug session.

This window displays trace data in different graphs in relation to a common time axis:

- Call Stack graph
- Data Log graph

- Interrupt Log graph

To display a graph:

1. Choose **Timeline** from the C-SPY driver menu to open the Timeline window.
2. In the Timeline window, click in the graph area and choose **Enable** from the context menu to enable a specific graph.
3. For the Data Log Graph, you need to set a Data Log breakpoint for each variable you want a graphical representation of in the Timeline window. See *Data Log breakpoints dialog box*, page 124.
4. Click **Go** on the toolbar to start executing your application. The graph appears.

To navigate in the graph, use any of these alternatives:

- Right-click and from the context menu choose **Zoom In** or **Zoom Out**. Alternatively, use the + and - keys. The graph zooms in or out depending on which command you used.
- Right-click in the graph and from the context menu choose **Navigate** and the appropriate command to move backwards and forwards on the graph. Alternatively, use any of the shortcut keys: arrow keys, Home, End, and Ctrl+End.
- Double-click on a sample of interest and the corresponding source code is highlighted in the editor window and in the Disassembly window.
- Click on the graph and drag to select a time interval. Press Enter or right-click and from the context menu choose **Zoom>Zoom to Selection**. The selection zooms in.

Point in the graph with the mouse pointer to get detailed tooltip information for that location.

Requirements

The display area can be populated with different graphs:

Graphs	Call Stack Graph	Data Log Graph	Interrupt Log Graph
C-SPY simulator	X	X	X
C-SPY hardware drivers	--	--	--

Table 9: Supported graphs in the Timeline window

For more information about requirements related to trace data, see *Requirements for using trace*, page 160.

Display area for the Call Stack Graph

The Call Stack Graph displays the sequence of calls and returns collected by trace.

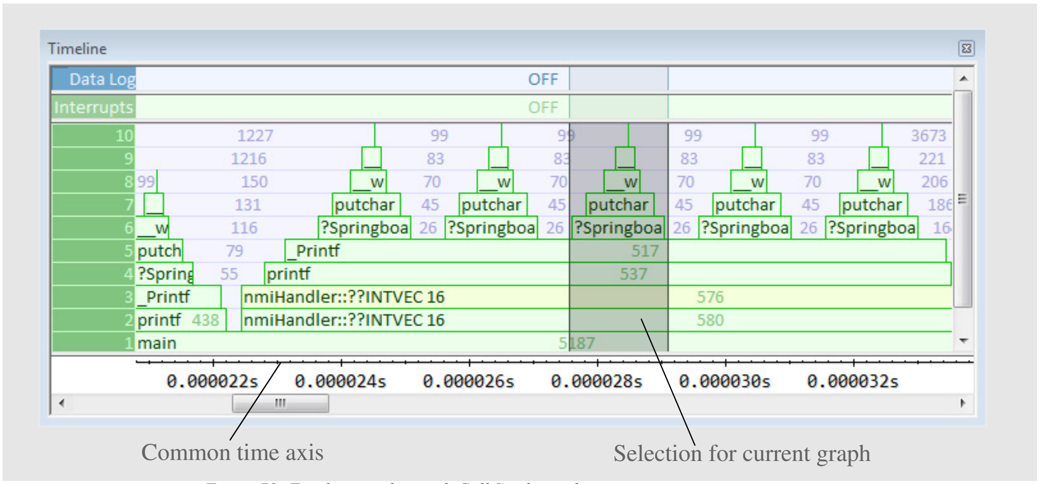


Figure 79: Timeline window with Call Stack graph

At the bottom of the graph you will usually find `main`, and above it, the functions called from `main`, and so on. The horizontal bars, which represent invocations of functions, use four different colors:

- Medium green for normal C functions with debug information
- Light green for functions known to the debugger only through an assembler label
- Medium or light yellow for interrupt handlers, with the same distinctions as for green.

The numbers represent the number of cycles spent in, or between, the function invocations.

At the bottom of the window, there is a common time axis that uses seconds as the time unit.

Display area for the Data Log graph

The Data Log graph displays the data logs generated by trace, for up to four different variables or address ranges specified as Data Log breakpoints.

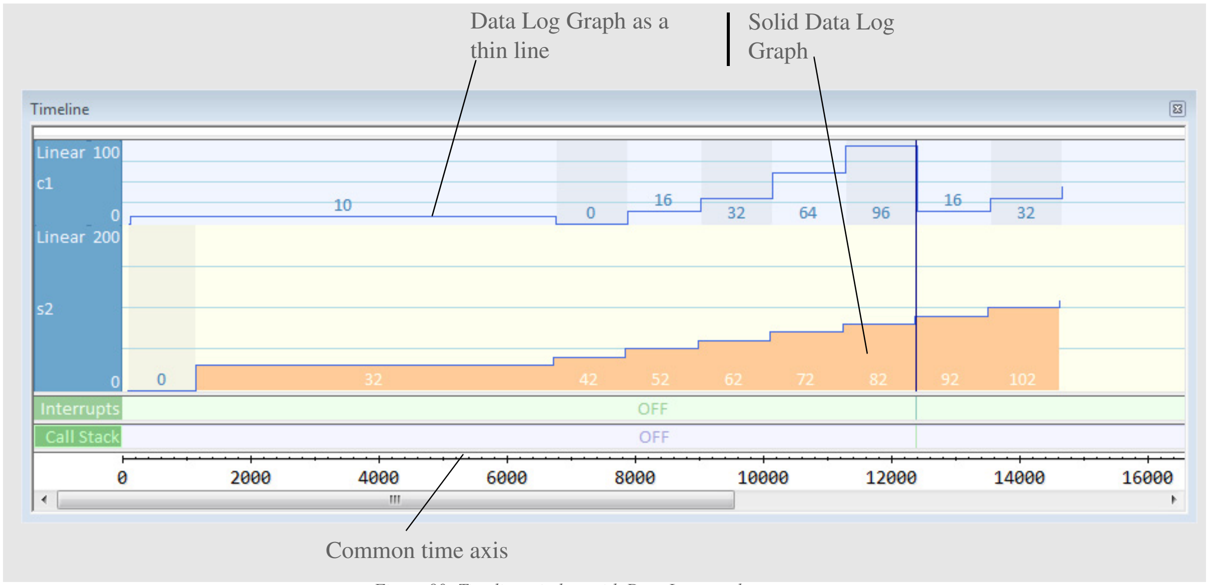


Figure 80: Timeline window with Data Log graph

Where:

- Each graph is labeled with—in the left-side area—the variable name or address for which you have specified the Data Log breakpoint.
- The graph itself displays how the value of the variable changes over time. The label area also displays the limits, or range, of the Y-axis for a variable. You can use the context menu to change these limits. The graph is a graphical representation of the information in the Data Log window, see *Data Log window*, page 96.
- The graph can be displayed either as a thin line or as a color-filled solid graph.
- A red vertical line indicates overflow, which means that the communication channel failed to transmit all data logs from the target system.

At the bottom of the window, there is a common time axis that uses seconds as the time unit.

Display area for the Interrupt Log graph

The Interrupt Log graph displays interrupts reported by the C-SPY simulator. In other words, the graph provides a graphical view of the interrupt events during the execution of your application.

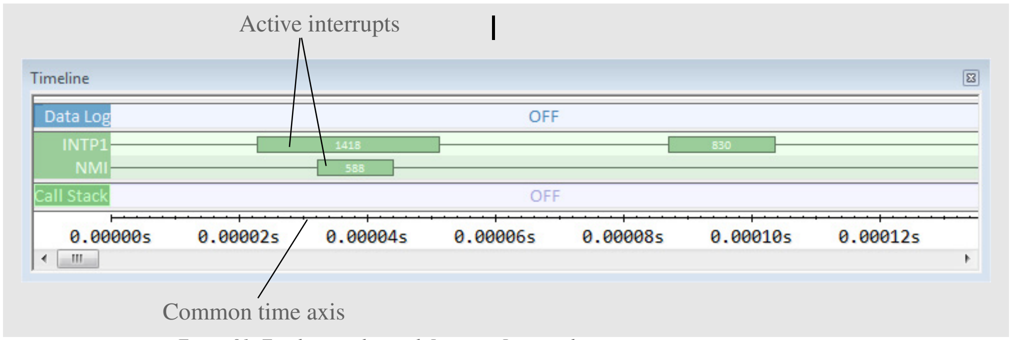


Figure 81: Timeline window with Interrupt Log graph

Where:

- The label area at the left end of the graph shows the names of the interrupts.
- The graph itself shows active interrupts as a thick green horizontal bar where the white figure indicates the time spent in the interrupt. This graph is a graphical representation of the information in the Interrupt Log window, see *Interrupt Log window*, page 209.

At the bottom of the window, there is a common time axis that uses seconds as the time unit.

Selection and navigation

Click and drag to select. The selection extends vertically over all graphs, but appears highlighted in a darker color for the selected graph. You can navigate backward and forward in the selected graph using the left and right arrow keys. Use the Home and End keys to move to the first or last relevant point, respectively. Use the navigation keys in combination with the Shift key to extend the selection.

Context menu

This context menu is available:

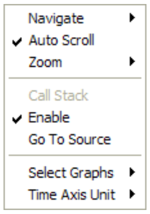


Figure 82: Timeline window context menu (for the Call Stack Graph context menu)

Note: The context menu contains some commands that are common to all graphs and some commands that are specific to each graph. The figure reflects the context menu for the Call Stack Graph, which means that the menu looks slightly different for the other graphs.

These commands are available:

Navigate (All graphs)

Commands for navigating over the graph(s); choose between:

Next moves the selection to the next relevant point in the graph. Shortcut key: right arrow.

Previous moves the selection backward to the previous relevant point in the graph. Shortcut key: left arrow.

First moves the selection to the first data entry in the graph. Shortcut key: Home.

Last moves the selection to the last data entry in the graph. Shortcut key: End.

End moves the selection to the last data in any displayed graph, in other words the end of the time axis. Shortcut key: Ctrl+End.

Auto Scroll (All graphs)

Toggles auto scrolling on or off. When on, the most recently collected data is automatically displayed if you have executed the command **Navigate>End**.

Zoom (All graphs)

Commands for zooming the window, in other words, changing the time scale; choose between:

Zoom to Selection makes the current selection fit the window. Shortcut key: Return.

Zoom In zooms in on the time scale. Shortcut key: +.

Zoom Out zooms out on the time scale. Shortcut key: -.

10ns, **100ns**, **1us**, etc makes an interval of 10 nanoseconds, 100 nanoseconds, 1 microsecond, respectively, fit the window.

1ms, **10ms**, etc makes an interval of 1 millisecond or 10 milliseconds, respectively, fit the window.

10m, **1h**, etc makes an interval of 10 minutes or 1 hour, respectively, fit the window.

Data Log (Data Log Graph)

A heading that shows that the Data Log-specific commands below are available.

Call Stack (Call Stack Graph)

A heading that shows that the Call stack-specific commands below are available.

Interrupt (Interrupt Log Graph)

A heading that shows that the Interrupt Log-specific commands below are available.

Enable (All graphs)

Toggles the display of the graph on or off. If you disable a graph, that graph will be indicated as `OFF` in the Timeline window. If no trace data has been collected for a graph, *no data* will appear instead of the graph.

Variable (Data Log Graph)

The name of the variable for which the Data Log-specific commands below apply. This menu command is context-sensitive, which means it reflects the Data Log Graph you selected in the Timeline window (one of up to four).

Solid Graph (Data Log Graph)

Displays the graph as a color-filled solid graph instead of as a thin line.

Viewing Range (Data Log Graph)

Displays a dialog box, see *Viewing Range dialog box*, page 175.

Size (Data Log Graph)

Determines the vertical size of the graph; choose between **Small**, **Medium**, and **Large**.

Show Numerical Value (Data Log Graph)

Shows the numerical value of the variable, in addition to the graph.

Go To Source (Common)

Displays the corresponding source code in an editor window, if applicable.

Select Graphs (Common)

Selects which graphs to be displayed in the Timeline window.

Time Axis Unit (Common)

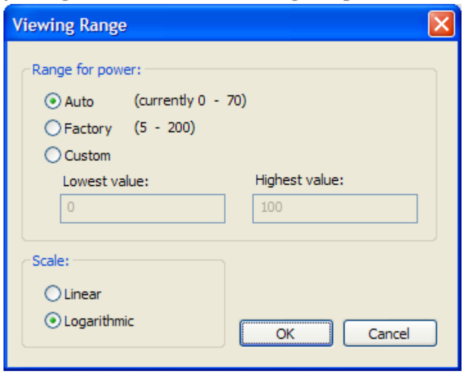
Selects the unit used in the time axis; choose between **Seconds** and **Cycles**.

Profile Selection

Enables profiling time intervals in the Function Profiler window. Note that this command is only available if the C-SPY driver supports PC Sampling.

Viewing Range dialog box

The **Viewing Range** dialog box is available from the context menu that appears when you right-click in the Data Log Graph in the Timeline window.



Range for power:
[✓ Auto] (currently 0 - 70)
Factory (5 - 200)
Custom
Lowest value: 0
Highest value: 100

Scale:
Linear
[✓ Logarithmic]

Figure 83: Viewing Range dialog box

Use this dialog box to specify the value range, that is, the range for the Y-axis for the graph.

Requirements

The C-SPY simulator

Range for ...

Selects the viewing range for the displayed values:

Auto

Uses the range according to the range of the values that are actually collected, continuously keeping track of minimum or maximum values. The currently computed range, if any, is displayed in parentheses. The range is rounded to reasonably *even* limits.

Factory

For the Data Log Graph: Uses the range according to the value range of the variable, for example 0–65535 for an unsigned 16-bit integer.

Custom

Use the text boxes to specify an explicit range.

Scale

Selects the scale type of the Y-axis:

- **Linear**
- **Logarithmic**.

Trace Start breakpoints dialog box

The **Trace Start** dialog box is available from the context menu that appears when you right-click in the Breakpoints window.

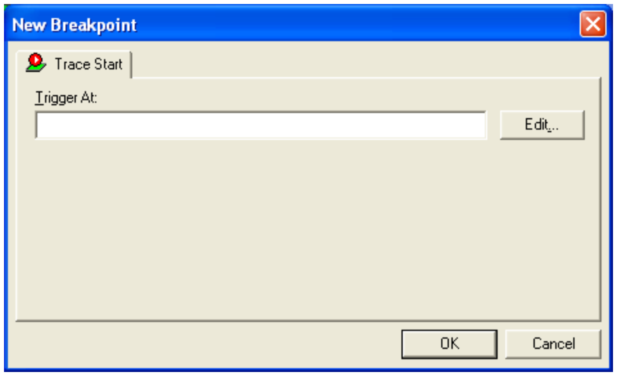


Figure 84: Trace Start breakpoints dialog box

Use this dialog box to set a Trace Start breakpoint where you want to start collecting trace data. If you want to collect trace data only for a specific range, you must also set a Trace Stop breakpoint where you want to stop collecting data.

See also, *Trace Stop breakpoints dialog box*, page 177.

To set a Trace Start breakpoint:

1 In the editor or Disassembly window, right-click and choose **Trace Start** from the context menu.

Alternatively, open the Breakpoints window by choosing **View>Breakpoints**.

2 In the Breakpoints window, right-click and choose **New Breakpoint>Trace Start**.

Alternatively, to modify an existing breakpoint, select a breakpoint in the Breakpoints window and choose **Edit** on the context menu.

3 In the **Trigger At** text box, specify an expression, an absolute address, or a source location. Click **OK**.

4 When the breakpoint is triggered, the trace data collection starts.

Requirements

The C-SPY simulator.

Trigger at

Specify the code location of the breakpoint. Alternatively, click the **Edit** button to open the **Enter Location** dialog box, see *Enter Location dialog box*, page 126.

Trace Stop breakpoints dialog box

The **Trace Stop** dialog box is available from the context menu that appears when you right-click in the Breakpoints window.

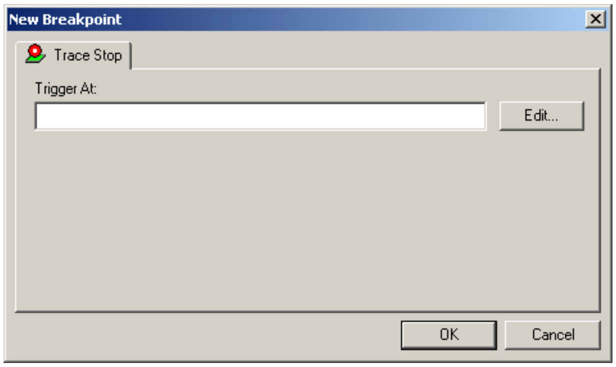


Figure 85: Trace Stop breakpoints dialog box

Use this dialog box to set a Trace Stop breakpoint where you want to stop collecting trace data. If you want to collect trace data only for a specific range, you might also need to set a Trace Start breakpoint where you want to start collecting data.

See also, *Trace Start breakpoints dialog box*, page 176.

To set a Trace Stop breakpoint:

1 In the editor or Disassembly window, right-click and choose **Trace Stop** from the context menu.

Alternatively, open the Breakpoints window by choosing **View>Breakpoints**.

2 In the Breakpoints window, right-click and choose **New Breakpoint>Trace Stop**.

Alternatively, to modify an existing breakpoint, select a breakpoint in the Breakpoints window and choose **Edit** on the context menu.

3 In the **Trigger At** text box, specify an expression, an absolute address, or a source location. Click **OK**.

4 When the breakpoint is triggered, the trace data collection stops.

Requirements

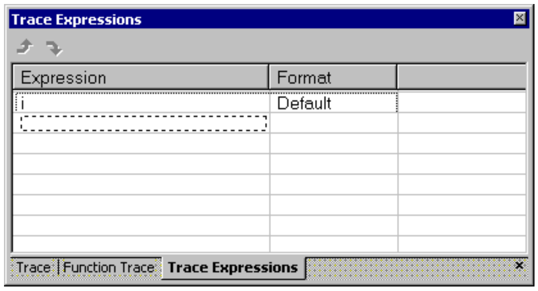
The C-SPY simulator.

Trigger at

Specify the code location of the breakpoint. Alternatively, click the **Edit** button to open the **Enter Location** dialog box, see *Enter Location dialog box*, page 126.

Trace Expressions window

The Trace Expressions window is available from the Trace window toolbar.



Expression	Format
i	Default

Figure 86: Trace Expressions window

Use this window to specify, for example, a specific variable (or an expression) for which you want to collect trace data.

Requirements

The C-SPY simulator.

Toolbar

The toolbar buttons change the order between the expressions:

Arrow up

Moves the selected row up.

Arrow down

Moves the selected row down.

Display area

Use the display area to specify expressions for which you want to collect trace data:

Expression

Specify any expression that you want to collect data from. You can specify any expression that can be evaluated, such as variables and registers.

Format

Shows which display format that is used for each expression. Note that you can change display format via the context menu.

Each row in this area will appear as an extra column in the Trace window.

Find in Trace dialog box

The **Find in Trace** dialog box is available by clicking the **Find** button on the Trace window toolbar or by choosing **Edit>Find and Replace>Find**.

Note that the **Edit>Find and Replace>Find** command is context-dependent. It displays the **Find in Trace** dialog box if the Trace window is the current window or the **Find** dialog box if the editor window is the current window.

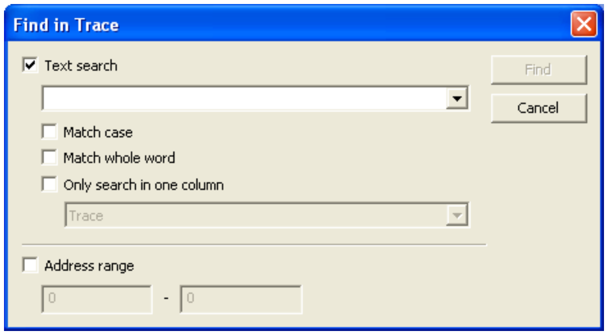


Figure 87: Find in Trace dialog box

Use this dialog box to specify the search criteria for advanced searches in the trace data.

The search results are displayed in the Find in Trace window—available by choosing the **View>Messages** command, see *Find in Trace window*, page 181.

See also *Searching in trace data*, page 161.

Requirements

One of these alternatives:

- The C-SPY Simulator
- A C-SPY E30 emulator
- A C-SPY E30A emulator

Text search

Specify the string you want to search for. To specify the search criteria, choose between:

Match Case

Searches only for occurrences that exactly match the case of the specified text. Otherwise `int` will also find `INT` and `Int` and so on.

Match whole word

Searches only for the string when it occurs as a separate word. Otherwise `int` will also find `print`, `sprintf` and so on.

Only search in one column

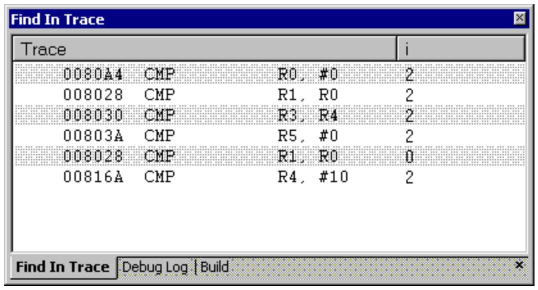
Searches only in the column you selected from the drop-down list.

Address Range

Specify the address range you want to display or search. The trace data within the address range is displayed. If you also have specified a text string in the **Text search** field, the text string is searched for within the address range.

Find in Trace window

The Find in Trace window is available from the **View>Messages** menu. Alternatively, it is automatically displayed when you perform a search using the **Find in Trace** dialog box or perform a search using the **Find in Trace** command available from the context menu in the editor window.



```
0080A4  CMP          R0, #0    2
008028  CMP          R1, R0    2
008030  CMP          R3, R4    2
00803A  CMP          R5, #0    2
008028  CMP          R1, R0    0
00816A  CMP          R4, #10   2
```

Figure 88: Find in Trace window

This window displays the result of searches in the trace data. Double-click an item in the Find in Trace window to bring up the same item in the Trace window.

Before you can view any trace data, you must specify the search criteria in the **Find in Trace** dialog box, see *Find in Trace dialog box*, page 179.

For more information, see *Searching in trace data*, page 161.

Requirements

One of these alternatives:

- The C-SPY Simulator
- A C-SPY E30 emulator
- A C-SPY E30A emulator

Display area

The Find in Trace window looks like the Trace window and shows the same columns and data, but *only* those rows that match the specified search criteria.

Profiling

This chapter describes how to use the profiler in C-SPY®. More specifically, this means:

- Introduction to the profiler
- Using the profiler
- Reference information on the profiler.

Introduction to the profiler

This section introduces the profiler.

These topics are covered:

- Reasons for using the profiler
- Briefly about the profiler
- Requirements for using the profiler.

REASONS FOR USING THE PROFILER

Function profiling can help you find the functions in your source code where the most time is spent during execution. You should focus on those functions when optimizing your code. A simple method of optimizing a function is to compile it using speed optimization. Alternatively, you can move the data used by the function into more efficient memory. For detailed information about efficient memory usage, see the *IAR C/C++ Compiler Reference Guide for R32C*.

