IAR Embedded Workbench®

C-SPY® Debugging Guide

for Texas Instruments’
MSP430 Microcontroller Family
COPYRIGHT NOTICE
Copyright © 1996–2010 IAR Systems AB.

No part of this document may be reproduced without the prior written consent of IAR Systems AB. The software described in this document is furnished under a license and may only be used or copied in accordance with the terms of such a license.

DISCLAIMER
The information in this document is subject to change without notice and does not represent a commitment on any part of IAR Systems. While the information contained herein is assumed to be accurate, IAR Systems assumes no responsibility for any errors or omissions.

In no event shall IAR Systems, its employees, its contractors, or the authors of this document be liable for special, direct, indirect, or consequential damage, losses, costs, charges, claims, demands, claim for lost profits, fees, or expenses of any nature or kind.

TRADEMARKS
IAR Systems, IAR Embedded Workbench, C-SPY, visualSTATE, From Idea To Target, IAR KickStart Kit, IAR PowerPac, IAR YellowSuite, IAR Advanced Development Kit, IAR, and the IAR Systems logotype are trademarks or registered trademarks owned by IAR Systems AB. J-Link is a trademark licensed to IAR Systems AB.

Microsoft and Windows are registered trademarks of Microsoft Corporation.

Texas Instruments is a registered trademark of Texas Instruments Incorporated.

Adobe and Acrobat Reader are registered trademarks of Adobe Systems Incorporated.

All other product names are trademarks or registered trademarks of their respective owners.

EDITION NOTICE
First edition: November 2010
Part number: UCS430-1

This guide applies to version 5.20 and later of IAR Embedded Workbench® for Texas Instruments' MSP430 microcontroller family.


Internal reference: Too6.1, IJOA.
Brief contents

Tables ...................................................................................................................... 13
Figures .................................................................................................................... 15
Preface .................................................................................................................... 19
The IAR C-SPY Debugger .................................................................................... 25
Getting started using C-SPY .............................................................................. 35
Executing your application .................................................................................... 51
Working with variables and expressions ............................................................ 71
Using breakpoints ................................................................................................. 87
Monitoring memory and registers ....................................................................... 119
Collecting and using trace data ........................................................................... 143
Using the profiler ................................................................................................. 159
Code coverage ...................................................................................................... 167
Using state storage ............................................................................................... 171
Using the sequencer ............................................................................................ 177
Using the cycle counter for 5xx devices ............................................................. 183
Simulating interrupts ......................................................................................... 191
Using C-SPY macros .......................................................................................... 205
The C-SPY Command Line Utility—cspybat ..................................................... 251
Debugger options ................................................................................................. 269
Additional information on C-SPY drivers ......................................................... 281
Index ..................................................................................................................... 289
Contents

Tables ...................................................................................................................... 13
Figures .................................................................................................................... 15
Preface .................................................................................................................... 19

Who should read this guide ............................................................... 19
How to use this guide ........................................................................... 19
What this guide contains ..................................................................... 20

Other documentation ........................................................................... 21
User and reference guides ................................................................. 21
The online help system ................................................................. 22
Web sites ......................................................................................... 22

Document conventions ........................................................................ 22
Typographic conventions ................................................................... 23
Naming conventions ........................................................................... 23

The IAR C-SPY Debugger ....................................................................... 25

Introduction to C-SPY ................................................................. 25
An integrated environment ................................................................. 25
General C-SPY debugger features ...................................................... 26
RTOS awareness ............................................................................... 27

Debugger concepts ............................................................................... 28
C-SPY and target systems ................................................................. 28
The debugger ...................................................................................... 29
The target system ............................................................................... 29
The application ..................................................................................... 29
C-SPY debugger systems ................................................................. 29
The ROM-monitor program ............................................................. 29
Third-party debuggers ........................................................................ 30
C-SPY plugin modules ....................................................................... 30

C-SPY drivers overview ...................................................................... 30
Differences between the C-SPY drivers .................................................. 30
Terminal input and output ......................................................... 56
Debug logging ........................................................................ 56

Reference information on application execution ...................... 57

Working with variables and expressions .................................. 71

Introduction to working with variables and expressions ......... 71
Briefly about working with variables and expressions ............ 71
C-SPY expressions .................................................................. 72
Limitations on variable information ...................................... 74
Viewing assembler variables ................................................. 75

Reference information on working with variables and
expressions ................................................................................ 76

Using breakpoints .................................................................... 87

Introduction to setting and using breakpoints ....................... 87
Reasons for using breakpoints ............................................. 87
Briefly about setting breakpoints ....................................... 88
Breakpoint types ................................................................. 88
Breakpoint icons ................................................................... 90
Breakpoints in the C-SPY simulator ..................................... 90
Breakpoints in the C-SPY FET Debugger driver ................. 90
Breakpoint consumers ......................................................... 91

Procedures for setting breakpoints ...................................... 93
Various ways to set a breakpoint .................................... 93
Toggling a simple code breakpoint .................................... 94
Setting breakpoints using the dialog box ......................... 94
Setting a data breakpoint in the Memory window ............ 95
Setting breakpoints using system macros ....................... 96
Useful breakpoint hints ...................................................... 97

Reference information on breakpoints ................................ 98

Monitoring memory and registers ...................................... 119

Introduction to monitoring memory and registers ............... 119
Briefly about monitoring memory and registers ............... 119
C-SPY memory zones ......................................................... 120
Stack display ................................................................. 121
Memory access checking ...................................................... 122
Reference information on memory and registers .............. 123
Collecting and using trace data............................................. 143
Introduction to using trace ................................................. 143
Reasons for using trace ...................................................... 143
Briefly about trace .............................................................. 143
Requirements for using trace .............................................. 144
Procedures for using trace .................................................... 144
Getting started with trace .................................................. 144
Trace data collection using breakpoints .............................. 144
Searching in trace data ...................................................... 145
Browsing through trace data ............................................... 145
Reference information on trace ......................................... 146
Using the profiler ................................................................. 159
Introduction to the profiler ................................................. 159
Reasons for using the profiler .............................................. 159
Briefly about the profiler ..................................................... 159
Requirements for using the profiler ................................. 160
Procedures for using the profiler ........................................ 160
Getting started using the profiler on function level .......... 161
Getting started using the profiler on instruction level ....... 161
Reference information on the profiler ............................... 162
Code coverage ................................................................. 167
Introduction to code coverage ............................................ 167
Reasons for using code coverage ....................................... 167
Briefly about code coverage .............................................. 167
Requirements for using code coverage .............................. 167
Reference information on code coverage .......................... 168
Using state storage ................................................................. 171

Introduction to state storage .................................................. 171
   Reasons for using state storage ............................................. 171
   Briefly about state storage .................................................. 171
   Requirements ................................................................. 172

Procedures for using state storage ........................................ 172
   Setting up state storage .................................................... 172

Reference information on state storage .................................. 173

Using the sequencer ............................................................ 177

Introduction to the sequencer .............................................. 177
   Reasons for using the sequencer ........................................... 177
   Briefly about the sequencer ................................................ 177
   Requirements for using the sequencer .................................. 178

Procedures for using the sequencer ...................................... 178
   Setting up the sequencer (simple setup) .............................. 178
   Setting up the sequencer (advanced setup) ......................... 178
   Using the sequencer to locate a problem ............................. 179

Reference information on the sequencer .............................. 181

Using the cycle counter for 5xx devices ............................... 183

Introduction to the cycle counter for 5xx ............................. 183
   Reasons for using the cycle counter for 5xx .......................... 183
   Briefly about the cycle counter for 5xx ............................... 183
   Requirements for using the cycle counter for 5xx .................. 184

Procedures for using the cycle counter applications ................ 184
   Counting all CPU cycles .................................................. 184
   Measuring the DMA load versus the CPU load ....................... 185
   Profiling a specific part of your application ....................... 185
   Measuring the Trigger hits .............................................. 186
   Measuring the number of CPU cycles for a task ................... 186

Reference information on the cycle counter for 5xx ............... 187
Simulating interrupts ................................................................. 191

Introduction to interrupt simulation .................................... 191
  Reasons for using the interrupt simulation system ............. 191
  Briefly about the interrupt simulation system .................. 192
  Interrupt characteristics .................................................... 193
  Interrupt simulation states .............................................. 193
  C-SPY system macros for interrupts ............................... 195
  Target-adapting the interrupt simulation system ............. 195

Procedures for simulating interrupts .................................. 196
  Simulating a simple interrupt ........................................ 196
  Simulating an interrupt in a multi-task system ............... 197

Reference information on simulating interrupts ................. 198

Using C-SPY macros ............................................................... 205

Introduction to C-SPY macros ............................................. 205
  Reasons for using C-SPY macros ................................. 205
  Briefly about using C-SPY macros ............................. 206
  Briefly about setup macro functions and files .............. 206
  Briefly about the macro language ............................... 207

Procedures for using C-SPY macros .................................. 207
  Registering C-SPY macros—an overview .................... 208
  Executing C-SPY macros—an overview ........................ 208
  Using the Macro Configuration dialog box .................. 209
  Registering and executing using setup macros and setup files .... 210
  Executing macros using Quick Watch ......................... 211
  Executing a macro by connecting it to a breakpoint .......... 212

Reference information on the macro language ................. 213
  Macro functions ......................................................... 213
  Macro variables ........................................................ 214
  Macro strings .......................................................... 214
  Macro statements ...................................................... 215
  Formatted output ....................................................... 216

Reference information on reserved setup macro function names ................................................................. 218
<table>
<thead>
<tr>
<th>Section</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>Reference information on C-SPY system macros</td>
<td>219</td>
</tr>
<tr>
<td>The C-SPY Command Line Utility—cspybat</td>
<td>251</td>
</tr>
<tr>
<td><strong>Using C-SPY in batch mode</strong></td>
<td>251</td>
</tr>
<tr>
<td>Invocation syntax</td>
<td>251</td>
</tr>
<tr>
<td>Output</td>
<td>252</td>
</tr>
<tr>
<td>Using an automatically generated batch file</td>
<td>253</td>
</tr>
<tr>
<td><strong>Summary of C-SPY command line options</strong></td>
<td>253</td>
</tr>
<tr>
<td><strong>Descriptions of C-SPY command line options</strong></td>
<td>255</td>
</tr>
<tr>
<td>Debugger options</td>
<td>269</td>
</tr>
<tr>
<td><strong>Setting debugger options</strong></td>
<td>269</td>
</tr>
<tr>
<td><strong>Reference information on general debugger options</strong></td>
<td>270</td>
</tr>
<tr>
<td><strong>Reference information on C-SPY driver options</strong></td>
<td>274</td>
</tr>
<tr>
<td>Additional information on C-SPY drivers</td>
<td>281</td>
</tr>
<tr>
<td><strong>Reference information on the C-SPY simulator</strong></td>
<td>281</td>
</tr>
<tr>
<td><strong>Reference information on the C-SPY FET Debugger driver</strong></td>
<td>282</td>
</tr>
<tr>
<td><strong>Resolving problems</strong></td>
<td>286</td>
</tr>
<tr>
<td>The device port pins do not work</td>
<td>286</td>
</tr>
<tr>
<td>Write failure during load</td>
<td>287</td>
</tr>
<tr>
<td>No contact with the target hardware</td>
<td>288</td>
</tr>
<tr>
<td><strong>Index</strong></td>
<td>289</td>
</tr>
</tbody>
</table>
## Tables

1: Typographic conventions used in this guide ......................................................... 23
2: Naming conventions used in this guide ................................................................. 23
3: Driver differences ............................................................................................... 30
4: C-SPY assembler symbols expressions ............................................................... 73
5: Handling name conflicts between hardware registers and assembler labels .......... 73
6: Effects of display format setting on different types of expressions ..................... 79
7: Effects of display format setting on different types of expressions ..................... 82
8: C-SPY macros for breakpoints ........................................................................... 96
9: Cycle counter registers ...................................................................................... 136
10: Project options for enabling the profiler ........................................................... 161
11: Project options for enabling code coverage ....................................................... 168
12: Sequencer settings - example .......................................................................... 180
13: State Storage Control settings—example ......................................................... 180
14: Cycle Counter 1, combinations of start, stop, and clear reactions ...................... 189
15: Timer interrupt settings .................................................................................... 197
16: Examples of C-SPY macro variables .................................................................. 214
17: C-SPY setup macros ....................................................................................... 218
18: Summary of system macros .............................................................................. 219
19: __cancelInterrupt return values ....................................................................... 221
20: __disableInterrupts return values ..................................................................... 222
21: __driverType return values ............................................................................. 223
22: __enableInterrupts return values ...................................................................... 223
23: __evaluate return values .................................................................................. 224
24: __isBatchMode return values .......................................................................... 224
25: __loadImage return values ................................................................................ 225
26: __openFile return values ................................................................................ 228
27: __readFile return values ................................................................................ 230
28: __setAdvancedTriggerBreak return values .................................................... 234
29: __setCodeBreak return values ......................................................................... 235
30: __setConditionalBreak return values .............................................................. 236
31: __setDataBreak return values .......................................................................... 238
32: __setLogBreak return values ............................................................................ 239
33: __setRangeBreak return values ...................................................................... 240
34: __setSimBreak return values .......................................................................... 241
35: __setTraceStartBreak return values ................................................................. 242
36: __setTraceStopBreak return values ................................................................. 243
37: __sourcePosition return values ....................................................................... 244
38: __unloadImage return values .......................................................................... 247
39: cspybat parameters ....................................................................................... 251
40: Options specific to the C-SPY drivers you are using ...................................... 269
Figures

1: C-SPY and target systems ................................................................. 28
2: C-SPY FET Debugger communication overview ............................... 33
3: Get Alternative File dialog box .......................................................... 39
4: Example applications ........................................................................ 42
5: Debug menu ...................................................................................... 44
6: Images window .................................................................................. 47
7: Images window context menu ............................................................ 48
8: Get Alternative File dialog box .......................................................... 49
9: Device Information window ................................................................. 50
10: C-SPY highlighting source location .................................................. 55
11: C-SPY Disassembly window .............................................................. 58
12: Disassembly window context menu ................................................... 60
13: Call Stack window ............................................................................ 62
14: Call Stack window context menu ....................................................... 62
15: Terminal I/O window .......................................................................... 63
16: Ctrl codes menu ................................................................................ 64
17: Input Mode dialog box ....................................................................... 64
18: Terminal I/O Log File dialog box ....................................................... 64
19: LCD window ..................................................................................... 65
20: LCD Settings dialog box ................................................................... 66
21: Debug Log window (message window) ............................................. 66
22: Debug Log window context menu ..................................................... 67
23: Log File dialog box .......................................................................... 67
24: Report Assert dialog box ................................................................... 68
25: Autostep settings dialog box .............................................................. 69
26: Viewing assembler variables in the Watch window .......................... 76
27: Auto window ................................................................................... 77
28: Locals window .................................................................................. 77
29: Watch window .................................................................................. 78
30: Watch window context menu ............................................................ 78
31: Live Watch window .......................................................................... 79
32: Statics window ................................................................. 80
33: Statics window context menu ........................................ 81
34: Select Statics dialog box .................................................... 82
35: Quick Watch window ......................................................... 83
36: Symbols window ............................................................. 84
37: Symbols window context menu ........................................ 85
38: Resolve Symbol Ambiguity dialog box .............................. 85
39: Breakpoint icons ............................................................. 90
40: Modifying breakpoints via the context menu ....................... 95
41: Breakpoints window ........................................................ 98
42: Breakpoints window context menu .................................... 99
43: Breakpoint Usage dialog box ............................................. 100
44: Code breakpoints dialog box ............................................. 101
45: Log breakpoints dialog box .............................................. 103
46: Data breakpoints dialog box .............................................. 104
47: Immediate breakpoints dialog box .................................... 106
48: Range breakpoints dialog box ........................................... 107
49: Conditional breakpoints dialog box ................................. 109
50: Advanced trigger dialog box ............................................ 112
51: Enter Location dialog box ............................................... 114
52: Breakpoint combiner dialog box ...................................... 115
53: Resolve Source Ambiguity dialog box ............................... 116
54: Zones in C-SPY ............................................................. 121
55: Memory window ............................................................ 123
56: Memory window context menu ....................................... 125
57: Memory Save dialog box ................................................ 127
58: Memory Restore dialog box ............................................. 128
59: Fill dialog box .............................................................. 128
60: Symbolic Memory window .............................................. 130
61: Symbolic Memory window context menu ......................... 131
62: Stack window ............................................................... 132
63: Stack window context menu ............................................ 134
64: Register window ........................................................... 135
65: Memory Access Setup dialog box ................................... 137
66: Edit Memory Access dialog box ................................................................. 139
67: Memory Dump dialog box ................................................................. 140
68: The Trace window ........................................................................ 146
69: Function Trace window ................................................................. 148
70: Timeline window ........................................................................ 149
71: Timeline window context menu ................................................................. 150
72: Trace Start breakpoints dialog box ....................................................... 152
73: Trace Stop breakpoints dialog box ....................................................... 153
74: Trace Expressions window ................................................................. 154
75: Find in Trace dialog box ...................................................................... 155
76: Find in Trace window ......................................................................... 156
77: Instruction count in Disassembly window .................................................. 162
78: Function Profiler window ................................................................. 163
79: Function Profiler window context menu ................................................ 165
80: Code Coverage window ........................................................................ 168
81: Code coverage context menu .............................................................. 170
82: State Storage Control window ............................................................. 174
83: State storage window ........................................................................ 175
84: Sequencer Control window ................................................................. 181
85: Cycle Counter 5xx Control window ....................................................... 187
86: Simulated interrupt configuration ......................................................... 193
87: Simulation states - example 1 .............................................................. 194
88: Simulation states - example 2 .............................................................. 194
89: Interrupt Setup dialog box ................................................................. 198
90: Edit Interrupt dialog box ...................................................................... 200
91: Forced Interrupt window ..................................................................... 202
92: Interrupt Log window .......................................................................... 203
93: Interrupt Log window context menu ..................................................... 204
94: Macro Configuration dialog box ........................................................... 209
95: Quick Watch window ......................................................................... 211
96: Debugger setup options ........................................................................ 270
97: Debugger images options ...................................................................... 272
98: Debugger extra options ......................................................................... 273
99: Debugger plugin options ...................................................................... 273
100: FET Debugger setup options ................................................................. 275
101: FET Debugger download options ....................................................... 277
102: FET Debugger breakpoints options .................................................. 278
103: Simulator setup options ...................................................................... 279
104: Simulator menu .................................................................................. 281
105: The Emulator menu ............................................................................ 283
106: General Clock Control dialog box ...................................................... 285
107: Extended Clock Control dialog box ..................................................... 286
Preface

Welcome to the C-SPY® Debugging Guide for MSP430. 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 MSP430 microcontroller.

Who should read this guide

Read this guide if you want to get the most out of the features available in C-SPY. In addition, you should have working knowledge of:

- The C or C++ programming language
- Application development for embedded systems
- The architecture and instruction set of the MSP430 microcontroller (refer to the chip manufacturer's documentation)
- 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 21.

How to use this guide

If you are new to using IAR Embedded Workbench, we suggest that you first read the guide Getting Started with IAR Embedded Workbench® for an overview of the tools and the features that the IDE offers.

If you already have had some experience using IAR Embedded Workbench, but need refreshing on how to work with the IAR Systems development tools, the tutorials which you can find in the IAR Information Center is a good place to begin. The process of managing projects and building, as well as editing, is described in the IDE Project Management and Building Guide, whereas information about how to use C-SPY for debugging is described in this guide.

This guide describes a number of topics, where each topic section contains an introduction which also covers concepts related to the topic. This will give you a good understanding of the features in C-SPY. Furthermore, the topic section provides procedures with step-by-step descriptions to help you use the features. Finally, each topic section gives all relevant reference information.
What this guide contains

We also recommend the Glossary which you can find in the IDE Project Management and Building Guide if you should encounter any unfamiliar terms in the IAR Systems user and reference guides.

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.

- **Working with 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.

- **Using breakpoints** describes the breakpoint system and the various ways to set breakpoints.

- **Monitoring 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.

- **Using state storage** describes how the state storage module can help you to examine how your code is executed, and find problems in a specific stage of the execution.

- **Using the sequencer** describes the sequencer module, a simple state machine that lets you break the execution or trigger the state storage module using a more complex method than a standard breakpoint.

- **Using the cycle counter for 5xx devices** describes the second cycle counter for MSP430F5xx devices, and how it can help you to profile your application or to measure how long some tasks take.
● **Simulating 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 C-SPY 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. For information about:

- System requirements and information about how to install and register the IAR Systems products, refer to 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, refer to the guide *Getting Started with IAR Embedded Workbench*.

- Using the IDE for project management and building, refer to the *IDE Project Management and Building Guide*.

- Programming for the IAR C/C++ Compiler for MSP430, refer to the *IAR C/C++ Compiler Reference Guide for MSP430*.

- Using the IAR XLINK Linker, the IAR XAR Library Builder, and the IAR XLIB Librarian, refer to the *IAR Linker and Library Tools Reference Guide*.

- Programming for the IAR Assembler for MSP430, refer to the *MSP430 IAR Assembler Reference Guide*.

- Using the IAR DLIB Library, refer to the *DLIB Library Reference information*, available in the online help system.
Document conventions

Using the IAR CLIB Library, refer to the IAR C Library Functions Reference Guide, available in the online help system.

Porting application code and projects created with a previous version of IAR Embedded Workbench for MSP430, refer to the IAR Embedded Workbench® Migration Guide for MSP430.


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

THE ONLINE HELP SYSTEM

The context-sensitive online help contains:

- Comprehensive information about debugging using the IAR C-SPY® Debugger
- 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. Note that if you select a function name in the editor window and press F1 while using the CLIB library, you will get reference information for the DLIB library.

WEB SITES

Recommended web sites:

- Finally, the Embedded C++ Technical Committee web site, www.caravan.net/ec2plus, contains information about the Embedded C++ standard.

Document conventions

When, in this text, 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 `430\doc`, the full path to the location is assumed, for example `c:\Program Files\IAR Systems\Embedded Workbench 6.0\430\doc`.