Alternatively, you can use *filtered profiling*, which means that you can exclude, for example, individual functions from being profiled. To profile only a specific part of your code, you can select a *time interval*—using the Timeline window—for which C-SPY produces profiling information.

Instruction profiling can help you fine-tune your code on a very detailed level, especially for assembler source code. Instruction profiling can also help you to understand where your compiled C/C++ source code spends most of its time, and perhaps give insight into how to rewrite it for better performance.

BRIEFLY ABOUT THE PROFILER

Function profiling information is displayed in the Function Profiler window, that is, timing information for the functions in an application. Profiling must be turned on explicitly using a button on the window's toolbar, and will stay enabled until it is turned off.

Instruction profiling information is displayed in the Disassembly window, that is, the number of times each instruction has been executed.

Profiling sources

The profiler can use different mechanisms, or *sources*, to collect profiling information. Depending on the available trace source features, one or more of the sources can be used for profiling:

- Trace (calls)

  The full instruction trace is analyzed to determine all function calls and returns. When the collected instruction sequence is incomplete or discontinuous, the profiling information is less accurate.
- Trace (flat)

  Each instruction in the full instruction trace or each PC Sample is assigned to a corresponding function or code fragment, without regard to function calls or returns. This is most useful when the application does not exhibit normal call/return sequences, such as when you are using an RTOS, or when you are profiling code which does not have full debug information.

REQUIREMENTS FOR USING THE PROFILER

The C-SPY simulator supports the profiler; there are no specific requirements.

Using the profiler

This section describes various tasks related to using the profiler.

More specifically, you will get information about:

- Getting started using the profiler on function level
- Getting started using the profiler on instruction level

GETTING STARTED USING THE PROFILER ON FUNCTION LEVEL

To display function profiling information in the Function Profiler window:

1 Build your application using these options:

Category	Setting
C/C++ Compiler	Output>Generate debug information
Linker	Output>Format>Debug information for C-SPY

Table 10: Project options for enabling the profiler

2 When you have built your application and started C-SPY, choose **Driver-menu>Function Profiler** to open the Function Profiler window, and click the **Enable** button to turn on the profiler. Alternatively, choose **Enable** from the context menu that is available when you right-click in the Function Profiler window.

3 Start executing your application to collect the profiling information.

4 Profiling information is displayed in the Function Profiler window. To sort, click on the relevant column header.

5 When you start a new sampling, you can click the **Clear** button—alternatively, use the context menu—to clear the data.

GETTING STARTED USING THE PROFILER ON INSTRUCTION LEVEL

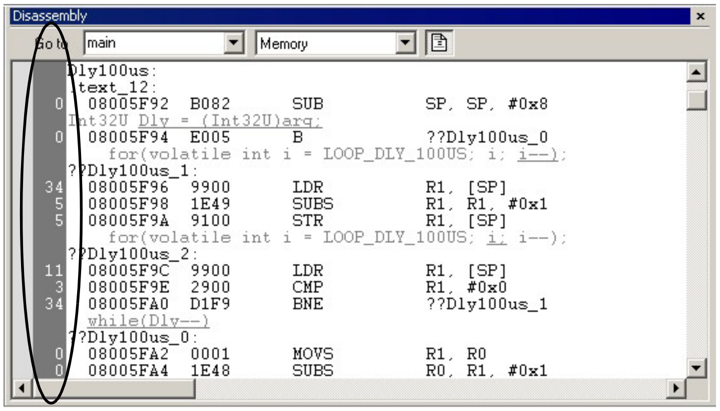
To display instruction profiling information in the Disassembly window:

1 When you have built your application and started C-SPY, choose **View>Disassembly** to open the Disassembly window, and choose **Instruction Profiling>Enable** from the context menu that is available when you right-click in the left-hand margin of the Disassembly window.

2 Make sure that the **Show** command on the context menu is selected, to display the profiling information.

3 Start executing your application to collect the profiling information.

4 When the execution stops, for instance because the program exit is reached or a breakpoint is triggered, you can view instruction level profiling information in the left-hand margin of the window.



Disassembly

Go to main Memory

```
     Dly100us:
     .text_12:
   0  08005F92  B082      SUB         SP, SP, #0x8
     Int32U Dly = (Int32U)arg;
   0  08005F94  E005      B           ??Dly100us_0
        for(volatile int i = LOOP_DLY_100US; i; i--);
     ??Dly100us_1:
  34  08005F96  9900      LDR         R1, [SP]
   5  08005F98  1E49      SUBS        R1, R1, #0x1
   5  08005F9A  9100      STR         R1, [SP]
        for(volatile int i = LOOP_DLY_100US; i; i--);
     ??Dly100us_2:
  11  08005F9C  9900      LDR         R1, [SP]
   3  08005F9E  2900      CMP         R1, #0x0
  34  08005FA0  D1F9      BNE         ??Dly100us_1
      while(Dly--)
     ??Dly100us_0:
   0  08005FA2  0001      MOVS        R1, R0
   0  08005FA4  1E48      SUBS        R0, R1, #0x1
```

Figure 89: Instruction count in Disassembly window

For each instruction, the number of times it has been executed is displayed.

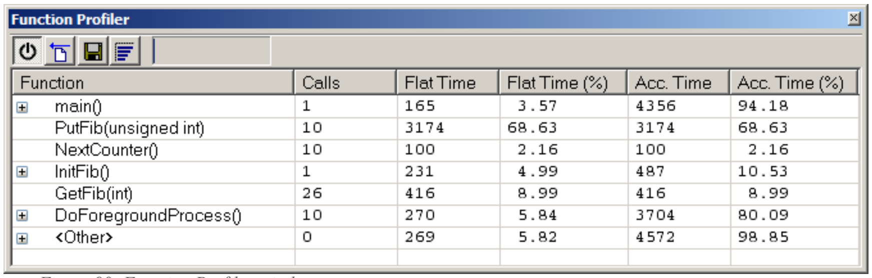
Reference information on the profiler

This section gives reference information about these windows and dialog boxes:

- *Function Profiler window*, page 186
- *Disassembly window*, page 66

Function Profiler window

The Function Profiler window is available from the C-SPY driver menu.



Function Profiler

Function	Calls	Flat Time	Flat Time (%)	Acc. Time	Acc. Time (%)
⊞ main()	1	165	3.57	4356	94.18
PutFib(unsigned int)	10	3174	68.63	3174	68.63
NextCounter()	10	100	2.16	100	2.16
⊞ InitFib()	1	231	4.99	487	10.53
GetFib(int)	26	416	8.99	416	8.99
⊞ DoForegroundProcess()	10	270	5.84	3704	80.09
⊞ <Other>	0	269	5.82	4572	98.85

Figure 90: Function Profiler window

This window displays function profiling information.

When Trace(flat) is selected, a checkbox appears on each line in the left-side margin of the window. Use these checkboxes to include or exclude lines from the profiling. Excluded lines are dimmed but not removed.

Requirements

The C-SPY simulator

Toolbar

The toolbar contains:

Enable/Disable

Enables or disables the profiler.

Clear

Clears all profiling data.

Save

Opens a standard **Save As** dialog box where you can save the contents of the window to a file, with tab-separated columns. Only non-expanded rows are sincluded in the list file.

Graphical view

Overlays the values in the percentage columns with a graphical bar.

Progress bar

Displays a backlog of profiling data that is still being processed. If the rate of incoming data is higher than the rate of the profiler processing the data, a backlog is accumulated. The progress bar indicates that the profiler is still processing data, but also approximately how far the profiler has come in the process. Note that because the profiler consumes data at a certain rate and the target system supplies data at another rate, the amount of data remaining to be processed can both increase and decrease. The progress bar can grow and shrink accordingly.

Display area

The content in the display area depends on which source that is used for the profiling information:

- For the Trace (calls) source, the display area contains one line for each function compiled with debug information enabled. When some profiling information has been collected, it is possible to expand rows of functions that have called other functions. The child items for a given function list all the functions that have been called by the parent function and the corresponding statistics.

- For the Trace (flat) source, the display area contains one line for each C function of your application, but also lines for sections of code from the runtime library or from other code without debug information, denoted only by the corresponding assembler labels. Each executed `PC` address from trace data is treated as a separate sample and is associated with the corresponding line in the Profiling window. Each line contains a count of those samples.

For information about which views that are supported in the C-SPY driver you are using, see *Requirements for using the profiler*, page 184.

More specifically, the display area provides information in these columns:

Function (All sources)

The name of the profiled C function.

Calls (Trace (calls))

The number of times the function has been called.

Flat time (Trace (calls))

The time in number of executed instructions spent inside the function.

Flat time (%) (Trace (calls))

Flat time expressed as a percentage of the total time.

Acc. time (Trace (calls))

The time in number of executed instructions spent in this function and everything called by this function.

Acc. time (%) (Trace (calls))

Accumulated time expressed as a percentage of the total time.

PC Samples (Trace (flat))

The number of PC samples associated with the function.

PC Samples (%) (Trace (flat))

The number of PC samples associated with the function as a percentage of the total number of samples.

Context menu

This context menu is available:

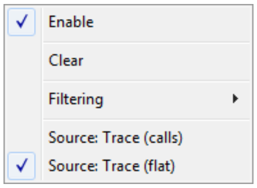


Figure 91: Function Profiler window context menu

The contents of this menu depend on the C-SPY driver you are using.

These commands are available:

Enable

Enables the profiler. The system will collect information also when the window is closed.

Clear

Clears all profiling data.

Filtering

Selects which part of your code to profile. Choose between:

Check All—Excludes all lines from the profiling.

Uncheck All—Includes all lines in the profiling.

Load—Reads all excluded lines from a saved file.

Save—Saves all excluded lines to a file. Typically, this can be useful if you are a group of engineers and want to share sets of exclusions.

These commands are only available when using Trace(flat).

Source*

Selects which source to be used for the profiling information. Choose between:

Trace (calls)—the instruction count for instruction profiling is only as complete as the collected trace data.

Trace (flat)—the instruction count for instruction profiling is only as complete as the collected trace data.

* The available sources depend on the C-SPY driver you are using.

For information about which views that are supported in the C-SPY driver you are using, see *Requirements for using the profiler*, page 184.

Code coverage

This chapter describes the code coverage functionality in C-SPY®, which helps you verify whether all parts of your code have been executed. More specifically, this means:

- Introduction to code coverage
- Reference information on code coverage.

Introduction to code coverage

This section covers these topics:

- Reasons for using code coverage
- Briefly about code coverage
- Requirements for using code coverage.

REASONS FOR USING CODE COVERAGE

The code coverage functionality is useful when you design your test procedure to verify whether all parts of the code have been executed. It also helps you identify parts of your code that are not reachable.

BRIEFLY ABOUT CODE COVERAGE

The Code Coverage window reports the status of the current code coverage analysis. For every program, module, and function, the analysis shows the percentage of code that has been executed since code coverage was turned on up to the point where the application has stopped. In addition, all statements that have not been executed are listed. The analysis will continue until turned off.

REQUIREMENTS FOR USING CODE COVERAGE

Code coverage is supported by the C-SPY Simulator.

Reference information on code coverage

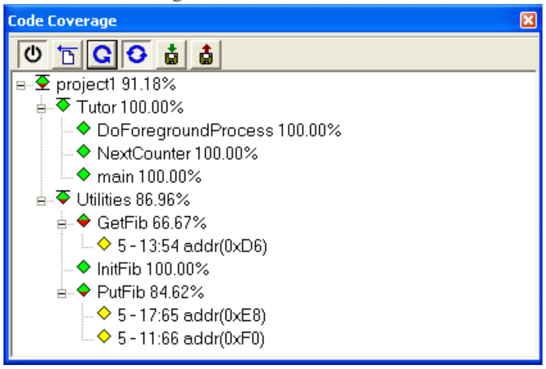
This section gives reference information about these windows and dialog boxes:

- *Code Coverage window*, page 192.

See also *Single stepping*, page 60.

Code Coverage window

The Code Coverage window is available from the **View** menu.


Code Coverage

```
project1 91.18%
  Tutor 100.00%
    DoForegroundProcess 100.00%
    NextCounter 100.00%
    main 100.00%
  Utilities 86.96%
    GetFib 66.67%
      5 - 13:54 addr(0xD6)
    InitFib 100.00%
    PutFib 84.62%
      5 - 17:65 addr(0xE8)
      5 - 11:66 addr(0xF0)
```

Figure 92: Code Coverage window

This window reports the status of the current code coverage analysis. For every program, module, and function, the analysis shows the percentage of code that has been executed since code coverage was turned on up to the point where the application has stopped. In addition, all statements that have not been executed are listed. The analysis will continue until turned off.

An asterisk (*) in the title bar indicates that C-SPY has continued to execute, and that the Code Coverage window must be refreshed because the displayed information is no longer up to date. To update the information, use the **Refresh** command.

To get started using code coverage:

1 Before using the code coverage functionality you must build your application using these options:

Category	Setting
C/C++ Compiler	Output>Generate debug information
Linker	Format>Debug information for C-SPY
Debugger	Plugins>Code Coverage

Table 11: Project options for enabling code coverage

2 After you have built your application and started C-SPY, choose **View>Code Coverage** to open the Code Coverage window.

3 Click the **Activate** button, alternatively choose **Activate** from the context menu, to switch on code coverage.

4 Start the execution. When the execution stops, for instance because the program exit is reached or a breakpoint is triggered, click the **Refresh** button to view the code coverage information.

Requirements

The C-SPY simulator

Display area

The code coverage information is displayed in a tree structure, showing the program, module, function, and statement levels. The window displays only source code that was compiled with debug information. Thus, startup code, exit code, and library code is not displayed in the window. Furthermore, coverage information for statements in inlined functions is not displayed. Only the statement containing the inlined function call is marked as executed. The plus sign and minus sign icons allow you to expand and collapse the structure.

These icons give you an overview of the current status on all levels:

Red diamond	Signifies that 0% of the modules or functions has been executed.
Green diamond	Signifies that 100% of the modules or functions has been executed.
Red and green diamond	Signifies that some of the modules or functions have been executed.
Yellow diamond	Signifies a statement that has not been executed.

The percentage displayed at the end of every program, module, and function line shows the amount of statements that has been covered so far, that is, the number of executed statements divided with the total number of statements.

For statements that have not been executed (yellow diamond), the information displayed is the column number range and the row number of the statement in the source window, followed by the address of the step point:

`<column_start>-<column_end>:row address.`

A statement is considered to be executed when one of its instructions has been executed. When a statement has been executed, it is removed from the window and the percentage is increased correspondingly.

Double-clicking a statement or a function in the Code Coverage window displays that statement or function as the current position in the source window, which becomes the active window. Double-clicking a module on the program level expands or collapses the tree structure.

Context menu

This context menu is available:

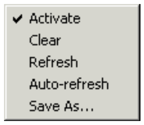


Figure 93: Code coverage window context menu

These commands are available:

Activate

Switches code coverage on and off during execution.

Clear

Clears the code coverage information. All step points are marked as not executed.

Refresh

Updates the code coverage information and refreshes the window. All step points that have been executed since the last refresh are removed from the tree.

Auto-refresh

Toggles the automatic reload of code coverage information on and off. When turned on, the code coverage information is reloaded automatically when C-SPY stops at a breakpoint, at a step point, and at program exit.

Save As

Saves the current code coverage result in a text file.

Save session

Saves your code coverage session data to a `*.dat` file. This is useful if you for some reason must abort your debug session, but want to continue the session later on. This command is available on the toolbar.

Restore session

Restores previously saved code coverage session data. This is useful if you for some reason must abort your debug session, but want to continue the session later on. This command is available on the toolbar.

Interrupts

This chapter describes how C-SPY® can help you test the logic of your interrupt service routines and debug the interrupt handling in the target system. Interrupt logging provides you with comprehensive information about the interrupt events. More specifically, this chapter gives:

- Introduction to interrupts
- Using the interrupt system
- Reference information on interrupts.

Introduction to interrupts

This section introduces you to interrupt logging and to interrupt simulation.

This section covers these topics:

- Briefly about interrupt logging
- Briefly about the interrupt simulation system
- Interrupt characteristics
- Interrupt simulation states
- C-SPY system macros for interrupt simulation
- Target-adapting the interrupt simulation system.

See also:

- *Reference information on C-SPY system macros*, page 229
- *Breakpoints*, page 101
- The *IAR C/C++ Compiler Reference Guide for R32C*.

BRIEFLY ABOUT INTERRUPT LOGGING

Interrupt logging provides you with comprehensive information about the interrupt events. This might be useful for example, to help you locate which interrupts you can fine-tune to become faster. You can log entrances and exits to and from interrupts. You can also log internal interrupt status information, such as triggered, expired, etc. The logs are displayed in the Interrupt Log window and a summary is available in the Interrupt Log Summary window. The Interrupt Graph in the Timeline window provides a graphical view of the interrupt events during the execution of your application program.

Requirements for interrupt logging

Interrupt logging is supported by the C-SPY simulator.

BRIEFLY ABOUT THE INTERRUPT SIMULATION SYSTEM

By simulating interrupts, you can test the logic of your interrupt service routines and debug the interrupt handling in the target system long before any hardware is available. If you use simulated interrupts in conjunction with C-SPY macros and breakpoints, you can compose a complex simulation of, for instance, interrupt-driven peripheral devices.

The C-SPY Simulator includes an interrupt simulation system where you can simulate the execution of interrupts during debugging. You can configure the interrupt simulation system so that it resembles your hardware interrupt system.

The interrupt system has the following features:

- Simulated interrupt support for the R32C/100 microcomputer
- Single-occasion or periodical interrupts based on the cycle counter
- Predefined interrupts for various devices
- Configuration of hold time, probability, and timing variation
- State information for locating timing problems
- Configuration of interrupts using a dialog box or a C-SPY system macro—that is, one interactive and one automating interface. In addition, you can instantly force an interrupt.
- A log window that continuously displays events for each defined interrupt.
- A status window that shows the current interrupt activities.

All interrupts you define using the **Interrupt Setup** dialog box are preserved between debug sessions, unless you remove them. A forced interrupt, on the other hand, exists only until it has been serviced and is not preserved between sessions.

The interrupt simulation system is activated by default, but if not required, you can turn off the interrupt simulation system to speed up the simulation. To turn it off, use either the **Interrupt Setup** dialog box or a system macro.

INTERRUPT CHARACTERISTICS

The simulated interrupts consist of a set of characteristics which lets you fine-tune each interrupt to make it resemble the real interrupt on your target hardware. You can specify a *first activation time*, a *repeat interval*, a *hold time*, a *variance*, and a *probability*.

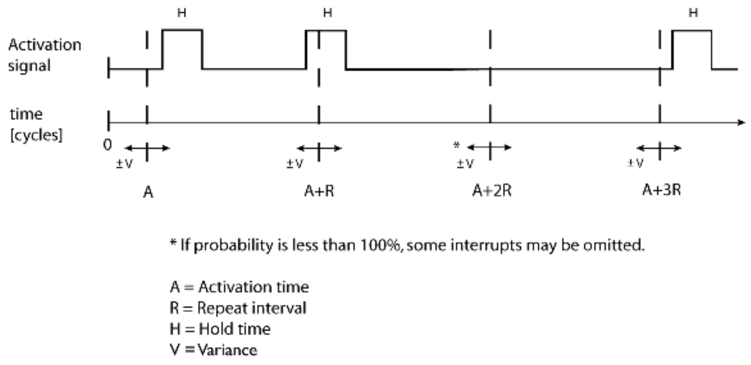


Figure 94: Simulated interrupt configuration

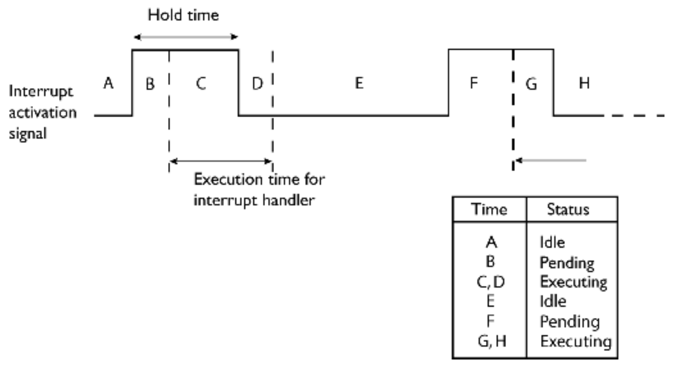
The interrupt simulation system uses the cycle counter as a clock to determine when an interrupt should be raised in the simulator. You specify the *first activation time*, which is based on the cycle counter. C-SPY will generate an interrupt when the cycle counter has passed the specified activation time. However, interrupts can only be raised between instructions, which means that a full assembler instruction must have been executed before the interrupt is generated, regardless of how many cycles an instruction takes.

To define the periodicity of the interrupt generation you can specify the *repeat interval* which defines the amount of cycles after which a new interrupt should be generated. In addition to the repeat interval, the periodicity depends on the two options *probability*—the probability, in percent, that the interrupt will actually appear in a period—and *variance*—a time variation range as a percentage of the repeat interval. These options make it possible to randomize the interrupt simulation. You can also specify a *hold time* which describes how long the interrupt remains pending until removed if it has not been processed. If the hold time is set to *infinite*, the corresponding pending bit will be set until the interrupt is acknowledged or removed.

INTERRUPT SIMULATION STATES

The interrupt simulation system contains status information that you can use for locating timing problems in your application. The Interrupt Status window displays the available status information. For an interrupt, these states can be displayed: *Idle*, *Pending*, *Executing*, or *Suspended*.

Normally, a repeatable interrupt has a specified repeat interval that is longer than the execution time. In this case, the status information at different times looks like this:

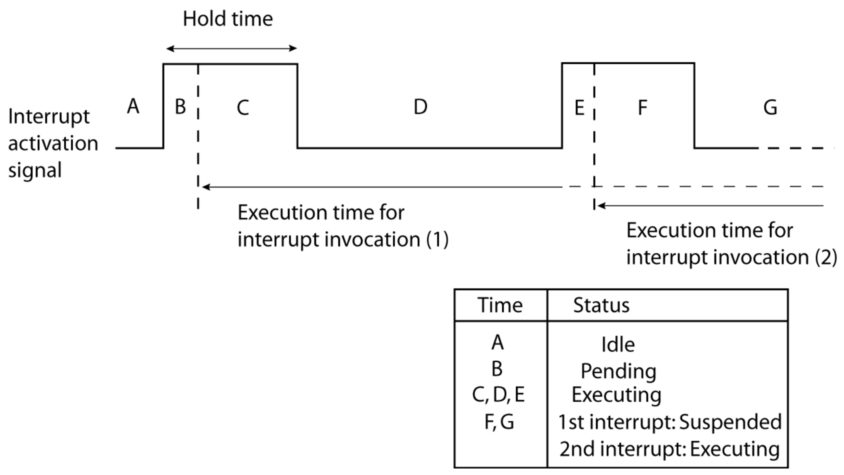

Hold time
Interrupt activation signal
A
B
C
D
E
F
G
H
Execution time for interrupt handler

Time	Status
A	Idle
B	Pending
C, D	Executing
E	Idle
F	Pending
G, H	Executing

Figure 95: Simulation states - example 1

Note: The interrupt activation signal—also known as the pending bit—is automatically deactivated the moment the interrupt is acknowledged by the interrupt handler.

However, if the interrupt repeat interval is shorter than the execution time, and the interrupt is reentrant (or non-maskable), the status information at different times looks like this:


Hold time
Interrupt activation signal
A
B
C
D
E
F
G
Execution time for interrupt invocation (1)
Execution time for interrupt invocation (2)

Time	Status
A	Idle
B	Pending
C, D, E	Executing
F, G	1st interrupt: Suspended 2nd interrupt: Executing

Figure 96: Simulation states - example 2

An execution time that is longer than the repeat interval might indicate that you should rewrite your interrupt handler and make it faster, or that you should specify a longer repeat interval for the interrupt simulation system.

C-SPY SYSTEM MACROS FOR INTERRUPT SIMULATION

Macros are useful when you already have sorted out the details of the simulated interrupt so that it fully meets your requirements. If you write a macro function containing definitions for the simulated interrupts, you can execute the functions automatically when C-SPY starts. Another advantage is that your simulated interrupt definitions will be documented if you use macro files, and if you are several engineers involved in the development project you can share the macro files within the group.

The C-SPY Simulator provides these predefined system macros related to interrupts:

`__enableInterrupts`

`__disableInterrupts`

`__orderInterrupt`

`__cancelInterrupt`

`__cancelAllInterrupts`

`__popSimulatorInterruptExecutingStack`

The parameters of the first five macros correspond to the equivalent entries of the **Interrupts** dialog box.

For more information about each macro, see *Reference information on C-SPY system macros*, page 229.

TARGET-ADAPTING THE INTERRUPT SIMULATION SYSTEM

The interrupt simulation system is easy to use. However, to take full advantage of the interrupt simulation system you should be familiar with how to adapt it for the processor you are using.

The interrupt simulation has the same behavior as the hardware. This means that the execution of an interrupt is dependent on the status of the global interrupt enable bit. The execution of maskable interrupts is also dependent on the status of the individual interrupt enable bits.

To perform these actions for various devices, the interrupt system must have detailed information about each available interrupt. Except for default settings, this information is provided in the device description files. The default settings are used if no device description file has been specified.

For information about device description files, see *Selecting a device description file*, page 33.

Using the interrupt system

This section describes various tasks related to interrupts.

More specifically, you will get information about:

- Simulating a simple interrupt
- Simulating an interrupt in a multi-task system
- Getting started using interrupt logging.

See also:

- *Registering and executing using setup macros and setup files*, page 220 for details about how to use a setup file to define simulated interrupts at C-SPY startup
- The tutorial *Simulating an interrupt* in the Information Center.

SIMULATING A SIMPLE INTERRUPT

This example demonstrates the method for simulating a timer interrupt. However, the procedure can also be used for other types of interrupts.

To simulate and debug an interrupt:

1 Assume this simple application which contains an interrupt service routine for a timer, which increments a tick variable. The main function sets the necessary status registers. The application exits when 100 interrupts have been generated.

```
#include "ior32c.h"
#include <intrinsics.h>

volatile int ticks = 0;
void main (void)
{
 /* Add your timer setup code here */

 while (ticks < 100);         /* Endless loop */
 printf("Done\n");
}

#pragma vector = BASICTIMER_VECTOR
__interrupt void basic_timer(void)
{
 ticks += 1;
}
```

2 Add your interrupt service routine to your application source code and add the file to your project.

3 Build your project and start the simulator.

4 Choose **Simulator>Interrupt Setup** to open the **Interrupts Setup** dialog box. Select the **Enable interrupt simulation** option to enable interrupt simulation. Click **New** to open the **Edit Interrupt** dialog box. For the BasicTimer example, verify these settings:

Option	Settings
Interrupt	BASICTIMER_VECTOR
First activation	4000
Repeat interval	2000
Hold time	10
Probability (%)	100
Variance (%)	0

Table 12: Timer interrupt settings

Click **OK**.

5 Execute your application. If you have enabled the interrupt properly in your application source code, C-SPY will:

- Generate an interrupt when the cycle counter has passed 4000
- Continuously repeat the interrupt after approximately 2000 cycles.

6 To watch the interrupt in action, choose **Simulator>Interrupt Log** to open the Interrupt Log window.

7 From the context menu, available in the Interrupt Log window, choose **Enable** to enable the logging. If you restart program execution, status information about entrances and exits to and from interrupts will now appear in the Interrupt Log window.

For information about how to get a graphical representation of the interrupts correlated with a time axis, see *Timeline window*, page 168.

SIMULATING AN INTERRUPT IN A MULTI-TASK SYSTEM

If you are using interrupts in such a way that the normal instruction used for returning from an interrupt handler is not used, for example in an operating system with task-switching, the simulator cannot automatically detect that the interrupt has finished executing. The interrupt simulation system will work correctly, but the status information in the **Interrupt Setup** dialog box might not look as you expect. If too many interrupts are executing simultaneously, a warning might be issued.

To simulate a normal interrupt exit:

1. Set a code breakpoint on the instruction that returns from the interrupt function.
2. Specify the `__popSimulatorInterruptExecutingStack` macro as a condition to the breakpoint.

   When the breakpoint is triggered, the macro is executed and then the application continues to execute automatically.

GETTING STARTED USING INTERRUPT LOGGING

1. Choose **C-SPY driver>Interrupt Log** to open the Interrupt Log window. Optionally, you can also choose:
   - **C-SPY driver>Interrupt Log Summary** to open the Interrupt Log Summary window
   - **C-SPY driver>Timeline** to open the Timeline window and view the Interrupt graph.
2. From the context menu in the Interrupt Log window, choose **Enable** to enable the logging.
3. Start executing your application program to collect the log information.
4. To view the interrupt log information, look in any of the Interrupt Log, Interrupt Log Summary, or the Interrupt graph in the Timeline window.
5. If you want to save the log or summary to a file, choose **Save to log file** from the context menu in the window in question.
6. To disable interrupt logging, from the context menu in the Interrupt Log window, toggle **Enable** off.

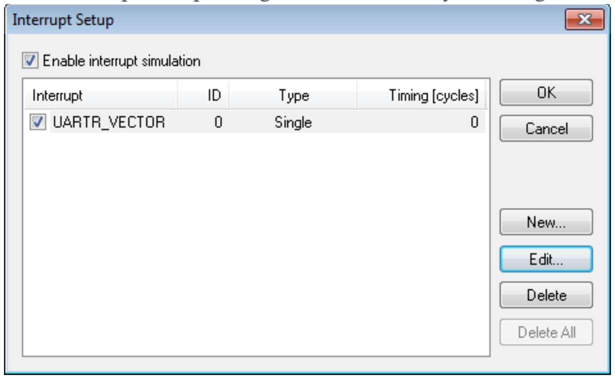
Reference information on interrupts

This section gives reference information about these windows and dialog boxes:

- *Interrupt Setup dialog box*, page 203
- *Edit Interrupt dialog box*, page 204
- *Forced Interrupt window*, page 206
- *Interrupt Status window*, page 207
- *Interrupt Log window*, page 209
- *Interrupt Log Summary window*, page 212.

Interrupt Setup dialog box

The **Interrupt Setup** dialog box is available by choosing **Simulator>Interrupt Setup**.



☑ Enable interrupt simulation

Interrupt	ID	Type	Timing [cycles]
☑ UARTR_VECTOR	0	Single	0

Figure 97: Interrupt Setup dialog box

This dialog box lists all defined interrupts. Use this dialog box to enable or disable the interrupt simulation system, as well as to enable or disable individual interrupts.

Requirements

The C-SPY simulator.

Enable interrupt simulation

Enables or disables interrupt simulation. If the interrupt simulation is disabled, the definitions remain but no interrupts are generated. Note that you can also enable and disable installed interrupts individually by using the check box to the left of the interrupt name in the list of installed interrupts.

Display area

This area contains these columns:

Interrupt

Lists all interrupts. Use the checkbox to enable or disable the interrupt.

ID

A unique interrupt identifier.

Type

Shows the type of the interrupt. The type can be one of:

`Forced`, a single-occasion interrupt defined in the Forced Interrupt Window.

`Single`, a single-occasion interrupt.

`Repeat`, a periodically occurring interrupt.

If the interrupt has been set from a C-SPY macro, the additional part `(macro)` is added, for example: `Repeat(macro)`.

Timing

The timing of the interrupt. For a `Single` and `Forced` interrupt, the activation time is displayed. For a `Repeat` interrupt, the information has the form: `Activation Time` + `n*Repeat Time`. For example, `2000` + `n*2345`. This means that the first time this interrupt is triggered, is at 2000 cycles and after that with an interval of 2345 cycles.

Buttons

These buttons are available:

New

Opens the **Edit Interrupt** dialog box, see *Edit Interrupt dialog box*, page 204.

Edit

Opens the **Edit Interrupt** dialog box, see *Edit Interrupt dialog box*, page 204.

Delete

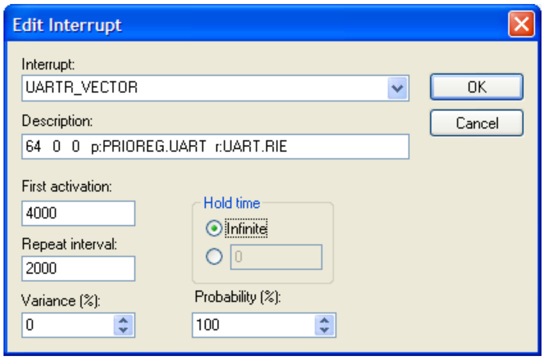
Removes the selected interrupt.

Delete All

Removes all interrupts.

Edit Interrupt dialog box

The **Edit Interrupt** dialog box is available from the **Interrupt Setup** dialog box.



Interrupt: UARTR_VECTOR

Description: 64 0 0 p:PRIOREG.UART r:UART.RIE

First activation: 4000

Repeat interval: 2000

Variance (%): 0

Hold time: [✓ Infinite] 0

Probability (%): 100

Figure 98: Edit Interrupt dialog box

Use this dialog box to interactively fine-tune the interrupt parameters. You can add the parameters and quickly test that the interrupt is generated according to your needs.

Note: You can only edit or remove non-forced interrupts.

Requirements

The C-SPY simulator.

Interrupt

Selects the interrupt that you want to edit. The drop-down list contains all available interrupts. Your selection will automatically update the **Description** box. The list is populated with entries from the device description file that you have selected.

Description

A description of the selected interrupt, if available. The description is retrieved from the selected device description file and consists of a string describing the vector table format, the vector number, the base priority, the hardware priority, the `SFR` register and request bit, separated by space characters. For interrupts specified using the system macro `__orderInterrupt`, the **Description** box is empty.

First activation

Specify the value of the cycle counter after which the specified type of interrupt will be generated.

Repeat interval

Specify the periodicity of the interrupt in cycles.

Variance %

Selects a timing variation range, as a percentage of the repeat interval, in which the interrupt might occur for a period. For example, if the repeat interval is 100 and the variance 5%, the interrupt might occur anywhere between T=95 and T=105, to simulate a variation in the timing.

Hold time

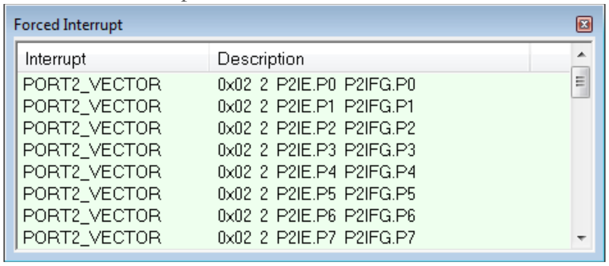
Specify how long, in cycles, the interrupt remains pending until removed if it has not been processed. If you select **Infinite**, the corresponding pending bit will be set until the interrupt is acknowledged or removed.

Probability %

Selects the probability, in percent, that the interrupt will actually occur within the specified period.

Forced Interrupt window

The Forced Interrupt window is available from the C-SPY driver menu.



Forced Interrupt

Interrupt	Description
PORT2_VECTOR	0x02 2 P2IE.P0 P2IFG.P0
PORT2_VECTOR	0x02 2 P2IE.P1 P2IFG.P1
PORT2_VECTOR	0x02 2 P2IE.P2 P2IFG.P2
PORT2_VECTOR	0x02 2 P2IE.P3 P2IFG.P3
PORT2_VECTOR	0x02 2 P2IE.P4 P2IFG.P4
PORT2_VECTOR	0x02 2 P2IE.P5 P2IFG.P5
PORT2_VECTOR	0x02 2 P2IE.P6 P2IFG.P6
PORT2_VECTOR	0x02 2 P2IE.P7 P2IFG.P7

Figure 99: Forced Interrupt window

Use this window to force an interrupt instantly. This is useful when you want to check your interrupt logistics and interrupt routines.

The hold time for a forced interrupt is infinite, and the interrupt exists until it has been serviced or until a reset of the debug session.

To force an interrupt:

1. Enable the interrupt simulation system, see *Interrupt Setup dialog box*, page 203.
2. Double-click the interrupt in the Forced Interrupt window, or activate by using the **Force** command available on the context menu.

Requirements

The C-SPY simulator.

Display area

This area lists all available interrupts and their definitions. The description field is editable and the information is retrieved from the selected device description file. See this file for a detailed description.

Context menu

This context menu is available:

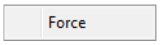


Figure 100: Forced Interrupt window context menu

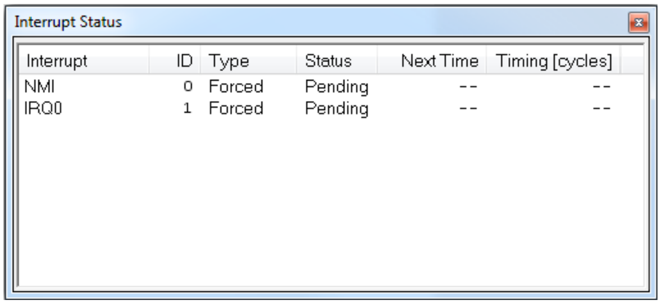
This command is available:

Force

Triggers the interrupt you selected in the display area.

Interrupt Status window

The Interrupt Status window is available from the C-SPY driver menu.



Interrupt Status

Interrupt	ID	Type	Status	Next Time	Timing [cycles]
NMI	0	Forced	Pending	--	--
IRQ0	1	Forced	Pending	--	--

Figure 101: Interrupt Status window

This window shows the status of all the currently active interrupts, in other words interrupts that are either executing or waiting to be executed.

Requirements

The C-SPY simulator.

Display area

This area contains these columns:

Interrupt

Lists all interrupts.

ID

A unique interrupt identifier.

Type

The type of the interrupt. The type can be one of:

`Forced`, a single-occasion interrupt defined in the Forced Interrupt window.

`Single`, a single-occasion interrupt.

`Repeat`, a periodically occurring interrupt.

If the interrupt has been set from a C-SPY macro, the additional part `(macro)` is added, for example: `Repeat(macro)`.

Status

The state of the interrupt:

Idle, the interrupt activation signal is low (deactivated).

Pending, the interrupt activation signal is active, but the interrupt has not been yet acknowledged by the interrupt handler.

Executing, the interrupt is currently being serviced, that is the interrupt handler function is executing.

Suspended, the interrupt is currently suspended due to execution of an interrupt with a higher priority.

(deleted) is added to Executing and Suspended if you have deleted a currently active interrupt. **(deleted)** is removed when the interrupt has finished executing.

Next Time

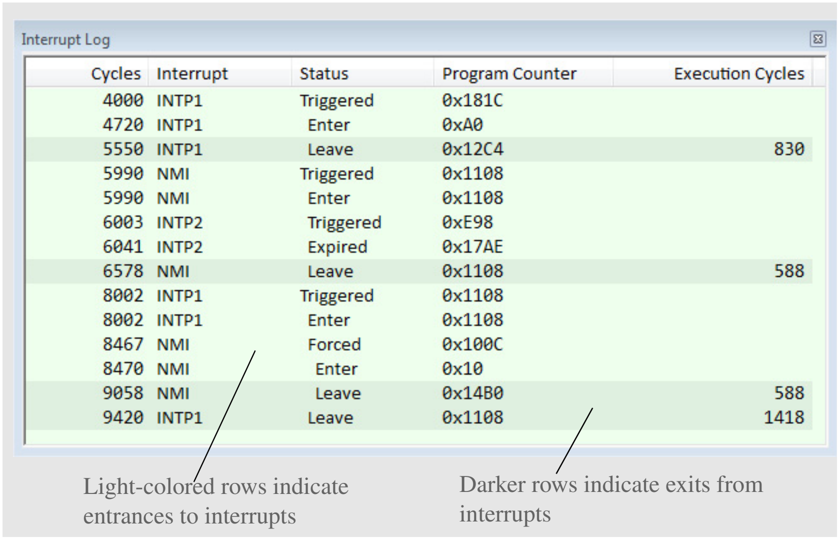
The next time an idle interrupt is triggered. Once a repeatable interrupt stats executing, a copy of the interrupt will appear with the state Idle and the next time set. For interrupts that do not have a next time—that is pending, executing, or suspended—the column will show `--`.

Timing

The timing of the interrupt. For a `Single` and `Forced` interrupt, the activation time is displayed. For a `Repeat` interrupt, the information has the form: `Activation Time + n*Repeat Time`. For example, `2000 + n*2345`. This means that the first time this interrupt is triggered, is at 2000 cycles and after that with an interval of 2345 cycles.

Interrupt Log window

The Interrupt Log window is available from the C-SPY driver menu.



Interrupt Log

Cycles	Interrupt	Status	Program Counter	Execution Cycles
4000	INTP1	Triggered	0x181C	
4720	INTP1	Enter	0xA0	
5550	INTP1	Leave	0x12C4	830
5990	NMI	Triggered	0x1108	
5990	NMI	Enter	0x1108	
6003	INTP2	Triggered	0xE98	
6041	INTP2	Expired	0x17AE	
6578	NMI	Leave	0x1108	588
8002	INTP1	Triggered	0x1108	
8002	INTP1	Enter	0x1108	
8467	NMI	Forced	0x100C	
8470	NMI	Enter	0x10	
9058	NMI	Leave	0x14B0	588
9420	INTP1	Leave	0x1108	1418

Light-colored rows indicate entrances to interrupts
Darker rows indicate exits from interrupts

Figure 102: Interrupt Log window

This window logs entrances to and exits from interrupts. The C-SPY simulator also logs internal state changes.

The information is useful for debugging the interrupt handling in the target system. When the Interrupt Log window is open, it is updated continuously at runtime.

Note: There is a limit on the number of saved logs. When this limit is exceeded, the entries in the beginning of the buffer are erased.

For more information, see *Getting started using interrupt logging*, page 202.

For information about how to get a graphical view of the interrupt events during the execution of your application, see *Timeline window*, page 168.

Requirements

The C-SPY simulator

Display area

This area contains these columns:

Time

The time for the interrupt entrance, based on an internally specified clock frequency.

This column is available when you have selected **Show Time** from the context menu.

Cycles

The number of cycles from the start of the execution until the event.

This column is available when you have selected **Show Cycles** from the context menu.

Interrupt

The interrupt as defined in the device description file.

Status

Shows the event status of the interrupt:

Triggered, the interrupt has passed its activation time.

Forced, the same as Triggered, but the interrupt was forced from the Forced Interrupt window.

Enter, the interrupt is currently executing.

Leave, the interrupt has been executed.

Expired, the interrupt hold time has expired without the interrupt being executed.

Rejected, the interrupt has been rejected because the necessary interrupt registers were not set up to accept the interrupt.

Program Counter

The value of the program counter when the event occurred.

Execution Time/Cycles

The time spent in the interrupt, calculated using the Enter and Leave timestamps. This includes time spent in any subroutines or other interrupts that occurred in the specific interrupt.

Interrupt Log window context menu

This context menu is available in the Interrupt Log window and in the Interrupt Log Summary window:

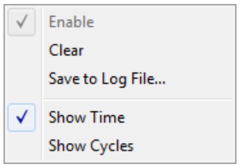


Figure 103: Interrupt Log window context menu

Note: The commands are the same in each window, but they only operate on the specific window.

These commands are available:

Enable

Enables the logging system. The system will log information also when the window is closed.

Clear

Deletes the log information. Note that this will happen also when you reset the debugger.

Save to log file

Displays a standard file selection dialog box where you can select the destination file for the log information. The entries in the log file are separated by `TAB` and `LF`. An `X` in the Approx column indicates that the timestamp is an approximation.

Show Time

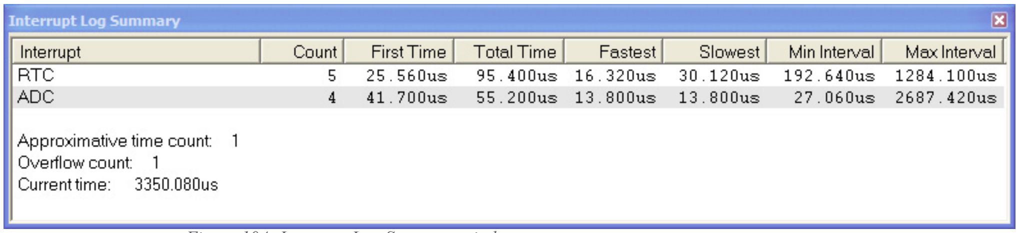
Displays the **Time** column in the Data Log window and in the Interrupt Log window, respectively.

Show Cycles

Displays the **Cycles** column in the Data Log window and in the Interrupt Log window, respectively.

Interrupt Log Summary window

The Interrupt Log Summary window is available from the C-SPY driver menu.



Interrupt	Count	First Time	Total Time	Fastest	Slowest	Min Interval	Max Interval
RTC	5	25.560us	95.400us	16.320us	30.120us	192.640us	1284.100us
ADC	4	41.700us	55.200us	13.800us	13.800us	27.060us	2687.420us

```
Approximative time count:   1
Overflow count:   1
Current time:     3350.080us
```

Figure 104: Interrupt Log Summary window

This window displays a summary of logs of entrances to and exits from interrupts.

For more information, see *Getting started using interrupt logging*, page 202.

For information about how to get a graphical view of the interrupt events during the execution of your application, see *Timeline window*, page 168.

Requirements

The C-SPY simulator

Display area

Each row in this area displays statistics about the specific interrupt based on the log information in these columns:

Interrupt

The type of interrupt that occurred.

At the bottom of the column, the current time or cycles is displayed—the number of cycles or the execution time since the start of execution. Overflow count and approximative time count is always zero.

Count

The number of times the interrupt occurred.

First time

The first time the interrupt was executed.

Total time**

The accumulated time spent in the interrupt.

Fastest**

The fastest execution of a single interrupt of this type.

Slowest**

The slowest execution of a single interrupt of this type.

Max interval

The longest time between two interrupts of this type.

The interval is specified as the time interval between the entry time for two consecutive interrupts.

** Calculated in the same way as for the Execution time/cycles in the Interrupt Log window.

Context menu

Identical to the context menu of the Interrupt Log window, see *Interrupt Log window*, page 209.