**TYPOGRAPHIC CONVENTIONS**

This guide uses the following typographic conventions:

<table>
<thead>
<tr>
<th>Style</th>
<th>Used for</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>computer</strong></td>
<td>• Source code examples and file paths.</td>
</tr>
<tr>
<td></td>
<td>• Text on the command line.</td>
</tr>
<tr>
<td></td>
<td>• Binary, hexadecimal, and octal numbers.</td>
</tr>
<tr>
<td><strong>parameter</strong></td>
<td>A placeholder for an actual value used as a parameter, for example <code>filename.h</code> where <code>filename</code> represents the name of the file.</td>
</tr>
<tr>
<td><code>[option]</code></td>
<td>An optional part of a command.</td>
</tr>
<tr>
<td>`[a</td>
<td>b</td>
</tr>
<tr>
<td>`{a</td>
<td>b</td>
</tr>
<tr>
<td><strong>bold</strong></td>
<td>Names of menus, menu commands, buttons, and dialog boxes that appear on the screen.</td>
</tr>
<tr>
<td><strong>italic</strong></td>
<td>• A cross-reference within this guide or to another guide.</td>
</tr>
<tr>
<td></td>
<td>• Emphasis.</td>
</tr>
<tr>
<td><code>...</code></td>
<td>An ellipsis indicates that the previous item can be repeated an arbitrary number of times.</td>
</tr>
<tr>
<td>![Icon]</td>
<td>Identifies instructions specific to the IAR Embedded Workbench® IDE interface.</td>
</tr>
<tr>
<td>![Icon]</td>
<td>Identifies instructions specific to the command line interface.</td>
</tr>
<tr>
<td>![Icon]</td>
<td>Identifies helpful tips and programming hints.</td>
</tr>
<tr>
<td>![Icon]</td>
<td>Identifies warnings.</td>
</tr>
</tbody>
</table>

*Table 1: Typographic conventions used in this guide*

**NAMING CONVENTIONS**

The following naming conventions are used for the products and tools from IAR Systems® referred to in this guide:

<table>
<thead>
<tr>
<th>Brand name</th>
<th>Generic term</th>
</tr>
</thead>
<tbody>
<tr>
<td>IAR Embedded Workbench® for MSP430</td>
<td>IAR Embedded Workbench®</td>
</tr>
<tr>
<td>IAR Embedded Workbench® IDE for MSP430</td>
<td>the IDE</td>
</tr>
</tbody>
</table>

*Table 2: Naming conventions used in this guide*
<table>
<thead>
<tr>
<th>Brand name</th>
<th>Generic term</th>
</tr>
</thead>
<tbody>
<tr>
<td>IAR C-SPY® Debugger for MSP430</td>
<td>C-SPY, the debugger</td>
</tr>
<tr>
<td>IAR C-SPY® Simulator</td>
<td>the simulator</td>
</tr>
<tr>
<td>IAR C/C++ Compiler™ for MSP430</td>
<td>the compiler</td>
</tr>
<tr>
<td>IAR Assembler™ for MSP430</td>
<td>the assembler</td>
</tr>
<tr>
<td>IAR XLINK Linker™</td>
<td>XLINK, the linker</td>
</tr>
<tr>
<td>IAR XAR Library Builder™</td>
<td>the library builder</td>
</tr>
<tr>
<td>IAR XLIB Librarian™</td>
<td>the librarian</td>
</tr>
<tr>
<td>IAR DLIB Library™</td>
<td>the DLIB library</td>
</tr>
<tr>
<td>IAR CLIB Library™</td>
<td>the CLIB library</td>
</tr>
</tbody>
</table>

Table 2: Naming conventions used in this guide (Continued)
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 FET Debugger driver.

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 control 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. Any variable and expression can be evaluated in one-shot views. You can easily both monitor and log values of a defined set of expressions during a longer period of time. You have instant control over local variables, and real-time data is displayed non-intrusively. Finally, the last referred variables are displayed automatically.

- **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 RTOSes are supported:

- IAR PowerPac
- Segger embOS
- Micrium μC/OS-II
- OSEK.

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.

Provided that one or more real-time operating system plugin modules are supported for the IAR Embedded Workbench version you are using, you can load one for use with C-SPY. A C-SPY RTOS awareness plugin module gives you a high level of control and visibility over an application built on top of a real-time operating system. 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 each chapter of this part of the 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](image)
Note: In IAR Embedded Workbench for MSP430, 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 30.

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 and the Stack window, both 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, refer to the documentation provided with IAR visualSTATE.

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

C-SPY drivers overview

The IAR C-SPY Debugger for the MSP430 microcontroller is available with drivers for these target systems:

- Simulator
- FET Debugger.

DIFFERENCES BETWEEN THE C-SPY DRIVERS

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

<table>
<thead>
<tr>
<th>Feature</th>
<th>Simulator</th>
<th>FET Debugger</th>
</tr>
</thead>
<tbody>
<tr>
<td>OP-fetch</td>
<td>x</td>
<td>x</td>
</tr>
<tr>
<td>Code breakpoints</td>
<td>x</td>
<td>x¹</td>
</tr>
<tr>
<td>Data breakpoints</td>
<td>x</td>
<td>--</td>
</tr>
</tbody>
</table>

Table 3: Driver differences
The IAR C-SPY Debugger

A limited number of hardware breakpoints, unlimited software breakpoints. For more information, see Breakpoints in the C-SPY FET Debugger driver, page 90.

2 The cycle counter 5xx is supported during single step, you can then view the value of the cycle counter in the Register window.

3 There are several levels of support for the Enhanced Emulation Module. The higher the level of support, the more debugging features are available.

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.

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).

Table 3: Driver differences (Continued)

<table>
<thead>
<tr>
<th>Feature</th>
<th>Simulator</th>
<th>FET Debugger</th>
</tr>
</thead>
<tbody>
<tr>
<td>Log breakpoints</td>
<td>x</td>
<td>x</td>
</tr>
<tr>
<td>Trace breakpoints</td>
<td>x</td>
<td>--</td>
</tr>
<tr>
<td>Execution in real time</td>
<td>--</td>
<td>x</td>
</tr>
<tr>
<td>Zero memory footprint</td>
<td>x</td>
<td>x</td>
</tr>
<tr>
<td>Simulated interrupts</td>
<td>x</td>
<td>--</td>
</tr>
<tr>
<td>Real interrupts</td>
<td>--</td>
<td>x</td>
</tr>
<tr>
<td>Interrupt logging</td>
<td>x</td>
<td>--</td>
</tr>
<tr>
<td>Live watch</td>
<td>x</td>
<td>--</td>
</tr>
<tr>
<td>Cycle counter&lt;sup&gt;2&lt;/sup&gt;</td>
<td>x</td>
<td>x</td>
</tr>
<tr>
<td>Code coverage</td>
<td>x</td>
<td>--</td>
</tr>
<tr>
<td>Data coverage</td>
<td>x</td>
<td>--</td>
</tr>
<tr>
<td>Function/instruction profiler</td>
<td>x</td>
<td>--</td>
</tr>
<tr>
<td>Enhanced Emulation Module support&lt;sup&gt;3&lt;/sup&gt;</td>
<td>--</td>
<td>x</td>
</tr>
<tr>
<td>Trace</td>
<td>x</td>
<td>--</td>
</tr>
</tbody>
</table>

1 A limited number of hardware breakpoints, unlimited software breakpoints. For more information, see Breakpoints in the C-SPY FET Debugger driver, page 90.
2 The cycle counter 5xx is supported during single step, you can then view the value of the cycle counter in the Register window.
3 There are several levels of support for the Enhanced Emulation Module. The higher the level of support, the more debugging features are available.
SELECTING THE SIMULATOR DRIVER

Before starting C-SPY, you must choose the simulator driver:

1. In the IDE, choose Project→Options and click the Setup tab in the Debugger category.
2. Choose Simulator from the Driver drop-down list.

The C-SPY FET Debugger driver

The C-SPY Flash Emulator Tool Debugger is a JTAG debugger that supports all Texas Instruments’ boards, and several third-party JTAG debug probes. It provides automatic flash downloading and takes advantage of on-chip debug facilities.

To make the C-SPY FET Debugger work, a communication driver must be installed on the host PC. This driver is automatically installed during the installation of the IAR Embedded Workbench IDE. Because the hardware debugger kernel is built into the microcontroller, no ordinary ROM-monitor program or extra specific hardware is needed to make the debugging work. You can also use the debugger on your own hardware design.

The C-SPY FET Debugger provides real-time debugging at a low cost.

At the time of writing this guide, the IAR C-SPY Debugger for the MSP430 microcontrollers is available with drivers for these JTAG debug probes supported by the FET debugger driver:

- Texas Instruments FET
- Olimex JTAG interface
- Elprotronic JTAG interface.

FEATURES

In addition to the general features of C-SPY, the FET Debugger driver also provides:

- Execution in real time with full access to the microcontroller
- High-speed communication through a JTAG interface
- Zero memory footprint on the target system
- Hardware code breakpoints
- Built-in flash loader.

Depending on the level of Enhanced Emulation Module (EEM) support, you might have access also to:

- State storage
● Sequencer
● Clock control
● Advanced cycle counter.

**Note:** Code coverage, function level profiling, and live watch are not supported by the C-SPY FET Debugger driver.

**COMMUNICATION OVERVIEW**

The C-SPY FET Debugger driver uses the USB or parallel port to communicate with the FET Interface module. The FET Interface module communicates with the JTAG interface on the hardware.

For further information, refer to the documentation supplied with the FET 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

For information about the hardware installation, see the documentation supplied with the FET Debugger from Texas Instruments. The following power-up sequence is recommended to ensure proper communication between the target board, FET Debugger, and C-SPY:

1. Power up the target board.
2. Power up FET Debugger.
3. Start the C-SPY debugging session.
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 the steps involved for setting 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 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.

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

   For information about these options, see Debugger options, page 269.

3 Click OK.
4 Choose Tools>Options>Debugger to configure:

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

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

See also Adapting for target hardware, page 40.

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 the debugger 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 detailed information about setup macro files and functions, see Briefly about setup macro functions and files, page 206. For an example about how to use a setup macro file, see the chapter Initializing target hardware before C-SPY starts, page 41.

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 msp430\config directory and they have the filename extension ddf.

For more information about device description files, see *Adapting for target hardware*, page 40.

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 273.

---

**Starting C-SPY**

When you have set up the debugger, you are ready to start a debug session; this section describes the steps involved.

More specifically, you will get information about:

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

You can choose to start the debugger 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:

![Get Alternative File dialog box](image)

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 49.

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.
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 37. They contain device-specific information such as:

- Memory information for device-specific memory zones, see C-SPY memory zones, page 120
- 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 Simulating interrupts, page 191.

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 37.

INITIALIZING TARGET HARDWARE BEFORE C-SPY STARTS

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.

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 msp430FET_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.

   ![Figure 4: Example applications](image)

   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 General Options>Target>Device and Debugger>Setup>Driver. As for other settings, the project is set up to suit the target system you selected.
Getting started using C-SPY

For further details about the C-SPY options and how to configure C-SPY to interact with the target board, see Debugger options, page 269.

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 Execute>Go or click the Go button to start the application.

Click the Stop button to stop execution.

Reference information on starting C-SPY

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

- C-SPY Debugger main window, page 43
- Images window, page 47
- Get Alternative File dialog box, page 49
- Device Information window, page 50.

For related information, see:

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

C-SPY Debugger main window

When you start the debugger, 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
- 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 when C-SPY is running:

**Debug**

Provides commands for executing and debugging the source application; see Debug menu, page 44. 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 281.

**Emulator**

Provides commands specific to the C-SPY FET Debugger driver. This menu is only available when the driver is used; see Emulator menu, page 283.

**Debug menu**

The Debug menu is available when C-SPY is running. 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.

![Debug menu](image)

These commands are available:

**Go**

F5

Executes from the current statement or instruction until a breakpoint or program exit is reached.
Getting started using C-SPY

Break

Stops the application execution.

Reset

Resets the target processor.

Stop Debugging

Crl+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 69.

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.

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 127.

Memory>Restore

Displays a dialog box where you can load the contents of a file in Intel-extended or Motorola s-record format to a specified memory zone, see Memory Restore dialog box, page 128.

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.
### C-SPY windows

These windows specific to C-SPY are available when C-SPY is running:

- 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
- LCD window
- Code Coverage window
- Function Profiler window

### 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 209.

### 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 67.

### 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 64.
Getting started using C-SPY

- 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.

```
<table>
<thead>
<tr>
<th>Name</th>
<th>Path</th>
</tr>
</thead>
<tbody>
<tr>
<td>cAll images</td>
<td>[Combines debug information from all images]</td>
</tr>
<tr>
<td>project</td>
<td>C:\Documents and Settings\My Documents\VAR Embedded Workbench\Debug\Eval\Project1.out</td>
</tr>
<tr>
<td>extraImage</td>
<td>C:\Documents and Settings\My Documents\VAR Embedded Workbench\Debug\Eval\ximage.out</td>
</tr>
</tbody>
</table>
```

*Figure 6. 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.

Display area

This area lists the loaded images in these columns:

- **Name**: The name of the loaded image.
- **Path**: The path to the loaded image.

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.

Context menu

This context menu is 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 39
- *Images*, page 272
- *__loadImage*, page 225.
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 dialog box](image)

**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 38.
Reference information on starting C-SPY

Device Information window

The Device Information window is available from the Emulator menu.

![Device Information Window](image)

**Device Information**

<table>
<thead>
<tr>
<th>Device</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>Device name</td>
<td>MSP430F5438</td>
</tr>
<tr>
<td>VCC voltage</td>
<td>3.0 V</td>
</tr>
<tr>
<td>VCC external voltage</td>
<td>0 V</td>
</tr>
</tbody>
</table>

*Figure 9: Device Information window*

Displays information about the target hardware being used.
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
- 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
Introduction to application execution

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 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 69.

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

```c
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, \( g(n-1) \):

```c
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:

```c
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:

```c
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:

```c
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.

Single-stepping and flash memory in the C-SPY FET Debugger

When you use the FET Debugger driver, be aware that single-stepping over instructions that manipulate the flash memory might cause some unexpected side-effects. Multiple internal machine cycles are required to clear and program the flash memory. When single-stepping over instructions that manipulate the flash memory, control is given back to C-SPY before these operations are complete. Consequently, C-SPY will update its memory window with erroneous information. A workaround to this behavior is to follow the flash access instruction with a \texttt{NOP} instruction, and then step past the \texttt{NOP} before reviewing the effects of the flash access instruction.

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.

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 MSP430 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 reference information, see Terminal I/O window, page 63 and Terminal I/O Log File dialog box, page 64.

**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 58
- Call Stack window, page 62
- Terminal I/O window, page 63
- Terminal I/O Log File dialog box, page 64
- LCD window, page 65
- LCD Settings dialog box, page 66
- Debug Log window, page 66
- Log File dialog box, page 67
- Report Assert dialog box, page 68
- Autostep settings dialog box, page 69.

For related information, see Terminal I/O options in IDE Project Management and Building Guide.
Disassembly window

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

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.

Toolbar

The toolbar contains:

Go to The location you want to view. This can be a memory address, or the name of a variable, function, or label.

Zone display Lists the available memory zones to display, see C-SPY memory zones, page 120.
**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 inspection of other frames, that is the next assembler instruction to be executed upon return to this frame.

- **Red dot** Indicates a breakpoint. Double-click in the gray left-side margin of the window to set a breakpoint. For more information, see Using breakpoints, page 87.

- **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:

<table>
<thead>
<tr>
<th>Move to PC</th>
</tr>
</thead>
<tbody>
<tr>
<td>Run to Cursor</td>
</tr>
<tr>
<td>Code Coverage</td>
</tr>
<tr>
<td>Instruction Profiling</td>
</tr>
<tr>
<td>Toggle Breakpoint (Code)</td>
</tr>
<tr>
<td>Toggle Breakpoint (Log)</td>
</tr>
<tr>
<td>Toggle Breakpoint (Trace Start)</td>
</tr>
<tr>
<td>Toggle Breakpoint (Trace Stop)</td>
</tr>
<tr>
<td>Enable/Disable Breakpoint</td>
</tr>
<tr>
<td>Set Host Statement</td>
</tr>
<tr>
<td>Copy Window Contents</td>
</tr>
<tr>
<td>Mixed-Mode</td>
</tr>
</tbody>
</table>

Figure 12: 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.
Executing your application

<table>
<thead>
<tr>
<th>Feature</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Toggle Breakpoint</strong></td>
<td><strong>(Code)</strong> 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 101.</td>
</tr>
<tr>
<td><strong>Toggle Breakpoint</strong></td>
<td><strong>(Log)</strong> 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 103.</td>
</tr>
<tr>
<td><strong>Toggle Breakpoint</strong></td>
<td><strong>(Trace Start)</strong> 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 152.</td>
</tr>
<tr>
<td><strong>Toggle Breakpoint</strong></td>
<td><strong>(Trace Stop)</strong> 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 153.</td>
</tr>
<tr>
<td><strong>Enable/Disable Breakpoint</strong></td>
<td>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.</td>
</tr>
<tr>
<td><strong>Edit Breakpoint</strong></td>
<td>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.</td>
</tr>
<tr>
<td><strong>Set Next Statement</strong></td>
<td>Sets the program counter to the address of the instruction at the insertion point.</td>
</tr>
<tr>
<td><strong>Copy Window Contents</strong></td>
<td>Copies the selected contents of the Disassembly window to the clipboard.</td>
</tr>
<tr>
<td><strong>Mixed-Mode</strong></td>
<td>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.</td>
</tr>
</tbody>
</table>
Call Stack window

The Call stack window is available from the View menu.

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.

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 13: Call Stack window

Figure 14: 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.

![Terminal I/O window](image)

*Figure 15: Terminal I/O window*

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

**To use this window, you must:**

1. **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.
**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.

![Ctrl codes menu](image)

Figure 16: 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.

![Input Mode dialog box](image)

Figure 17: 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**.

![Terminal I/O Log File dialog box](image)

Figure 18: Terminal I/O Log File dialog box
Use this dialog box to select a destination log file for terminal I/O from C-SPY.

**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.

**LCD window**

The LCD window is available from the View menu.

Figure 19: LCD window

This window simulates a 7- or 14-segments LCD display.

**Toolbar**

Displays the **LCD Settings** dialog box, where you can configure the LCD window.
LCD Settings dialog box

The LCD Settings dialog box is available from the LCD window.

![LCD Settings dialog box](image)

Figure 20: LCD Settings dialog box

Use this dialog box to configure the LCD window.

**LCD configuration file**

Specify the LCD display to simulate, using the browse button. Choose between a 7-segment display and a 14-segment display.

**LCD control register address**

Specify the address of the LCD control register.

Debug Log window

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

![Debug Log window](image)

Figure 21: 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 when C-SPY is running. 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>
```

**Context menu**

This context menu is available:

- **Copy**
- **Select All**
- **Clear All**

*Figure 22: 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**.

*Figure 23: Log File dialog box*

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

Use the browse button, to override the default file and location of the log file (the default filename extension is log).

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.

![Figure 24: Report Assert dialog box](image)

**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.

![Autostep settings dialog box](image)

*Figure 25: Autostep settings dialog box*

Use this dialog box to customize autostepping.

The drop-down menu lists the available step commands.

**Delay**

Specify the delay between each step in milliseconds.
Working with 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
- 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
- Viewing assembler variables.

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.
Introduction to working with variables and expressions

- 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 Trace-related windows let you inspect the program flow up to a specific state. For more information, see Collecting and using trace data, page 143.

Details about using these windows
All the windows are easy to use. You can add, modify, and remove expressions, and change the display format.

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 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 and the Quick Watch window where it is not relevant.

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:

\[ i + j \]
\[ i = 42 \]
\[ \#asm_label \]
\[ \#R2 \]
\[ \#PC \]
\[ my\_macro\_func(19) \]

**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 register names. That is, general purpose registers, such as R4–R15, and special purpose registers, such as the program counter and the status register. 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 40.

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.
\#label7 | Refers to the integral address of label7 within its zone.

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.

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 135.
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 details of C-SPY macro functions and how to use them, see Briefly about the macro language, page 207.

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 details of C-SPY macro variables and how to use them, see Reference information on the macro language, page 213.