C-SPY macros

C-SPY® includes a comprehensive macro language which allows you to automate the debugging process and to simulate peripheral devices.

This chapter describes the C-SPY macro language, its features, for what purpose these features can be used, and how to use them. More specifically, this means:

- Introduction to C-SPY macros
- Using C-SPY macros
- Reference information on the macro language
- Reference information on reserved setup macro function names
- Reference information on C-SPY system macros.

Introduction to C-SPY macros

This section covers these topics:

- Reasons for using C-SPY macros
- Briefly about using C-SPY macros
- Briefly about setup macro functions and files
- Briefly about the macro language.

REASONS FOR USING C-SPY MACROS

You can use C-SPY macros either by themselves or in conjunction with complex breakpoints and interrupt simulation to perform a wide variety of tasks. Some examples where macros can be useful:

- Automating the debug session, for instance with trace printouts, printing values of variables, and setting breakpoints.
- Hardware configuring, such as initializing hardware registers.
- Feeding your application with simulated data during runtime.

- Simulating peripheral devices, see the chapter *Interrupts*. This only applies if you are using the simulator driver.
- Developing small debug utility functions, for instance reading I/O input from a file, see the file `setupsimple.mac` located in the directory `\R32C\tutor\`.

BRIEFLY ABOUT USING C-SPY MACROS

To use C-SPY macros, you should:

- Write your macro variables and functions and collect them in one or several *macro files*
- Register your macros
- Execute your macros.

For registering and executing macros, there are several methods to choose between. Which method you choose depends on which level of interaction or automation you want, and depending on at which stage you want to register or execute your macro.

BRIEFLY ABOUT SETUP MACRO FUNCTIONS AND FILES

There are some reserved *setup macro function names* that you can use for defining macro functions which will be called at specific times, such as:

- Once after communication with the target system has been established but before downloading the application software
- Once after your application software has been downloaded
- Each time the reset command is issued
- Once when the debug session ends.

To define a macro function to be called at a specific stage, you should define and register a macro function with one of the reserved names. For instance, if you want to clear a specific memory area before you load your application software, the macro setup function `execUserPreload` should be used. This function is also suitable if you want to initialize some CPU registers or memory-mapped peripheral units before you load your application software.

You should define these functions in a *setup macro file*, which you can load before C-SPY starts. Your macro functions will then be automatically registered each time you start C-SPY. This is convenient if you want to automate the initialization of C-SPY, or if you want to register multiple setup macros.

For more information about each setup macro function, see *Reference information on reserved setup macro function names*, page 228.

BRIEFLY ABOUT THE MACRO LANGUAGE

The syntax of the macro language is very similar to the C language. There are:

- *Macro statements*, which are similar to C statements.
- *Macro functions*, which you can define with or without parameters and return values.
- Predefined built-in *system macros*, similar to C library functions, which perform useful tasks such as opening and closing files, setting breakpoints, and defining simulated interrupts.
- *Macro variables*, which can be global or local, and can be used in C-SPY expressions.
- *Macro strings*, which you can manipulate using predefined system macros.

For more information about the macro language components, see *Reference information on the macro language*, page 223.

Example

Consider this example of a macro function which illustrates the various components of the macro language:

```
__var oldVal;
CheckLatest(val)
{
 if (oldval != val)
 {
  __message "Message: Changed from ", oldval, " to ", val, "\n";
  oldval = val;
 }
}
```

Note: Reserved macro words begin with double underscores to prevent name conflicts.

Using C-SPY macros

This section describes various tasks related to registering and executing C-SPY macros.

More specifically, you will get information about:

- Registering C-SPY macros—an overview
- Executing C-SPY macros—an overview
- Registering and executing using setup macros and setup files
- Registering and executing using setup macros and setup files
- Executing macros using Quick Watch

- Executing a macro by connecting it to a breakpoint

For more examples using C-SPY macros, see:

- The tutorial about simulating an interrupt, which you can find in the Information Center
- *Initializing target hardware before C-SPY starts*, page 37.

REGISTERING C-SPY MACROS—AN OVERVIEW

C-SPY must know that you intend to use your defined macro functions, and thus you must *register* your macros. There are various ways to register macro functions:

- You can register macros interactively in the **Macro Configuration** dialog box, see *Using the Macro Configuration dialog box*, page 219.
- You can register macro functions during the C-SPY startup sequence, see *Registering and executing using setup macros and setup files*, page 220.
- You can register a file containing macro function definitions, using the system macro `__registerMacroFile`. This means that you can dynamically select which macro files to register, depending on the runtime conditions. Using the system macro also lets you register multiple files at the same moment. For information about the system macro, see *__registerMacroFile*, page 242.

Which method you choose depends on which level of interaction or automation you want, and depending on at which stage you want to register your macro.

EXECUTING C-SPY MACROS—AN OVERVIEW

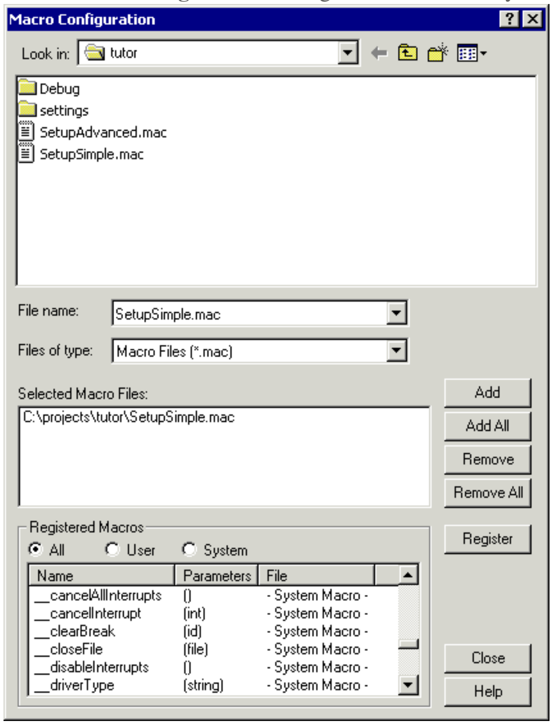
There are various ways to execute macro functions:

- You can execute macro functions during the C-SPY startup sequence and at other predefined stages during the debug session by defining setup macro functions in a setup macro file, see *Registering and executing using setup macros and setup files*, page 220.
- The Quick Watch window lets you evaluate expressions, and can thus be used for executing macro functions. For an example, see *Executing macros using Quick Watch*, page 221.
- A macro can be connected to a breakpoint; when the breakpoint is triggered the macro is executed. For an example, see *Executing a macro by connecting it to a breakpoint*, page 222.

Which method you choose depends on which level of interaction or automation you want, and depending on at which stage you want to execute your macro.

USING THE MACRO CONFIGURATION DIALOG BOX

The **Macro Configuration** dialog box is available by choosing **Debug>Macros**.



Debug
settings
SetupAdvanced.mac
SetupSimple.mac

File name: SetupSimple.mac

Files of type: Macro Files (*.mac)

Selected Macro Files:
C:\projects\tutor\SetupSimple.mac

Registered Macros: [✓ All] User System

Name	Parameters	File
__cancelAllInterrupts	()	- System Macro -
__cancelInterrupt	(int)	- System Macro -
__clearBreak	(id)	- System Macro -
__closeFile	(file)	- System Macro -
__disableInterrupts	()	- System Macro -
__driverType	(string)	- System Macro -

Figure 105: Macro Configuration dialog box

Use this dialog box to list, register, and edit your macro files and functions. The dialog box offers you an interactive interface for registering your macro functions which is convenient when you develop macro functions and continuously want to load and test them.

Macro functions that have been registered using the dialog box are deactivated when you exit the debug session, and will not automatically be registered at the next debug session.

To register a macro file:

Select the macro files you want to register in the file selection list, and click Add or Add All to add them to the Selected Macro Files list. Conversely, you can remove files from the Selected Macro Files list using Remove or Remove All.

2 Click Register to register the macro functions, replacing any previously defined macro functions or variables. Registered macro functions are displayed in the scroll list under Registered Macros.

Note: System macros cannot be removed from the list, they are always registered.

To list macro functions:

1 Select All to display all macro functions, select User to display all user-defined macros, or select System to display all system macros.

2 Click either Name or File under Registered Macros to display the column contents sorted by macro names or by file. Clicking a second time sorts the contents in the reverse order.

To modify a macro file:

Double-click a user-defined macro function in the Name column to open the file where the function is defined, allowing you to modify it.

REGISTERING AND EXECUTING USING SETUP MACROS AND SETUP FILES

It can be convenient to register a macro file during the C-SPY startup sequence. To do this, specify a macro file which you load before starting the debug session. Your macro functions will be automatically registered each time you start the debugger.

If you use the reserved setup macro function names to define the macro functions, you can define exactly at which stage you want the macro function to be executed.

To define a setup macro function and load it during C-SPY startup:

1 Create a new text file where you can define your macro function.

For example:

```
execUserSetup()
{
 ...
 __registerMacroFile("MyMacroUtils.mac");
 __registerMacroFile("MyDeviceSimulation.mac");

}
```

This macro function registers the additional macro files `MyMacroUtils.mac` and `MyDeviceSimulation.mac`. Because the macro function is defined with the function name `execUserSetup`, it will be executed directly after your application has been downloaded.

2 Save the file using the filename extension `mac`.

3 Before you start C-SPY, choose **Project>Options>Debugger>Setup**. Select **Use Setup file** and choose the macro file you just created.

The macros will now be registered during the C-SPY startup sequence.

EXECUTING MACROS USING QUICK WATCH

The Quick Watch window lets you dynamically choose when to execute a macro function.

1 Consider this simple macro function that checks the status of a watchdog timer interrupt enable bit:

```
WDTstatus()
{
  if ((WDreg & 0x01) != 0)  /* Checks the status of WDreg */
    return "Timer enabled"; /* C-SPY macro string used */
  else
    return "Timer disabled"; /* C-SPY macro string used */
}
```

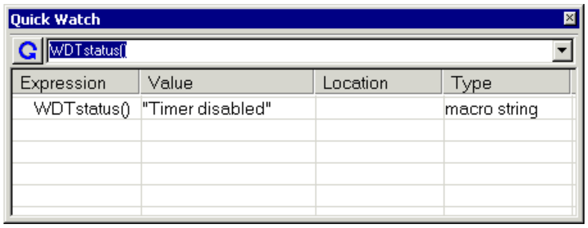
2 Save the macro function using the filename extension `mac`.

3 To register the macro file, choose **Debug>Macros**. The **Macro Configuration** dialog box appears.

4 Locate the file, click **Add** and then **Register**. The macro function appears in the list of registered macros.

5 Choose **View>Quick Watch** to open the Quick Watch window, type the macro call `WDTstatus()` in the text field and press Return,

Alternatively, in the macro file editor window, select the macro function name `WDTstatus()`. Right-click, and choose **Quick Watch** from the context menu that appears.



Expression	Value	Location	Type
WDTstatus()	"Timer disabled"		macro string

Figure 106: Quick Watch window

The macro will automatically be displayed in the Quick Watch window.

For more information, see *Quick Watch window*, page 92.

EXECUTING A MACRO BY CONNECTING IT TO A BREAKPOINT

You can connect a macro to a breakpoint. The macro will then be executed when the breakpoint is triggered. The advantage is that you can stop the execution at locations of particular interest and perform specific actions there.

For instance, you can easily produce log reports containing information such as how the values of variables, symbols, or registers change. To do this you might set a breakpoint on a suspicious location and connect a log macro to the breakpoint. After the execution you can study how the values of the registers have changed.

To create a log macro and connect it to a breakpoint:

1. Assume this skeleton of a C function in your application source code:

```
int fact(int x)
{
 ...
}
```

2. Create a simple log macro function like this example:

```
logfact()
{
 __message "fact(" ,x, ")";
}
```

The `__message` statement will log messages to the Log window.

Save the macro function in a macro file, with the filename extension `mac`.

3. To register the macro, choose **Debug>Macros** to open the **Macro Configuration** dialog box and add your macro file to the list **Selected Macro Files**. Click **Register** and your macro function will appear in the list **Registered Macros**. Close the dialog box.

4. To set a code breakpoint, click the **Toggle Breakpoint** button on the first statement within the function `fact` in your application source code. Choose **View>Breakpoints** to open the Breakpoints window. Select your breakpoint in the list of breakpoints and choose the **Edit** command from the context menu.

5. To connect the log macro function to the breakpoint, type the name of the macro function, `logfact()`, in the **Action** field and click **Apply**. Close the dialog box.

6. Execute your application source code. When the breakpoint is triggered, the macro function will be executed. You can see the result in the Log window.

- Note that the expression in the **Action** field is evaluated only when the breakpoint causes the execution to really stop. If you want to log a value and then automatically continue execution, you can either:

  Use a Log breakpoint, see *Log breakpoints dialog box*, page 119
- Use the **Condition** field instead of the **Action** field. For an example, see *Performing a task and continuing execution*, page 110.

7 You can easily enhance the log macro function by, for instance, using the `__fmessage` statement instead, which will print the log information to a file. For information about the `__fmessage` statement, see *Formatted output*, page 226.

For an example where a serial port input buffer is simulated using the method of connecting a macro to a breakpoint, see the tutorial *Simulating an interrupt* in the Information Center.

Reference information on the macro language

This section gives reference information on the macro language:

- *Macro functions*, page 223
- *Macro variables*, page 224
- *Macro strings*, page 224
- *Macro statements*, page 225
- *Formatted output*, page 226.

MACRO FUNCTIONS

C-SPY macro functions consist of C-SPY variable definitions and macro statements which are executed when the macro is called. An unlimited number of parameters can be passed to a macro function, and macro functions can return a value on exit.

A C-SPY macro has this form:

```
macroName (parameterList)
{
  macroBody
}
```

where `parameterList` is a list of macro parameters separated by commas, and `macroBody` is any series of C-SPY variable definitions and C-SPY statements.

Type checking is neither performed on the values passed to the macro functions nor on the return value.

MACRO VARIABLES

A macro variable is a variable defined and allocated outside your application. It can then be used in a C-SPY expression, or you can assign application data—values of the variables in your application—to it. For more information about C-SPY expressions, see *Expressions*, page 225.

The syntax for defining one or more macro variables is:

```
__var nameList;
```

where `nameList` is a list of C-SPY variable names separated by commas.

A macro variable defined outside a macro body has global scope, and it exists throughout the whole debugging session. A macro variable defined within a macro body is created when its definition is executed and destroyed on return from the macro.

By default, macro variables are treated as signed integers and initialized to 0. When a C-SPY variable is assigned a value in an expression, it also acquires the type of that expression. For example:

Expression	What it means
`myvar = 3.5;`	`myvar` is now type `float`, value `3.5`.
`myvar = (int*)i;`	`myvar` is now type pointer to `int`, and the value is the same as `i`.

Table 13: Examples of C-SPY macro variables

In case of a name conflict between a C symbol and a C-SPY macro variable, C-SPY macro variables have a higher precedence than C variables. Note that macro variables are allocated on the debugger host and do not affect your application.

MACRO STRINGS

In addition to C types, macro variables can hold values of *macro strings*. Note that macro strings differ from C language strings.

When you write a string literal, such as `"Hello!"`, in a C-SPY expression, the value is a macro string. It is not a C-style character pointer `char*`, because `char*` must point to a sequence of characters in target memory and C-SPY cannot expect any string literal to actually exist in target memory.

You can manipulate a macro string using a few built-in macro functions, for example `__strFind` or `__subString`. The result can be a new macro string. You can concatenate macro strings using the `+` operator, for example `str + "tail"`. You can also access individual characters using subscription, for example *str*`[3]`. You can get the length of a string using `sizeof(str)`. Note that a macro string is not NULL-terminated.

The macro function `__toString` is used for converting from a NULL-terminated C string in your application (`char*` or `char[]`) to a macro string. For example, assume this definition of a C string in your application:

```
char const *cstr = "Hello";
```

Then examine these macro examples:

```
__var str;          /* A macro variable */
str = cstr          /* str is now just a pointer to char */
sizeof str          /* same as sizeof (char*), typically 2 or 4 */
str = __toString(cstr,512)  /* str is now a macro string */
sizeof str          /* 5, the length of the string */
str[1]              /* 101, the ASCII code for 'e' */
str += " World!"    /* str is now "Hello World!" */
```

See also *Formatted output*, page 226.

MACRO STATEMENTS

Statements are expected to behave in the same way as the corresponding C statements would do. The following C-SPY macro statements are accepted:

Expressions

```
expression;
```

For more information about C-SPY expressions, see *Expressions*, page 225.

Conditional statements

```
if (expression)
  statement

if (expression)
  statement
else
  statement
```

Loop statements

```
for (init_expression; cond_expression; update_expression)
  statement

while (expression)
  statement
```

```
do
  statement
while (expression);
```

Return statements

```
return;

return expression;
```

If the return value is not explicitly set, `signed int 0` is returned by default.

Blocks

Statements can be grouped in blocks.

```
{
  statement1
  statement2
  .
  .
  .
  statementN
}
```

FORMATTED OUTPUT

C-SPY provides various methods for producing formatted output:

`__message argList;`	Prints the output to the Debug Log window.
`__fmessage file, argList;`	Prints the output to the designated file.
`__smessage argList;`	Returns a string containing the formatted output.

where *`argList`* is a comma-separated list of C-SPY expressions or strings, and *`file`* is the result of the `__openFile` system macro, see *__openFile*, page 237.

To produce messages in the Debug Log window:

```
var1 = 42;
var2 = 37;
__message "This line prints the values ", var1, " and ", var2,
" in the Log window.";
```

This produces this message in the Log window:

```
This line prints the values 42 and 37 in the Log window.
```

To write the output to a designated file:

```
__fmessage myfile, "Result is ", res, "!\n";
```

To produce strings:

```
myMacroVar = __smessage 42, " is the answer.";
```

`myMacroVar` now contains the string `"42 is the answer."`.

Specifying display format of arguments

To override the default display format of a scalar argument (number or pointer) in *`argList`*, suffix it with a `:` followed by a format specifier. Available specifiers are:

`%b`	for binary scalar arguments
`%o`	for octal scalar arguments
`%d`	for decimal scalar arguments
`%x`	for hexadecimal scalar arguments
`%c`	for character scalar arguments

These match the formats available in the Watch and Locals windows, but number prefixes and quotes around strings and characters are not printed. Another example:

```
__message "The character '", cvar:%c, "' has the decimal value
", cvar;
```

Depending on the value of the variables, this produces this message:

```
The character 'A' has the decimal value 65
```

Note: A character enclosed in single quotes (a character literal) is an integer constant and is not automatically formatted as a character. For example:

```
__message 'A', " is the numeric value of the character ",
'A':%c;
```

would produce:

```
65 is the numeric value of the character A
```

Note: The default format for certain types is primarily designed to be useful in the Watch window and other related windows. For example, a value of type `char` is formatted as 'A' (0x41), while a pointer to a character (potentially a C string) is formatted as 0x8102 "Hello", where the string part shows the beginning of the string (currently up to 60 characters).

When printing a value of type `char*`, use the `%x` format specifier to print just the pointer value in hexadecimal notation, or use the system macro `__toString` to get the full string value.

Reference information on reserved setup macro function names

There are reserved setup macro function names that you can use for defining your setup macro functions. By using these reserved names, your function will be executed at defined stages during execution. For more information, see *Briefly about setup macro functions and files*, page 216.

This table summarizes the reserved setup macro function names:

Macro	Description
`execUserPreload`	Called after communication with the target system is established but before downloading the target application. Implement this macro to initialize memory locations and/or registers which are vital for loading data properly.
`execUserExecutionStarted`	Called when the debugger is about to start or resume execution. The macro is not called when performing a one-instruction assembler step, in other words, Step or Step Into in the Disassembly window. This setup macro is supported by the C-SPY E8a driver, the C-SPY E30 driver, and the C-SPY E30A driver.
`execUserExecutionStopped`	Called when the debugger has stopped execution. The macro is not called when performing a one-instruction assembler step, in other words, Step or Step Into in the Disassembly window. This setup macro is supported by the C-SPY E8a driver, the C-SPY E30 driver, and the C-SPY E30A driver.
`execUserSetup`	Called once after the target application is downloaded. Implement this macro to set up the memory map, breakpoints, interrupts, register macro files, etc.
`execUserPreReset`	Called each time just before the reset command is issued. Implement this macro to set up any required device state.
`execUserReset`	Called each time just after the reset command is issued. Implement this macro to set up and restore data.
`execUserExit`	Called once when the debug session ends. Implement this macro to save status data etc.

Table 14: C-SPY setup macros

If you define interrupts or breakpoints in a macro file that is executed at system start (using `execUserSetup`) we strongly recommend that you also make sure that they are removed at system shutdown (using `execUserExit`). An example is available in `SetupSimple.mac`, see the tutorials in the Information Center.

The reason for this is that the simulator saves interrupt settings between sessions and if they are not removed they will get duplicated every time `execUserSetup` is executed again. This seriously affects the execution speed.

Reference information on C-SPY system macros

This section gives reference information about each of the C-SPY system macros.

This table summarizes the pre-defined system macros:

Macro	Description
`__cancelAllInterrupts`	Cancels all ordered interrupts
`__cancelInterrupt`	Cancels an interrupt
`__clearBreak`	Clears a breakpoint
`__closeFile`	Closes a file that was opened by `__openFile`
`__delay`	Delays execution
`__disableInterrupts`	Disables generation of interrupts
`__driverType`	Verifies the driver type
`__enableInterrupts`	Enables generation of interrupts
`__evaluate`	Interprets the input string as an expression and evaluates it.
`__isBatchMode`	Checks if C-SPY is running in batch mode or not.
`__loadImage`	Loads an image.
`__memoryRestore`	Restores the contents of a file to a specified memory zone
`__memorySave`	Saves the contents of a specified memory area to a file
`__openFile`	Opens a file for I/O operations
`__orderInterrupt`	Generates an interrupt
`__popSimulatorInterruptExecutingStack`	Informs the interrupt simulation system that an interrupt handler has finished executing
`__readFile`	Reads from the specified file
`__readFileByte`	Reads one byte from the specified file

Table 15: Summary of system macros

Macro	Description
`__readMemory8`, `__readMemoryByte`	Reads one byte from the specified memory location
`__readMemory16`	Reads two bytes from the specified memory location
`__readMemory32`	Reads four bytes from the specified memory location
`__registerMacroFile`	Registers macros from the specified file
`__resetFile`	Rewinds a file opened by `__openFile`
`__setCodeBreak`	Sets a code breakpoint
`__setDataBreak`	Sets a data breakpoint
`__setLogBreak`	Sets a log breakpoint
`__setSimBreak`	Sets a simulation breakpoint
`__setTraceStartBreak`	Sets a trace start breakpoint
`__setTraceStopBreak`	Sets a trace stop breakpoint
`__sourcePosition`	Returns the file name and source location if the current execution location corresponds to a source location
`__strFind`	Searches a given string for the occurrence of another string
`__subString`	Extracts a substring from another string
`__targetDebuggerVersion`	Returns the version of the target debugger
`__toLower`	Returns a copy of the parameter string where all the characters have been converted to lower case
`__toString`	Prints strings
`__toUpper`	Returns a copy of the parameter string where all the characters have been converted to upper case
`__unloadImage`	Unloads a debug image.
`__writeFile`	Writes to the specified file
`__writeFileByte`	Writes one byte to the specified file
`__writeMemory8`, `__writeMemoryByte`	Writes one byte to the specified memory location
`__writeMemory16`	Writes a two-byte word to the specified memory location
`__writeMemory32`	Writes a four-byte word to the specified memory location

Table 15: Summary of system macros (Continued)

__cancelAllInterrupts

Syntax `__cancelAllInterrupts()`

Return value `int 0`

Applicability The C-SPY Simulator.