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:

```c
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`.

**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:

![Figure 26: Viewing assembler variables in the Watch window](image)

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.

Reference information on working with variables and expressions

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

- Auto window, page 77
- Locals window, page 77
- Watch window, page 78
- Live Watch window, page 79
- Statics window, page 80
- Select Statics dialog box, page 82
- Quick Watch window, page 83
- Symbols window, page 84
- Resolve Symbol Ambiguity dialog box, page 85.
For trace-related reference information, see Reference information on trace, page 146.

**Auto window**

The Auto window is available from the View menu.

![Auto window](image)

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.

**Context menu**

For a description of the context menu, see Watch window, page 78.

**Locals window**

The Locals window is available from the View menu.

![Locals window](image)

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.
Reference information on working with variables and expressions

**Context menu**

For a description of the context menu, see *Watch window*, page 78.

**Watch window**

The Watch window is available from the View menu.

Use this window to monitor the values of C-SPY expressions or variables. 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.

**Context menu**

This context menu is available:

![Context menu](image)

*Figure 30: 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, see Table 6, *Effects of display format setting on different types of expressions*. Your selection of display format is saved between debug sessions.

- **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 75.

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

<table>
<thead>
<tr>
<th>Type of expression</th>
<th>Effects of display format setting</th>
</tr>
</thead>
<tbody>
<tr>
<td>Variable</td>
<td>The display setting affects only the selected variable, not other variables.</td>
</tr>
<tr>
<td>Array element</td>
<td>The display setting affects the complete array, that is, the same display format is used for each array element.</td>
</tr>
<tr>
<td>Structure field</td>
<td>All elements with the same definition—the same field name and C declaration type—are affected by the display setting.</td>
</tr>
</tbody>
</table>

*Table 6: Effects of display format setting on different types of expressions*

**Live Watch window**

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

**Context menu**

For a description of the context menu, see *Watch window*, page 78.

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.

<table>
<thead>
<tr>
<th>Expression</th>
<th>Value</th>
<th>Location</th>
<th>Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>cel_count+Temp()</td>
<td>0</td>
<td>DATA:0x00002</td>
<td>int</td>
</tr>
<tr>
<td>x&lt;array&gt;</td>
<td>1</td>
<td>DATA:0x00062</td>
<td>unsigned int</td>
</tr>
<tr>
<td>[1]</td>
<td>0</td>
<td>DATA:0x00062</td>
<td>unsigned int</td>
</tr>
<tr>
<td>[2]</td>
<td>0</td>
<td>DATA:0x00062</td>
<td>unsigned int</td>
</tr>
<tr>
<td>[3]</td>
<td>0</td>
<td>DATA:0x00062</td>
<td>unsigned int</td>
</tr>
<tr>
<td>[4]</td>
<td>0</td>
<td>DATA:0x00062</td>
<td>unsigned int</td>
</tr>
<tr>
<td>[5]</td>
<td>0</td>
<td>DATA:0x00062</td>
<td>unsigned int</td>
</tr>
<tr>
<td>[6]</td>
<td>0</td>
<td>DATA:0x00062</td>
<td>unsigned int</td>
</tr>
<tr>
<td>[7]</td>
<td>0</td>
<td>DATA:0x00062</td>
<td>unsigned int</td>
</tr>
<tr>
<td>[8]</td>
<td>0</td>
<td>DATA:0x00062</td>
<td>unsigned int</td>
</tr>
<tr>
<td>[9]</td>
<td>0</td>
<td>DATA:0x00062</td>
<td>unsigned int</td>
</tr>
</tbody>
</table>

*Figure 32: Statics window*
Working with variables and expressions

This window displays the values of variables with static storage duration, typically that is variables with file scope but also 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.

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. 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:

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

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, see Table 6, *Effects of display format setting on different types of expressions*. Your selection of display format is saved between debug sessions.
Select Statics

Displays a dialog box where you can select a subset of variables to be displayed in the Statics window, see Select Statics dialog box, page 82.

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

<table>
<thead>
<tr>
<th>Type of expression</th>
<th>Effects of display format setting</th>
</tr>
</thead>
<tbody>
<tr>
<td>Variable</td>
<td>The display setting affects only the selected variable, not other variables.</td>
</tr>
<tr>
<td>Array element</td>
<td>The display setting affects the complete array, that is, the same display format is used for each array element.</td>
</tr>
<tr>
<td>Structure field</td>
<td>All elements with the same definition—the same field name and C declaration type—are affected by the display setting.</td>
</tr>
</tbody>
</table>

Table 7: Effects of display format setting on different types of expressions

Select Statics dialog box

The Select Statics dialog box is available from the context menu in the Statics window.

Use this dialog box to select which variables should be displayed in the Statics window.

Show all variables with static storage duration

Makes all variables be displayed in the Statics window, including new variables that are added to your application between debug sessions.
Show selected variables only

Selects which variables to be displayed in the Statics window. Note that if you add a new variable to your application between two debug sessions, this variable will not automatically be displayed in the Statics window. Select the checkbox next to a variable to make that variable be displayed. Alternatively, click Select All.

Quick Watch window

The Quick Watch window is available from the View menu and from the context menu in the editor 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:
   1. In the Quick Watch window, type the expression you want to examine in the Expressions text box.
   2. Click the Recalculate button to calculate the value of the expression.

For an example, see Executing macros using Quick Watch, page 211.
Context menu

For a description of the context menu, see Watch window, page 78.

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.

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.

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:

![Symbols window context menu](image)

*Figure 37: 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.

![Resolve Symbol Ambiguity dialog box](image)

*Figure 38: Resolve Symbol Ambiguity dialog box*

- **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.
Reference information on working with variables and expressions
Using breakpoints

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

- Introduction to setting and using breakpoints
- Procedures for 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 FET Debugger driver
- 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 dialog box also lists all internally used breakpoints, see Breakpoint consumers, page 91.

Breakpoints are set with a higher precision than single lines, using the same mechanism as when stepping; for more details about the precision, see Single stepping, page 52.

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.

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.
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.

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.

Range breakpoints
Range breakpoints can be set on a data or an address range, and the action can be specified to occur on an access inside or outside the specified range. These breakpoints are only available if you are using a device that supports the Enhanced Emulation Module at the required level.

Advanced trigger breakpoints
Advanced trigger breakpoints can be set with various operators on the address bus, the data bus, or on a register, to be triggered by a certain kind of access. These breakpoints are only available if you are using a device that supports the Enhanced Emulation Module at the required level.

Conditional breakpoints
Conditional breakpoints can be set with various operators on the address bus, the data bus, or on a register, to be triggered by a certain kind of access. You can also specify a conditional value. These breakpoints are only available if you are using a device that supports the Enhanced Emulation Module at the required level.
BREAKPOINT ICONS

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

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 dialog box.

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 FET DEBUGGER DRIVER

Using the C-SPY driver for the C-SPY FET Debugger driver, you can set code breakpoints. If you are using a device that supports the Enhanced Emulation Module at the required level you also have access to an extended breakpoint system with support for:

- breakpoints on addresses, data, and registers
- defining which type of access that should trigger the breakpoint: read, write, or fetch
- range breakpoints
- setting conditional breakpoints
● triggering different actions: stopping the execution, or starting the state storage module
● emulated breakpoints.

The Enhanced Emulation Module (at the required level) also gives you access to the sequencer module which is a state machine that uses breakpoints for triggering new states.

**Hardware breakpoints**

To set breakpoints, the C-SPY FET Debugger uses the hardware breakpoints available on the device. The number of hardware breakpoints is limited, and when all hardware breakpoints have been used C-SPY can use software breakpoints.

For information about the number of available hardware breakpoints for each device, see the release notes or the hardware documentation.

**Software breakpoints**

There are two types of software breakpoints: virtual breakpoints and emulated breakpoints. See Breakpoints, page 278, for information about how to specify which type to use.

When virtual breakpoints are used, C-SPY is forced into single-step mode after all hardware breakpoints have been used. However, if your device supports the Enhanced Emulation Module at the required level, you can use emulated breakpoints for access to an unlimited number of breakpoints.

When emulated breakpoints are used, the instruction where the breakpoint is set will be replaced by a special instruction that the debugger recognizes. When the debugger encounters such an instruction, it stops. This mechanism uses one hardware breakpoint to emulate an unlimited number of breakpoints.

To prevent the debugger from executing in single-step mode if you do not use emulated software breakpoints, you can disable the use of virtual breakpoints and—in the CLIB runtime environment—fine-tune the use of breakpoint consumers. This will increase the performance of the debugger, but you will only have access to the available number of hardware breakpoints. For information about the breakpoint consumers of the debugger system, see Breakpoint consumers, page 91.

Adding and removing software breakpoints generates many write accesses to the code memory. This can significantly reduce the lifetime of flash memories.

**BREAKPOINT CONSUMERS**

Exceeding the number of available hardware breakpoints, when you do not use emulated software 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.

**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 dialog box 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 when the debugger system is running. This means that they are not visible in the Breakpoints window.
- The command Edit>Run to Cursor is used
- The linker option With I/O emulation modules has been selected.

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

In the CLIB runtime environment, C-SPY will set a breakpoint if:

- the library functions putchar and getchar are used (low-level routines used by functions like printf and scanf)
- the application has an exit label.

You can disable the setting of system breakpoints on the putchar and getchar functions and on the exit label; see Breakpoints, page 278.

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

When the Run to option is selected and all hardware breakpoints have already been used, a virtual breakpoint will be set even if you have deselected the Use virtual breakpoints option. When you start the debugger under these conditions, C-SPY will prompt you to choose whether you want to execute in single-step mode or stop at the first instruction.
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.

Procedures for setting breakpoints

This section gives you step-by-step descriptions about how to set and use 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:

- Using the Toggle Breakpoint command toggles a code breakpoint. This command is available both from the Tools menu and from the context menus in the editor window and in the Disassembly window.
- Right-clicking in the left-side margin of the editor window or the Disassembly window toggles a 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.
TOGGING A SIMPLE CODE BREAKPOINT

Toggling a code breakpoint is a quick method of setting a breakpoint. The following
methods are available both in the editor window and in the Disassembly window:

- Double-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.

**To set a new breakpoint:**

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

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:

1. 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.

2. On the context menu, choose the appropriate command.

3. 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 this window will be triggered for both read and write accesses. All breakpoints defined in the Memory window are preserved between debug sessions.
Procedures for setting breakpoints

**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:

<table>
<thead>
<tr>
<th>C-SPY macro for breakpoints</th>
<th>Simulator</th>
<th>FET Debugger</th>
</tr>
</thead>
<tbody>
<tr>
<td>__setAdvancedTriggerBreak</td>
<td>—</td>
<td>Yes</td>
</tr>
<tr>
<td>__setCodeBreak</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>__setConditionalBreak</td>
<td>—</td>
<td>Yes</td>
</tr>
<tr>
<td>__setDataBreak</td>
<td>Yes</td>
<td>—</td>
</tr>
<tr>
<td>__setLogBreak</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>__setRangeBreak</td>
<td>—</td>
<td>Yes</td>
</tr>
<tr>
<td>__setSimBreak</td>
<td>Yes</td>
<td>—</td>
</tr>
<tr>
<td>__setTraceStartBreak</td>
<td>Yes</td>
<td>—</td>
</tr>
<tr>
<td>__setTraceStopBreak</td>
<td>Yes</td>
<td>—</td>
</tr>
<tr>
<td>__clearBreak</td>
<td>Yes</td>
<td>Yes</td>
</tr>
</tbody>
</table>

*Table 8: C-SPY macros for breakpoints*

For details of each breakpoint macro, see Reference information on C-SPY system macros, page 219.
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 210.

USEFUL BREAKPOINT HINTS

Below are some useful hints related to setting breakpoints.

Performing a task and continue 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:

```c
__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.

Using breakpoints when programming flash memory

When programming the flash memory, do not set a breakpoint on the instruction immediately following the write to flash operation. A simple work-around is to follow the write to flash operation with a NOP instruction, and set a breakpoint on the instruction following the NOP instruction.
Reference information on breakpoints

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

- Breakpoints window, page 98
- Breakpoint Usage dialog box, page 100
- Code breakpoints dialog box, page 101
- Log breakpoints dialog box, page 103
- Data breakpoints dialog box, page 104
- Immediate breakpoints dialog box, page 106
- Range breakpoints dialog box, page 107
- Conditional breakpoints dialog box, page 109
- Advanced Trigger breakpoints dialog box, page 112
- Enter Location dialog box, page 114
- Breakpoint combiner dialog box, page 115

For related information, see also:

- Reference information on C-SPY system macros, page 219
- Reference information on trace, page 146.

Breakpoints window

The Breakpoints window is available from the View menu.

![Breakpoints window](image)

*Figure 41: 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.
Using breakpoints

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:

- **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.

*Figure 42: Breakpoints window context menu*
The **Breakpoint Usage** dialog box 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 dialog box 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 and software breakpoints are not enabled, 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.
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.

![Code breakpoints dialog box](image)

*Figure 44: 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. This figure reflects the C-SPY simulator. For information about support for breakpoints in the C-SPY FET Debugger driver, see Breakpoints in the C-SPY FET Debugger driver, page 90.

**Break At**

Specify the 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 114.
Reference information on breakpoints

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:

<table>
<thead>
<tr>
<th>Action</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Auto</td>
<td>The size will be set automatically, typically to 1.</td>
</tr>
<tr>
<td>Manual</td>
<td>Specify the size of the breakpoint range in the text box</td>
</tr>
</tbody>
</table>

Action

Determines whether there is an action connected to the breakpoint. Specify an expression, for instance a C-SPY macro function, which is evaluated when the breakpoint is triggered and the condition is true.

Conditions

Specify simple or complex conditions:

<table>
<thead>
<tr>
<th>Conditions</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Expression</td>
<td>Specify a valid expression conforming to the C-SPY expression syntax.</td>
</tr>
<tr>
<td>Condition true</td>
<td>The breakpoint is triggered if the value of the expression is true.</td>
</tr>
<tr>
<td>Condition changed</td>
<td>The breakpoint is triggered if the value of the expression has changed since it was last evaluated.</td>
</tr>
<tr>
<td>Skip count</td>
<td>The number of times that the breakpoint condition must be fulfilled before a break occurs (integer).</td>
</tr>
</tbody>
</table>
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.

![Log breakpoints dialog box](image)

Figure 45: Log breakpoints dialog box

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

**Break At**

Specify the location of the breakpoint. Alternatively, click the Edit button to open the Enter Location dialog box; see Enter Location dialog box, page 114.

**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 216.

**Conditions**

Specify simple or complex conditions:

**Expression**

Specify a valid expression conforming to the C-SPY expression syntax.
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.

Break At

Specify the location for the breakpoint in the Break At text box. Alternatively, click the Edit button to open the Enter Location dialog box; see Enter Location dialog box, page 114.

Access Type

Selects the type of memory access that triggers data breakpoints:

Read/Write

Reads from or writes to location.

Read

Reads from 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. 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. Select between two different ways 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.

Action

Determines whether there is an action connected to the breakpoint. Specify an expression, for instance a C-SPY macro function, which is evaluated when the breakpoint is triggered and the condition is true.

Conditions

Specify simple or complex conditions:

Expression
Specify a valid expression conforming to the C-SPY expression syntax.

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 a break occurs (integer).
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.

Figure 47: 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.

Break At

Specify the location for the breakpoint in the Break At text box. Alternatively, click the Edit button to open the Enter Location dialog box; see Enter Location dialog box, page 114.

Access Type

Selects the type of memory access that triggers immediate breakpoints:

Read

Reads from location.

Write

Writes to location.

Action

Determines whether there is an action connected to the breakpoint. Specify an expression, for instance a C-SPY macro function, which is evaluated when the breakpoint is triggered and the condition is true.
Range breakpoints dialog box

The Range breakpoints dialog box is available from the context menu in the Breakpoints window and the Memory window.

Start value

Specify the start value for the range breakpoint in the Start value text box; an expression, an absolute address, or a source location. Alternatively, click the Edit button to open the Enter Location dialog box; see Enter Location dialog box, page 114.

Range delimiter

Sets the end location of the range. Choose the type of delimiter and specify the value in the text box:

- **End value**: The same type of value as for the Start value.
- **Length**: The length of the range in bytes (in hexadecimal notation).
- **Automatic**: Bases the range automatically on the type of expression the breakpoint is set on. For example, if you set the breakpoint on a 12-byte structure, the range of the breakpoint will be 12 bytes.
**Reference information on breakpoints**

**Type**

Selects which breakpoint type to use:

- **Address (MAB)**: Sets a breakpoint on a specified address, or anything that can be evaluated to one. The breakpoint is triggered when the specified location is accessed. If you have set the breakpoint on a specific machine instruction, the breakpoint will be triggered and the execution will stop exactly before the instruction will be executed.

- **Data (MDB)**: Sets a breakpoint on a specified value. It is the value on the data bus that triggers the breakpoint.

**Access type**

Selects the type of access that triggers the selected breakpoint:

- **Read**: Read from location.
- **Write**: Write to location.
- **Read/Write**: Read from or write to location.
- **Fetch**: At instruction fetch.

**Action**

Selects the action that occurs when the breakpoint is triggered:

- **Break**: The execution stops when the breakpoint is triggered.
- **State Storage Trigger**: Defines the breakpoint as a state storage trigger. To control the behavior of the state storage module, see *State Storage Control window*, page 174.
Using breakpoints

Action when

Selects when the action shall occur:

- **Inside range** The action occurs at an access inside the specified range.
- **Outside range** The action occurs at an access outside of the specified range.

Conditional breakpoints dialog box

The Conditional breakpoints dialog box is available from the context menu in the Breakpoints window and the Disassembly window.

![Conditional breakpoints dialog box](image)

Use the Conditional breakpoints dialog box to set a conditional breakpoint.

Conditional breakpoints are available for the C-SPY FET Debugger driver on devices that support the Enhanced Emulation Module at the required level.

**Break At**

Specify the location for the breakpoint in the Break At text box. Alternatively, click the Edit button to open the Enter Location dialog box; see Enter Location dialog box, page 114.
Reference information on breakpoints

**Type**

Selects which breakpoint type to use:

- **Address bus (MAB)**
  Sets a breakpoint on the address specified in the Break At text box, or anything that can be evaluated to one. The breakpoint is triggered when the specified location is accessed. If you have set the breakpoint on a specific machine instruction, the breakpoint will be triggered and the execution will stop exactly before the instruction will be executed.

- **Data bus (MDB)**
  Sets a breakpoint on a specified value. It is the value on the data bus that triggers the breakpoint.

- **Register**
  Sets a breakpoint on a register. Specify the register, or anything that can be evaluated to such, in the Break At text box. In the Register Value text box, type the value that should trigger the breakpoint.

**Operator**

Selects a condition operator for when the breakpoint should be triggered:

- `==` Equal to.
- `>=` Greater than or equal to.
- `<=` Less than or equal to.
- `!=` Not equal to.

**Access**

Selects the type of access that triggers the selected breakpoint:

- **Read**
  Read from location.
- **Write**
  Write to location.
- **Read/Write**
  Read from or write to location.
- **Fetch**
  At instruction fetch.
Using breakpoints

Mask

Specify a bit mask value that the breakpoint address or value will be masked with. (On the FET hardware the mask is inverted, but this is not the case in the FET Debugger driver.)

Condition

Optionally, specify an additional condition to a conditional breakpoint. This means that a conditional breakpoint can be a single data breakpoint or a combination of two breakpoints that must occur at the same time. These settings can be specified for the additional condition:

 MDB/Register Value The extra conditional data value.
 Mask The bit mask value that the breakpoint value will be masked with.
 Operator The operator of condition, either ==, >=, <=, or !=.
 Access The access type of the condition, either Read, Write, or Read/Write.

Action

Selects the action that occurs when the breakpoint is triggered:

 Break The execution stops when the breakpoint is triggered.
 State Storage Trigger State storage starts when the breakpoint is triggered. To control the behavior of the state storage module, see State Storage Control window, page 174.
Advanced Trigger breakpoints dialog box

The Advanced Trigger breakpoints dialog box is available from the context menu in the Breakpoints window and the Disassembly window.

Use the Advanced Trigger breakpoints dialog box to set an advanced trigger breakpoint.

Advanced trigger breakpoints are available for the C-SPY FET Debugger driver on devices that support the Enhanced Emulation Module at the required level.

**Break At**

Specify the location for the breakpoint in the Break At text box. Alternatively, click the Edit button to open the Enter Location dialog box; see Enter Location dialog box, page 114.

**Type**

Selects which breakpoint type to use:

- **Address bus (MAB)** Sets a breakpoint on a specified address, or anything that can be evaluated to one. The breakpoint is triggered when the specified location is accessed. If you have set the breakpoint on a specific machine instruction, the breakpoint will be triggered and the execution will stop exactly before the instruction will be executed.
Using breakpoints

**Operator**

Selects a condition operator for when the breakpoint should be triggered:

- `==` Equal to.
- `>=` Greater than or equal to.
- `<=` Less than or equal to.
- `!=` Not equal to.

**Mask**

Specify a bit mask value that the breakpoint address or value will be masked with. (On the FET hardware the mask is inverted, but this is not the case in the FET Debugger driver.)

**Value**

Specify the data value in the specified register that should be triggered.

**Access type**

Selects the type of access that triggers the selected breakpoint.

**Action**

Selects the action that occurs when the breakpoint is triggered:

- **Break** The execution stops when the breakpoint is triggered.
- **State Storage Trigger** State storage starts when the breakpoint is triggered. To control the behavior of the state storage module, see State Storage Control window, page 174.
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 51: 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:

Expression

Any expression that evaluates to a valid address, such as a function or variable name.

Code breakpoints are set on functions and data breakpoints are set on variable names. For example, my_var refers to the location of the variable my_var, and arr[3] refers to the third element of the array arr.

Absolute address

An absolute location on the form zone:hexaddress or simply hexaddress. Zone refers to C-SPY memory zones and specifies in which memory the address belongs. For example Memory:0x42.
Using breakpoints

Breakpoint combiner dialog box

The Breakpoint combiner dialog box is available from the Emulator menu.

Source location

A location in the C source code using the syntax:

\{file\_path\}.row.column.

*file\_path* 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:\my\_projects\Utilities.c\}.22.3 sets a breakpoint on the third character position on line 22 in the source file *Utilities.c*.

Note: Only available for devices that support the Enhanced Emulation Module at the required level. The settings are not saved when the debug session is closed.

Use this dialog box to combine two already defined breakpoints.

Select a breakpoint and right-click to display a list of breakpoints to combine it with. When two breakpoints have been combined, the defined action will not occur until both breakpoints have been reached.

Note: Only available for devices that support the Enhanced Emulation Module at the required level. The settings are not saved when the debug session is closed.
Resolve Source Ambiguity dialog box

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

![Figure 53: Resolve Source Ambiguity dialog box](Figure)

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**.
- Click **Cancel**.

**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.
Monitoring 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
- 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.
Introduction to monitoring memory and registers

- 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 several 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, it is inconvenient to list all registers concurrently in the Register window. Instead the registers are divided into logical 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 different register groups.

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 MSP430 architecture has one zone, Memory, which covers the whole MSP430.
memory range. If you load a device description file, additional zones that adhere better to the specific device memory layout are defined.

Figure 54: Zones in C-SPY

Memory zones are used in several contexts, most importantly in the Memory and Disassembly windows. Use the Zone box in these windows to choose which memory zone to display.

Device-specific zones

Memory information for device-specific zones are defined in the device description files. By default, there is only one zone in the debugger, Memory. If you load a device description file, additional zones that adhere to the specific memory layout are defined.

These zones are available, depending on the device description file you are using: SFR, RAM, INFO, and Flash.

If your hardware does not have the same memory layout as any of the predefined device description files, you can define customized zones by adding them to the file.

For further information, see Selecting a device description file, page 37 and Modifying a device description file, page 40.

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
Introduction to monitoring memory and registers

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

**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. To read more about this, see the IAR C/C++ Compiler Reference Guide for MSP430.

**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.
Reference information on memory and registers

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

- Memory window, page 123
- Memory Save dialog box, page 127
- Memory Restore dialog box, page 128
- Fill dialog box, page 128
- Symbolic Memory window, page 130
- Stack window, page 132
- Register window, page 135
- Memory Access Setup dialog box, page 137
- Edit Memory Access dialog box, page 139
- Memory Dump dialog box, page 140.

Memory window

The Memory window is available from the View menu.

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.

**Toolbar**

The toolbar contains:

- **Go to**
  The location you want to view. This can be a memory address, or the name of a variable, function, or label.

- **Zone display**
  Selects a memory zone to display; see *C-SPY memory zones*, page 120.

- **Context menu button**
  Displays the context menu, see *Context menu*, page 125.

- **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.
Monitoring memory and registers

**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 56: Memory window context menu](image)

These commands are available:

- **Copy, Paste**
  - Standard editing commands.

- **Zone**
  - Selects a memory zone to display; see *C-SPY memory zones*, page 120.

- **1x Units**
  - Displays the memory contents in units of 8 bits.

- **2x Units**
  - Displays the memory contents in units of 16 bits.

- **4x Units**
  - Displays the memory contents in units of 32 bits.

- **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 128.

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 127.

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 128.

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 95.
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.

![Memory Save dialog box](image)

Use this dialog box to save the contents of a specified memory area to a file.

**Zone**
- Selects a memory zone.

**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 msp430-txt 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.

Zone

Selects a memory zone.

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.

Use this dialog box to fill a specified area of memory with a value.
Monitoring memory and registers

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.

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 when the debugger is running.

![Symbolic Memory window](image)

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.

### Toolbar

The toolbar contains:

- **Go to**: The memory location or symbol you want to view.
- **Zone display**: Selects a memory zone to display; *C-SPY memory zones*, page 120.
- **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:

![Context menu](image)

**Figure 61: 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 in units of 8 bits. This applies only to rows which do not contain a variable.
- **2x Units**: Displays the memory contents in units of 16 bits.
Stack window

The Stack window is available from the View menu.

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 several 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 91.

For information about options specific to the Stack window, see the IDE Project Management and Building Guide.
Monitoring memory and registers

Toolbar

Stack
Selects which stack to view. This applies to microcontrollers 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:

<table>
<thead>
<tr>
<th>Command</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Show Variables</td>
<td>Displays separate columns named <strong>Variables</strong>, <strong>Value</strong>, and <strong>Frame</strong> in the Stack window. Variables located at memory addresses listed in the Stack window are displayed in these columns.</td>
</tr>
<tr>
<td>Show Offsets</td>
<td>Displays locations in the <strong>Location</strong> column as offsets from the stack pointer. When deselected, locations are displayed as absolute addresses.</td>
</tr>
<tr>
<td>1x Units</td>
<td>Displays data in the <strong>Data</strong> column as single bytes.</td>
</tr>
<tr>
<td>2x Units</td>
<td>Displays data in the <strong>Data</strong> column as 2-byte groups.</td>
</tr>
<tr>
<td>4x Units</td>
<td>Displays data in the <strong>Data</strong> column as 4-byte groups.</td>
</tr>
<tr>
<td>Options</td>
<td>Opens the IDE Options dialog box where you can set options specific to the Stack window, see the IDE Project Management and Building Guide.</td>
</tr>
</tbody>
</table>

*Figure 63: Stack window context menu*
Register window

The Register window is available from the View menu.

![Register window](image)

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 37.

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, read about register filter options in the IDE Project Management and Building Guide.

**Toolbar**

**CPU Registers**

Selects which register group to display, by default CPU Registers. Additional registers are defined in a specific register definition file—with the filename extension `.sfr`—which is included from the register section of the device description file. These registers are the device-specific memory-mapped control and status registers for the peripheral units on the MSP430 microcontrollers.
**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**.

**Note:** When the FET Debugger is used, the cycle counter registers can only be used while single-stepping in the Disassembly window.

<table>
<thead>
<tr>
<th>Register</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>CYCLOCOUNTER</td>
<td>Cycle counter, cleared at reset.</td>
</tr>
<tr>
<td>CCTIMER1, CCTIMER2</td>
<td>Trip counters, cleared at reset. These registers can be edited.</td>
</tr>
<tr>
<td>CCSTEP</td>
<td>The number of cycles in the last step.</td>
</tr>
</tbody>
</table>

*Table 9: Cycle counter registers*
Memory Access Setup dialog box

The Memory Access Setup dialog box is available from the Simulator menu.

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 139.

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.
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 details, see **Edit Memory Access dialog box**, page 139.

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 139.

**Edit**

Opens the **Edit Memory Access** dialog box, where you can edit the selected memory area. See **Edit Memory Access dialog box**, page 139.

**Delete**

Deletes the selected memory area definition.

**Delete All**

Deletes all defined memory area definitions.

**Note**: 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.

Use this dialog box to specify the memory ranges, and assign an access type to each memory range, for which you want to detect illegal accesses during the simulation.

Memory range

Defines the memory area for which you want to check the memory accesses:

- **Zone**: Selects a memory zone; see C-SPY memory zones, page 120.
- **Start address**: Specify the start address for the address range, in hexadecimal notation.
- **End address**: Specify the end address for the address range, in hexadecimal notation.

Access type

Selects an access type to the memory range; choose between:

- Read and write
- Read only
- Write only.
Memory Dump dialog box

The Memory Dump dialog box is available from the Emulator menu.

![Memory Dump dialog box](image)

Use this dialog box to write memory contents to a file.

**Dump File Name**

Specify the name of the destination file where the contents of the selected part of the memory will be saved in text format. You can find the file using the Browse button.

**Start Address**

Specify the start address for the memory section you want to save to a file.

**Dump Length**

Specify the length of the memory section you want to save to a file, in hexadecimal notation. The maximum length is FFFF, which limits the number of bytes that can be written to 65535. Consequently, you cannot write memory from address 0x0000 to 0xFFFF, inclusive, as this would require a length specifier of 65536 bytes (0x10000).

**Add address information**

Adds address information to the file.
Append register contents

Appends register contents to the file. Choose between:

- Program Counter (R0)
- Stack Pointer (R1)
- Status Register (R2)
- Register R4 to R15

Format

Selects format for the text that is written to the file. Choose between:

- Word
- Byte
Collecting and using trace data

This chapter gives you information about collecting and using trace data in C-SPY®. More specifically, this means:

- Introduction to using trace
- Procedures for using trace
- 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.

For related information, see also:

- Using the profiler, page 159
- Using state storage, page 171.

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

Trace data is a continuously collected sequence of every executed instruction for a selected portion of the execution. Once generated, C-SPY can collect the data and you can visualize and analyze it in various windows and dialog boxes.
Procedures for using trace

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 to control the collection of trace data.

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

To use trace-related functionality in C-SPY, you must use the C-SPY simulator. Trace data cannot be collected from the FET Debugger.

Procedures for using trace

This section gives you step-by-step descriptions about how to collect and use 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

To collect trace data using the C-SPY simulator, no specific build settings are required.

1. After you have built your application and started C-SPY, choose Simulator>Trace to open the Trace window, and click the Activate button to enable collecting trace data.

2. 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 146.

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.
Collecting and using trace data

The C-SPY system macros __setTraceStartBreak and __setTraceStopBreak can also be used.

For details about these breakpoints, see Trace Start breakpoints dialog box, page 152 and Trace Stop breakpoints dialog box, page 153, 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.

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, follow these steps:

1. In 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 detailed information about the different options, see Find in Trace dialog box, page 155.

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 detailed reference information, see Find in Trace window, page 156.

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 window*, page 146
- *Function Trace window*, page 148
- *Timeline window*, page 149
- *Trace Start breakpoints dialog box*, page 152
- *Trace Stop breakpoints dialog box*, page 153
- *Trace Expressions window*, page 154
- *Find in Trace window*, page 156
- *Find in Trace dialog box*, page 155.

Trace window

The Trace window is available from the *Simulator* menu when you are using the C-SPY simulator.

![Figure 68: The Trace window](image-url)
This window displays a collected sequence of executed machine instructions. In addition, the window can display trace data for expressions.

**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 145.

- **Find**
  Displays a dialog box where you can perform a search; see *Find in Trace dialog box*, page 155.

- **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.

- **Edit Expressions**
  (C-SPY simulator only)
  Opens the Trace Expressions window; see *Trace Expressions window*, page 154.

**Display area**

This area contains these columns:

- **#**
  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 154.

Function Trace window

The Function Trace window is available from the Simulator menu when you are using the C-SPY simulator.

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.

Toolbar

For information about the toolbar, see Trace toolbar, page 147.

Display area

For information about the columns in the display area, see:

- Display area, page 147
Timeline window

The Timeline window is available from the Simulator menu during a debug session.

This window displays trace data—for interrupt logs and for the call stack—as graphs in relation to a common time axis.

Display area

The display area can be populated with different graphs:

- Interrupt Log Graph
- Call Stack Graph

At the bottom of the window, there is a common time axis that uses seconds as the time unit.

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, 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. This graph is a graphical representation of the information in the Interrupt Log window, see Interrupt Log window, page 203.
Call Stack Graph

The Call Stack Graph displays the sequence of calls and returns collected by trace. 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.

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:

```
Navigate
  Auto Scroll
  Zoom

Interrupts
  Disable
  Go To Source

Select Graphs
  Time-Roll Window
```

Figure 71: Timeline window context menu

Note: The context menu contains some commands that are common to all graphs and some commands that are specific to each graph.
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 recent collected data is automatically displayed.

**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.

**Call Stack**  Call Stack Graph  A heading that shows that the Call stack-specific commands below are available.
Reference information on trace

The Trace Start dialog box is available from the context menu that appears when you right-click in the Breakpoints window.

![ Trace Start breakpoints dialog box ](image)

This dialog box is available for the C-SPY simulator.

**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.
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.

**Trigger At**

Specify the location for the breakpoint in the text box. Alternatively, click the Edit browse button to open the Enter Location dialog box; see Enter Location dialog box, page 114.

**Trace Stop breakpoints**

The Trace Stop dialog box is available from the context menu that appears when you right-click in the Breakpoints window.

![Trace Stop breakpoints dialog box](image)

This dialog box is available for the C-SPY simulator.

**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.

**Trigger At**

Specify the location for the breakpoint in the text box. Alternatively, click the **Edit** browse button to open the **Enter Location** dialog box; see **Enter Location dialog box**, page 114.

**Trace Expressions window**

The Trace Expressions window is available from the Trace window toolbar in the C-SPY simulator.

![Figure 74: Trace Expressions window](image)

Use this window to specify, for example, a specific variable (or an expression) for which you want to collect trace data.

**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.

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 156.

See also Searching in trace data, page 145.
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 specifying `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.

![Figure 76: Find in Trace window](image)

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 155.

See also, Searching in trace data, page 145.
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.
Reference information on trace
Using the profiler

This chapter describes how to use the profiler in C-SPY®. More specifically, this means:

- Introduction to the profiler
- Procedures for 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 the memory which uses the most efficient addressing mode. For detailed information about efficient memory usage, see the IAR C/C++ Compiler Reference Guide for MSP430.

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 hardware 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 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, and there are no specific requirements for using the profiler.

Profiling is not available for the C-SPY FET Debugger.

---

**Procedures for using the profiler**

This section gives you step-by-step descriptions about how to use 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, follow these steps:

1. Make sure you build your application using these options:

<table>
<thead>
<tr>
<th>Category</th>
<th>Setting</th>
</tr>
</thead>
<tbody>
<tr>
<td>C/C++ Compiler</td>
<td>Output&gt;Generate debug information</td>
</tr>
<tr>
<td>Linker</td>
<td>Output&gt;Format&gt;Debug information for C-SPY</td>
</tr>
</tbody>
</table>

   *Table 10: Project options for enabling the profiler*

2. When you have built your application and started C-SPY, choose **Simulator>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, follow these steps:

1. Build your application and start C-SPY.

2. Choose **View>Disassembly** to open the Disassembly window, and choose **Enable** from the context menu that is available when you right-click in the Profiler window.

3. Make sure that the **Show** command on the context menu is selected, to display the profiling information.

4. Start executing your application to collect the profiling information.
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 Disassembly window.

![Figure 77: Instruction count in Disassembly window](image)

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 163
- *Disassembly window*, page 58.
Function Profiler window

The Function Profiler window is available from the Simulator menu.

![Function Profiler window](image)

This window displays function profiling information.

**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 included 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.

More specifically, the display area provides information in these columns:

<table>
<thead>
<tr>
<th>Column</th>
<th>Source</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Function</td>
<td>All sources</td>
<td>The name of the profiled C function.</td>
</tr>
<tr>
<td>Calls</td>
<td>Trace (calls)</td>
<td>The number of times the function has been called.</td>
</tr>
<tr>
<td>Flat time</td>
<td>Trace (calls)</td>
<td>The time in cycles spent inside the function.</td>
</tr>
<tr>
<td>Flat time (%)</td>
<td>Trace (calls)</td>
<td>Flat time in cycles expressed as a percentage of the total time.</td>
</tr>
<tr>
<td>Acc. time</td>
<td>Trace (calls)</td>
<td>The time in cycles spent in this function and everything called by this function.</td>
</tr>
<tr>
<td>Acc. time (%)</td>
<td>Trace (calls)</td>
<td>Accumulated time in cycles expressed as a percentage of the total time.</td>
</tr>
<tr>
<td>PC Samples</td>
<td>Trace (flat)</td>
<td>The number of PC samples associated with the function.</td>
</tr>
<tr>
<td>PC Samples (%)</td>
<td>Trace (flat)</td>
<td>The number of PC samples associated with the function as a percentage of the total number of samples.</td>
</tr>
</tbody>
</table>
Context menu

This context menu is available:

<table>
<thead>
<tr>
<th>Command</th>
</tr>
</thead>
<tbody>
<tr>
<td>Enable</td>
</tr>
<tr>
<td>Clear</td>
</tr>
<tr>
<td>Source*</td>
</tr>
<tr>
<td>Source: Trace (calls)</td>
</tr>
<tr>
<td>Source: Trace (flat)</td>
</tr>
</tbody>
</table>

Figure 79: Function Profiler window context menu

These commands are available:

- **Enable**: Enables the profiler. The system will collect information also when the window is closed.
- **Clear**: Clears all profiling data.
- **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.
Reference information on the profiler
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 only 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 168.

For related information, see Single stepping, page 52.

Code Coverage window

The Code Coverage window is available from the View menu.

![Code Coverage window](image)

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:

Before using the code coverage functionality you must build your application using these options:

<table>
<thead>
<tr>
<th>Category</th>
<th>Setting</th>
</tr>
</thead>
<tbody>
<tr>
<td>C/C++ Compiler</td>
<td>Output&gt;Generate debug information</td>
</tr>
<tr>
<td>Linker</td>
<td>Format&gt;Debug information for C-SPY</td>
</tr>
</tbody>
</table>

Table 11: Project options for enabling code coverage
After you have built your application and started C-SPY, choose View>Code Coverage to open the Code Coverage window.

Click the Activate button, alternatively choose Activate from the context menu, to switch on code coverage.

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.

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.
Reference information on code coverage

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:

[Image of context menu]

<table>
<thead>
<tr>
<th>Command</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Activate</strong></td>
<td>Switches code coverage on and off during execution.</td>
</tr>
<tr>
<td><strong>Clear</strong></td>
<td>Clears the code coverage information. All step points are marked as not executed.</td>
</tr>
<tr>
<td><strong>Refresh</strong></td>
<td>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.</td>
</tr>
<tr>
<td><strong>Auto-refresh</strong></td>
<td>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.</td>
</tr>
<tr>
<td><strong>Save As</strong></td>
<td>Saves the current code coverage result in a text file.</td>
</tr>
<tr>
<td><strong>Save session</strong></td>
<td>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.</td>
</tr>
<tr>
<td><strong>Restore session</strong></td>
<td>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.</td>
</tr>
</tbody>
</table>
Using state storage

This chapter describes how to use the state storage module. More specifically, this means:

- Introduction to state storage
- Procedures for using state storage
- Reference information on state storage.

Introduction to state storage

This section introduces the state storage module, a limited variant of a traditional trace module available for the C-SPY FET Debugger driver.

These topics are covered:

- Reasons for using state storage
- Briefly about state storage
- Requirements.

For related information, see also:

- Using the sequencer, page 177.

REASONS FOR USING STATE STORAGE

State storage allows you to examine the last eight states or instructions that were executed before a specific point was reached or a specific event occurred, or the next eight states that will be executed after a specific point is reached or a specific event occurs.

BRIEFLY ABOUT STATE STORAGE

The state storage module is a limited variant of a traditional trace module. It can store eight states and can be used for monitoring program states or program flow, without interfering with the execution. It uses a built-in buffer to store MAB, MDB, and CPU control signal information.
REQUIREMENTS
The state storage module is only available if you are using the C-SPY FET Debugger driver and a device that supports the Enhanced Emulation Module at the required level.

Procedures for using state storage
This section gives you step-by-step descriptions about how to use certain features related to state storage.

More specifically, you will get information about:
- Setting up state storage.

SETTING UP STATE STORAGE
To use the state storage module, you must:
1 Define one or several range breakpoints or conditional breakpoints.
2 In the breakpoints dialog box, select the action State Storage Trigger for these breakpoints. This means that the breakpoint is defined as a state storage trigger. (State storage can also be triggered from the Sequencer Control window.)
   Note: Depending on the behavior you want when the state storage module is triggered, it is useful to consider the combination of the Action options and the options available in the State Storage Control window. See the examples following immediately after these steps.
3 Choose Emulator>State Storage Control to open the State Storage Control window.
4 Select the option Enable state storage. Set the options Buffer wrap around, Trigger action, and Storage action according to your preferences.
   In the list State Storage Triggers, all breakpoints defined as state storage triggers are listed.
   For more details about the options, see State Storage Control window, page 174.
5 Click Apply.
6 Choose Emulator>State Storage window to open the State Storage window.
7 Choose Debug>Go to execute your application. Before you can view the state storage information, you must stop the execution. You can do this, for instance, by using the Break command.
   For information about the window contents, see State Storage window, page 175.
Example
As an example, assume the following setup:

- There is a conditional breakpoint which has both of the action options selected—Break and State Storage Trigger
- The state storage options Instruction fetch and Buffer wrap around are selected in the State Storage Control window.

This will start the state storage immediately when you start executing your application. When the breakpoint is triggered, the execution will stop and the last eight states can be inspected in the State Storage window.

However, if you do not want the state storage module to start until a specific state is reached, you would usually not want the execution to stop, because no state information has been stored yet.

In this case, select the State Storage Trigger action in the Conditional breakpoints dialog box (making sure that Break is deselected), and deselect the option Buffer wrap around in the State Storage Control window.

When the breakpoint is triggered, the execution will not stop, but the state storage will start. Because the option Buffer wrap around is deselected, you have ensured that the data in the buffer will not be overwritten.

When another breakpoint (which has Break selected) is triggered, or if you stop the execution by clicking the Break button, the State Storage window will show eight states starting with the breakpoint that was used for starting the state storage module.

Reference information on state storage
This section gives reference information about these windows:

- State Storage Control window, page 174
- State Storage window, page 175.
State Storage Control window

The State Storage Control window is available from the **Emulator** menu.

![State Storage Control window](image.png)

Use this window to define how to use the state storage module. The window is only available for devices that support the Enhanced Emulation Module at the required level.

**Enable state storage**
Enables the state storage module.

**Buffer wrap around**
Determines whether the state storage buffer should wrap around. If you select this option, the state storage buffer is continuously overwritten until the execution is stopped or a breakpoint is triggered. Only the eight last states are stored.

Alternatively, in order not to overwrite the information in the state storage buffer, deselect this option. To guarantee that the eight first states will be stored, you should also click **Reset**.

**Reset**
Resets the state storage module.

**Trigger action**
Selects which action to take when breakpoints defined as state storage triggers are triggered:

- **Start on trigger** Starts state storage when the breakpoint is triggered.

![Figure 82: State Storage Control window](image.png)
Using state storage

Storage action on

Selects when the state information should be collected:

- **Stop on trigger**
  Starts state storage immediately when execution starts. State storage stops when the breakpoint is triggered.

- **None**
  Starts state storage immediately when execution starts. State storage does not stop when the breakpoint is triggered. However, if execution stops, state storage also stops but it will resume when execution resumes.

**State storage triggers**

Lists all the breakpoints that are defined as state storage triggers. That is, the breakpoints that have the action State Storage Trigger selected.

**State Storage window**

The State Storage window is available from the Emulator menu.

<table>
<thead>
<tr>
<th>Address</th>
<th>Instruction</th>
<th>SP</th>
<th>PC</th>
<th>Control Signals</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x1000</td>
<td>31h A001</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>0x1004</td>
<td>80112111</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>0x0600</td>
<td>*****</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>0x0600</td>
<td>*****</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>0x0600</td>
<td>*****</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>0x0600</td>
<td>*****</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Figure 83: State storage window

This window displays state storage information for eight states. Invalid data is displayed in red color.
Reference information on state storage

Toolbar

The toolbar contains:

- **Update**: Refreshes the data in the State Storage window, alternatively appends new data.
- **Automatic update**: Updates the data in the state storage window automatically each time new data is available in the state storage buffer.
- **Automatic restart**: Resets the state storage module for consecutive data readouts after each readout.
- **Append data**: Appends collected data from the state storage buffer to the data that is already present in the State Storage window. The new data is added below the data that is already present.

Display area

This area contains these columns:

- **Address bus**: The stored value of the address bus.
- **Instruction**: The instruction.
- **Mnemonic**: The mnemonic.
- **Data bus**: The stored content of the data bus.
- **Control signals (byte)**: The stored value of the control signals during storage.
  - Bit 1: Instruction fetch
  - Bit 2: Byte/Word
  - Bit 3: Interrupt request
  - Bit 4: CPU off
  - Bit 5: The value of the Power Up Clear (PUC) signal
  - Bit 6: ZERO|HALT (which one depends on the used device)
  - Bit 7: Break trigger
- **Control signals (bits)**: Each bit in the stored value of the control signals during storage.
Using the sequencer

This chapter describes the sequencer and how to take advantage of it. More specifically, this means:

- Introduction to the sequencer
- Procedures for using the sequencer
- Reference information on the sequencer.

Introduction to the sequencer

This section introduces the sequencer module.

These topics are covered:

- Reasons for using the sequencer, page 177
- Briefly about the sequencer, page 177
- Requirements for using the sequencer, page 178.

For related information, see also:

- Using state storage, page 171.

REASONS FOR USING THE SEQUENCER

The sequencer is useful if you, for instance, want to stop the execution or start the state storage module under certain conditions, for instance a specific program flow. If you combine this with letting the state storage module continuously store information, you will have useful state information logged in the State Storage window when the execution stops.

BRIEFLY ABOUT THE SEQUENCER

The sequencer module is a simple state machine that lets you break the execution or trigger the state storage module using a more complex method than a standard breakpoint.

In a simple setup, you can define three transition triggers, where the last one triggers an action. In an advanced setup, the state machine can have four states (0-3). State 0 is the starting state, and state 3 is the state that triggers a breakpoint (the action state).
Procedures for using the sequencer

Breakpoint can be designed either to stop execution, or to trigger the state storage module.

REQUIREMENTS FOR USING THE SEQUENCER

The sequencer module is only available for the C-SPY FET Debugger driver and if you are using a device that supports the Enhanced Emulation Module at the required level.

Procedures for using the sequencer

This section gives you step-by-step descriptions about how to use certain features of the sequencer module.

More specifically, you will get information about:

- Setting up the sequencer (simple setup)
- Setting up the sequencer (advanced setup)
- Using the sequencer to locate a problem.

SETTING UP THE SEQUENCER (SIMPLE SETUP)

In a simple setup, you can define three transition triggers, where the last one triggers an action.

To define a simple sequencer setup:

1. Choose Emulator>Sequencer Control to open the Sequencer Control window.
2. Select the option Enable Sequencer.
3. Use the Transition trigger drop-down lists to define three breakpoints, where the last breakpoint should act as a transition trigger.

SETTING UP THE SEQUENCER (ADVANCED SETUP)

In an advanced setup, the state machine can have four states (0-3). State 0 is the starting state, and state 3 is the state that triggers a breakpoint (the action state). This breakpoint can be designed either to stop execution, or to trigger the state storage module.

To define an advanced sequencer setup:

1. Choose Emulator>Sequencer Control to open the Sequencer Control window.
2. Select the option Enable Sequencer.
3. Click the Advanced button. This will let you define 4 states (0-3) with two transition triggers each (a and b).
From the eight available hardware breakpoints (0-7) of the device, breakpoint number 7 will be reserved for state 3. The Transition trigger drop-down lists let you define one breakpoint each, where the breakpoint should act as a transition trigger.

4 For each transition trigger, define which state should be the next state after the transition.

Use the following options:

State Storage Trigger Select a breakpoint from the drop-down list to move the state machine from one state to another. Note: to do this you must first define the required conditional breakpoints.

Next state Select a state to define which state should be the next state after the transition.

5 Select an action to determine the result of the final transition trigger. If you select the option Break, the execution will stop. If you select the option State Storage Trigger, the state storage module will be triggered.

**USING THE SEQUENCER TO LOCATE A PROBLEM**

Consider this example:

```c
void my_putchar(char c)
{
    ... /* Code suspected to be erroneous */
    ... 
}

void my_function(void)
{
    ...
    my_putchar('a');
    ...
    my_putchar('x');
    ...
    my_putchar('@');
    ...
}
```

In this example, the customized putchar function my_putchar has for some reason a problem with special characters. To locate the problem, it might be useful to stop execution when the function is called, but only when it is called with one of the problematic characters as the argument.
To locate the problem:

1. Set a hardware breakpoint on the statement `my_putchar('@');`.

2. Set another breakpoint on the suspected code within the function `my_putchar()`.

3. Define these breakpoints as transition triggers. Choose Emulator>Sequencer Control to open the Sequencer Control window. Select the option Enable sequencer.