Description Cancels all ordered interrupts.

__cancelInterrupt

Syntax `__cancelInterrupt(`*`interrupt_id`*`)`

Parameters *`interrupt_id`*

The value returned by the corresponding `__orderInterrupt` macro call (`unsigned long`).

Return value

Result	Value
Successful	`int 0`
Unsuccessful	Non-zero error number

Table 16: __cancelInterrupt return values

Applicability The C-SPY Simulator.

Description Cancels the specified interrupt.

__clearBreak

Syntax `__clearBreak(`*`break_id`*`)`

Parameters *`break_id`*

The value returned by any of the set breakpoint macros.

Return value `int 0`

Description Clears a user-defined breakpoint.

See also *Breakpoints*, page 101.

__closeFile

Syntax `__closeFile(`*`fileHandle`*`)`

Parameters

`fileHandle`

A macro variable used as filehandle by the `__openFile` macro.

Return value `int 0`

Description Closes a file previously opened by `__openFile`.

__delay

Syntax `__delay(`*`value`*`)`

Parameters

`value`

The number of milliseconds to delay execution.

Return value `int 0`

Description Delays execution the specified number of milliseconds.

__disableInterrupts

Syntax `__disableInterrupts()`

Return value

Result	**Value**
Successful	`int 0`
Unsuccessful	Non-zero error number

Table 17: __disableInterrupts return values

Applicability The C-SPY Simulator.

Description Disables the generation of interrupts.

__driverType

Syntax

`__driverType(driver_id)`

Parameters

driver_id

A string corresponding to the driver you want to check for. Choose one of these:

`"sim"` corresponds to the simulator driver.

`"emue8a"` corresponds to the C-SPY E8a Emulator driver

`"emue30"` corresponds to the C-SPY E30 Emulator driver

`"emue30a"` corresponds to the C-SPY E30A Emulator driver

Return value

Result	Value
Successful	`1`
Unsuccessful	`0`

Table 18: __driverType return values

Description

Checks to see if the current C-SPY driver is identical to the driver type of the *driver_id* parameter.

Example

```
__driverType("sim")
```

If the simulator is the current driver, the value `1` is returned. Otherwise `0` is returned.

__enableInterrupts

Syntax

`__enableInterrupts()`

Return value

Result	Value
Successful	`int 0`
Unsuccessful	Non-zero error number

Table 19: __enableInterrupts return values

Applicability

The C-SPY Simulator.

Description

Enables the generation of interrupts.

__evaluate

Syntax

`__evaluate(`*`string, valuePtr`*`)`

Parameters

`string`

Expression string.

`valuePtr`

Pointer to a macro variable storing the result.

Return value

Result	Value
Successful	`int 0`
Unsuccessful	`int 1`

Table 20: __evaluate return values

Description

This macro interprets the input string as an expression and evaluates it. The result is stored in a variable pointed to by *`valuePtr`*.

Example

This example assumes that the variable `i` is defined and has the value 5:

```
__evaluate("i + 3", &myVar)
```

The macro variable `myVar` is assigned the value 8.

__isBatchMode

Syntax

`__isBatchMode()`

Return value

Result	Value
True	`int 1`
False	`int 0`

Table 21: __isBatchMode return values

Description

This macro returns True if the debugger is running in batch mode, otherwise it returns False.

__loadImage

Syntax

`__loadImage(`*`path, offset, debugInfoOnly`*`)`

Parameters

path

A string that identifies the path to the image to download. The path must either be absolute or use argument variables. For information about argument variables, see the *IDE Project Management and Building Guide*.

offset

An integer that identifies the offset to the destination address for the downloaded image.

debugInfoOnly

A non-zero integer value if no code or data should be downloaded to the target system, which means that C-SPY will only read the debug information from the debug file. Or, 0 (zero) for download.

Return value

Value	**Result**
Non-zero integer number	A unique module identification.
`int 0`	Loading failed.

Table 22: __loadImage return values

Description

Loads an image (debug file).

Example 1

Your system consists of a ROM library and an application. The application is your active project, but you have a debug file corresponding to the library. In this case you can add this macro call in the `execUserSetup` macro in a C-SPY macro file, which you associate with your project:

```
__loadImage(ROMfile, 0x8000, 1);
```

This macro call loads the debug information for the ROM library *ROMfile* without downloading its contents (because it is presumably already in ROM). Then you can debug your application together with the library.

Example 2

Your system consists of a ROM library and an application, but your main concern is the library. The library needs to be programmed into flash memory before a debug session. While you are developing the library, the library project must be the active project in the IDE. In this case you can add this macro call in the `execUserSetup` macro in a C-SPY macro file, which you associate with your project:

```
__loadImage(ApplicationFile, 0x8000, 0);
```

The macro call loads the debug information for the application and downloads its contents (presumably into RAM). Then you can debug your library together with the application.

See also — *Images*, page 270 and *Loading multiple images*, page 35.

__memoryRestore

Syntax — `__memoryRestore(zone, filename)`

Parameters

`zone`
: A string that specifies the memory zone, see *C-SPY memory zones*, page 130.

`filename`
: A string that specifies the file to be read. The filename must include a path, which must either be absolute or use argument variables. For information about argument variables, see the *IDE Project Management and Building Guide*.

Return value — `int 0`

Description — Reads the contents of a file and saves it to the specified memory zone.

Example

```
__memoryRestore("Memory", "c:\\temp\\saved_memory.hex");
```

See also — *Memory Restore dialog box*, page 140.

__memorySave

Syntax — `__memorySave(start, stop, format, filename)`

Parameters

`start`
: A string that specifies the first location of the memory area to be saved.

`stop`
: A string that specifies the last location of the memory area to be saved.

`format`
: A string that specifies the format to be used for the saved memory. Choose between:

`intel-extended`

`motorola`

`motorola-s19`

`motorola-s28`

`motorola-s37`.

`filename`

A string that specifies the file to write to. The filename must include a path, which must either be absolute or use argument variables. For information about argument variables, see the *IDE Project Management and Building Guide*.

Return value

`int 0`

Description

Saves the contents of a specified memory area to a file.

Example

```
__memorySave("Memory:0x00", "Memory:0xFF", "intel-extended",
"c:\\temp\\saved_memory.hex");
```

See also

Memory Save dialog box, page 139.

__openFile

Syntax

`__openFile(`*`filename, access`*`)`

Parameters

`filename`

The file to be opened. The filename must include a path, which must either be absolute or use argument variables. For information about argument variables, see the *IDE Project Management and Building Guide*.

`access`

The access type (string).

These are mandatory but mutually exclusive:

`"a"` append, new data will be appended at the end of the open file

`"r"` read

`"w"` write

These are optional and mutually exclusive:

`"b"` binary, opens the file in binary mode

`"t"` ASCII text, opens the file in text mode

This access type is optional:

`"+"` together with `r`, `w`, or `a`; `r+` or `w+` is *read* and *write*, while `a+` is *read* and *append*

Return value

Result	Value
Successful	The file handle
Unsuccessful	An invalid file handle, which tests as False

Table 23: __openFile return values

Description

Opens a file for I/O operations. The default base directory of this macro is where the currently open project file (`*.ewp`) is located. The argument to `__openFile` can specify a location relative to this directory. In addition, you can use argument variables such as `$PROJ_DIR$` and `$TOOLKIT_DIR$` in the path argument.

Example

```
__var myFileHandle;          /* The macro variable to contain */
                             /* the file handle */
myFileHandle = __openFile("$PROJ_DIR$\\Debug\\Exe\\test.tst",
"r");
if (myFileHandle)
{
    /* successful opening */
}
```

See also

For information about argument variables, see the *IDE Project Management and Building Guide*.

__orderInterrupt

Syntax

```
__orderInterrupt(specification, first_activation,
                 repeat_interval, variance, infinite_hold_time,
                 hold_time, probability)
```

Parameters

`specification`

The interrupt (string). The specification can either be the full specification used in the device description file (`ddf`) or only the name. In the latter case the interrupt system will automatically get the description from the device description file.

`first_activation`

The first activation time in cycles (integer)

`repeat_interval`

The periodicity in cycles (integer)

`variance`

The timing variation range in percent (integer between 0 and 100) *`infinite_hold_time`*

1 if infinite, otherwise 0.

`hold_time`

The hold time (integer)

`probability`

The probability in percent (integer between 0 and 100)

Return value

The macro returns an interrupt identifier (`unsigned long`).

If the syntax of *`specification`* is incorrect, it returns -1.

Applicability

The C-SPY Simulator.

Description

Generates an interrupt.

Example

This example generates a repeating interrupt using an infinite hold time first activated after 4000 cycles

```
__orderInterrupt( "USARTR_VECTOR", 4000, 2000, 0, 1, 0, 100 );
```

__popSimulatorInterruptExecutingStack

Syntax

```
__popSimulatorInterruptExecutingStack(void)
```

Return value

`int 0`

Applicability

The C-SPY Simulator.

Description

Informs the interrupt simulation system that an interrupt handler has finished executing, as if the normal instruction used for returning from an interrupt handler was executed.

This is useful if you are using interrupts in such a way that the normal instruction for returning from an interrupt handler is not used, for example in an operating system with task-switching. In this case, the interrupt simulation system cannot automatically detect that the interrupt has finished executing.

See also

Simulating an interrupt in a multi-task system, page 201.

__readFile

Syntax `__readFile(`*`fileHandle`*`, `*`valuePtr`*`)`

Parameters *`fileHandle`*

A macro variable used as filehandle by the `__openFile` macro.

`valuePtr`

A pointer to a variable.

Return value

Result	Value
Successful	0
Unsuccessful	Non-zero error number

Table 24: __readFile return values

Description Reads a sequence of hexadecimal digits from the given file and converts them to an `unsigned long` which is assigned to the *`value`* parameter, which should be a pointer to a macro variable.

Example

```
__var number;
if (__readFile(myFileHandle, &number) == 0)
{
  // Do something with number
}
```

__readFileByte

Syntax `__readFileByte(`*`fileHandle`*`)`

Parameters *`fileHandle`*

A macro variable used as filehandle by the `__openFile` macro.

Return value `-1` upon error or end-of-file, otherwise a value between `0` and `255`.

Description Reads one byte from a file.

Example

```
__var byte;
while ( (byte = __readFileByte(myFileHandle)) != -1 )
{
  /* Do something with byte */
}
```

__readMemory8, __readMemoryByte

Syntax

`__readMemory8(`*`address`*`, `*`zone`*`)`
`__readMemoryByte(`*`address`*`, `*`zone`*`)`

Parameters

`address`

The memory address (integer).

`zone`

A string that specifies the memory zone, see *C-SPY memory zones*, page 130.

Return value

The macro returns the value from memory.

Description

Reads one byte from a given memory location.

Example

```
__readMemory8(0x0108, "Memory");
```

__readMemory16

Syntax

`__readMemory16(`*`address`*`, `*`zone`*`)`

Parameters

`address`

The memory address (integer).

`zone`

A string that specifies the memory zone, see *C-SPY memory zones*, page 130.

Return value

The macro returns the value from memory.

Description

Reads a two-byte word from a given memory location.

Example

```
__readMemory16(0x0108, "Memory");
```

__readMemory32

Syntax

`__readMemory32(`*`address`*`, `*`zone`*`)`

Parameters

`address`

The memory address (integer).

`zone`

A string that specifies the memory zone, see *C-SPY memory zones*, page 130.

Return value	The macro returns the value from memory.
Description	Reads a four-byte word from a given memory location.
Example	`__readMemory32(0x0108, "Memory");`

__registerMacroFile

Syntax `__registerMacroFile(`*`filename`*`)`

Parameters *`filename`*

A file containing the macros to be registered (string). The filename must include a path, which must either be absolute or use argument variables. For information about argument variables, see the *IDE Project Management and Building Guide*.

Return value `int 0`

Description Registers macros from a setup macro file. With this function you can register multiple macro files during C-SPY startup.

Example `__registerMacroFile("c:\\testdir\\macro.mac");`

See also *Registering and executing using setup macros and setup files*, page 220.

__resetFile

Syntax `__resetFile(`*`fileHandle`*`)`

Parameters *`fileHandle`*

A macro variable used as filehandle by the `__openFile` macro.

Return value `int 0`

Description Rewinds a file previously opened by `__openFile`.

__setCodeBreak

Syntax

`__setCodeBreak(`*`location, count, condition, cond_type, action`*`)`

Parameters

`location`

A string that defines the code location of the breakpoint, either a valid C-SPY expression whose value evaluates to a valid address, an absolute location, or a source location. For more information about the location types, see Enter Location dialog box, page 25.

`count`

The number of times that a breakpoint condition must be fulfilled before a break occurs (integer).

`condition`

The breakpoint condition (string).

`cond_type`

The condition type; either `"CHANGED"` or `"TRUE"` (string).

`action`

An expression, typically a call to a macro, which is evaluated when the breakpoint is detected.

Return value

Result	Value
Successful	An unsigned integer uniquely identifying the breakpoint. This value must be used to clear the breakpoint.
Unsuccessful	0

Table 25: __setCodeBreak return values

Description

Sets a code breakpoint, that is, a breakpoint which is triggered just before the processor fetches an instruction at the specified location.

Examples

```
__setCodeBreak("{D:\\src\\prog.c}.12.9", 3, "d>16", "TRUE",
"ActionCode()");
```

This example sets a code breakpoint on the label `main` in your source:

```
__setCodeBreak("main", 0, "1", "TRUE", "");
```

See also

Breakpoints, page 101.

__setDataBreak

Syntax

```
__setDataBreak(location, count, condition, cond_type, access,
               action)
```

Parameters

location

A string that defines the data location of the breakpoint, either a valid C-SPY expression whose value evaluates to a valid address or an absolute location. For more information about the location types, see Enter Location dialog box, page 25.

count

The number of times that a breakpoint condition must be fulfilled before a break occurs (integer).

condition

The breakpoint condition (string).

cond_type

The condition type; either `"CHANGED"` or `"TRUE"` (string).

access

The memory access type: `"R"`, for read, `"W"` for write, or `"RW"` for read/write.

action

An expression, typically a call to a macro, which is evaluated when the breakpoint is detected.

Return value

Result	Value
Successful	An unsigned integer uniquely identifying the breakpoint. This value must be used to clear the breakpoint.
Unsuccessful	0

Table 26: __setDataBreak return values

Applicability

The C-SPY Simulator.

Description

Sets a data breakpoint, that is, a breakpoint which is triggered directly after the processor has read or written data at the specified location.

Example

```
__var brk;
brk = __setDataBreak("Memory:0x2710", 3, "d>6", "TRUE",
      "W", "ActionData()");
...
__clearBreak(brk);
```

See also

Breakpoints, page 101.

__setDataLogBreak

Syntax

```
__setDataLogBreak(location, access,,
```

Parameters

location

A string that defines the data location of the breakpoint, either a valid C-SPY expression whose value evaluates to a valid address or an absolute location. For more information about the location types, see Enter Location dialog box, page 25.

access

The memory access type: `"R"`, for read, `"W"` for write, or `"RW"` for read/write.

Return value

Result	Value
Successful	An unsigned integer uniquely identifying the breakpoint. This value must be used to clear the breakpoint.
Unsuccessful	0

Table 27: __setDataLogBreak return values

Applicability

The C-SPY Simulator

Description

Sets a data log breakpoint, that is, a breakpoint which is triggered when the processor reads or writes data at the specified location. Note that a data log breakpoint does not stop the execution it just generates a data log.

Example

```
__var brk;
brk = __setDataLogBreak("Memory:0x4710", "R", );
...
__clearBreak(brk);
```

See also

Breakpoints, page 101 and *Getting started using data logging*, page 84.

__setLogBreak

Syntax

```
__setLogBreak(location, message, msg_type, condition,
              cond_type)
```

Parameters

location

A string that defines the code location of the breakpoint, either a valid C-SPY expression whose value evaluates to a valid address, an absolute location, or a source location. For more information about the location types, see Enter Location dialog box, page 25.

message

The message text.

msg_type

The message type; choose between:

`TEXT`, the message is written word for word.

`ARGS`, the message is interpreted as a comma-separated list.

of C-SPY expressions or strings.

condition

The breakpoint condition (string).

cond_type

The condition type; either `"CHANGED"` or `"TRUE"` (string).

Return value

Result	Value
Successful	An unsigned integer uniquely identifying the breakpoint. The same value must be used when you want to clear the breakpoint.
Unsuccessful	0

Table 28: __setLogBreak return values

Description

Sets a log breakpoint, that is, a breakpoint which is triggered when an instruction is fetched from the specified location. If you have set the breakpoint on a specific machine instruction, the breakpoint will be triggered and the execution will temporarily halt and print the specified message in the C-SPY Debug Log window.

Example

```
__var logBp1;
__var logBp2;

logOn()
{
  logBp1 = __setLogBreak ("{C:\\temp\\Utilities.c}.23.1",
    "\"Entering trace zone at :\", #PC:%X", "ARGS", "1", "TRUE");
  logBp2 = __setLogBreak ("{C:\\temp\\Utilities.c}.30.1",
    "Leaving trace zone...", "TEXT", "1", "TRUE");
}

logOff()
{
  __clearBreak(logBp1);
  __clearBreak(logBp2);
}
```

See also

Formatted output, page 226 and *Breakpoints*, page 101.

__setSimBreak

Syntax

`__setSimBreak(`*`location, access, action`*`)`

Parameters

`location`

A string that defines the data location of the breakpoint, either a valid C-SPY expression whose value evaluates to a valid address or an absolute location. For more information about the location types, see Enter Location dialog box, page 25.

`count`

The number of times that a breakpoint condition must be fulfilled before a break occurs (integer).

`condition`

The breakpoint condition (string).

`cond_type`

The condition type; either `"CHANGED"` or `"TRUE"` (string).

`action`

An expression, typically a call to a macro, which is evaluated when the breakpoint is detected.

Return value

Result	Value
Successful	An unsigned integer uniquely identifying the breakpoint. This value must be used to clear the breakpoint.
Unsuccessful	0

Table 29: __setSimBreak return values

Applicability

The C-SPY Simulator.

Description

Use this system macro to set *immediate* breakpoints, which will halt instruction execution only temporarily. This allows a C-SPY macro function to be called when the processor is about to read data from a location or immediately after it has written data. Instruction execution will resume after the action.

This type of breakpoint is useful for simulating memory-mapped devices of various kinds (for instance serial ports and timers). When the processor reads at a memory-mapped location, a C-SPY macro function can intervene and supply the appropriate data. Conversely, when the processor writes to a memory-mapped location, a C-SPY macro function can act on the value that was written.

__setTraceStartBreak

Syntax

`__setTraceStartBreak(`*`location`*`)`

Parameters

`location`

A string that defines the code location of the breakpoint, either a valid C-SPY expression whose value evaluates to a valid address, an absolute location, or a source location. For more information about the location types, see Enter Location dialog box, page 25.

`count`

The number of times that a breakpoint condition must be fulfilled before a break occurs (integer)

`condition`

The breakpoint condition (string)

`cond_type`

The condition type; either `"CHANGED"` or `"TRUE"` (string)

`action`

An expression, typically a call to a macro, which is evaluated when the breakpoint is detected

Return value

Result	Value
Successful	An unsigned integer uniquely identifying the breakpoint. The same value must be used when you want to clear the breakpoint.
Unsuccessful	0

Table 30: __setTraceStartBreak return values

Applicability

The C-SPY Simulator.

Description

Sets a breakpoint at the specified location. When that breakpoint is triggered, the trace system is started.

Example

```
__var startTraceBp;
__var stopTraceBp;

traceOn()
{
  startTraceBp = __setTraceStartBreak
    ("{C:\\TEMP\\Utilities.c}.23.1");
  stopTraceBp = __setTraceStopBreak
    ("{C:\\temp\\Utilities.c}.30.1");
}

traceOff()
{
  __clearBreak(startTraceBp);
  __clearBreak(stopTraceBp);
}
```

See also

Breakpoints, page 101.

__setTraceStopBreak

Syntax

`__setTraceStopBreak(`*`location`*`)`

Parameters

`location`

A string that defines the code location of the breakpoint, either a valid C-SPY expression whose value evaluates to a valid address, an absolute location, or a source location. For more information about the location types, see Enter Location dialog box, page 25.

count

The number of times that a breakpoint condition must be fulfilled before a break occurs (integer)

condition

The breakpoint condition (string)

cond_type

The condition type; either `"CHANGED"` or `"TRUE"` (string)

action

An expression, typically a call to a macro, which is evaluated when the breakpoint is detected

Return value

Result	Value
Successful	An unsigned integer uniquely identifying the breakpoint. The same value must be used when you want to clear the breakpoint.
Unsuccessful	`int 0`

Table 31: __setTraceStopBreak return values

Applicability

The C-SPY Simulator.

Description

Sets a breakpoint at the specified location. When that breakpoint is triggered, the trace system is stopped.

Example

See *__setTraceStartBreak*, page 248.

See also

Breakpoints, page 101.

__sourcePosition

Syntax

`__sourcePosition(`*`linePtr, colPtr`*`)`

Parameters

linePtr

Pointer to the variable storing the line number

colPtr

Pointer to the variable storing the column number

Return value

Result	Value
Successful	Filename string
Unsuccessful	Empty ("") string

Table 32: __sourcePosition return values

Description

If the current execution location corresponds to a source location, this macro returns the filename as a string. It also sets the value of the variables, pointed to by the parameters, to the line and column numbers of the source location.

__strFind

Syntax

`__strFind(macroString, pattern, position)`

Parameters

`macroString`

A macro string.

`pattern`

The string pattern to search for

`position`

The position where to start the search. The first position is `0`

Return value

The position where the pattern was found or `-1` if the string is not found.

Description

This macro searches a given string (*`macroString`*) for the occurrence of another string (*`pattern`*).

Example

```
__strFind("Compiler", "pile", 0)  = 3
__strFind("Compiler", "foo", 0)   = -1
```

See also

Macro strings, page 224.

__subString

Syntax

`__subString(macroString, position, length)`

Parameters

`macroString`

A macro string.

`position`

The start position of the substring. The first position is 0.

`length`
The length of the substring

Return value
A substring extracted from the given macro string.

Description
This macro extracts a substring from another string (*`macroString`*).

Example
```
__subString("Compiler", 0, 2)
```
The resulting macro string contains `Co`.
```
__subString("Compiler", 3, 4)
```
The resulting macro string contains `pile`.

See also
Macro strings, page 224.