4. In this simple example you will only need two transition triggers. Make sure the following options are selected:

<table>
<thead>
<tr>
<th>Option</th>
<th>Setting</th>
</tr>
</thead>
<tbody>
<tr>
<td>Transition trigger 0</td>
<td>The breakpoint which is set on the function call <code>my_putchar('@');</code></td>
</tr>
<tr>
<td>Transition trigger 1</td>
<td>The breakpoint which is set on the suspected code within the function <code>my_putchar()</code></td>
</tr>
<tr>
<td>Action</td>
<td>Break</td>
</tr>
</tbody>
</table>

5. Set up the state storage module. Choose Emulator>State Storage Control to open the State Storage Control window. Make sure the following options are selected:

<table>
<thead>
<tr>
<th>Option</th>
<th>Setting</th>
</tr>
</thead>
<tbody>
<tr>
<td>Enable state storage</td>
<td>Selected</td>
</tr>
<tr>
<td>Buffer wrap around</td>
<td>Selected</td>
</tr>
<tr>
<td>Storage action</td>
<td>Instruction fetch</td>
</tr>
<tr>
<td>Trigger action</td>
<td>None</td>
</tr>
</tbody>
</table>

6. Start the program execution. The state storage module will continuously store trace information. Execution stops when the function `my_putchar()` has been called by the function call `my_putchar('@');`.

7. Choose Emulator>State Storage Window to open the State Storage window. You can now examine the stored trace information. For more information, see State Storage window, page 175.
When the sequencer is in state 3, C-SPY’s breakpoint mechanism—which is used for all breakpoints, not only transition triggers—can be locked. Therefore, you should always end the session with one of these steps:

- Disabling the sequencer module. This will restore all breakpoint actions.
- Resetting the state machine by clicking the Reset States button. The sequencer will still be active and trigger on the defined setup during the program execution.

Reference information on the sequencer

This section gives reference information about the Sequencer Control window.

Sequencer Control window

The Sequencer Control window is available from the Emulator menu.

Use this window to break the execution or trigger the state storage module, using a more complex method than a standard breakpoint. The window is only available for devices that support the Enhanced Emulation Module at the required level.

For related information about state storage, see Using state storage, page 171.

Enable Sequencer

Enables the sequencer.
Reference information on the sequencer

Action

Controls the result of the final transition trigger:

- Break: Stops the execution.
- State Storage Trigger: Triggers to move the state machine from one state to another and activates the state storage module. Requires that you select a conditional breakpoint, that you have defined, from the Transition trigger drop-down list.

Current state

Displays the current state of the state machine.

Reset Trigger

Selects a trigger that will reset the state machine.

Reset States

Sets the state machine to state 0.

Advanced

Displays the advanced setup options. This will let you define 4 states (numbered 0–3) with two transition triggers each (a and b). For each transition trigger, you can define which state should be the next state after the transition.

State 0–3

Controls the transition triggers and the state that follows the transitions.

- Transition trigger a: Selects a breakpoint to act as a transition trigger.
- Transition trigger b: Selects a breakpoint to act as a transition trigger.
- Next state: Select the next state after the transition.

For state 3—the final transition trigger—you must also define an Action: Break or State Storage Trigger.
Using the cycle counter for 5xx devices

This chapter describes the cycle counter for MSP430F5xx devices. More specifically, this means:

- Introduction to the cycle counter for 5xx
- Procedures for using the cycle counter applications
- Reference information on the cycle counter for 5xx.

Introduction to the cycle counter for 5xx

This section introduces Cycle counter 1, a second cycle counter for MSP430F5xx devices in addition to the cycle counter displayed in the Register window.

These topics are covered:

- Reasons for using the cycle counter for 5xx
- Briefly about the cycle counter for 5xx
- Requirements for using the cycle counter for 5xx.

For related information, see also:

- Register window, page 135.

REASONS FOR USING THE CYCLE COUNTER FOR 5XX

The cycle counter for MSP430F5xx devices can help you, for example, to measure the DMA load, to profile a part of your application, or to measure how long some tasks take.

BRIEFLY ABOUT THE CYCLE COUNTER FOR 5XX

The cycle counter provides one or two 40-bit counters to count the number of cycles used by the CPU to execute certain tasks. On some devices, the cycle counter operation can be controlled using triggers. This allows, for example, conditional profiling, such as profiling a specific section of code.

The Cycle Counter 5xx Control window contains preconfigured applications, as well as the opportunity to customize your own use of the cycle counter.
REQUIREMENTS FOR USING THE CYCLE COUNTER FOR 5XX

The cycle counter for 5xx is only available if you are using the C-SPY FET Debugger driver and an MSP430F5xx device that supports the Enhanced Emulation Module at the required level.

Procedures for using the cycle counter applications

This section gives you descriptions of the Cycle counter 1 applications available in the Cycle Counter 5xx Control window; that is, step-by-step descriptions of how to use the extra cycle counter. More specifically, you will get information about:

- Counting all CPU cycles
- Measuring the DMA load versus the CPU load
- Profiling a specific part of your application
- Measuring the Trigger hits
- Measuring the number of CPU cycles for a task.

COUNTING ALL CPU CYCLES

The Cycle counter application in the Cycle Counter 5xx Control window makes Cycle counter 1 behave like an ordinary cycle counter, counting all CPU cycles.

To count all CPU cycles:

1. Choose Emulator>Cycle Counter for 5xx to open the Cycle Counter 5xx Control window.

2. Select the Cycle counter option.

3. Click the Reset Counter 1 button to reset Cycle counter 1.

4. Execute your program and then stop the execution.

Cycle counter 1 in the Cycle Counter Values area now shows the number of CPU cycles that were executed.
MEASURING THE DMA LOAD VERSUS THE CPU LOAD

The DMA load vs. CPU load application in the Cycle Counter 5xx Control window measures the DMA load versus the CPU load by comparing the number of DMA bus cycles with the total number of bus cycles.

To measure the DMA load versus the CPU load:
1 Choose Emulator>Cycle Counter for 5xx to open the Cycle Counter 5xx Control window.
2 Select the DMA load vs. CPU load option.
3 Click the Reset Counter 1 button to reset Cycle counter 1.
4 Execute your program and then stop the execution.

In the Cycle Counter Values area, Cycle counter 1 now shows the number of DMA bus cycles, which can be compared with the number of CPU cycles shown by cycle counter 0.

PROFILING A SPECIFIC PART OF YOUR APPLICATION

The Profiling application in the Cycle Counter 5xx Control window lets you profile a specific part of your program. Two reaction triggers or breakpoints define the start and stop points for the cycle counter. Cycle counter 1 starts to count cycles at the first trigger point and stops counting cycles at the second trigger.

Note: You cannot define which trigger point should be used as the start point or stop point; the first of the two that is reached will start the cycle counter. The execution does not stop at any of the triggers. You can compare the amount of time spent in a specific function with the result for cycle counter 0, which counts all cycles for the entire application.

To profile a section of your application:
1 Set up an advanced trigger at the start point of the code section that you want to measure. For information about setting advanced trigger breakpoints, see Advanced Trigger breakpoints dialog box, page 112.
2 Set up a second advanced trigger at the stop point of the code section that you want to measure.
3 Choose Emulator>Cycle Counter for 5xx to open the Cycle Counter 5xx Control window.
4 Select the Profiling option.
5 Select the start trigger point from the drop-down list Reaction trigger 1.
6 Select the stop trigger point from the drop-down list Reaction trigger 2.
Procedures for using the cycle counter applications

7 Click the **Reset Counter 1** button to reset Cycle counter 1.

8 Execute your program and then stop the execution.

In the **Cycle Counter Values** area, Cycle counter 1 shows the number of CPU cycles spent in the selected code section or function. Cycle counter 0 shows all counted CPU cycles.

**MEASURING THE TRIGGER HITS**

The **Trigger hits** application in the Cycle Counter 5xx Control window measures the number of times a certain point in your program has been reached. Cycle counter 1 counts the number of times a trigger point has been triggered.

**To measure how many times a point in your application is reached:**

1 Set up an advanced trigger at the trigger point that you want to measure. For information about setting advanced trigger breakpoints, see *Advanced Trigger breakpoints dialog box*, page 112.

2 Choose **Emulator>Cycle Counter for 5xx** to open the Cycle Counter 5xx Control window.

3 Select the **Trigger hits** option.

4 Select the trigger point from the drop-down list **Reaction trigger 1**.

5 Click the **Reset Counter 1** button to reset Cycle counter 1.

6 Execute your program and then stop the execution.

In the **Cycle Counter Values** area, Cycle counter 1 now shows the number of times the trigger point has been triggered.

**MEASURING THE NUMBER OF CPU CYCLES FOR A TASK**

The **Trip counter** application in the Cycle Counter 5xx Control window measures the number of CPU cycles required to execute a certain task or function. Cycle counter 1 starts to count cycles at the start trigger and stops counting cycles at the stop trigger. The execution stops when the stop trigger is reached.

**To measure the number of cycles required to execute a task or function:**

1 Set up an advanced trigger at the start point of the code section that you want to measure. For information about setting advanced trigger breakpoints, see *Advanced Trigger breakpoints dialog box*, page 112.

2 Set up a second advanced trigger at the stop point of the code section that you want to measure.
3 Choose Emulator>Cycle Counter for 5xx to open the Cycle Counter 5xx Control window.

4 Select the Trip counter option.

5 Select the start trigger point from the drop-down list Reaction trigger 1.

6 Select the stop trigger point from the drop-down list Reaction trigger 2.

7 Click the Reset Counter 1 button to reset Cycle counter 1.

8 Execute your program. The execution stops when the stop trigger is reached.

In the Cycle Counter Values area, Cycle counter 1 now shows the number of CPU cycles required to execute the task or function.

Reference information on the cycle counter for 5xx

This section gives reference information about the Cycle Counter 5xx Control window.

Cycle Counter 5xx Control window

The Cycle Counter 5xx Control window is available from the Emulator menu when the debugger is running.

Use this window to set the behavior of Cycle counter 1, the extra cycle counter for MSP430F5xx devices.
Applications

The Cycle Counter 5xx Control window contains preconfigured applications, as well as the opportunity to customize your own use of the cycle counter. Use the cycle counter applications to configure for what purpose to use the cycle counter. Choose between:

- **Cycle counter**: Makes Cycle counter 1 count all CPU cycles; see Counting all CPU cycles, page 184.
- **DMA load vs. CPU load**: Compares the number of DMA bus cycles with the total number of bus cycles; see Measuring the DMA load versus the CPU load, page 185.
- **Profiling**: Profiles a specific part of your program; see Profiling a specific part of your application, page 185.
- **Trigger hits**: Measures the number of times a certain point in your program has been reached; see Measuring the Trigger hits, page 186.
- **Trip counter**: Measures the number of CPU cycles required to execute a certain task or function; see Measuring the number of CPU cycles for a task, page 186.
- **Custom**: Make a custom application using the options in the Cycle Counter 1 area.

Cycle counter values

- **Cycle counter 0**: The value of cycle counter 0. Cycle counter 0 is the same cycle counter as the one displayed in the Register window.
- **Cycle counter 1**: The value of Cycle counter 1. Cycle counter 1 is controlled by the settings in the Cycle Counter 5xx Control window.

Mode

- **Reaction**: Increments the counter on reactions.
- **Fetch**: Increments the counter on all instruction fetch cycles.
- **Bus incl. DMA**: Increments the counter on all bus cycles (including DMA cycles).
- **CPU excl. DMA**: Increments the counter on all CPU bus cycles (excluding DMA cycles).
- **DMA**: Increments the counter on all DMA bus cycles.
Using the cycle counter for 5xx devices

Start

Reaction
  Uses the cycle counter reaction to start the cycle counter.

CPU starts
  Starts counting cycles when the CPU starts to execute.

Stop

Reaction
  Uses the cycle counter reaction to stop the cycle counter.

CPU stops
  Stops counting when the CPU stops the execution.

Clear

Reaction
  Clears the cycle counter on the cycle count reaction.

No event
  Does not clear the counter.

Combinations of start, stop, and clear reactions

<table>
<thead>
<tr>
<th>Start</th>
<th>Stop</th>
<th>Clear</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Reaction</td>
<td>Reaction</td>
<td>Reaction</td>
<td>The cycle counter reaction starts the cycle counter and clears it at start.</td>
</tr>
<tr>
<td>Reaction</td>
<td>Reaction</td>
<td>Reaction</td>
<td>The cycle counter reaction stops and clears the cycle counter.</td>
</tr>
<tr>
<td>Reaction</td>
<td>Reaction</td>
<td>Reaction</td>
<td>If the cycle counter is stopped, the cycle counter reaction starts the counter.</td>
</tr>
<tr>
<td>Reaction</td>
<td>Reaction</td>
<td>Reaction</td>
<td>If the cycle counter is running, the cycle counter reaction stops the counter.</td>
</tr>
</tbody>
</table>

Table 14: Cycle Counter 1, combinations of start, stop, and clear reactions

Reaction trigger 1, Reaction trigger 2

Selects breakpoints to act as a reaction triggers.

Reset Counter 1

Resets the value and state of Cycle counter 1.
Reference information on the cycle counter for 5xx
Simulating interrupts

By simulating interrupts, you can debug the program logic long before any hardware is available. This chapter 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. More specifically, this means:

- Introduction to interrupt simulation
- Procedures for simulating interrupts
- Reference information on simulating interrupts.

Introduction to interrupt simulation

This section introduces the C-SPY interrupt simulation system. These topics are covered:

- Reasons for using the interrupt simulation system
- Briefly about the interrupt simulation system
- Interrupt characteristics
- Interrupt simulation states
- C-SPY system macros for interrupts
- Target-adapting the interrupt simulation system.

For related information, see also:

- Reference information on C-SPY system macros, page 219
- Using breakpoints, page 87
- Timeline window, page 149

Reasons for using the interrupt simulation system

Simulated interrupts let you test the logic of your interrupt service routines and debug the interrupt handling in the target system. 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.
Introduction to interrupt simulation

BRIEFLY ABOUT THE INTERRUPT SIMULATION SYSTEM

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 MSP430 microcontroller
- 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 which continuously displays events for each defined interrupt.

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 86: 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 Setup dialog box displays the available status information. For an interrupt, these states can be displayed: Idle, Pending, Executing, Executed, Removed, Rejected, or Expired.
For a repeatable interrupt that has a specified repeat interval which is longer than the execution time, the status information at different times can look like this:

**Figure 87: 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.

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 can look like this:

**Figure 88: Simulation states - example 2**
In this case, the execution time of the interrupt handler is too long compared to the repeat interval, which 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 INTERRUPTS**

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 detailed reference information about each macro, see *Reference information on C-SPY system macros*, page 219.

**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 details of device description files, see *Selecting a device description file*, page 37.
Procedures for simulating interrupts

This section gives you step-by-step descriptions about how to use the interrupt simulation system.

More specifically, you will get information about:

- Simulating a simple interrupt
- Simulating an interrupt in a multi-task system.

For related information, see also:

- Registering and executing using setup macros and setup files, page 210 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:

Assume this simple application which contains an interrupt service routine for the BasicTimer, which increments a tick variable. The main function sets the necessary status registers. The application exits when 100 interrupts have been generated.

```c
#include "io430x41x.h"
#include <intrinsics.h>
volatile int ticks = 0;
void main (void)
{
    /* Timer setup code */
    WDTCTL = WDTPW + WDTHOLD; /* Stop WDT */
    IE2 |= BTIE; /* Enable BT interrupt */
    BTCTL = BTSEL+BTP1+BTP0;
    __enable_interrupt(); /* Enable interrupts */

    while (ticks < 100); /* Endless loop */
    printf("Done\n");
}
```
/* Timer interrupt service routine */
#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 Choose Project>Options>Debugger>Setup and select a device description file. The device description file contains information about the interrupt that C-SPY needs to be able to simulate it. Use the Use device description file browse button to locate the .ddf file.

4 Build your project and start the simulator.

5 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 Timer example, verify these settings:

<table>
<thead>
<tr>
<th>Option</th>
<th>Settings</th>
</tr>
</thead>
<tbody>
<tr>
<td>Interrupt</td>
<td>BASICTIMER_VECTOR</td>
</tr>
<tr>
<td>First activation</td>
<td>4000</td>
</tr>
<tr>
<td>Repeat interval</td>
<td>2000</td>
</tr>
<tr>
<td>Hold time</td>
<td>10</td>
</tr>
<tr>
<td>Probability (%)</td>
<td>100</td>
</tr>
<tr>
<td>Variance (%)</td>
<td>0</td>
</tr>
</tbody>
</table>

Table 15: Timer interrupt settings

Click OK.

6 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.

7 To watch the interrupt in action, choose Simulator>Interrupt Log to open the Interrupt Log window.

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.

---

**Reference information on simulating interrupts**

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

- **Interrupt Setup dialog box**, page 198
- **Edit Interrupt dialog box**, page 200
- **Forced Interrupt window**, page 202
- **Interrupt Log window**, page 203.

**Interrupt Setup dialog box**

The **Interrupt Setup** dialog box is available by choosing Simulator>Interrupt Setup.

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.
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.
- **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.
  
  For repeatable interrupts there might be additional information about how many interrupts of the same type that are simultaneously executing ($n$ executing). If $n$ is larger than one, there is a reentrant interrupt in your interrupt simulation system that never finishes executing, which might indicate that there is a problem in your application.

- **Status**: Shows 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 acknowledged yet by the interrupt handler.
  - **Executing**, the interrupt is currently being serviced, that is the interrupt handler function is executing.
  - **Executed**, this is a single-occasion interrupt and it has been serviced.
  - **Removed**, the interrupt has been removed, but because the interrupt is currently executing it is visible until it is finished.
  - **Rejected**, the interrupt has been rejected because the necessary interrupt registers are not set up to accept the interrupt.
  - **Expired**, this is a single-occasion interrupt which was not serviced while the interrupt activation signal was active.

- **Next Activation**: Shows the next activation time in cycles.
Buttons

These buttons are available:

**New**
Opens the **Edit Interrupt** dialog box, see *Edit Interrupt dialog box*, page 200.

**Edit**
Opens the **Edit Interrupt** dialog box, see *Edit Interrupt dialog box*, page 200.

**Delete**
Removes the selected interrupt.

**Delete All**
Removes all interrupts.

**Note:** You can only edit or remove non-forced interrupts.

**Edit Interrupt dialog box**

The **Edit Interrupt** dialog box is available from the **Interrupt Setup** dialog box.

![Figure 90: Edit Interrupt dialog box](image)

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.

**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.
Simulating interrupts

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 address, priority, enable bit, and pending 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 Simulator menu.

![Forced Interrupt window](image)

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 198.
2. Double-click the interrupt in the Forced Interrupt window, or select the interrupt and click *Trigger*.

**Display 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.

**Trigger**

Triggers the interrupt you selected in the display area.
Interrupt Log window

The Interrupt Log window is available from the Simulator menu.

![Interrupt Log window](image)

This window displays runtime information about the interrupts that you have activated in the Edit Interrupts dialog box or forced via the Forced Interrupt window. The information is useful for debugging the interrupt handling in the target system.

When the Interrupt Log window is open it is updated continuously during runtime.

**Note:** If the window becomes full of entries, the first entries are erased.

**Display area**

This area contains these columns:

- **Cycles**: The point in time, measured in cycles, when the event occurred. Displayed if Show Cycles is selected on the context menu.
- **Time**: The point in time, measured in seconds, when the event occurred. Displayed if Show Time is selected on 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 has been forced from the Forced Interrupt window.
  - **Executing**: The interrupt is currently executing.
  - **Finished**: 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 are not set up to accept the interrupt.
Reference information on simulating interrupts

Context menu

Program Counter  The value of the program counter when the event occurred.

Execution Cycles  The duration of the interrupt in execution cycles. Displayed if Show Cycles is selected on the context menu.

Execution Time  The duration of the interrupt in seconds. Displayed if Show Time is selected on the context menu.

These commands are available:

- Enable
- Show Time
- Show Cycles
- Clear
- Save to Log File

These commands are available:

- Enable  Enables interrupt logging.
- Show Time  Uses seconds as the unit of measurement when displaying when events occur and the duration of interrupts.
- Show Cycles  Uses cycles as the unit of measurement when displaying when events occur and the duration of interrupts.
- Clear  Clears all contents of the window.
- Save to Log File  Opens a standard Save dialog box to let you save the contents of the window as a text file.

Related information

For related information, see Timeline window, page 149.
Using 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
- Procedures for 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.
Introduction to C-SPY macros

- Simulating peripheral devices, see the chapter Simulating interrupts. This only applies if you are using the simulator driver.
- Developing small debug utility functions, for instance calculating the stack depth.

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 detailed information about each setup macro function, see Reference information on reserved setup macro function names, page 218.
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 a detailed description of the macro language components, see Reference information on the macro language, page 213.

Example

Consider this example of a macro function which illustrates the various components of the macro language:

```c
__var oldValue;
CheckLatest(val)
{
  if (oldval != val)
  {
    __message "Message: Changed from ", oldval, ", to ", val, ";
    oldval = val;
  }
}
```

**Note:** Reserved macro words begin with double underscores to prevent name conflicts.

Procedures for using C-SPY macros

This section gives you step-by-step descriptions about how to register and execute C-SPY macros.

More specifically, you will get information about:

- Registering C-SPY macros—an overview
- Executing C-SPY macros—an overview
- Using the Macro Configuration dialog box
- Registering and executing using setup macros and setup files
Procedures for using C-SPY macros

- Executing macros using Quick Watch
- Executing a macro by connecting it to a breakpoint.

For more examples using C-SPY macros, see:

- The tutorial Simulating an interrupt in the Information Center
- Initializing target hardware before C-SPY starts, page 41.

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 209.
- You can register macro functions during the C-SPY startup sequence, see Registering and executing using setup macros and setup files, page 210.
- 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 details about the system macro, see __registerMacroFile, page 232.

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 210.
- 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 211.
- 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 212.

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.

![Macro Configuration dialog box](image)

Figure 94: 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:

1. 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 debugger. 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:
```c
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.
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:

   ```c
   WDTstatus()
   {
     if (#IE1 & 0x01 != 0)      /* Checks the status of WDTIE */
       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.

The macro will automatically be displayed in the Quick Watch window.

For reference information, see Quick Watch window, page 83.
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:

   ```c
   int fact(int x)
   {
   ...
   }
   ```

2. Create a simple log macro function like this example:

   ```c
   logfact()
   {
       __message "fact(" ,x, ")";
   }
   ```

   The __message statement will log messages to the Log window.

3. Save the macro function in a macro file, with the filename extension .mac.

4. 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.

5. 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.

6. 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.

7. 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 103
- Use the **Condition** field instead of the **Action** field. For an example, see *Performing a task and continue execution*, page 97.

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 216.

For a complete 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 213
- **Macro variables**, page 214
- **Macro strings**, page 214
- **Macro statements**, page 215
- **Formatted output**, page 216.

**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 detailed information about C-SPY expressions, see C-SPY expressions, page 72.

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
```

<table>
<thead>
<tr>
<th>Expression</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>myvar = 3.5;</td>
<td>myvar is now type float, value 3.5.</td>
</tr>
<tr>
<td>myvar = (int*)i;</td>
<td>myvar is now type pointer to int, and the value is the same as i.</td>
</tr>
</tbody>
</table>

Table 16: 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 \texttt{\_\_toString} is used for converting from a NULL-terminated C string in your application (\texttt{char*} or \texttt{char[]}) to a macro string. For example, assume this definition of a C string in your application:

\begin{verbatim}
char const *cstr = "Hello";
\end{verbatim}

Then examine these macro examples:

\begin{verbatim}
\_\_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!" */
\end{verbatim}

See also \textit{Formatted output}, page 216.

**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**

\begin{verbatim}
expression;
\end{verbatim}

For detailed information about C-SPY expressions, see \textit{C-SPY expressions}, page 72.

**Conditional statements**

\begin{verbatim}
if (expression)  
  statement
\end{verbatim}

\begin{verbatim}
if (expression)  
  statement  
else  
  statement
\end{verbatim}

**Loop statements**

\begin{verbatim}
for (init_expression; cond_expression; update_expression)  
  statement
\end{verbatim}

\begin{verbatim}
while (expression)  
  statement
\end{verbatim}
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 227.

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:

```c
__fmessage myfile, "Result is ", res, "!\n";
```

To produce strings:

```c
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 colon `:` 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:

```c
__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:

```c
__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 206*.

This table summarizes the reserved setup macro function names:

<table>
<thead>
<tr>
<th>Macro</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>execUserPreload</td>
<td>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.</td>
</tr>
<tr>
<td>execUserSetup</td>
<td>Called once after the target application is downloaded. Implement this macro to set up the memory map, breakpoints, interrupts, register macro files, etc.</td>
</tr>
<tr>
<td>execUserReset</td>
<td>Called each time the reset command is issued. Implement this macro to set up and restore data.</td>
</tr>
<tr>
<td>execUserExit</td>
<td>Called once when the debug session ends. Implement this macro to save status data etc.</td>
</tr>
</tbody>
</table>

**Table 17: 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 *Simulating an interrupt* 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:

<table>
<thead>
<tr>
<th>Macro</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>__cancelAllInterrupts</td>
<td>Cancels all ordered interrupts</td>
</tr>
<tr>
<td>__cancelInterrupt</td>
<td>Cancels an interrupt</td>
</tr>
<tr>
<td>__clearBreak</td>
<td>Clears a breakpoint</td>
</tr>
<tr>
<td>__closeFile</td>
<td>Closes a file that was opened by __openFile</td>
</tr>
<tr>
<td>__delay</td>
<td>Delays execution</td>
</tr>
<tr>
<td>__disableInterrupts</td>
<td>Disables generation of interrupts</td>
</tr>
<tr>
<td>__driverType</td>
<td>Verifies the driver type</td>
</tr>
<tr>
<td>__enableInterrupts</td>
<td>Enables generation of interrupts</td>
</tr>
<tr>
<td>__evaluate</td>
<td>Interprets the input string as an expression and evaluates it.</td>
</tr>
<tr>
<td>__isBatchMode</td>
<td>Checks if C-SPY is running in batch mode or not.</td>
</tr>
<tr>
<td>__loadImage</td>
<td>Loads an image.</td>
</tr>
<tr>
<td>__memoryRestore</td>
<td>Restores the contents of a file to a specified memory zone</td>
</tr>
<tr>
<td>__memorySave</td>
<td>Saves the contents of a specified memory area to a file</td>
</tr>
<tr>
<td>__openFile</td>
<td>Opens a file for I/O operations</td>
</tr>
<tr>
<td>__orderInterrupt</td>
<td>Generates an interrupt</td>
</tr>
<tr>
<td>__popSimulatorInterruptExecutingStack</td>
<td>Informs the interrupt simulation system that an interrupt handler has finished executing</td>
</tr>
<tr>
<td>__readFile</td>
<td>Reads from the specified file</td>
</tr>
<tr>
<td>__readFileByte</td>
<td>Reads one byte from the specified file</td>
</tr>
<tr>
<td>__readMemory8, __readMemoryByte</td>
<td>Reads one byte from the specified memory location</td>
</tr>
<tr>
<td>__readMemory16</td>
<td>Reads two bytes from the specified memory location</td>
</tr>
<tr>
<td>__readMemory32</td>
<td>Reads four bytes from the specified memory location</td>
</tr>
<tr>
<td>__registerMacroFile</td>
<td>Registers macros from the specified file</td>
</tr>
<tr>
<td>__resetFile</td>
<td>Rewinds a file opened by __openFile</td>
</tr>
</tbody>
</table>

Table 18: Summary of system macros
<table>
<thead>
<tr>
<th>Macro</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>__setAdvancedTriggerBreak</code></td>
<td>Sets an advanced trigger breakpoint</td>
</tr>
<tr>
<td><code>__setCodeBreak</code></td>
<td>Sets a code breakpoint</td>
</tr>
<tr>
<td><code>__setConditionalBreak</code></td>
<td>Sets a conditional breakpoint</td>
</tr>
<tr>
<td><code>__setDataBreak</code></td>
<td>Sets a data breakpoint</td>
</tr>
<tr>
<td><code>__setLogBreak</code></td>
<td>Sets a log breakpoint</td>
</tr>
<tr>
<td><code>__setRangeBreak</code></td>
<td>Sets a range breakpoint</td>
</tr>
<tr>
<td><code>__setSimBreak</code></td>
<td>Sets a simulation breakpoint</td>
</tr>
<tr>
<td><code>__setTraceStartBreak</code></td>
<td>Sets a trace start breakpoint</td>
</tr>
<tr>
<td><code>__setTraceStopBreak</code></td>
<td>Sets a trace stop breakpoint</td>
</tr>
<tr>
<td><code>__sourcePosition</code></td>
<td>Returns the file name and source location if the current execution location corresponds to a source location</td>
</tr>
<tr>
<td><code>__strFind</code></td>
<td>Searches a given string for the occurrence of another string</td>
</tr>
<tr>
<td><code>__subString</code></td>
<td>Extracts a substring from another string</td>
</tr>
<tr>
<td><code>__targetDebuggerVersion</code></td>
<td>Returns the version of the target debugger</td>
</tr>
<tr>
<td><code>__toLower</code></td>
<td>Returns a copy of the parameter string where all the characters have been converted to lower case</td>
</tr>
<tr>
<td><code>__toString</code></td>
<td>Prints strings</td>
</tr>
<tr>
<td><code>__toUpper</code></td>
<td>Returns a copy of the parameter string where all the characters have been converted to upper case</td>
</tr>
<tr>
<td><code>__unloadImage</code></td>
<td>Unloads a debug image</td>
</tr>
<tr>
<td><code>__writeFile</code></td>
<td>Writes to the specified file</td>
</tr>
<tr>
<td><code>__writeFileByte</code></td>
<td>Writes one byte to the specified file</td>
</tr>
<tr>
<td><code>__writeMemory8</code>,</td>
<td>Writes one byte to the specified memory location</td>
</tr>
<tr>
<td><code>__writeMemoryByte</code></td>
<td>Writes a two-byte word to the specified memory location</td>
</tr>
<tr>
<td><code>__writeMemory16</code></td>
<td>Writes a four-byte word to the specified memory location</td>
</tr>
</tbody>
</table>

Table 18: Summary of system macros (Continued)
__cancelAllInterrupts

Syntax

__cancelAllInterrupts()

Return value

int 0

Description

Cancels all ordered interrupts.

Applicability

This system macro is only available in the C-SPY Simulator.

__cancelInterrupt

Syntax

__cancelInterrupt(interrupt_id)

Parameter

interrupt_id          The value returned by the corresponding
                      __orderInterrupt macro call (unsigned long)

Return value

<table>
<thead>
<tr>
<th>Result</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>Successful</td>
<td>int 0</td>
</tr>
<tr>
<td>Unsuccessful</td>
<td>Non-zero error</td>
</tr>
</tbody>
</table>

Description

Cancels the specified interrupt.

Applicability

This system macro is only available in the C-SPY Simulator.

__clearBreak

Syntax

__clearBreak(break_id)

Parameter

break_id          The value returned by any of the set breakpoint macros

Return value

int 0

Description

Clears a user-defined breakpoint.

See also

Using breakpoints, page 87.
**__closeFile**

Syntax: __closeFile(fileHandle)

Parameter:

- **fileHandle**: The 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)

Parameter:

- **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 20: __disableInterrupts return values

Description: Disables the generation of interrupts.

Applicability: This system macro is only available in the C-SPY Simulator.
__driverType

Syntax

__driverType(driver_id)

Parameter

driver_id A string corresponding to the driver you want to check for. Choose one of these:
"sim" corresponds to the simulator driver.
"fet" corresponds to the FET debugger driver.

Return value

<table>
<thead>
<tr>
<th>Result</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>Successful</td>
<td>1</td>
</tr>
<tr>
<td>Unsuccessful</td>
<td>0</td>
</tr>
</tbody>
</table>

Table 21: __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

<table>
<thead>
<tr>
<th>Result</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>Successful</td>
<td>int 0</td>
</tr>
<tr>
<td>Unsuccessful</td>
<td>Non-zero error number</td>
</tr>
</tbody>
</table>

Table 22: __enableInterrupts return values

Description
Enables the generation of interrupts.

Applicability
This system macro is only available in the C-SPY Simulator.
__evaluate

Syntax

__evaluate(string, valuePtr)

Parameter

<table>
<thead>
<tr>
<th>string</th>
<th>Expression string</th>
</tr>
</thead>
<tbody>
<tr>
<td>valuePtr</td>
<td>Pointer to a macro variable storing the result</td>
</tr>
</tbody>
</table>

Return value

<table>
<thead>
<tr>
<th>Result</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>Successful</td>
<td>int 0</td>
</tr>
<tr>
<td>Unsuccessful</td>
<td>int 1</td>
</tr>
</tbody>
</table>

Table 23: __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

<table>
<thead>
<tr>
<th>Result</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>True</td>
<td>int 1</td>
</tr>
<tr>
<td>False</td>
<td>int 0</td>
</tr>
</tbody>
</table>

Table 24: __isBatchMode return values

Description

This macro returns True if the debugger is running in batch mode, otherwise it returns False.
Using C-SPY macros

__loadImage

Syntax

__loadImage(path, offset, debugInfoOnly)

Parameter

path
A string that identifies the path to the image to download. The path must either be absolute or use argument variables. Read more about argument variables in the IDE Project Management and Building Guide.

offset
An integer that identifies the offset to the destination address for the downloaded image.

downloadInfoOnly
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

<table>
<thead>
<tr>
<th>Value</th>
<th>Result</th>
</tr>
</thead>
<tbody>
<tr>
<td>Non-zero integer number</td>
<td>A unique module identification.</td>
</tr>
<tr>
<td>int 0</td>
<td>Loading failed.</td>
</tr>
</tbody>
</table>

Table 25: __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:

```c
__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:

```c
__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


**__memoryRestore**

Syntax

```
__memoryRestore(zone, filename)
```

Parameters

- `zone` The memory zone name (string); for a list of available zones, see C-SPY memory zones, page 120.
- `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 128.

**__memorySave**

Syntax

```
__memorySave(start, stop, format, file)
```

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
Using C-SPY macros

__memorySave

**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 127.

___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.
__openFile

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

```c
__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.

<table>
<thead>
<tr>
<th>Result</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>Successful</td>
<td>The file handle</td>
</tr>
<tr>
<td>Unsuccessful</td>
<td>An invalid file handle, which tests as False</td>
</tr>
</tbody>
</table>

Table 26: __openFile return values
__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.

Description

Generates an interrupt.

Applicability

This system macro is only available in the C-SPY Simulator.

Example

This example generates a repeating interrupt using an infinite hold time first activated after 4000 cycles:

```c
__orderInterrupt( 'USART0RX_VECTOR', 4000, 2000, 0, 1, 0, 100 );
```

__popSimulatorInterruptExecutingStack

Syntax

__popSimulatorInterruptExecutingStack(void)

Return value

This macro has no return value.
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.

Applicability

This system macro is only available in the C-SPY Simulator.

See also

Simulating an interrupt in a multi-task system, page 197.

__readFile

Syntax

__readFile(fileHandle, valuePtr)

Parameters

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>fileHandle</td>
<td>A macro variable used as filehandle by the __openFile macro</td>
</tr>
<tr>
<td>valuePtr</td>
<td>A pointer to a variable</td>
</tr>
</tbody>
</table>

Return value

<table>
<thead>
<tr>
<th>Result</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>Successful</td>
<td>0</td>
</tr>
<tr>
<td>Unsuccessful</td>
<td>Non-zero error number</td>
</tr>
</tbody>
</table>

Table 27: __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

```c
__var number;
if (__readFile(myFileHandle, &number) == 0)
{
    // Do something with number
}
```
__readFileByte

Syntax
__readFileByte(fileHandle)

Parameter
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 The memory zone name (string); for a list of available zones, see C-SPY memory zones, page 120.

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)
Reference information on C-SPY system macros

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
The memory zone name (string); for a list of available zones, see C-SPY memory zones, page 120.

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)

Parameter
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.

C-SPY® Debugging Guide
for MSP430
Example

```c
__registerMacroFile("c:\\testdir\\macro.mac");
```

See also


### __resetFile

**Syntax**

```c
__resetFile(fileHandle)
```

**Parameter**

- **fileHandle**: A macro variable used as filehandle by the `__openFile` macro.

**Return value**

`int 0`

**Description**

Rewinds a file previously opened by `__openFile`.

### __setAdvancedTriggerBreak

**Syntax**

```c
__setAdvancedTriggerBreak(type, condition, access, action, mask cond_value)
```

**Parameters**

- **All parameters are strings.**
  - **type**: The breakpoint type; either "Address", "Data", or "Register".
  - **condition**: The breakpoint condition operator, either "==", ">=", "<=", or "!=".
Reference information on C-SPY system macros

**access**  
The memory access type. Choose between:

- "Read"
- "Write"
- "ReadWrite"
- "Fetch"
- "FetchHold"
- "NoFetch"
- "NoFetchRead"
- "NoFetchNoDMA"
- "DMA"
- "NoDMA"
- "WriteNoDMA"
- "NoFetchReadNoDMA"
- "ReadNoDMA"
- "ReadDMA"
- "WriteDMA"

**action**  
The action type: "Break", "Trigger", or "BreakTrigger".

**mask**  
A 16-bit value that the breakpoint address or value will be masked with.

**cond_value**  
An extra conditional data value.

<table>
<thead>
<tr>
<th>Return value</th>
<th>Result</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Successful</td>
<td>An unsigned integer uniquely identifying the breakpoint. This value must be used to clear the breakpoint.</td>
</tr>
<tr>
<td></td>
<td>Unsuccessful</td>
<td>0</td>
</tr>
</tbody>
</table>

Table 28: __setAdvancedTriggerBreak return values

**Description**  
Sets an advanced trigger breakpoint.

**Applicability**  
This system macro is only available in the C-SPY FET Debugger driver.

**Example**  
```c
__var brk;
brk = __setAdvancedTriggerBreak("Register", ">=", "Write",
                                 "Trigger", "0x0000", "0x4000");
...
__clearBreak(brk);
```

**See also**  
*Using breakpoints*, page 87.
__setCodeBreak

Syntax

__setCodeBreak(location, count, condition, cond_type, action)

Parameters

location A string with a location description. This can be either:

- A source location on the form {filename}.line.col (for example {D:\src\prog.c}.12.9)
- An absolute location on the form zone:hexaddress or simply hexaddress (for example Memory:0x42)
- An expression whose value designates a location (for example main)

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

<table>
<thead>
<tr>
<th>Result</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>Successful</td>
<td>An unsigned integer uniquely identifying the breakpoint. This value must be used to clear the breakpoint.</td>
</tr>
<tr>
<td>Unsuccessful</td>
<td>0</td>
</tr>
</tbody>
</table>

Table 29: __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

Using breakpoints, page 87.
__setConditionalBreak

Syntax

__setConditionalBreak(location, type, operator, access, action,
mask, cond_value, cond_operator,
cond_access, cond_mask)

Parameters

All parameters are strings.

location The breakpoint location. Choose between:

A source location on the form "{filename}.line.col" (for example "{D:\src\prog.c}.12.9")
An absolute location on the form "zone:hexaddress" or simply "hexaddress" (for example "Memory:0x42")
An expression whose value designates a location (for example "my_global_variable").
A register (for example "R10")

type The breakpoint type: "Address", "Data", or "Register".
operator The breakpoint operator: "==", ">=", "<=", or "!=".
access The memory access type: "Read", "Write", "ReadWrite", or "Fetch".
action The action type: "Break", "Trigger", or "BreakTrigger".
mask A 16-bit value that the breakpoint address or value will be masked with.
cond_value An extra conditional data value.
cond_operator The condition operator: "==", ">=", "<=", or "!=".
cond_access The access type of the condition: "Read" or "Write".
cond_mask The mask value of the condition.

Return value

Result | Value
--- | ---
Successful | An unsigned integer uniquely identifying the breakpoint. This value must be used to clear the breakpoint.
Unsuccessful | 0

Table 30: __setConditionalBreak return values

Description

Sets a conditional breakpoint.
Applicability

This system macro is only available in the C-SPY FET Debugger driver.

Example

```c
__var brk;
brk = __setConditionalBreak("R10", "Register", "0x5000", ">=", "Write", "Trigger", "0x0000", "0x4000", "=" , "Write", "0x00FF");
...
__clearBreak(brk);
```

See also

*Using breakpoints*, page 87.

__setDataBreak

**Syntax**

```
__setDataBreak(location, count, condition, cond_type, access, action)
```

**Parameters**

- **location**
  A string with a location description. This can be either:
  - A *source location* on the form `{filename}.line.col` (for example `{D:\src\prog.c}.12.9`), although this is not very useful for data breakpoints
  - An *absolute location* on the form `zone:hexaddress` or simply `hexaddress` (for example Memory:0x42)
  - An *expression* whose value designates a location (for example `my_global_variable`).

- **count**
  The number of times 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

<table>
<thead>
<tr>
<th>Result</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>Successful</td>
<td>An unsigned integer uniquely identifying the breakpoint. This value</td>
</tr>
<tr>
<td></td>
<td>must be used to clear the breakpoint.</td>
</tr>
<tr>
<td>Unsuccessful</td>
<td>0</td>
</tr>
</tbody>
</table>

Table 31: __setDataBreak return values

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.

Applicability
This system macro is only available in the C-SPY Simulator.

Example
```c
__var brk;
brk = __setDataBreak("Memory:0x4710", 3, "d=6", "TRUE", "W", "ActionData()");
...
__clearBreak(brk);
```

See also
Using breakpoints, page 87.

__setLogBreak

Syntax
```c
__setLogBreak(location, message, msg_type, condition, cond_type)
```

Parameters

- `location` A string with a location description. This can be either:
  - A source location on the form `{filename}.line.col` (for example `{D:\src\prog.c}.12.9`)
  - An absolute location on the form `zone:hexaddress` or simply `hexaddress` (for example `Memory:0x42`)
  - An expression whose value designates a location (for example `main`)

- `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.
Using C-SPY macros

__setLogBreak

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

```c
__var logBp1;
__var logBp2;

logOn()
{
    logBp1 = __setLogBreak("{C:\temp\Utilities.c}.23.1",
                           "{C:\temp\Utilities.c}.23.1",
                           "{C:\temp\Utilities.c}.23.1",
                           "Entering trace zone at :", #PC:%X", "ARGS", "1", "TRUE");
    logBp2 = __setLogBreak("{C:\temp\Utilities.c}.30.1",
                           "{C:\temp\Utilities.c}.30.1",
                           "{C:\temp\Utilities.c}.30.1",
                           "Leaving trace zone...", "TEXT", "1", "TRUE");
}

logOff()
{
    __clearBreak(logBp1);
    __clearBreak(logBp2);
}
```

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.

See also

Formatted output, page 216 and Using breakpoints, page 87.

---

**condition**

The breakpoint condition (string)

**cond_type**

The condition type; either "CHANGED" or "TRUE" (string)

**Return value**

<table>
<thead>
<tr>
<th>Result</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>Successful</td>
<td>An unsigned integer uniquely identifying the breakpoint. The same value must be used when you want to clear the breakpoint.</td>
</tr>
<tr>
<td>Unsuccessful</td>
<td>0</td>
</tr>
</tbody>
</table>

Table 32: __setLogBreak return values
__setRangeBreak

Syntax

__setRangeBreak(start_loc, end_loc, end_cond, type, access, action, action_when)

Parameters

All parameters are strings.

- **start_loc**: The start location. Choose between:
  - A *source location* on the form "{filename}.line.col" (for example "{D:\src\prog.c}.12.9")
  - An *absolute location* on the form "zone:hexaddress" or simply "hexaddress" (for example "Memory:0x42")
  - An *expression* whose value designates a location (for example "my_global_variable").

- **end_loc**: The end location. This can be either the same as for start_loc above or the length of the range.

- **end_cond**: The type of end condition: "Location", 'Length', or "Automatic".

- **type**: The breakpoint type: "Address" or "Data".

- **access**: The memory access type: "Read", "Write", "ReadWrite", or "Fetch".

- **action**: The action type: "Break", "Trigger", or "BreakTrigger".

- **action_when**: Specifies if the action should happen at an access inside or outside of the specified range: "Inside" or "Outside".

Return value

<table>
<thead>
<tr>
<th>Result</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>Successful</td>
<td>An unsigned integer uniquely identifying the breakpoint. This value</td>
</tr>
<tr>
<td></td>
<td>must be used to clear the breakpoint.</td>
</tr>
<tr>
<td>Unsuccessful</td>
<td>0</td>
</tr>
</tbody>
</table>

Table 33: __setRangeBreak return values

Description

Sets a range breakpoint.

Applicability

This system macro is only available in the C-SPY FET Debugger driver.
Example

```c
__var brk;
brk = __setRangeBreak("Memory:0x1240", "Memory:0x1360",
   "Location", "Address", "Fetch", "Trigger", "Inside");
...__clearBreak(brk);
```

See also

*Using breakpoints*, page 87.

__setSimBreak

**Syntax**

```c
__setSimBreak(location, access, action)
```

**Parameters**

- **location**: A string with a location description. This can be either:
  - A source location on the form `{filename}.line.col` (for example `{D:\src\prog.c}.12.9`)
  - An absolute location on the form `zone:hexaddress` or simply `hexaddress` (for example `Memory:0x42`)
  - An expression whose value designates a location (for example `main`)

- **access**: The memory access type: "R" for read or "W" for write

- **action**: An expression, typically a call to a macro function, which is evaluated when the breakpoint is detected

**Return value**

<table>
<thead>
<tr>
<th>Result</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>Successful</td>
<td>An unsigned integer uniquely identifying the breakpoint. This value must be used to clear the breakpoint.</td>
</tr>
<tr>
<td>Unsuccessful</td>
<td>0</td>
</tr>
</tbody>
</table>

*Table 34: __setSimBreak return values*

**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.

Applicability

This system macro is only available in the C-SPY Simulator.

__setTraceStartBreak

Syntax

```c
__setTraceStartBreak(location)
```

Parameters

- `location` A string with a location description. This can be either:
  - A source location on the form `{filename}.line.col` (for example `{D:\src\prog.c}.12.9`)
  - An absolute location on the form `zone:hexaddress` or simply `hexaddress` (for example `Memory:0x42`)
  - An expression whose value designates a location (for example `main`)

Return value

<table>
<thead>
<tr>
<th>Result</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>Successful</td>
<td>An unsigned integer uniquely identifying the breakpoint. The same value must be used when you want to clear the breakpoint.</td>
</tr>
<tr>
<td>Unsuccessful</td>
<td>0</td>
</tr>
</tbody>
</table>

Table 35: __setTraceStartBreak return values

Description

Sets a breakpoint at the specified location. When that breakpoint is triggered, the trace system is started.

Applicability

This system macro is only available in the C-SPY Simulator.

Example

```c
__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 \textit{Using breakpoints}, page 87.