__targetDebuggerVersion

Syntax
`__targetDebuggerVersion`

Return value
A string that represents the version number of the C-SPY debugger processor module.

Description
This macro returns the version number of the C-SPY debugger processor module.

Example
```
__var toolVer;
toolVer = __targetDebuggerVersion();
__message "The target debugger version is, ", toolVer;
```

__toLower

Syntax
`__toLower(`*`macroString`*`)`

Parameters
`macroString`
A macro string.

Return value
The converted macro string.

Description
This macro returns a copy of the parameter *`macroString`* where all the characters have been converted to lower case.

Example
```
__toLower("IAR")
```
The resulting macro string contains `iar`.

```
__toLower("Mix42")
```

The resulting macro string contains `mix42`.

See also *Macro strings*, page 224.

__toString

Syntax `__toString(C_string, maxlength)`

Parameters

`C_string`
Any null-terminated C string.

`maxlength`
The maximum length of the returned macro string.

Return value Macro string.

Description This macro is used for converting C strings (`char*` or `char[]`) into macro strings.

Example Assuming your application contains this definition:

```
char const * hptr = "Hello World!";
```

this macro call:

```
__toString(hptr, 5)
```

would return the macro string containing `Hello`.

See also *Macro strings*, page 224.

__toUpper

Syntax `__toUpper(macroString)`

Parameters

`macroString`
A macro string.

Return value The converted string.

Description This macro returns a copy of the parameter *`macroString`* where all the characters have been converted to upper case.

Example

```
__toUpper("string")
```

The resulting macro string contains `STRING`.

See also

Macro strings, page 224.

__unloadImage

Syntax

`__unloadImage(`*`module_id`*`)`

Parameters

`module_id`

An integer which represents a unique module identification, which is retrieved as a return value from the corresponding `__loadImage` C-SPY macro.

Return value

Value	Result
`module_id`	A unique module identification (the same as the input parameter).
`int 0`	The unloading failed.

Table 33: __unloadImage return values

Description

Unloads debug information from an already downloaded image.

See also

Loading multiple images, page 35 and *Images*, page 270.

__writeFile

Syntax

`__writeFile(`*`fileHandle, value`*`)`

Parameters

`fileHandle`

A macro variable used as filehandle by the `__openFile` macro.

`value`

An integer.

Return value

`int 0`

Description

Prints the integer value in hexadecimal format (with a trailing space) to the file *`file`*.

Note: The `__fmessage` statement can do the same thing. The `__writeFile` macro is provided for symmetry with `__readFile`.

__writeFileByte

Syntax `__writeFileByte(`*`fileHandle, value`*`)`

Parameters *`fileHandle`*

A macro variable used as filehandle by the `__openFile` macro.

`value`

An integer.

Return value `int 0`

Description Writes one byte to the file *`fileHandle`*.

__writeMemory8, __writeMemoryByte

Syntax `__writeMemory8(`*`value, address, zone`*`)`
`__writeMemoryByte(`*`value, address, zone`*`)`

Parameters *`value`*

An integer.

`address`

The memory address (integer).

`zone`

A string that specifies the memory zone, see *C-SPY memory zones*, page 130.

Return value `int 0`

Description Writes one byte to a given memory location.

Example

```
__writeMemory8(0x2F, 0x8020, "Memory");
```

__writeMemory16

Syntax `__writeMemory16(`*`value, address, zone`*`)`

Parameters *`value`*

An integer.

address
: The memory address (integer).

zone
: A string that specifies the memory zone, see *C-SPY memory zones*, page 130.

Return value `int 0`

Description Writes two bytes to a given memory location.

Example

```
__writeMemory16(0x2FFF, 0x8020, "Memory");
```

__writeMemory32

Syntax `__writeMemory32(`*`value, address, zone`*`)`

Parameters

value
: An integer.

address
: The memory address (integer).

zone
: A string that specifies the memory zone, see *C-SPY memory zones*, page 130.

Return value `int 0`

Description Writes four bytes to a given memory location.

Example

```
__writeMemory32(0x5555FFFF, 0x8020, "Memory");
```

The C-SPY Command Line Utility—cspybat

This chapter describes how you can execute C-SPY® in batch mode, using the C-SPY Command Line Utility—cspybat.exe. More specifically, this means:

- Using C-SPY in batch mode
- Summary of C-SPY command line options
- Reference information on C-SPY command line options.

Using C-SPY in batch mode

You can execute C-SPY in batch mode if you use the command line utility `cspybat`, installed in the directory `common\bin`.

INVOCATION SYNTAX

The invocation syntax for `cspybat` is:

```
cspybat processor_DLL driver_DLL debug_file [cspybat_options]
        --backend driver_options
```

Note: In those cases where a filename is required—including the DLL files—you are recommended to give a full path to the filename.

Parameters

The parameters are:

Parameter	Description
processor_DLL	The processor-specific DLL file; available in `r32c\bin`.
driver_DLL	The C-SPY driver DLL file; available in `r32c\bin`.
debug_file	The object file that you want to debug (filename extension `d53`).
cspybat_options	The command line options that you want to pass to `cspybat`. Note that these options are optional. For information about each option, see *Reference information on C-SPY command line options*, page 260.

Table 34: cspybat parameters

Parameter	Description
`--backend`	Marks the beginning of the parameters to the C-SPY driver; all options that follow will be sent to the driver. Note that this option is mandatory.
`driver_options`	The command line options that you want to pass to the C-SPY driver. Note that some of these options are mandatory and some are optional. For information about each option, see *Reference information on C-SPY command line options*, page 260.

Table 34: cspybat parameters (Continued)

Example

In the following example, *`EW_DIR`* represents the full path of the directory where you have installed IAR Embedded Workbench and *`PROJ_DIR`* is the path of the project directory.

Starting cspybat using the simulator driver

```
EW_DIR\common\bin\cspybat EW_DIR\r32c\bin\r32cproc.dll
EW_DIR\r32c\bin\r32csim.dll PROJ_DIR\myproject.d53 --plugin
EW_DIR\r32c\bin\r32cbat.dll --backend -d sim -p
EW_DIR\r32c\config\debugger\r32c.ddf
```

Starting cspybat using the E8a driver

```
EW_DIR\common\bin\cspybat EW_DIR\r32c\bin\r32cproc.dll
EW_DIR\r32c\bin\r32cemue8a.dll PROJ_DIR\myproject.d53 --plugin
EW_DIR\r32c\bin\r32cbat.dll --backend -d emue8a -p
EW_DIR\r32c\config\debugger\R5F64189.ddf
```

Starting cspybat using the E30A driver

To run `cspybat` for the E30A emulator, the following preparation steps must be performed:

1 Start IAR Embedded Workbench and enter your emulator hardware settings in the **Hardware Setup** dialog box. These settings will be stored in a file with the following name and location: *`PROJ_DIR`*`\settings\project_name.dni`

2 Set up the environment variable `CSPYBAT_INIFILE` to point to the `.dni` file where your hardware settings are stored. For example:

```
set CSPYBAT_INIFILE = PROJ_DIR\settings\project_name.dni
```

This example starts `cspybat` using the E30A emulator driver:

```
EW_DIR\common\bin\cspybat EW_DIR\r32c\bin\r32cproc.dll
EW_DIR\r32c\bin\r32cemue30a.dll PROJ_DIR\myproject.d53 --plugin
EW_DIR\r32c\bin\r32cbat.dll --backend -d emue30a -p
EW_DIR\r32c\config\debugger\R5F64213.ddf --drv_communication USB
```

OUTPUT

When you run `cspybat`, these types of output can be produced:

- Terminal output from `cspybat` itself

  All such terminal output is directed to `stderr`. Note that if you run `cspybat` from the command line without any arguments, the `cspybat` version number and all available options including brief descriptions are directed to `stdout` and displayed on your screen.
- Terminal output from the application you are debugging

  All such terminal output is directed to `stdout`, provided that you have used the `--plugin` option. See *--plugin*, page 264.
- Error return codes

  `cspybat` return status information to the host operating system that can be tested in a batch file. For *successful*, the value `int 0` is returned, and for *unsuccessful* the value `int 1` is returned.

USING AN AUTOMATICALLY GENERATED BATCH FILE

When you use C-SPY in the IDE, C-SPY generates a batch file *`projectname`*`.cspy.bat` every time C-SPY is initialized. You can find the file in the directory `$PROJ_DIR$\settings`. This batch file contains the same settings as in the IDE, and you can use it from the command line to start `cspybat`. The file also contains information about required modifications.

Summary of C-SPY command line options

GENERAL CSPYBAT OPTIONS

`--backend`	Marks the beginning of the parameters to be sent to the C-SPY driver (mandatory).
`--code_coverage_file`	Enables the generation of code coverage information and places it in a specified file.
`--cycles`	Specifies the maximum number of cycles to run.

`--download_only`	Downloads a code image without starting a debug session afterwards.
`--macro`	Specifies a macro file to be used.
`--plugin`	Specifies a plugin file to be used.
`--silent`	Omits the sign-on message.
`--timeout`	Limits the maximum allowed execution time.

OPTIONS AVAILABLE FOR ALL C-SPY DRIVERS

`-B`	Enables batch mode (mandatory).
`-d`	Specifies the C-SPY driver to be used.
`-p`	Specifies the device description file to be used.

OPTIONS AVAILABLE FOR THE SIMULATOR DRIVER

`--disable_interrupts`	Disables the interrupt simulation.
`--mapu`	Activates memory access checking.

OPTIONS AVAILABLE FOR THE C-SPY HARDWARE DRIVER

`--drv_communication`	Specifies the communication link to be used.
`--verify_download`	Verifies the executable image.

Reference information on C-SPY command line options

This section gives detailed reference information about each `cspybat` option and each option available to the C-SPY drivers.

-B

Syntax	`-B`
Applicability	All C-SPY drivers.
Description	Use this option to enable batch mode.

--backend

Syntax `--backend {`*`driver options`*`}`

Parameters *`driver options`*

Any option available to the C-SPY driver you are using.

Applicability Sent to `cspybat` (mandatory).

Description Use this option to send options to the C-SPY driver. All options that follow `--backend` will be passed to the C-SPY driver, and will not be processed by `cspybat` itself.

--code_coverage_file

Syntax `--code_coverage_file` *`file`*

Parameters *`file`*

The name of the destination file for the code coverage information.

Applicability Sent to `cspybat`.

Description Use this option to enable the generation of code coverage information. The code coverage information will be generated after the execution has completed and you can find it in the specified file.

Note that this option requires that the C-SPY driver you are using supports code coverage. If you try to use this option with a C-SPY driver that does not support code coverage, an error message will be directed to `stderr`.

See also *Code coverage*, page 191.

--cycles

Syntax `--cycles` *`cycles`*

Parameters *`cycles`*

The number of cycles to run.

Applicability Sent to `cspybat`.

Description	Use this option to specify the maximum number of cycles to run. If the target program executes longer than the number of cycles specified, the target program will be aborted. Using this option requires that the C-SPY driver you are using supports a cycle counter, and that it can be sampled while executing.

-d

Syntax `-d`

Parameters

`sim`	Specifies the simulator driver.
`emue8a`	Specifies the E8a emulator driver.
`emue30`	Specifies the E30 emulator driver.
`emue30a`	Specifies the E30A emulator driver.

Applicability All C-SPY drivers.

Description Use this option to specify the C-SPY driver to be used.

--disable_interrupts

Syntax `--disable_interrupts`

Applicability The C-SPY Simulator driver.

Description Use this option to disable the interrupt simulation.

To set this option, choose **Simulator>Interrupt Setup** and deselect the **Enable interrupt simulation** option.

--download_only

Syntax `--download_only`

Applicability Sent to `cspybat`.

Description Use this option to download the code image without starting a debug session afterwards.

To set related options, choose:

Project>Download

--drv_communication

Syntax	`--drv_communication=`*`connection number`*
Parameters	

`connection`	Specifies the USB connection.
`number`	Specifies the serial number of the USB connection.

Applicability	The C-SPY E30 driver and the C-SPY E30A driver.
Description	Use this option to choose a communication link.

Project>Options>Debugger>*driver*>Communication>USB

--macro

Syntax	`--macro `*`filename`*
Parameters	*`filename`* The C-SPY macro file to be used (filename extension `mac`).
Applicability	Sent to `cspybat`.
Description	Use this option to specify a C-SPY macro file to be loaded before executing the target application. This option can be used more than once on the command line.
See also	*Briefly about using C-SPY macros*, page 216.

--mapu

Syntax	`--mapu`
Applicability	The C-SPY simulator driver.

Description	Specify this option to use the segment information in the debug file for memory access checking. During the execution, the simulator will then check for accesses to unspecified memory ranges. If any such access is found, the C function call stack and a message will be printed on `stderr` and the execution will stop.
See also	*Memory access checking*, page 132.

To set related options, choose:

Simulator>Memory Access Setup

-p

Syntax	`-p filename`
Parameters	*`filename`* The device description file to be used.
Applicability	All C-SPY drivers.
Description	Use this option to specify the device description file to be used.
See also	*Selecting a device description file*, page 33.

--plugin

Syntax	`--plugin filename`
Parameters	*`filename`* The plugin file to be used (filename extension `dll`).
Applicability	Sent to `cspybat`.
Description	Certain C/C++ standard library functions, for example `printf`, can be supported by C-SPY—for example, the C-SPY Terminal I/O window—instead of by real hardware devices. To enable such support in `cspybat`, a dedicated plugin module called `r32cbat.dll` located in the `r32c\bin` directory must be used. Use this option to include this plugin during the debug session. This option can be used more than once on the command line.

Note: You can use this option to include also other plugin modules, but in that case the module must be able to work with `cspybat` specifically. This means that the C-SPY plugin modules located in the `common\plugin` directory cannot normally be used with `cspybat`.

--silent

Syntax

```
--silent
```

Applicability

Sent to `cspybat`.

Description

Use this option to omit the sign-on message.

--suppress_download

Syntax

```
--suppress_download
```

Applicability

All C-SPY hardware debugger drivers.

Description

Use this option to suppress the downloading of the executable image to a non-volatile type of target memory. The image corresponding to the debugged application must already exist in the target.

If this option is combined with the option `--verify_download`, the debugger will read back the executable image from memory and verify that it is identical to the debugged application.

Project>Options>Debugger>*driver*>Download>Verify download

--timeout

Syntax

```
--timeout milliseconds
```

Parameters

`milliseconds`
The number of milliseconds before the execution stops.

Applicability

Sent to `cspybat`.

Description

Use this option to limit the maximum allowed execution time.

This option is not available in the IDE.

--verify_download

Syntax `--verify_download`

Applicability All C-SPY hardware debugger drivers.

Description Use this option to verify that the downloaded code image can be read back from target memory with the correct contents.

Project>Options>Debugger>*driver*>Download>Verify download

Debugger options

This chapter describes the C-SPY® options available in the IAR Embedded Workbench® IDE. More specifically, this means:

- Setting debugger options
- Reference information on debugger options
- Reference information on C-SPY hardware driver options.

Setting debugger options

Before you start the C-SPY debugger you might need to set some options—both C-SPY generic options and options required for the target system (C-SPY driver-specific options). This section gives detailed information about the options in the **Debugger** category.

To set debugger options in the IDE:

1. Choose **Project>Options** to display the **Options** dialog box.
2. Select **Debugger** in the **Category** list.

   For more information about the generic options, see *Reference information on debugger options*, page 268.
3. On the **Setup** page, select the appropriate C-SPY driver from the **Driver** drop-down list.
4. To set the driver-specific options, select the appropriate driver from the **Category** list. Depending on which C-SPY driver you are using, different options are available.

C-SPY driver	Available options pages
C-SPY emulator	*Reference information on C-SPY hardware driver options*, page 272

Table 35: Options specific to the C-SPY drivers you are using

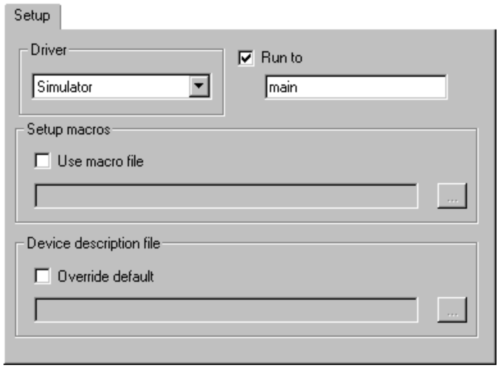
5. To restore all settings to the default factory settings, click the **Factory Settings** button.
6. When you have set all the required options, click **OK** in the **Options** dialog box..

Reference information on debugger options

This section gives reference information on C-SPY debugger options.

Setup

The **Setup** options select the C-SPY driver, the setup macro file, and device description file to use, and specify which default source code location to run to.



Simulator

[✓] Run to: main
[ ] Use macro file
[ ] Override default

Figure 107: Debugger setup options

Driver

Selects the C-SPY driver for the target system you have.

Run to

Specifies the location C-SPY runs to when the debugger starts after a reset. By default, C-SPY runs to the `main` function.

To override the default location, specify the name of a different location you want C-SPY to run to. You can specify assembler labels or whatever can be evaluated as such, for example function names.

If the option is deselected, the program counter will contain the regular hardware reset address at each reset.

Setup macros

Registers the contents of a setup macro file in the C-SPY startup sequence. Select **Use macro file** and specify the path and name of the setup file, for example `SetupSimple.mac`. If no extension is specified, the extension `mac` is assumed. A browse button is available for your convenience.

Device description file

A default device description fileis selected automatically based on your project settings. To override the default file, select **Override default** and specify an alternative file. A browse button is available for your convenience.

For information about the device description file, see *Modifying a device description file*, page 36.

Device description files for each R32C device are provided in the directory `r32c\config` and have the filename extension `ddf`.

Extra Options

The Extra Options page provides you with a command line interface to C-SPY.

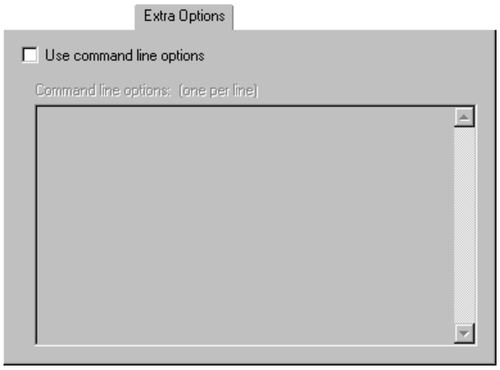


Figure 108: Debugger extra options

Use command line options

Specify additional command line arguments to be passed to C-SPY (not supported by the GUI).

Images

The Images options control the use of additional debug files to be downloaded.

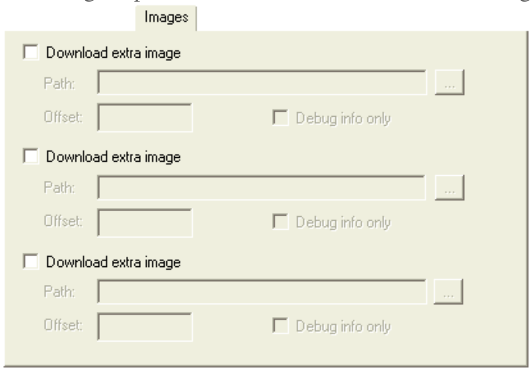


Figure 109: Debugger images options

Note: Flash loading will not be performed; you can only download images to RAM using the **Images** options.

Download extra Images

Controls the use of additional debug files to be downloaded:

Path

Specify the debug file to be downloaded. A browse button is available for your convenience.

Offset

Specify an integer that determines the destination address for the downloaded debug file.

Debug info only

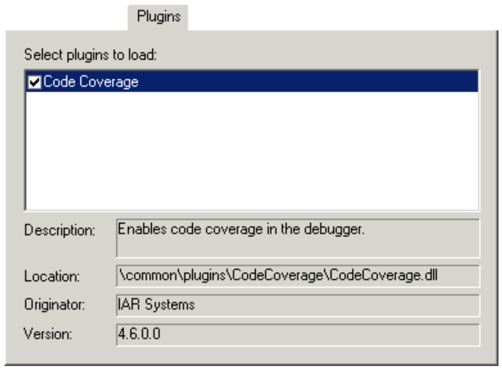
Makes the debugger download only debug information, and not the complete debug file.

If you want to download more than three images, use the related C-SPY macro, see *__loadImage*, page 234.

For more information, see *Loading multiple images*, page 35.

Plugins

The **Plugins** options select the C-SPY plugin modules to be loaded and made available during debug sessions.



Plugins

Select plugins to load:

☑ Code Coverage

Description: Enables code coverage in the debugger.

Location: \common\plugins\CodeCoverage\CodeCoverage.dll

Originator: IAR Systems

Version: 4.6.0.0

Figure 110: Debugger plugin options

Select plugins to load

Selects the plugin modules to be loaded and made available during debug sessions. The list contains the plugin modules delivered with the product installation.

Description

Describes the plugin module.

Location

Informs about the location of the plugin module.

Generic plugin modules are stored in the `common\plugins` directory. Target-specific plugin modules are stored in the `r32c\plugins` directory.

Originator

Informs about the originator of the plugin module, which can be modules provided by IAR Systems or by third-party vendors.

Version

Informs about the version number.

Reference information on C-SPY hardware driver options

This section gives reference information on C-SPY hardware driver options.

Communication

With the options on the **Communication** page you can modify the behavior of the communication with the emulator.

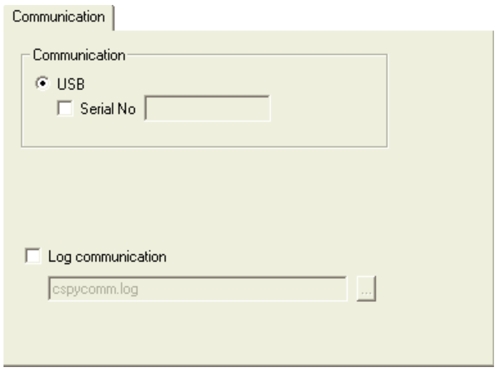


Figure 111: Emulator communication options

Note: This page is not available for the E8a emulator driver.

USB

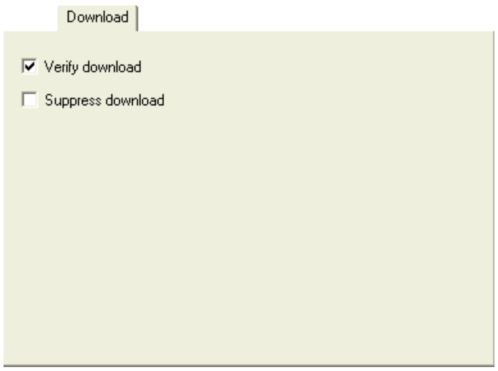
Use this option if an emulator is connected to your host computer via a USB cable. If more than one emulator is connected, choose which one to use with the **Serial No** option.

Log communication

Use this option to log the communication between C-SPY and the target system to a file. To interpret the result, a detailed knowledge of the communication protocol is required.

Download

The **Download** page contains the options related to downloading.



[✓] Verify download

[ ] Suppress download

Figure 112: Download page

Verify download

Verifies that the downloaded code image can be read back from target memory with the correct contents.