\textbf{\texttt{\_\_setTraceStopBreak}}

\textbf{Syntax}
\begin{verbatim}
\_\_setTraceStopBreak(location)
\end{verbatim}

\textbf{Parameters}
\begin{itemize}
\item \textit{location} A string with a location description. This can be either:
\begin{itemize}
\item A source location on the form \{filename\}.line.col (for example \{D:\src\prog.c\}.12.9)
\item An absolute location on the form \textit{zone:hexaddress} or simply \textit{hexaddress} (for example Memory:0x42)
\item An expression whose value designates a location (for example main)
\end{itemize}
\end{itemize}

\textbf{Return value}
\begin{tabular}{ll}
\hline
Result & Value \tabularnewline
\hline
Successful & An unsigned integer uniquely identifying the breakpoint. The same value must be used when you want to clear the breakpoint. \\
Unsuccessful & int 0 \\
\hline
\end{tabular}

\textit{Table 36: \_\_setTraceStopBreak return values}

\textbf{Description}
Sets a breakpoint at the specified location. When that breakpoint is triggered, the trace system is stopped.

\textbf{Applicability}
This system macro is only available in the C-SPY Simulator.

\textbf{Example}
See \texttt{\_\_setTraceStartBreak}, page 242.

\textbf{See also}
\textit{Using breakpoints}, page 87.
__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

<table>
<thead>
<tr>
<th>Result</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>Successful</td>
<td>Filename string</td>
</tr>
<tr>
<td>Unsuccessful</td>
<td>Empty (&quot;&quot;) string</td>
</tr>
</tbody>
</table>

Table 37: __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` The macro string to search in
- `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 for the occurrence of another string.

Example
__strFind("Compiler", "pile", 0) = 3
__strFind("Compiler", "foo", 0) = -1

See also
Macro strings, page 214.
using C-SPY macros

__subString

Syntax
__subString(macroString, position, length)

Parameters

- macroString: The macro string from which to extract a substring
- 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.

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 214.

__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)

Parameter

- macroString: Any macro string
### __toLower__

**Return value**  
The converted macro string.

**Description**  
This macro returns a copy of the parameter string 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 214.

### __toString__

**Syntax**  
```c
__toString(C_string, maxlength)
```

**Parameter**  
- `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:

```c
char const * hptr = "Hello World!";
```

this macro call:

```c
__toString(hptr, 5)
```

would return the macro string containing Hello.

**See also**  
Macro strings, page 214.

### __toUpper__

**Syntax**  
```c
__toUpper(macroString)
```

**Parameter**  
`macroString` is any macro string.
Using C-SPY macros

__toupper

<table>
<thead>
<tr>
<th>Return value</th>
<th>Description</th>
<th>Example</th>
</tr>
</thead>
<tbody>
<tr>
<td>The converted string.</td>
<td>This macro returns a copy of the parameter macroString where all the characters have been converted to upper case.</td>
<td><code>__toupper('string')</code></td>
</tr>
<tr>
<td></td>
<td></td>
<td>The resulting macro string contains STRING.</td>
</tr>
<tr>
<td>See also</td>
<td></td>
<td>Macro strings, page 214.</td>
</tr>
</tbody>
</table>

__unloadImage

Syntax

```
__unloadImage(module_id)
```

Parameter

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>module_id</td>
<td>An integer which represents a unique module identification, which is retrieved as a return value from the corresponding __loadImage C-SPY macro.</td>
</tr>
</tbody>
</table>

Return value

<table>
<thead>
<tr>
<th>Value</th>
<th>Result</th>
</tr>
</thead>
<tbody>
<tr>
<td>module_id</td>
<td>A unique module identification (the same as the input parameter).</td>
</tr>
<tr>
<td>int 0</td>
<td>The unloading failed.</td>
</tr>
</tbody>
</table>

See also

Unloads debug information from an already downloaded image.

See also


__writeFile

Syntax

```
__writeFile(file, value)
```

Parameter

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>fileHandle</td>
<td>A macro variable used as filehandle by the __openFile macro</td>
</tr>
<tr>
<td>value</td>
<td>An integer</td>
</tr>
</tbody>
</table>

Return value

<table>
<thead>
<tr>
<th>value</th>
<th>Result</th>
</tr>
</thead>
<tbody>
<tr>
<td>int 0</td>
<td></td>
</tr>
</tbody>
</table>

Table 38: __unloadImage return values
Reference information on C-SPY system macros

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(file, value)

Parameters
fileHandle A macro variable used as filehandle by the __openFile
macro
value An integer in the range 0-255

Return value
int 0

Description
Writes one byte to the file file.

__writeMemory8, __writeMemoryByte

Syntax
__writeMemory8(value, address, zone)
__writeMemoryByte(value, address, zone)

Parameters
value The value to be written (integer)
address The memory address (integer)
zone The memory zone name (string); for a list of available zones,
see C-SPY memory zones, page 120.

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 The value to be written (integer)
address The memory address (integer)
zone The memory zone name (string); for a list of available zones, see C-SPY memory zones, page 120.

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 The value to be written (integer)
address The memory address (integer)
zone The memory zone name (string); for a list of available zones, see C-SPY memory zones, page 120.

Return value int 0

Description Writes four bytes to a given memory location.

Example __writeMemory32(0x5555FFFF, 0x8020, "Memory");
Reference information on C-SPY system macros
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
- Descriptions of 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:

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>processor_DLL</td>
<td>The processor-specific DLL file; available in 430\bin.</td>
</tr>
<tr>
<td>driver_DLL</td>
<td>The C-SPY driver DLL file; available in 430\bin.</td>
</tr>
<tr>
<td>debug_file</td>
<td>The object file that you want to debug (filename extension d43).</td>
</tr>
<tr>
<td>cspybat_options</td>
<td>The command line options that you want to pass to cspybat. Note that these options are optional. For information about each option, see Descriptions of C-SPY command line options, page 255.</td>
</tr>
</tbody>
</table>

*Table 39: cspybat parameters*
Using C-SPY in batch mode

### Example

This example starts `cspybat` using the simulator driver:

```plaintext
EW_DIR\common\bin\cspybat  EW_DIR\430\bin\430proc.dll
EW_DIR\430\bin\430sim.dll  PROJ_DIR\myproject.d43 --plugin
EW_DIR\430\bin\430bat.dll  --backend -d sim -B --cpu 430 -p
EW_DIR\430\bin\config\devicedescription.ddf
```

where `EW_DIR` is the full path of the directory where you have installed IAR Embedded Workbench

and where `PROJ_DIR` is the path of your project directory.

This example shows how to start `cspybat` using the FET debugger driver:

```plaintext
EW_DIR\common\bin\cspybat  EW_DIR\430\bin\430proc.dll
EW_DIR\430\bin\430fet.dll  PROJ_DIR\myproject.d43 --plugin
EW_DIR\430\bin\430bat.dll  --backend -d fet -B --connection ti_usb
--derivative msp430f449 --erase_main_and_info
```

**Note:** The path to the 430\bin directory must be included in the environment path when the parallel port FET or the TI USB FET is used.

### 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.

---

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>--backend</td>
<td>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.</td>
</tr>
<tr>
<td>driver_options</td>
<td>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 Descriptions of C-SPY command line options, page 255.</td>
</tr>
</tbody>
</table>

Table 39: C-SPY parameters (Continued)
● 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 with minimal modifications, 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).
--core
Specifies the core to be used.
-d
Specifies the C-SPY driver to be used.
-p
Specifies the device description file to be used.
Summary of C-SPY command line options

OPTIONS AVAILABLE FOR THE SIMULATOR DRIVER

--disable_interrupts  Disables the interrupt simulation.
--mapu              Activates memory access checking.
--odd_word_check     Stops the execution if an access to an odd address is found.

OPTIONS AVAILABLE FOR THE C-SPY FET DEBUGGER DRIVER

--allow_access_to_BSL  Enables erase/write access to BSL flash memory.
--allow_locked_flash_access Enables erase/write access to locked flash memory.
--attach             Attaches the debugger to a running target.
--connection        Specifies the communication channel to be used.
--derivative        Specifies the device.
--disable_memory_cache Disables the memory cache in the FET debugger.
--eem               Specifies the level of Enhanced Emulation Mode.
--erase_main        Erases main flash memory before download.
--erase_main_and_info Erases the main and Information flash memories before download.
--erase_retain      Retains unchanged memory during download.
--hardware_multiplier Generates code for the hardware multiplier peripheral unit.
--hwmult_type      Specifies the type of hardware multiplier to be used.
--lptx              Specifies the parallel port to be used.
--port              Specifies the serial port to be used.
--protocol          Specifies the debug protocol to be used.
**Descriptions of C-SPY command line options**

This section gives detailed reference information about each `cspybat` option and each option available to the C-SPY drivers.

---

**--allow_access_to_BSL**

**Syntax**

`--allow_access_to_BSL`

**Applicability**

The C-SPY FET Debugger driver.

**Description**

Use this option to enable erase/write access to BSL flash memory.

*Project>Options>Debugger>FET Debugger>Download>Allow erase/write access to BSL flash memory*

---

**--allow_locked_flash_access**

**Syntax**

`--allow_locked_flash_access`

**Applicability**

The C-SPY FET Debugger driver.

**Description**

Use this option to enable erase/write access to locked flash memory.

*Project>Options>Debugger>FET Debugger>Download>Allow erase/write access to locked flash memory*
Descriptions of C-SPY command line options

--attach
Syntax: --attach
Applicability: The C-SPY FET Debugger driver.
Description: Use this option to make the debugger attach to a running application at its current location, without resetting the target system.

-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.

--connection
Syntax: --connection Port
Parameters:
Port The communication channel to be used; choose between: ti_usb, ti_lpt, jlink, olimex, olimex_parallel, elprotronic, softbaugh_lpt, softbaugh_usb, and softbaugh_usb_pro.
Applicability
The C-SPY FET Debugger driver.

Description
Use this option to specify the communication channel to be used between C-SPY and the target system.

Project>Options>Debugger>FET Debugger>Setup>Connection

--core

Syntax
--core {430X|430Xv2}

Parameters
430X|430Xv2 The core you are using. For a description of the cores, see the IAR C/C++ Compiler Reference Guide for MSP430.

Applicability
All C-SPY drivers.

Description
Use this option to specify the core you are using. This option reflects the corresponding compiler option.

--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 167.
Descriptions of C-SPY command line options

---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 {sim|fet}
Parameters
sim Specifies the simulator driver.
fet Specifies the FET debugger driver.
Applicability
All C-SPY drivers.
Description
Use this option to specify the C-SPY driver to be used.

--derivative
Syntax
--derivative device
Parameters
device Specifies the device to be used.
Applicability
The C-SPY FET Debugger driver.
Description
Use this option to select the device for which you will build your application.

C-SPY® Debugging Guide
for MSP430
--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.

--disable_memory_cache

Syntax: `--disable_memory_cache`

Applicability: The C-SPY FET Debugger driver.

Description: Use this option to disable the memory cache in the FET debugger.

To set this option, choose **Project>Options>Debugger>FET Debugger>Setup>Disable memory cache**

--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**
Descriptions of C-SPY command line options

--eem
Syntax: `--eem {level}`
Parameters: 
- `level` Implementation level of Enhanced Emulation Module. Choose between:
  - EMEX_HIGH
  - EMEX MEDIUM
  - EMEX LOW
  - EMEX NONE

Applicability: The C-SPY FET Debugger driver.
Description: Use this option to specify the implementation level of Enhanced Emulation Module for a MSP430 device. The default value is EMEX_NONE. The value of this option depends on the device you are using.

This option is not available in the IDE.

--erase_main
Syntax: `--erase_main`
Applicability: The C-SPY FET Debugger driver.
Description: Use this option to erase the main flash memory before download. The Information memory is not erased.

Project>Options>Debugger>FET Debugger>Download>Erase main memory

--erase_main_and_info
Syntax: `--erase_main_and_info`
Applicability: The C-SPY FET Debugger driver.
Description
Use this option to erase both flash memories—main and Information memory—before download.

Project>Options>Debugger>FET Debugger>Download>Erase main and Information memory

--erase_retain

Syntax
--erase_retain

Applicability
The C-SPY FET Debugger driver.

Description
Use this option to make C-SPY read the main and Information memories into a buffer. Only the flash segments that are needed are erased. If data that is to be written into a segment matches the data on the target, the data on the target is left as is, and no download is performed. The new data effectively replaces the old data, and unchanged old data is retained.

Project>Options>Debugger>FET Debugger>Download>Retain unchanged memory

--hardware_multiplier

Syntax
--hardware_multiplier {16|32}

Parameters

16
The size of the multiplicands in bits.

Note: This parameter can only be used in combination with the --hwmult_type parameters 1 and 2.

32
The size of the multiplicands in bits.

Note: This parameter can only be used in combination with the --hwmult_type parameters 4 and 8.

Applicability
The C-SPY FET Debugger driver.

Description
Use this option to generate code for the MSP430 hardware multiplier peripheral unit. Use this option only when you have chosen a device with a hardware multiplier.
Note: This option requires that you also specify the `--hwmult_type` option.

To set related options, choose:

Project>Options>General Options>Target>Hardware multiplier

---

--hwmult_type

Syntax

```
--hwmult_type {1|2|4|8}
```

Parameters

1 16 bits

Note: This parameter can only be combined with the `--hardware_multiplier parameter 16`.

2 16 bits, the 2xx Family

Note: This parameter can only be combined with the `--hardware_multiplier parameter 16`.

4 32 bits

Note: This parameter can only be combined with the `--hardware_multiplier parameter 32`.

8 32 bits, the 5xx Families

Note: This parameter can only be combined with the `--hardware_multiplier parameter 32`.

Applicability

The C-SPY FET Debugger driver.

Description

Use this option to generate code for the MSP430 hardware multiplier peripheral unit. Use this option only when you have chosen a device with a hardware multiplier.

Note: This option requires that you also specify the `--hardware_multiplier` option.

To set related options, choose:

Project>Options>General Options>Target>Hardware multiplier
The C-SPY Command Line Utility—cspybat

--lpt

Syntax:  --lpt x

Applicability: The C-SPY FET Debugger driver.

Description: Use this option to specify which parallel port the FET Debugger is connected to. x can be 1, 2, or 3.

Project>Options>Debugger>FET Debugger>Setup>Parallel port

--macro

Syntax:  --macro filename

Parameters: filename

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 206.

--mapu

Syntax:  --mapu

Applicability: Sent to 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 ranges. If any such access is found, a message will be printed on stdout and the execution will stop.

See also: Memory access checking, page 122.

To set related options, choose:

Simulator>Memory Access Setup
--odd_word_check

Syntax

--odd_word_check

Applicability
The C-SPY simulator driver.

Description
Use this option to make the simulator issue a warning if there is a word access to an odd address.

Project>Options>Debugger>Simulator>Setup>Check for word access on odd address

-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 37.

--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 430bat.dll located in the 430\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 csybpat specifically. This means that the C-SPY plugin modules located in the common\plugin directory cannot normally be used with csybpat.

--port
Syntax
--port port
Parameters
port The serial port to be used, can be COMx or Automatic. x is the number of the COM port.
Applicability The C-SPY FET Debugger driver.
Description Use this option to specify which serial port the FET Debugger is connected to.
Project>Options>Debugger>FET Debugger>Setup>Connection

--protocol
Syntax
--protocol {spy-bi-wire|spy-bi-wire-jtag|4wire}
Parameters
spy-bi-wire The Spy-Bi-Wire JTAG protocol.
spy-bi-wire-jtag 4-wire JTAG protocol for devices that also support Spy-Bi-Wire.
4wire The ordinary 4-wire JTAG protocol.
Applicability The C-SPY FET Debugger driver.
Description Use this option to specify the debug protocol. Spy-Bi-Wire works for the parallel port FET module and the TI USB FET module.
Project>Options>Debugger>FET Debugger>Setup>Debug protocol
--set_exit_breakpoint

Syntax: `--set_exit_breakpoint`

Applicability: The C-SPY FET Debugger driver.

Description: Use this option to set a system breakpoint for exit.

Project>Options>Debugger>FET Debugger>Breakpoints>System breakpoints on>exit

--set_getchar_breakpoint

Syntax: `--set_getchar_breakpoint`

Applicability: The C-SPY FET Debugger driver.

Description: Use this option to set a system breakpoint for getchar.

Project>Options>Debugger>FET Debugger>Breakpoints>System breakpoints on>getchar

--set_putchar_breakpoint

Syntax: `--set_putchar_breakpoint`

Applicability: The C-SPY FET Debugger driver.

Description: Use this option to set a system breakpoint for putchar.

Project>Options>Debugger>FET Debugger>Breakpoints>System breakpoints on>putchar

--settlingtime

Syntax: `--settlingtime=milliseconds`

Applicability: The C-SPY FET Debugger driver.
Description

Use this option to specify the delay between switching on the target VCC and starting the identification of the MSP430 device. Give the value in milliseconds in the range 0-9999 ms. This option can only be used with a USB connection.

**Project>Options>Debugger>FET Debugger>Setup>Target VCC>Settling time (in ms)**

---

**--silent**

**Syntax**

```
--silent
```

**Applicability**

Sent to cspybat.

**Description**

Use this option to omit the sign-on message.

---

**--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.

---

**--use_emulated_breakpoints**

**Syntax**

```
--use_emulated_breakpoints
```

**Applicability**

The C-SPY FET Debugger driver.

**Description**

Use this option to allow C-SPY to use emulated breakpoints.

**Project>Options>Debugger>FET Debugger>Breakpoints>Use software breakpoints>Emulated breakpoints**
**--use_virtual_breakpoints**

Syntax: `--use_virtual_breakpoints`

Applicability: The C-SPY FET Debugger driver.

Description: Use this option to allow C-SPY to use virtual breakpoints.

Project>Options>Debugger>FET Debugger>Breakpoints>Use software breakpoints>Virtual breakpoints

**--vccvoltage**

Syntax: `--vccvoltage=volts`

Applicability: The C-SPY FET Debugger driver.

Description: Use this option to specify the voltage provided by the USB interface. Give the value in Volts with one decimal’s precision in the range 1.0-4.0 V. This option can only be used with a USB connection.

Project>Options>Debugger>FET Debugger>Setup>Target VCC>Target VCC (in Volt)

**--verify_all**

Syntax: `--verify_all`

Applicability: The C-SPY FET Debugger driver.

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>FET Debugger>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 general debugger options
- Reference information on C-SPY driver options.

Setting debugger options

Before you start the C-SPY debugger you must 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 reference information on the generic options, see:
   - Setup, page 270
   - Images, page 272
   - Extra Options, page 273
   - Plugins, page 273.
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 sets of option pages appear.

C-SPY driver | Available options pages |
-------------|-------------------------|
FET Debugger | Setup for FET Debugger, page 275 |
|              | Download, page 277 |
|              | Breakpoints, page 278 |
Simulator     | Setup for simulator, page 279 |

Table 40: Options specific to the C-SPY drivers you are using
To restore all settings to the default factory settings, click the **Factory Settings** button.

When you have set all the required options, click **OK** in the **Options** dialog box.

---

**Reference information on general debugger options**

This section gives reference information on general C-SPY debugger options, located on these option pages:

- **Setup**, page 270
- **Images**, page 272
- **Extra Options**, page 273
- **Plugins**, page 273.

**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.

---

**Driver**

Selects the C-SPY driver for the target system you are using:

- **FET Debugger**
- **Simulator**
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 file is 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 details about the device description file, see Modifying a device description file, page 40.

Device description files for each MSP430 device are provided in the directory 430\config and have the filename extension ddf.
Images

The Images options control the use of additional debug files to be downloaded.

Download Extra Image

Enables the options for the additional debug file to be downloaded:

Path
Specify the debug file to be downloaded. A browse button is available for your convenience.

Offset
An integer that specifies the destination address for the downloaded image.

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 225.

For more information, see Loading multiple images, page 39.
Extra Options

The **Extra Options** page provides you with a command line interface to C-SPY.

**Figure 98: Debugger extra options**

**Use command line options**

Specify additional command line arguments to be passed to C-SPY (not supported by the GUI).

Plugins

The **Plugins** options select the C-SPY plugin modules to be loaded and made available during debug sessions.

**Figure 99: 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.

<table>
<thead>
<tr>
<th>Description</th>
<th>Describes the plugin module.</th>
</tr>
</thead>
<tbody>
<tr>
<td>Location</td>
<td>Informs about the location of the plugin module. Generic plugin modules are stored in the <code>common\plugins</code> directory. Target-specific plugin modules are stored in the <code>430\plugins</code> directory.</td>
</tr>
<tr>
<td>Originator</td>
<td>Informs about the originator of the plugin module, which can be modules provided by IAR Systems or by third-party vendors.</td>
</tr>
<tr>
<td>Version</td>
<td>Informs about the version number.</td>
</tr>
</tbody>
</table>

---

**Reference information on C-SPY driver options**

This section gives reference information on C-SPY driver options, located on these option pages:

- Setup for FET Debugger, page 275
- Download, page 277
- Breakpoints, page 278
- Setup for simulator, page 279.
Setup for FET Debugger

The FET Debugger Setup options control the C-SPY FET Debugger driver.

Connection

Controls the communication between C-SPY and the target device.

The C-SPY FET Debugger can communicate with the target device via a number of different emulators. Select the emulator you are using.

If your emulator is connected to the host computer via a parallel port, you must also specify which parallel port to use: Parallel port 1, Parallel port 2, or Parallel port 3. If your emulator is connected to the host computer via a USB port, the debugger will automatically connect to the correct port.

Some emulator drivers support multiple emulators connected to the same host computer. Each emulator requires its own instance of IAR Embedded Workbench and each instance must identify its emulator. To identify an emulator, click the browse button to display a list of all detected emulators. To identify a connection, click the port in the list and the Mode LED on the attached emulator will light up.

Debug protocol

Determines the debug interface to use:

Automatic selection Selects the debug interface automatically. (If Connection is set to Automatic, C-SPY correctly determines which interface to use.)
Reference information on C-SPY driver options

**Attach to running target**

Makes the debugger attach to a running application at its current location, without resetting the target system.

*Note: Attach to running target* must be deselected when you download the application for the first time.