Suppress download

Disables the downloading of code, while preserving the present content of the flash. This command is useful if you want to debug an application that already resides in target memory.

If this option is combined with the **Verify download** option, the debugger will read back the code image from non-volatile memory and verify that it is identical to the debugged application.

Additional information on C-SPY drivers

This chapter describes the additional menus and features provided by the C-SPY® drivers. You will also find some useful hints about resolving problems.

Reference information on the C-SPY simulator

This section gives additional reference information the C-SPY simulator, reference information not provided elsewhere in this documentation.

More specifically, this means:

- *Simulator menu*, page 275

Simulator menu

When you use the simulator driver, the **Simulator** menu is added to the menu bar.

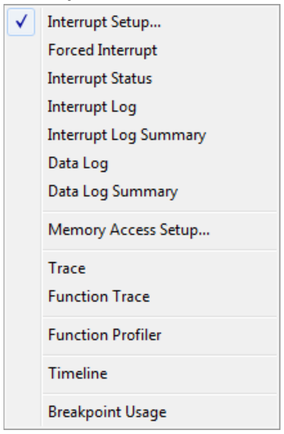


Figure 113: Simulator menu

Menu commands

These commands are available on the menu:

Interrupt Setup

Displays a dialog box where you can configure C-SPY interrupt simulation, see *Interrupt Setup dialog box*, page 203.

Forced Interrupts

Opens a window from where you can instantly trigger an interrupt, see *Forced Interrupt window, page 206*.

Interrupt Status

Opens a window from where you can instantly trigger an interrupt, see *Interrupt Status window, page 207*.

Interrupt Log

Opens a window which displays the status of all defined interrupts, see *Interrupt Log window*, page 209.

Interrupt Log Summary

Opens a window which displays a summary of the status of all defined interrupts, see *Interrupt Log Summary window*, page 212.

Data Log

Opens a window which logs accesses to up to four different memory locations or areas, see *Data Log window*, page 96.

Data Log Summary

Opens a window which displays a summary of data accesses to specific memory location or areas, see *Data Log Summary window*, page 98.

Memory Access Setup

Displays a dialog box to simulate memory access checking by specifying memory areas with different access types, see *Memory Access Setup dialog box*, page 154.

Trace

Opens a window which displays the collected trace data, see *Trace window*, page 165.

Function Trace

Opens a window which displays the trace data for function calls and function returns, see *Function Trace window*, page 168.

Function Profiler

Opens a window which shows timing information for the functions, see *Function Profiler window*, page 186.

Timeline

Opens a window which shows trace data for interrupt logs and for the call stack, see *Timeline window*, page 168.

Breakpoint Usage

Displays a window which lists all active breakpoints, see *Breakpoint Usage window*, page 114.

Reference information on the C-SPY emulator drivers

This section gives additional reference information on the C-SPY emulator drivers, reference information not provided elsewhere in this documentation.

More specifically, this means:

- *Emulator menu*, page 277
- *Time Measurement Event dialog box*, page 282
- *Interval Time Measurement Event dialog box*, page 283
- *Data Break Event dialog box*, page 123
- *Interval Time Measurement window*, page 285

Emulator menu

When you are using the C-SPY emulator drivers, the **Emulator** menu is added to the menu bar.

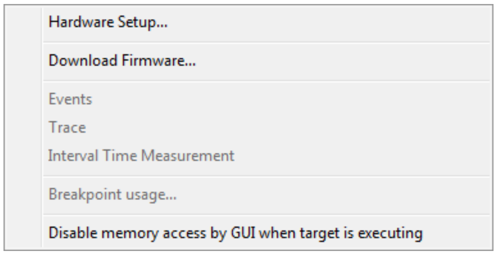

Hardware Setup...
Download Firmware...
Events
Trace
Interval Time Measurement
Breakpoint usage...
Disable memory access by GUI when target is executing

Figure 114: The Emulator menu

Menu commands

These commands are available on the menu:

Hardware Setup

Displays the **Hardware Setup** dialog box, in which the basic configuration for the emulator is done, see *Hardware Setup*, page 55.

Download Firmware

Opens the **Download Firmware** dialog box for selecting a firmware file to download to the target board, see *Downloading firmware*, page 38.

Events

Opens the Events window where events can be viewed and modified, see *Events window*, page 279.

Trace

Opens a window which displays the collected trace data, see *Trace window*, page 165.

Interval Time Measurement (E30A only)

Opens a window which shows interval time measurement results, see *Function Profiler window*, page 186.

Breakpoint Usage

Displays a window which lists all active breakpoints, see *Breakpoint Usage window*, page 114.

Disable memory access by GUI when target is executing

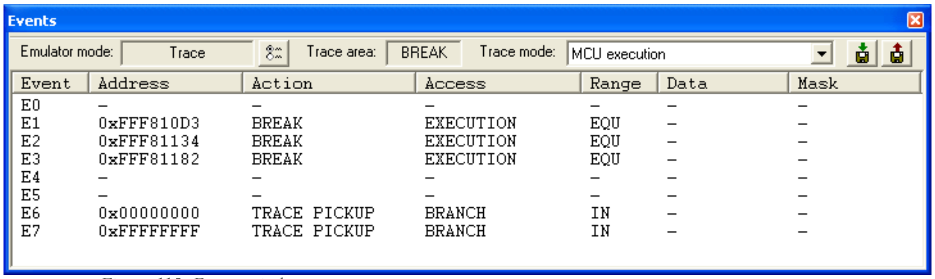
Disables all memory accesses by the GUI when target is executing.

Show Emulator Setting (E8a only)

Displays the **Emulator Setting** dialog box, see *Emulator Setting*, page 53.

Events window

The Events window is available from the **Emulator** menu.



Event	Address	Action	Access	Range	Data	Mask
E0	–	–	–	–	–	–
E1	0xFFF810D3	BREAK	EXECUTION	EQU	–	–
E2	0xFFF81134	BREAK	EXECUTION	EQU	–	–
E3	0xFFF81182	BREAK	EXECUTION	EQU	–	–
E4	–	–	–	–	–	–
E5	–	–	–	–	–	–
E6	0x00000000	TRACE PICKUP	BRANCH	IN	–	–
E7	0xFFFFFFFF	TRACE PICKUP	BRANCH	IN	–	–

Figure 115: Events window

Use this window to display and set data break events, trace events, and RAM monitor events.

A total of 8 events, comprised of break events, trace events, and RAM monitor events in combination can be specified.

Execution time measurement uses two events—one that defines a Time Start action and one that defines a Time End action—to measure the program execution time from the start to the end of execution. The result in cycles will be displayed in the cycle counter register. These events are set up automatically from the Events window context menu.

Requirements

One of these alternatives:

- A C-SPY E30 emulator
- A C-SPY E30A emulator

Toolbar

The toolbar contains:

Emulator mode

Displays the selected emulator mode. E30A only.

Edit settings

Opens the **Hardware Setup** dialog box where you can change the emulator mode. This option is only available for E30A.

Trace area

Displays the selected trace area, for more information see *Trace Event dialog box*, page 163.

Trace mode

Selects the trace mode to be used. Trace modes are only available for the C-SPY E30A emulator driver.

Choose between the following trace modes:

MCU execution	Gives priority to the MCU execution, and the trace range is 512 bytes.
MCU execution with disassembler	Gives priority to the MCU execution, and the trace range is 512 bytes. Includes disassembly.
Trace priority	Gives priority to the trace data output, and the MCU execution is delayed. *
Trace priority with disassembler	Gives priority to the trace data output, and the MCU execution is delayed. Includes disassembly. *

* For the trace priority and time measurement modes, pay attention to the following:

1) C-SPY will not automatically stop on breakpoints. When the CPU stops, you must click the **Stop** button to make C-SPY halt the execution.

2) If you do not want C-SPY to stop on main, disable the options **Project>Options>Debugger>Run to main** and **Tools>Options>Stack>Stack pointer(s) not valid until program reaches main**.

Save events to file

Saves the event log and the emulator mode setting to a file. This is available for E30A only.

Load events from file

Loads the event log and the emulator mode setting from a file. This is available for E30A only.

The display area

The Events window contains the following columns:

Event

The event number.

Address

The address of the event.

Action

The event type.

Access

The event access condition.

Range

The range condition. `EQU` means an address match event, `IN` means an address range event.

Data

Data used in data compare break.

Mask

Mask used in data compare break.

The context menu

If you right-click an emulator event in the Events window, a context menu appears with the following commands:

Edit Event

Edits the selected event.

Note: Address match breaks cannot be edited from the Events window. Use the Breakpoints window (**Edit>Breakpoints**) to edit them.

Delete Event

Deletes the selected event.

Note: Address match breaks can not be deleted in the Events window. Use the Breakpoints window (**Edit>Breakpoints**) to delete them.

New Trace Event

Opens the **Trace Event** dialog box, where you can create a new trace event. See *Trace Event dialog box*, page 163.

New Data Break Event

Opens the **Data Break Event** dialog box, where you can create a new data break event. See *Data Break Event dialog box*, page 123.

New RAM Monitor Event

Creates a new RAM monitor event. The variable values displayed in the Live Watch window can be periodically updated by the RAM monitor function. In this case, the real-time capability of execution is not lost. The RAM monitor event must be set on event E5.

New Execution Time Measurement Event

Creates two events, one at event E0 that defines the Time Start action and one at event E4 that defines the Time End action, to use for execution time measurement. This command is available for E30A only.

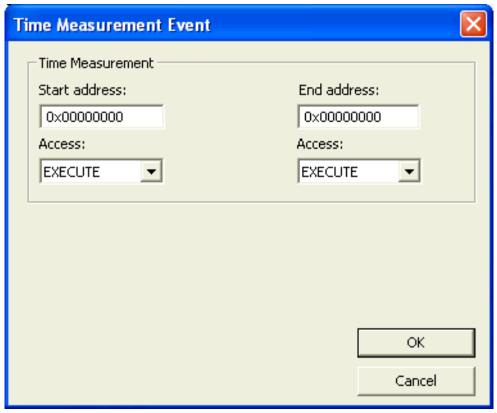
New Interval Time Measurement Event

Opens the **Interval Time Measurement** dialog box where you can specify an interval time measurement event. See *Interval Time Measurement Event dialog box*, page 283. E30A only.

Note: Some types of events cannot be edited. They must be deleted and then created as new events.

Time Measurement Event dialog box

The **Time Measurement Event** dialog box is available from the Events window context menu.


Time Measurement Event

Time Measurement

Start address: 0x00000000

Access: EXECUTE

End address: 0x00000000

Access: EXECUTE

Figure 116: Time Measurement Event dialog box

Use this window to measure the maximum, minimum, and average execution times in clock cycles and measurement counts of a specified memory section.

By specifying E0 as the start event and E4 as the end event, you can measure the execution time in a specified interval.

The result of the measurement will be displayed in the cycle counters in the Register window.

Requirements

A C-SPY E30 emulator.

Start address

The start address of the time measurement.

End address

The end address of the time measurement.

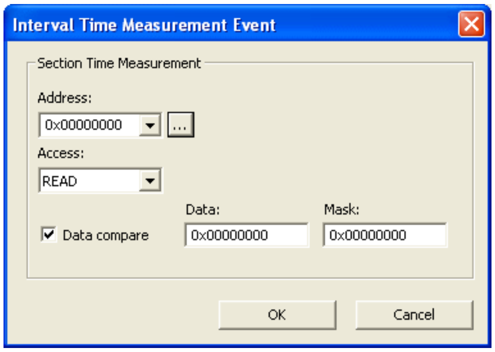
Access

EXECUTE specifies an instruction execution event. The event is established when an instruction is executed from the specified address.

READ or WRITE or R/W specifies a memory access event. The event is established when memory is accessed at the specified address or under conditions set for the specified address range.

Interval Time Measurement Event dialog box

The **Interval Time Measurement Event** dialog box is available from the Events window context menu.


Interval Time Measurement Event

Section Time Measurement

Address: 0x00000000

Access: READ

[✓] Data compare — Data: 0x00000000 — Mask: 0x00000000

Figure 117: Interval Time Measurement Event dialog box

Use this dialog box to set up events to measure the execution times in microseconds between data accesses in a specified section of memory.

In the Interval Time Measurement window, you specify which of the events to use as start and stop events for the interval; see *Interval Time Measurement window*, page 285. The result of the measurement is also displayed in the Interval Time Measurement window.

Interval time measurement measures the execution time in microseconds between two data accesses. You set up the events in this dialog box and the measurement in the Interval Time Measresment window. See *Interval Time Measurement window*, page 285.

Requirements

A C-SPY E30A emulator.

Address

The address of the memory section to measure. Use the browse button to specify a predefined symbol instead of a hardwired address.

Accress

Specifies a data memory access type. Choose between **READ**, **WRITE**, and **READ/WRITE**.

Data compare

Turns on data comparison.

Data

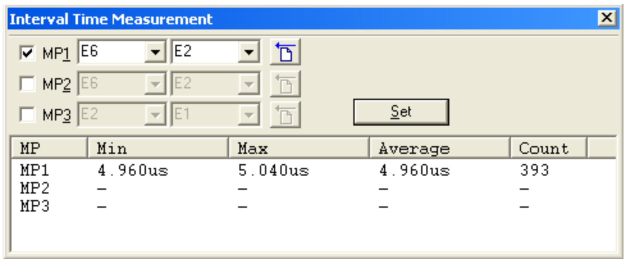
Specifies the data that triggers the event if matched.

Mask

Specifies the mask for the data somparison.

Interval Time Measurement window

The Interval Time Measurement window is available from the **Emulator** menu.



[✓] MP1 E6 E2
[ ] MP2 E6 E2
[ ] MP3 E2 E1

MP	Min	Max	Average	Count
MP1	4.960us	5.040us	4.960us	393
MP2	-	-	-	-
MP3	-	-	-	-

Figure 118: Interval Time Measurement window

This window displays the time between *data accesses* in a specified section of memory. This window is also used for setting the time measurement points.

Three check boxes allow you to specify up to three measurement points, numbered **MP1–MP3**. For every measurement point, you must specify a start event and a stop event. These events must be of the Interval Time Measurement Event type, see *Interval Time Measurement Event dialog box*, page 283. When you are done, click **Set**.

Click the **Clear** button to clear the measurement data for the measurement point in question.

Requirements

A C-SPY E30A emulator.

Display area

This area contains these columns:

MP

Identifies the measurement point.

Min

The fastest execution time for the measured interval, in microseconds.

Max

The slowest execution time for the measured interval, in microseconds.

Average

The average execution time for the measured interval, in microseconds.

Count

The number of times the time interval has been executed.

Resolving problems

Debugging using the C-SPY hardware debugger systems requires interaction between many systems, independent from each other. For this reason, setting up this debug system can be a complex task. If something goes wrong, it might at first be difficult to locate the cause of the problem.

This section includes suggestions for resolving the most common problems that can occur when debugging with the C-SPY hardware debugger systems.

For problems concerning the operation of the evaluation board, refer to the documentation supplied with it, or contact your hardware distributor.

WRITE FAILURE DURING LOAD

There are several possible reasons for write failure during load. The most common is that your application has been incorrectly linked:

- Check the contents of your linker configuration file and make sure that your application has not been linked to the wrong address
- Check that you are using the correct linker configuration file.

If you are using the IAR Embedded Workbench IDE, the linker configuration file is automatically selected based on your choice of device.

To choose a device:

1 Choose **Project>Options.**

2 Select the **General Options** category.

3 Click the **Target** tab.

4 Choose the appropriate device from the **Device** drop-down list.

To override the default linker configuration file:

5 Choose **Project>Options.**

6 Select the **Linker** category.

7 Click the **Config** tab.

8 Choose the appropriate linker configuration file in the **Linker configuration file** area.

NO CONTACT WITH THE TARGET HARDWARE

There are several possible reasons for C-SPY to fail to establish contact with the target hardware. Do this:

- Check the communication devices on your host computer
- Verify that the cable is properly plugged in and not damaged or of the wrong type
- Make sure that the evaluation board is supplied with sufficient power
- Check that the correct options for communication have been specified in the IAR Embedded Workbench IDE.

Examine the linker configuration file to make sure that the application has not been linked to the wrong address.

SLOW STEPPING SPEED

If you find that the stepping speed is slow, these troubleshooting tips might speed up stepping:

- If you are using a hardware debugger system, keep track of how many hardware breakpoints that are used and make sure some of them are left for stepping.

  Stepping in C-SPY is normally performed using breakpoints. When C-SPY performs a step command, a breakpoint is set on the next statement and the application executes until it reaches this breakpoint. If you are using a hardware debugger system, the number of hardware breakpoints—typically used for setting a stepping breakpoint in code that is located in flash/ROM memory—is limited. If you, for example, step into a C `switch` statement, breakpoints are set on each branch; this might consume several hardware breakpoints. If the number of available hardware breakpoints is exceeded, C-SPY switches into single stepping on assembly level, which can be very slow.

  For more information, see *Breakpoints in the C-SPY hardware drivers*, page 104 and *Breakpoint consumers*, page 105.
- Disable trace data collection, using the **Enable/Disable** button in both the Trace and the Function Profiling windows. Trace data collection might slow down stepping because the collected trace data is processed after each step. Note that it is not sufficient to just close the corresponding windows to disable trace data collection.
- Choose to view only a limited selection of SFR registers. You can choose between two alternatives. Either type `#`*`SFR_name`* (where *`SFR_name`* reflects the name of the SFR you want to monitor) in the Watch window, or create your own filter for displaying a limited group of SFRs in the Register window. Displaying many SFR registers might slow down stepping because all registers must be read from the hardware after each step. See *Defining application-specific register groups*, page 133.

- Close the Memory and Symbolic Memory windows if they are open, because the visible memory must be read after each step and that might slow down stepping.
- Close any window that displays expressions such as Watch, Live Watch, Locals, Statics if it is open, because all these windows read memory after each step and that might slow down stepping.
- Close the Stack window if it is open. Choose **Tools>Options>Stack** and disable the **Enable graphical stack display and stack usage tracking** option if it is enabled.
- If possible, increase the communication speed between C-SPY and the target board/emulator.

A

Abort (Report Assert option) 77
absolute location, specifying for a breakpoint. 127
Access Type (Data breakpoints option). 121, 125–126
Access type (Edit Memory Access option) 157
Access (Data Break Event option) 124
Access (Edit SFR option) 154
Access (Interval Time Measurement Event option). 284
Access (Time Measurement Event option) 283
Action (Code breakpoints option). 116–118, 122, 126
Add to Watch Window (Symbolic Memory window context menu) 144
Add (SFR Setup window context menu). 152
Add (Watch window context menu) 88
address breakpoints. 116
Address Range (Find in Trace option) 180
Address (Edit SFR option) 154
Address (Interval Time Measurement Event option). 284
Address 1 (Data Break Event option) 123
Address 2 (Data Break Event option) 123
Ambiguous symbol (Resolve Symbol Ambiguity option). . 95
application, built outside the IDE 34
assembler labels, viewing 84
assembler source code, fine-tuning 183
assembler symbols, using in C-SPY expressions 81
assembler variables, viewing. 84
assumptions, programming experience 17
Auto Scroll (Timeline window context menu) 173
Auto window 86
Autostep settings dialog box 77
Autostep (Debug menu) 48

B

-B (C-SPY command line option). 260
--backend (C-SPY command line option). 261
backtrace information
  generated by compiler 64
  viewing in Call Stack window 71
batch mode, using C-SPY in. 257
Big Endian (Memory window context menu) 138
blocks, in C-SPY macros 226
bold style, in this guide 20
Break At (breakpoints option). 115, 117–118, 121
Break on Throw (Debug menu). 48
Break on Uncaught Exception (Debug menu). 48
Break (Debug menu). 47
breakpoint condition, example 110
Breakpoint Usage window 114
Breakpoint Usage (Emulator menu) 278
Breakpoint Usage (Simulator menu). 277
breakpoints
  address 116
  code, example 243
  connecting a C-SPY macro 222
  consumers of 105
  data 121
  data log 124
  description of. 102
  disabling used by Stack window 106
  icons for in the IDE 104
  in Memory window 108
  listing all 114
  reasons for using 101
  setting
    in memory window 108
    using system macros 109
    using the dialog box 107
  single-stepping if not available. 32
  toggling 107
  types of 102
  useful tips. 110
Breakpoints dialog box
  Code 115
  Data 121
  Data Log 124
  Execution Address. 116

Immediate 125
Log 119
Software. 118
Trace Start 176
Trace Stop 177
Breakpoints window 112
Browse (Trace toolbar) 166
byte order, setting in Memory window 137

C

C function information, in C-SPY. 64
C symbols, using in C-SPY expressions 80
C variables, using in C-SPY expressions 80
call chain, displaying in C-SPY 64
Call stack information. 64
Call Stack window 71
for backtrace information. 65
Call Stack (Timeline window context menu) 174
__cancelAllInterrupts (C-SPY system macro) 231
__cancelInterrupt (C-SPY system macro). 231
CCR (E30A Emulator) 56
Clear All (Debug Log window context menu) 75
Clear trace data (Trace toolbar). 165
Clear (Interrupt Log window context menu). 211
__clearBreak (C-SPY system macro) 231
clock control register (E30A Emulator) 56
Clock (E30A) (Hardware Setup option) 56
__closeFile (C-SPY system macro) 232
code breakpoints
overview 102
toggling 107
Code Coverage window 192
Code Coverage (Disassembly window context menu) 69
--code_coverage_file (C-SPY command line option) 261
code, covering execution of 192
command line options. 260
typographic convention 20
command prompt icon, in this guide. 20
Communication Baud Rate (Emulator Setting option) 55
computer style, typographic convention 20
conditional statements, in C-SPY macros 225
Conditions (Code breakpoints option) . . 116–117, 119–120, 122
Connecting dialog box, E8/E8a. 41
context menu, in windows. 83
conventions, used in this guide 20
Copy Window Contents (Disassembly window context menu) 70
Copy (Debug Log window context menu) 75
copyright notice 2
cspybat 257
current position, in C-SPY Disassembly window 68
cursor, in C-SPY Disassembly window. 68
--cycles (C-SPY command line option) 261
C-SPY
batch mode, using in 257
debugger systems, overview of 26
differences between drivers 28
environment overview 23
plugin modules, loading. 33
setting up 31–32
starting the debugger 33
C-SPY drivers
overview 28
specifying 268
C-SPY expressions 80
evaluating. 92
in C-SPY macros 225
Tooltip watch, using. 79
Watch window, using 79
C-SPY macro "__message" style (Log breakpoints option) 120
C-SPY macros
blocks. 226
conditional statements 225
C-SPY expressions 225
dialog box, using 219
examples 217

checking status of register 221
checking the status of WDT 221
creating a log macro 222
executing 217
connecting to a breakpoint 222
using Quick Watch 221
using setup macro and setup file 220
functions 81, 223
loop statements 225
macro statements 225
setup macro file 216
executing. 220
setup macro functions 216
summary 228
system macros, summary of. 229
using 215
variables. 81, 224
C-SPY options 267
Extra Options. 269
Images 270
Plugins. 271
Setup 268
C-SPYLink 28
C++ terminology. 20

D

-d (C-SPY command line option) 262
Data acquisition interval of program execution (E30A) (Hardware Setup option) 58
Data Break Event (E30A emulator dialog box). 123
data breakpoints, overview 103
Data compare (Data Break Event option) 124
Data compare (Interval Time Measurement Event option) 284
Data Coverage (Memory window context menu) 138
data coverage, in Memory window 136
data log breakpoints, overview 103
Data Log Summary window 98
Data Log window 96
Data Log (Timeline window context menu) 174
Data (Interval Time Measurement Event option) 284
ddf (filename extension), selecting a file. 33
Debug Log window. 74
Debug Log window context menu 75
Debug menu (C-SPY main window). 47
Debug the program using the CPU rewrite mode (E30A) (Hardware Setup option). 56
Debug (Report Assert option) 77
debugger concepts, definitions of 26
debugger system overview 26
Debugging of CPU rewrite mode (E8/E8a emulator options). 54
Debugging of CPU rewrite mode (Emulator Setting option) 54
debugging projects
externally built applications 34
loading multiple images. 35
debugging, RTOS awareness. 25
__delay (C-SPY system macro) 232
Delay (Autostep Settings option) 77
Delete (Breakpoints window context menu) 113
Delete (SFR Setup window context menu) 152
Delete/revert All Custom SFRs (SFR Setup window context menu) 153
Description (Edit Interrupt option) 205
description (interrupt property). 205
Device description file (debugger option) 269
device description files 33
definition of 36
memory zones 131
modifying 36
register zone. 131
specifying interrupts 238
Device (Emulator Setting option) 53
Disable All (Breakpoints window context menu) 113
Disable memory access by GUI when target is executing (Emulator menu). 278
Disable Reset of Target (E30A) (Hardware Setup option) . 58
Disable (Breakpoints window context menu) 113

__disableInterrupts (C-SPY system macro) 232
--disable_interrupts (C-SPY command line option) 262
Disassembly window 66
  context menu 68
disclaimer 2
DLIB
  documentation 19
Do not communicate with MCU while target is executing (E30A) (Hardware Setup option) 58
Do not show this dialog box again (E8/E8a emulator options). 54
Do not show this dialog box again (Emulator Setting option) 54
do (macro statement) 226
document conventions. 20
documentation
  overview of guides. 18
  overview of this guide 17
Download Firmware (Emulator menu) 278
download options 273
--download_only (C-SPY command line option) 262
Driver (debugger option). 268
__driverType (C-SPY system macro) 233
--drv_communication (C-SPY command line option). . . . 263

E

Edit Expressions (Trace toolbar). 166
Edit Interrupt dialog box 204
Edit Memory Access dialog box 156
Edit Memory Range dialog box 153
Edit Settings (Trace toolbar) 166
Edit (Breakpoints window context menu). 113
Edit (SFR Setup window context menu). 152
edition, of this guide 2
Embedded C++ Technical Committee 20
Emulator mode (E30A) (Hardware Setup option). 57
Emulator Setting dialog box 40
Emulator Setting dialog box (E8a) 53
Emulator, menu. 277
Enable All (Breakpoints window context menu). 113
Enable interrupt simulation (Interrupt Setup option). 203
Enable Log File (Log File option). 76
Enable (Breakpoints window context menu). 113
Enable (Interrupt Log window context menu). 211
Enable (Timeline window context menu) 174
__enableInterrupts (C-SPY system macro). 233
Enable/Disable Breakpoint (Call Stack window context menu) 72
Enable/Disable Breakpoint (Disassembly window context menu) 70
Enable/Disable (Trace toolbar) 165
End address (Edit Memory Access option). 157
End address (Time Measurement Event option) 283
endianness. *See* byte order
Enter Location dialog box. 126
Erase Flash and Connect (Emulator Setting option) 53
Erase Flash and Connect (E8/E8a emulator options) 53
__evaluate (C-SPY system macro) 234
Events (Emulator menu) 278
Events (emulator window) 279
examples
  C-SPY macros 217
  interrupts
    interrupt logging 202
    timer 200
  macros
    checking status of register. 221
    checking status of WDT 221
    creating a log macro 222
    using Quick Watch 221
  performing tasks and continue execution 110
  tracing incorrect function arguments 110
execUserExit (C-SPY setup macro) 228
execUserFlashInit (C-SPY setup macro). 228
execUserPreload (C-SPY setup macro). 228
execUserPreReset (C-SPY setup macro). 228
execUserReset (C-SPY setup macro) 228
execUserSetup (C-SPY setup macro) 228

Execute the user program after ending the debugger (Emulator Setting option) 54
Execute the user program after ending the debugger (E8/E8a emulator options) 54
executed code, covering 192
execution history, tracing 162
execution time measurement. 279
creating events 282
expressions. *See* C-SPY expressions
Extra Options, for C-SPY 269
E2 Data Flash is not erased on download (E30A) (Hardware Setup option) 56
E30 Emulator, hardware setup 55
E30A Emulator, hardware setup 55
E30/E30A Emulator, setting up 272
E30/E30A options
Log communication 272
USB 272

F

Factory Settings (E30A) (Hardware Setup option) 58
File format (Memory Save option) 139
file types
device description, specifying in IDE 33
macro 33, 268
filename extensions
ddf, selecting device description file 33
mac, using macro file 33
Filename (Memory Restore option) 140
Filename (Memory Save option). 139
Fill dialog box. 140
Find in Trace dialog box 179
Find in Trace window 181
Find (Memory window context menu) 138
Find (Trace toolbar) 166
Firmware Location & WDT dialog box 40
first activation time (interrupt property)
definition of 197
First activation (Edit Interrupt option). 205
flash memory, load library module to 235
for (macro statement) 225
Forced Interrupt window. 206
Forced Interrupts (Simulator menu) 276
formats, C-SPY input 25
Function Profiler window 186
Function Profiler (Driver1 menu) 278
Function Profiler (Simulator menu) 277
Function Trace window. 168
Function Trace (Simulator menu) 276
functions
call stack information for. 64
C-SPY running to when starting 32, 268
most time spent in, locating 183

G

Go to Source (Breakpoints window context menu). 113
Go to Source (Call Stack window context menu) 72
Go To Source (Timeline window context menu). 174
Go (Debug menu) 47, 63

H

Hardware installation, E8a driver 30
Hardware Setup dialog box (Emulator menu). 55
Hardware Setup (Emulator menu). 278
highlighting, in C-SPY 64
Hold time (Edit Interrupt option) 205
hold time (interrupt property), definition of 197

I

icons, in this guide 20
ID Code verification dialog box 42
ID Code Verification dialog box (E30/E30A Emulator) . . . 45
if else (macro statement). 225
if (macro statement) 225
Ignore (Report Assert option) 77

illegal memory accesses, checking for 132
Images window. 50
Images, loading multiple. 270
immediate breakpoints, overview 103
Include (Log File option) 76
input formats, C-SPY 25
Input Mode dialog box 73
input, special characters in Terminal I/O window 73
installation directory 20
Instruction Profiling (Disassembly window context menu). 69
Intel-extended, C-SPY input format 25
Intel-extended, C-SPY output format 27
Interrupt Log Summary window 212
Interrupt Log Summary (Simulator menu) 276
Interrupt Log window 209
Interrupt Log (Simulator menu) 276
Interrupt Setup dialog box 203
Interrupt Setup (Simulator menu) 276
Interrupt Status window 207
interrupt system, using device description file 199
Interrupt (Edit Interrupt option) 205
Interrupt (Timeline window context menu). 174
interrupts
  adapting C-SPY system for target hardware 199
  simulated, introduction to 195
  timer, example 200
  using system macros 199
interval time measurement 284
Interval Time Measurement Event (E30A emulator dialog box). 283
Interval Time Measurement, emulator window. 285
__isBatchMode (C-SPY system macro) 234
italic style, in this guide 20
I/O register. *See* SFR

K

Keep Flash and Connect (Emulator Setting option) 54

L

labels (assembler), viewing. 84
Length (Fill option). 141
lightbulb icon, in this guide. 20
Little Endian (Memory window context menu) 137
Live Watch window 89
__loadImage (C-SPY system macro) 234
loading multiple debug files, list currently loaded. 50
loading multiple images 35
Locals window 86
log breakpoints, overview 103
Log communication (E30/E30A option). 272
Log File dialog box. 75
Logging>Set Log file (Debug menu) 49
Logging>Set Terminal I/O Log file (Debug menu). 49
loop statements, in C-SPY macros 225

M

mac (filename extension), using a macro file 33
--macro (C-SPY command line option) 263
Macro Configuration dialog box. 219
macro files, specifying 33, 268
macro statements 225
macros
  executing 217
  using 215
Macros (Debug menu) 49
main function, C-SPY running to when starting 32, 268
--mapu (C-SPY command line option) 263
Mask (Interval Time Measurement Event option). 284
MCU Group (Emulator Setting option). 53
mcu (filename extension) 57
memory access checking. 132
Memory access checking (Memory Access Setup option) 155
Memory Access Setup dialog box. 154
Memory Access Setup (Simulator menu) 276
memory accesses, illegal. 132

Memory Fill (Memory window context menu) 138
memory map 154
Memory map (E30A) (Hardware Setup option) 57
Memory range (Edit Memory Access option) 157
Memory Restore dialog box 140
Memory Restore (Memory window context menu) 138
Memory Save dialog box 139
Memory Save (Memory window context menu) 138
Memory window 135
memory zones 130
  in device description file 131
__memoryRestore (C-SPY system macro) 236
__memorySave (C-SPY system macro) 236
Memory>Restore (Debug menu) 48
Memory>Save (Debug menu) 48
menu bar, C-SPY-specific 46
Message (Log breakpoints option) 120
MISRA C, documentation 19
Mixed Mode (Disassembly window context menu) 70
Monitor start address (E30A) (Hardware Setup option) . . . 57
Motorola, C-SPY input format 25
Motorola, C-SPY output format 27
Move to PC (Disassembly window context menu) 68

N

Name (Edit SFR option) 153
naming conventions 21
Navigate (Timeline window context menu) 173
New Breakpoint (Breakpoints window context menu) . . . 113
Next Statement (Debug menu) 48
Next Symbol (Symbolic Memory window context menu) 143

O

__openFile (C-SPY system macro) 237
Operation (Fill option) 141
operators, sizeof in C-SPY 82
optimizations, effects on variables 82
options
  in the IDE 267
  on the command line 260, 269
Options (Stack window context menu) 147
__orderInterrupt (C-SPY system macro) 238
Originator (debugger option) 271
Overwrite data in FLASH without erasing the FLASH area block (E30A) (Hardware Setup option) 56

P

-p (C-SPY command line option) 264
parameters
  tracing incorrect values of 64
  typographic convention 20
part number, of this guide 2
peripheral units
  device-specific 36
peripherals register. *See* SFR
phase-locked loop (E30A Emulator) 56
Please select one symbol (Resolve Symbol Ambiguity option) 95
PLL (E30A Emulator) 56
--plugin (C-SPY command line option) 264
plugin modules (C-SPY) 28
  loading 33
Plugins (C-SPY options) 271
__popSimulatorInterruptExecutingStack (C-SPY system macro) 239
pop-up menu. *See* context menu
Power Supply dialog box, E8/E8a 42
Power target from Emulator (Emulator Setting option) 54
Power target from Emulator (E8/E8a emulator options) . . . 54
prerequisites, programming experience 17
Previous Symbol (Symbolic Memory window context menu) 143
probability (interrupt property) 205
  definition of 197
Probability % (Edit Interrupt option) 205
Profile Selection (Timeline window context menu) 175

profiling
  on function level 185
  on instruction level. 185
profiling information, on functions and instructions 184
profiling sources
  trace (calls) 184, 187
  trace (flat) 184, 188
program execution, in C-SPY 59
Program Flash (Emulator Setting option) 54
Program Flash (E8/E8a emulator options) 54
programming experience. 17
program, *see also* application
projects, for debugging externally built applications. 34
publication date, of this guide 2

Q

Quick Watch window 92
  executing C-SPY macros 221

R

RAM monitor events, creating 281
Range for (Viewing Range option) 175
Range (Data Break Event option) 123
__readFile (C-SPY system macro) 240
__readFileByte (C-SPY system macro) 240
__readMemoryByte (C-SPY system macro). 241
__readMemory8 (C-SPY system macro) 241
__readMemory16 (C-SPY system macro) 241
__readMemory32 (C-SPY system macro) 241
read-protected E30/E30A Emulator 45
reference information, typographic convention. 20
Refresh (Debug menu) 48
register groups 130
  predefined, enabling. 148
Register window 148
registered trademarks 2
__registerMacroFile (C-SPY system macro). 242
registers, displayed in Register window 148
Remove (Watch window context menu) 88
Repeat interval (Edit Interrupt option) 205
repeat interval (interrupt property), definition of. 197
Replace (Memory window context menu) 138
Report Assert dialog box 76
Reset (Debug menu) 47
__resetFile (C-SPY system macro). 242
Resolve Source Ambiguity dialog box 128
Restore (Memory Restore option). 140
return (macro statement). 226
ROM-monitor, definition of 27
RTOS awareness debugging 25
RTOS awareness (C-SPY plugin module). 25
Run to Cursor (Call Stack window context menu) 72
Run to Cursor (Debug menu) 48
Run to Cursor (Disassembly window context menu) 69
Run to Cursor, command for executing. 64
Run to (C-SPY option) 32
Run to (debugger option) 268

S

Save Custom SFRs (SFR Setup window context menu) . . 153
Save to log file (Interrupt Log window context menu) . . . 211
Save (Memory Save option) 139
Save (Trace toolbar) 166
Scale (Viewing Range option). 176
Select All (Debug Log window context menu) 75
Select Graphs (Timeline window context menu). 175
Select plugins to load (debugger option). 271
Set Data Breakpoint (Memory window context menu) . . . 138
Set Next Statement (Debug menu) 48
Set Next Statement (Disassembly window context menu) . 70
__setCodeBreak (C-SPY system macro). 243
__setDataBreak (C-SPY system macro) 244
__setDataLogBreak (C-SPY system macro) 245
__setLogBreak (C-SPY system macro) 245
__setSimBreak (C-SPY system macro) 247

__setTraceStartBreak (C-SPY system macro). 248
__setTraceStopBreak (C-SPY system macro). 249
setup macro functions 216
  reserved names. 228
Setup macros (debugger option) 268
Setup (C-SPY options) 268
setupsimple.mac 216
SFR
  in Register window 149
  using as assembler symbols 81
SFR Setup window 150
shortcut menu. *See* context menu
Show all images (Images window context menu) 51
Show All (SFR Setup window context menu). 152
Show Arguments (Call Stack window context menu) 72
Show As (Watch window context menu) 89
Show Custom SFRs only (SFR Setup
window context menu) 152
Show Cycles (Interrupt Log window context menu) 211
Show Factory SFRs only (SFR Setup
window context menu) 152
Show Numerical Value (Timeline window context menu) 174
Show offsets (Stack window context menu) 146
Show only (Image window context menu) 51
Show Time (Interrupt Log window context menu) 211
Show variables (Stack window context menu) 146
--silent (C-SPY command line option) 265
simulating interrupts, enabling/disabling 203
Simulator menu. 275
simulator, introduction 29
Size (breakpoints option) 115, 122
Size (Edit SFR option) 154
Size (Timeline window context menu) 174
sizeof 82
software breakpoints
  setting 118
  use of 105
Solid Graph (Timeline window context menu) 174
__sourcePosition (C-SPY system macro) 250
special function registers (SFR)
  in Register window 149
  using as assembler symbols 81
stack usage, computing 132
Stack window 144
stack.mac 216
standard C, sizeof operator in C-SPY 82
Start address (Edit Memory Access option) 157
Start address (Fill option) 141
Start address (Memory Save option). 139
Start address (Time Measurement Event option). 283
Statics window 90
stdin and stdout, redirecting to C-SPY window 72
Step Into (Debug menu) 47
Step Into, description 61
Step Out (Debug menu) 48
Step Out, description. 62
Step Over (Debug menu) 47
Step Over, description. 61
step points, definition of 60
Stop address (Memory Save option) 139
Stop Debugging (Debug menu). 47
__strFind (C-SPY system macro) 251
__subString (C-SPY system macro) 251
Suppress download (debugger option) 273
--suppress_download (C-SPY command line option) 265
Symbolic Memory window. 142
Symbols window 93
symbols, using in C-SPY expressions. 80

T

target system, definition of 26
__targetDebuggerVersion (C-SPY system macro) 252
Terminal IO Log Files (Terminal IO Log Files option). . . . 74
Terminal I/O Log Files dialog box 74
Terminal I/O window 65, 72
terminology. 20
Text search (Find in Trace option) 180

Time Axis Unit (Timeline window context menu) 175
Time Measurement Event (E30 emulator dialog box). . . . 282
Timeline window 168
Timeline (Simulator menu). 277
--timeout (C-SPY command line option) 265
timer interrupt, example 200
Toggle Breakpoint (Code) (Call
Stack window context menu) 72
Toggle Breakpoint (Code) (Disassembly
window context menu) 69
Toggle Breakpoint (Log) (Call
Stack window context menu) 72
Toggle Breakpoint (Log) (Disassembly
window context menu) 69
Toggle Breakpoint (Trace Start) (Disassembly
window context menu) 70
Toggle Breakpoint (Trace Stop) (Disassembly
window context menu) 70
Toggle source (Trace toolbar) 166
__toLower (C-SPY system macro) 252
tools icon, in this guide 20
__toString (C-SPY system macro) 253
__toUpper (C-SPY system macro) 253
trace area (E30A emulator), setting up 164
Trace Event (E30A emulator dialog box) 163
Trace Expressions window 178
trace point (E30A emulator), setting up 164
trace start and stop breakpoints, overview. 103
Trace Start breakpoints dialog box 176
Trace Stop breakpoints dialog box 177
Trace window 165
trace (calls), profiling source. 184, 187
Trace (Emulator menu). 278
trace (flat), profiling source. 184, 188
Trace (Simulator menu) 276
trace, in Timeline window. 168
trademarks 2
Trigger at (breakpoints option) 120, 124–125, 177–178
Trigger (Forced Interrupt window context menu) 206
typographic conventions 20

U

UBROF. 25
Unavailable, C-SPY message 83
Universal Binary Relocatable Object Format. *See* UBROF
__unloadImage(C-SPY system macro). 254
USB (E30/E30A option). 272
Use command line options (debugger option). 269
Use ECC for E2 Data Flash (Emulator Setting option) 54
Use ECC for E2 Data Flash (E30A)
(Hardware Setup option). 56
Use Extra Images (debugger option). 270
Use manual ranges (Memory Access Setup option) 155
Use ranges based on (Memory Access Setup option) 155
user application, definition of 27

V

Value (Fill option). 141
variables
- effects of optimizations 82
- information, limitation on 82
- using in C-SPY expressions. 80

variance (interrupt property), definition of 197
Variance % (Edit Interrupt option) 205
Verify download (debugger option). 273
--verify_download (C-SPY command line option) 266
version number
- of this guide. 2

Viewing Range dialog box 175
Viewing Range (Timeline window context menu) 174
visualSTATE, C-SPY plugin module for. 28

W

warnings icon, in this guide 21
Watch window 87
- using 79

web sites, recommended. 19

while (macro statement) 225
windows, specific to C-SPY 49
With I/O emulation modules (linker option), using 72
__writeFile (C-SPY system macro) 254
__writeFileByte (C-SPY system macro) 255
__writeMemoryByte (C-SPY system macro) 255
__writeMemory8 (C-SPY system macro) 255
__writeMemory16 (C-SPY system macro) 255
__writeMemory32 (C-SPY system macro) 256

X

Xin clock (E30A Emulator) 56

Z

zone
  defined in device description file 131
  in C-SPY 130
Zone (Edit SFR option) 154
Zone (Memory Save option) 139–141
Zoom (Timeline window context menu) 173

Symbols

__cancelAllInterrupts (C-SPY system macro) 231
__cancelInterrupt (C-SPY system macro) 231
__clearBreak (C-SPY system macro) 231
__closeFile (C-SPY system macro) 232
__delay (C-SPY system macro) 232
__disableInterrupts (C-SPY system macro) 232
__driverType (C-SPY system macro) 233
__enableInterrupts (C-SPY system macro) 233
__evaluate (C-SPY system macro) 234
__fmessage (C-SPY macro statement) 226
__isBatchMode (C-SPY system macro) 234
__loadImage (C-SPY system macro) 234
__memoryRestore (C-SPY system macro) 236
__memorySave (C-SPY system macro) 236
__message (C-SPY macro statement) 226
__openFile (C-SPY system macro) 237
__orderInterrupt (C-SPY system macro) 238
__popSimulatorInterruptExecutingStack (C-SPY system macro) 239
__readFile (C-SPY system macro) 240
__readFileByte (C-SPY system macro) 240
__readMemoryByte (C-SPY system macro) 241
__readMemory8 (C-SPY system macro) 241
__readMemory16 (C-SPY system macro) 241
__readMemory32 (C-SPY system macro) 241
__registerMacroFile (C-SPY system macro) 242
__resetFile (C-SPY system macro) 242
__setCodeBreak (C-SPY system macro) 243
__setDataBreak (C-SPY system macro) 244
__setDataLogBreak (C-SPY system macro) 245
__setLogBreak (C-SPY system macro) 245
__setSimBreak (C-SPY system macro) 247
__setTraceStartBreak (C-SPY system macro) 248
__setTraceStopBreak (C-SPY system macro) 249
__smessage (C-SPY macro statement) 226
__sourcePosition (C-SPY system macro) 250
__strFind (C-SPY system macro) 251
__subString (C-SPY system macro) 251
__targetDebuggerVersion (C-SPY system macro) 252
__toLower (C-SPY system macro) 252
__toString (C-SPY system macro) 253
__toUpper (C-SPY system macro) 253
__unloadImage (C-SPY system macro) 254
__writeFile (C-SPY system macro) 254
__writeFileByte (C-SPY system macro) 255
__writeMemoryByte (C-SPY system macro) 255
__writeMemory8 (C-SPY system macro) 255
__writeMemory16 (C-SPY system macro) 255
__writeMemory32 (C-SPY system macro) 256
-B (C-SPY command line option) 260
-d (C-SPY command line option) 262
-p (C-SPY command line option) 264
--backend (C-SPY command line option) 261
--code_coverage_file (C-SPY command line option) 261

--cycles (C-SPY command line option) 261
--disable_interrupts (C-SPY command line option) 262
--download_only (C-SPY command line option) 262
--drv_communication (C-SPY command line option). . . . 263
--macro (C-SPY command line option) 263
--mapu (C-SPY command line option) 263
--plugin (C-SPY command line option) 264
--silent (C-SPY command line option) 265
--suppress_download (C-SPY command line option) 265
--timeout (C-SPY command line option) 265
--verify_download (C-SPY command line option) 266

Numerics

1x Units (Memory window context menu) 137, 146
1x Units (Symbolic Memory window context menu) 144
2x Units (Memory window context menu) 137, 144, 146
4x Units (Memory window context menu) 137, 144, 146
8x Units (Memory window context menu) 137