**To avoid unexpected behavior when using this option:**

1. Deselect the option *Attach to running target*.
2. Click the *Download and debug* button to start the debugger.
3. Make sure that the command *Leave target running* is selected on the *Emulator* menu.
4. Start the execution.
5. Stop the debugger.
6. Select the option *Attach to running target*.
7. Choose *Project>Options>Debugger>Setup* and deselect the *Run to main* option.
8. Click the *Debug without downloading* button to start the debugger.

**Disable memory cache**

Disables the memory cache in the FET debugger.

**Target VCC**

Specify the voltage provided by the USB interface:

- **Override default**
  - Overrides the default voltage. To see what the default voltage is, see *Device Information window*, page 50.

- **Target VCC**
  - Specify the voltage with one decimal’s precision in the range 1.0–4.0 V. This option can only be used with a USB connection.
Debug LPMx5

 Enables debugging of applications that use the LPMx5 low-power mode.

Download

By default, C-SPY downloads the application to RAM or flash when a debug session starts. The Download options let you modify the behavior of the download.

**Settling time**

Specify a delay that will be used between switching on the target VCC and starting the identification of the MSP430 device.

**Debug LPMx5**

Enables debugging of applications that use the LPMx5 low-power mode.

**Download**

By default, C-SPY downloads the application to RAM or flash when a debug session starts. The Download options let you modify the behavior of the download.

![Figure 101: FET Debugger download options](image)

**Verify download**

Use this option to verify that the downloaded code image can be read back from target memory with the correct contents.

**Allow erase/write access to locked flash memory**

Enables erase/write access to Info Segment A. This option can only be used with an MSP430F2xx device.

**Allow erase/write access to BSL flash memory**

Enables erase/write access to BSL flash memory. This option can only be used with an MSP430F5xx device.

**External code download**

Saves user code to external SPI memory.
**Reference information on C-SPY driver options**

**Erase main memory**
Erases only the main flash memory before download. The Information memory is not erased.

**Erase main and Information memory**
Erases both flash memories—main and Information memory—before download.

**Retain unchanged memory**
Reads the main and Information memories into a buffer. Only the flash segments that are needed are erased. If data that is to be written into a segment matches the data on the target, the data on the target is left as is, and nothing is downloaded. The new data effectively replaces the old data, and unchanged old data is retained.

**JTAG password**
If a JTAG device is password-protected, supply the needed password here.

**Breakpoints**

The **Breakpoints** options control the use of breakpoints.

![Breakpoints](image)

*Figure 102: FET Debugger breakpoints options*
Use software breakpoints

Allows C-SPY to use software breakpoints when all available hardware breakpoints have been used. Choose between:

**Virtual breakpoints**  Makes C-SPY use virtual breakpoints.

**Emulated breakpoints**  Makes C-SPY use emulated breakpoints: the instruction where the breakpoint is set will be replaced by a special instruction that the debugger recognizes.

For information, see *Breakpoints in the C-SPY FET Debugger driver*, page 90.

System breakpoints on

Controls the use of system breakpoints in the CLIB runtime environment. If the C-SPY Terminal I/O window is not required or if you do not need a breakpoint on the `exit` label, you can save hardware breakpoints by not reserving system breakpoints. Select or deselect the options `exit`, `putchar`, and `getchar` respectively, if you want, or do not want, C-SPY to use system breakpoints for these. For more information, see *Breakpoint consumers*, page 91.

In the DLIB runtime environment, C-SPY will always set a system breakpoint on the `__DebugBreak` label. You cannot disable this behavior.

Setup for simulator

The simulator **Setup** options control the behavior of the C-SPY Simulator.
Peripheral simulation

These options set up peripheral simulation, which requires a plug-in from a third-party vendor. For information, see the PDF `EW_PeripheralSimulationGuide.pdf` in the `EW_DIR\430\doc` directory and the examples in the `EW_DIR\430\plugins\simulation` directory.

Check for word access on odd address

Makes the simulator issue a warning if there is a word access to an odd address.
Additional information on C-SPY drivers

This chapter describes the additional menus and features provided by the C-SPY® drivers. More specifically, this means:

- Reference information on the C-SPY simulator
- Reference information on the C-SPY FET Debugger driver
- 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 281.

Simulator menu

When you use the simulator driver, the Simulator menu is added to the menu bar.
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 198.

**Forced Interrupts**  
Opens a window from where you can instantly trigger an interrupt; see *Forced Interrupt window*, page 202.

**Interrupt Log**  
Opens a window which displays the status of all defined interrupts; see *Interrupt Log window*, page 203.

**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 137.

**Trace**  
Opens a window which displays the collected trace data; see *Trace window*, page 146.

**Function Trace**  
Opens a window which displays the trace data for function calls and function returns; see *Function Trace window*, page 148.

**Function Profiler**  
Opens a window which shows timing information for the functions; see *Function Profiler window*, page 163.

**Breakpoint Usage**  
Displays a dialog box which lists all active breakpoints; see *Breakpoint Usage dialog box*, page 100.

**Timeline**  
Opens a window which displays trace data for interrupts and for the call stack as graphs; see *Timeline window*, page 149.

---

**Reference information on the C-SPY FET Debugger driver**

This section gives additional reference information on the C-SPY FET Debugger driver, reference information not provided elsewhere in this documentation.

More specifically, this means:

- *Emulator menu*, page 283
- *General Clock Control dialog box*, page 285
- *Extended Clock Control dialog box*, page 286.
Emulator menu

When you are using the C-SPY FET Debugger driver, the Emulator menu is added to the menu bar.

![Emulator menu](image)

These commands are available on the menu:

* **Connected device**
  The name of the device used for debugging.

* **Device information**
  Opens the Device Information window with information about the device used for debugging, see Device Information window, page 50.

* **Release JTAG on Go**
  Sets the JTAG drivers in tri-state so that the device is released from JTAG control—TEST pin is set to GND—when Go is activated.

* **Resynchronize JTAG**
  Regains control of the device.
  It is not possible to resynchronize JTAG while the device is operating.
Reference information on the C-SPY FET Debugger driver

Init New Device  Initializes the device according to the settings in the Projects>Options>FET Debugger category. The current program file is downloaded to the device memory, and the device is then reset. This command can be used to program multiple devices with the same program from within the same C-SPY session.

It is not possible to choose Init New Device while the device is operating, thus the command will be dimmed.

Secure - Blow JTAG Fuse  Blows the fuse on the target device. After the fuse is blown, no communication with the device is possible.

Only available if you are using a USB-connected debug probe.

Breakpoint Usage  Displays a dialog box which lists all active breakpoints; see Breakpoint Usage dialog box, page 100.

Advanced> Clock Control  Lets you control the clocks on the device. Depending on the hardware support, either the General Clock Control dialog box or the Extended Clock Control dialog box is displayed; see General Clock Control dialog box, page 285 and Extended Clock Control dialog box, page 286, respectively.

Advanced> Emulation Mode  Specifies the device to be emulated. The device must be reset (or reinitialized by using the menu command Init New Device) following a change to the emulation mode.

Advanced> Memory Dump  Displays the Memory Dump dialog box, which lets you write device memory contents to a file; see Memory Dump dialog box, page 140.

Advanced> Breakpoint Combiner  Displays the Breakpoint combiner dialog box, which lets you combine two already defined breakpoints; see Breakpoint combiner dialog box, page 115.

State Storage Control  Opens the State Storage Control window, which lets you define the use of the state storage module; see State Storage Control window, page 174.

State Storage Window  Opens the State Storage window which contains state storage information according to your definitions; see State Storage window, page 175.
Sequencer Control

Opens the Sequencer Control window, which lets you define a state machine; see Sequencer Control window, page 181.

“Power on” Reset

The device is reset by cycling power to the device.

GIE on/off

Clears the General Interrupt Enable bit (GIE) in the Processor Status register.

Cycle Counter for 5xx

Opens the Cycle Counter 5xx Control window, where you can define and use the advanced cycle counter on the MSP430x5xx devices; see Cycle Counter 5xx Control window, page 187.

Leave Target Running

Leaves the application running on the target hardware after the debug session is closed.

Force Single Stepping

Forces single step debugging.

Force hardware RST/NMI

Forces an RST/NMI clear reset when the Reset button is pressed.

Note: Not all Emulator>Advanced submenus are available on all MSP430 devices.

General Clock Control dialog box

The General Clock Control dialog box is available from the Emulator menu.

Use this dialog box to control the clocks of the device.

Select the clock modules you want to stop when the execution stops. The other clocks will keep running.

Which clock modules that are displayed depends on the available clocks on the connected device.

Figure 106: General Clock Control dialog box
Extended Clock Control dialog box

The Extended Clock Control dialog box is available from the Emulator menu.

![Extended Clock Control dialog box](image)

**Figure 107: Extended Clock Control dialog box**

Use this dialog box for module level control over the clocks of the device.

Select the clock modules you want to stop when the execution stops. The other clocks will keep running.

Which clock modules that are displayed depends on the available clocks on the connected device.

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.

To troubleshoot the Flash Emulation Tool, see appendix A in the document *MSP-FET430 Flash Emulation Tool User’s Guide* at the Texas Instruments web site, [www.ti.com](http://www.ti.com). The document has the literature number SLAU138K.

**THE DEVICE PORT PINS DO NOT WORK**

On some MSP430 devices, the device port pins are shared with the JTAG pins that C-SPY uses to debug the device. Normally, C-SPY maintains the pins in JTAG mode so that the device can be debugged. During this time the port functionality of the shared pins is not available.
To release the JTAG pins:
Choose **Emulator>Release JTAG on Go** to set the JTAG drivers to tri-state and release the device from JTAG control (the TEST pin is set to GND) when **Go** is activated. Any active on-chip breakpoints are retained and the shared JTAG port pins revert to their port functions.

**Note:** Be aware of the following:
- The JTAG pins will only be released if there are \( n \) or fewer active breakpoints.
- When you measure the electrical current of the device, the JTAG control signals must be released, otherwise the device will be powered by the signals on the JTAG pins and the measurements will be incorrect.
- If you release the JTAG pins, C-SPY has no access to the device and cannot determine if an active breakpoint has been triggered. C-SPY must be manually told to stop the device, at which time the state of the device will be determined (that is, has a breakpoint been reached?).

**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 correct linker configuration file.

If you are using the IAR Embedded Workbench, 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:**
1. Choose **Project>Options**.
2. Select the **Linker** category.
3. Click the **Config** tab.
4. 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; see Setup for FET Debugger, page 275.

Examine the linker configuration file to make sure that the application has not been linked to the wrong address.
Abort (Report Assert option) ........................................ 68
Absolute address (Type setting) .................................... 114
absolute location, specifying for a breakpoint ..................... 114
Access type (Advanced Trigger breakpoints option) ............. 113
Access Type (Data breakpoints option) .......................... 104
Access type (Edit Memory Access option) ........................ 139
Access Type (Immediate breakpoints option) ...................... 106
Access (Conditional breakpoints option) .......................... 110
Action (Advanced Trigger breakpoints option) .................... 113
Action (Code breakpoints option) .................................. 102
Action (Conditional breakpoints option) .......................... 111
Action (Data breakpoints option) ................................... 105
Action (Immediate breakpoints option) ............................ 106
Action (Sequencer option) ............................................. 182
Add to Watch Window (Symbolic Memory window context menu) ........................................ 132
Add (Watch window context menu) ................................... 79
Address Range (Find in Trace option) .............................. 156
Advanced Trigger breakpoints dialog box ......................... 112
advanced trigger breakpoints, overview .......................... 89
Advanced (Sequencer option) .......................................... 182
Advanced>Breakpoint Combiner (Emulator menu) ................ 284
Advanced>Clock Control (Emulator menu) ........................ 284
Advanced>Emulation Mode (Emulator menu) ...................... 284
Advanced>Memory Dump (Emulator menu) ........................ 284
Allow erase/write access to locked flash memory (C-SPY FET option) ........................................ 277
Allow erase/write access to BSL flash memory (C-SPY FET option) ........................................ 277
--allow_access_to_BSL (C-SPY command line option) ........ 255
--allow_locked_flash_access (C-SPY command line option) ........................................ 255
Ambiguous symbol (Resolve Symbol Ambiguity option) ......... 85
AND (Operation setting) .............................................. 129
application flow, monitoring ........................................ 171
Applications (Cycle Counter 5xx option) ........................... 188
application, built outside the IDE .................................... 38
assembler symbols, using in C-SPY expressions .................... 73
assembler variables, viewing ........................................ 75
assumptions, programming experience ............................. 19
--attach (C-SPY command line option) ............................ 256
Attach to running target (C-SPY FET option) ....................... 276
Auto Scroll (Timeline window context menu) ...................... 151
Auto window .......................................................... 77
Autostep settings dialog box ......................................... 69
Autostep (Debug menu) ................................................. 45

B

-B (C-SPY command line option) .................................... 256
--backend (C-SPY command line option) ........................ 256
backtrace information
generated by compiler ............................................. 55
viewing in Call Stack window ........................................ 62
batch mode, using C-SPY in ........................................... 251
Big Endian (Memory window context menu) ......................... 125
blocks, in C-SPY macros ............................................. 216
bold style, in this guide .............................................. 23
Break At (Advanced Trigger breakpoints option) ................ 112
Break At (Code breakpoints option) ................................ 101
Break At (Conditional breakpoints option) ........................ 109
Break At (Data breakpoints option) ................................ 104
Break At (Immediate breakpoints option) ........................ 106
Break At (Log breakpoints option) .................................. 103
Break (Debug menu) ................................................... 45
Breakpoint combiner dialog box ..................................... 115
breakpoint condition, example ..................................... 97
Breakpoint Usage dialog box ......................................... 100
Breakpoint Usage (Emulator menu) ................................. 284
Breakpoint Usage (Simulator menu) ................................. 282
breakpoints
advanced trigger ..................................................... 112
example ......................................................... 234
code .......................................................... 101
element ......................................................... 235
command prompt icon, in this guide | 23
computer style, typographic convention | 23
Condition (Conditional breakpoints option) | 111
conditional breakpoints
  overview | 89
  triggering state storage | 173
Conditional breakpoints dialog box | 109
conditional statements, in C-SPY macros | 215
Conditions (Code breakpoints option) | 102
Conditions (Data breakpoints option) | 105
Conditions (Log breakpoints option) | 103
  --connection (C-SPY command line option) | 256
Connection (C-SPY FET option) | 275
current position, in Disassembly window | 59
conventions, used in this guide | 22
Copy Window Contents
  (Disassembly window context menu) | 61
  Copy (Debug Log window context menu) | 67
  Copy (Operation setting) | 129
copyright notice | 2
--core (C-SPY command line option) | 257
CPU cycles, counting (MSP430F5xx devices) | 184, 186
cspybat | 251
current position, in C-SPY Disassembly window | 59
Current state (Sequencer option) | 182
cursor, in C-SPY Disassembly window | 59
Cycle Counter for 5xx (Emulator menu) | 285
Cycle counter values (Cycle Counter 5xx option) | 188
Cycle Counter 5xx Control window | 187
  --cycles (C-SPY command line option) | 258
C-SPY
  batch mode, using in | 251
  command line options | 255
debugger systems, overview of | 28
differences between drivers | 30
environment overview | 25
plugin modules, loading | 37
setting up | 35-36
starting the debugger | 37
C-SPY drivers
  FET Debugger | 32
  simulator | 31
  specifying | 270
C-SPY expressions | 72
  evaluating | 83
  in C-SPY macros | 215
tooltip watch, using | 71
Watch window, using | 71
C-SPY macro "__message"
  style (Log breakpoints option) | 103
C-SPY macros | 205
  blocks | 216
  conditional statements | 215
  C-SPY expressions | 215
dialog box | 209
  using | 209
  examples | 207
  checking status of register | 211
  checking the status of WDT | 211
  creating a log macro | 212
  executing | 207
  connecting to a breakpoint | 212
  using Quick Watch | 211
  using setup macro and setup file | 210
  functions | 74, 213
  loop statements | 215
  macro statements | 215
  setup macro file | 206
  executing | 210
  setup macro functions | 206
  summary | 218
  system macros, summary of | 219
  using | 205
  variables | 74, 214
C-SPY options | 269
Breakpoints | 278
Download | 277
Extra Options | 273
Images | 272

291
Plugins .................................................. 273
Setup (FET) ............................................. 275
Setup (general) ....................................... 270
Setup (simulator) ................................. 279
C-SPYLink ............................................. 30
C++ terminology ................................. 22
-d (C-SPY command line option) .......... 258
data breakpoints, overview ................. 89
Data Coverage (Memory window context menu) .......... 126
data coverage, in Memory window ............ 124
ddf (filename extension), selecting a file .... 37
Debug file segment information
(Use ranges based on setting) .................. 138
Debug info only (Download Extra Image setting) .................. 272
Debug Log window ............................... 66
Debug Log window context menu .............. 67
Debug LPMx5 (C-SPY FET option) .......... 277
Debug menu (C-SPY main window) .......... 44
Debug protocol (C-SPY FET option) ........ 275
Debug (Report Assert option) ................. 68
debugger concepts, definitions of .......... 28
debugger drivers
FET Debugger .......................................... 32
simulator ............................................. 31
debugger system overview ..................... 28
debugging projects
externally built applications ................. 38
loading multiple images ..................... 39
debugging, RTOS awareness .................. 27
__delay (C-SPY system macro) ............... 222
Delay (Autostep Settings option) .......... 69
Delete (Breakpoints window context menu) .......... 99
--derivative (C-SPY command line option) ........ 258
Description (Edit Interrupt option) ........ 201
description (interrupt property) .......... 201
Device description file (debugger option) .......... 271
Device description file (Use ranges based on setting) .......... 137
device description files ......................... 37
definition of ........................................ 40
memory zones ...................................... 121
modifying .......................................... 40
register zone ........................................ 121
specifying interrupts ............................ 229
Device Information window .................... 50
Device information (Emulator menu) ........ 283
Disable All (Breakpoints window context menu) .......... 99
Disable memory cache (C-SPY FET option) .......... 276
Disable (Breakpoints window context menu) .......... 99
__disableInterrupts (C-SPY system macro) .......... 222
--disable_interrupts (C-SPY command line option) .......... 259
--disable_memory_cache (C-SPY command line option) .......... 259
Disassembly window .............................. 58
disclaimer ........................................... 2
DLIB, documentation ............................. 21
DMA load, measuring (MSP430F5xx devices) .......... 185
do (macro statement) ............................. 216
document conventions ............................ 22
documentation
overview of guides .............................. 21
overview of this guide .......................... 20
this guide ............................................ 19
Download Extra Image (debugger option) .......... 272
Download (FET debugger options) ........ .... 277
--download_only (C-SPY command line option) .......... 259
Driver (debugger option) ....................... 270
__driverType (C-SPY system macro) .......... 223

disclaimer ........................................... 2
DLIB, documentation ............................. 21
DMA load, measuring (MSP430F5xx devices) .......... 185
do (macro statement) ............................. 216
document conventions ............................ 22
documentation
overview of guides .............................. 21
overview of this guide .......................... 20
this guide ............................................ 19
Download Extra Image (debugger option) .......... 272
Download (FET debugger options) ........ .... 277
--download_only (C-SPY command line option) .......... 259
Driver (debugger option) ....................... 270
__driverType (C-SPY system macro) .......... 223
edition, of this guide ........................................ 2
--eem (C-SPY command line option) ................. 260
Embedded C++ Technical Committee .................. 22
emulated breakpoints ..................................... 91
  configuring ............................................. 279
Emulator menu (reference information) ................ 283
Enable All (Breakpoints window context menu) ....... 99
Enable interrupt simulation (Interrupt Setup option) .... 199
Enable Log File (Log File option) ...................... 68
Enable (Breakpoints window context menu) .......... 99
Enable (Timeline window context menu) ............. 152
__enableInterrupts (C-SPY system macro) ............ 223
Enable/Disable Breakpoint
  (Call Stack window context menu) ................. 63
Enable/Disable Breakpoint
  (Disassembly window context menu) ............... 61
Enable/Disable (Trace toolbar) ....................... 147
End address (Edit Memory Access option) .......... 139
Enter Location dialog box ................................ 114
Erase main and Information memory
  (C-SPY FET option) .................................. 278
Erase main memory (C-SPY FET option) .............. 278
--erase_main (C-SPY command line option) .......... 260
--erase_main_and_info (C-SPY command line option) .... 260
--erase_retain (C-SPY command line option) ........ 261
__evaluate (C-SPY command line option) ............. 261
examples
  C-SPY macros ......................................... 207
  interrupts ............................................. 196
  macros
    checking status of register ....................... 211
    checking status of WDT ............................ 211
    creating a log macro ................................ 212
    using Quick Watch .................................. 211
    performing tasks and continue execution .......... 97
    timer interrupt .................................... 196
  execUserExit (C-SPY setup macro) ............... 218
  execUserPreload (C-SPY setup macro) ............. 218
  execUserReset (C-SPY setup macro) ............... 218
  execUserSetup (C-SPY setup macro) ............... 218
  executed code, tracking progress of .............. 167
  execution history, tracing ......................... 145
  execution, stopping using the sequencer .......... 177
  Expression (Type setting) ......................... 114
  expressions. See C-SPY expressions
  Extended Clock Control dialog box ................. 286
  External code download (C-SPY FET option) ....... 277
  Extra Options (C-SPY) ................................ 273

F

FET Debugger (C-SPY driver) .......................... 32
  communication ....................................... 286
  features ............................................. 30
  hardware installation ................................ 34
FET Debugger (debugger option) ....................... 270
File format (Memory Save option) ..................... 127
file types
  device description, specifying in IDE .............. 37
  macro ................................................. 37
  filename extensions
    ddf, selecting device description file ............ 37
    mac, using macro file ............................ 37
    sfr, register definitions for C-SPY ............... 135
Filename (Memory Restore option) ..................... 128
Filename (Memory Save option) ....................... 127
Fill dialog box ......................................... 128
Find in Trace dialog box ................................ 155
Find in Trace window .................................. 156
Find (Memory window context menu) ................... 126
Find (Trace toolbar) .................................. 147
first activation time (interrupt property)
  definition of ........................................ 193
First activation (Edit Interrupt option) ........... 201
flash memory, load library module to ............... 225
flash memory, single-stepping in C-SPY emulator .... 54
for (macro statement) ................................ 215
Force hardware RST/NMI (Emulator menu) ............ 285
Force Single Stepping (Emulator menu) ............. 285
Forced Interrupt window ........................................ 202
Forced Interrupts (Simulator menu) ................................. 282
formats, C-SPY input ............................................. 27
Function Profiler window ......................................... 163
Function Profiler (Simulator menu) ................................. 282
Function Trace window ............................................ 148
Function Trace (Simulator menu) ................................. 282
functions, C-SPY running to when starting .................. 36, 271
fuse, blowing on the target device ............................... 284

G
General Clock Control dialog box ................................. 285
GIE on/off (Emulator menu) ..................................... 285
Go to Source (Breakpoints window context menu) ...... 99
Go to Source (Call Stack window context menu) ...... 63
Go To Source (Timeline window context menu) ...... 152
Go (Debug menu) ................................................. 44, 54

H
hardware breakpoints in FET debugger ......................... 91
--hardware_multiplier (C-SPY command line option) ... 261
highlighting, in C-SPY ............................................. 55
Hold time (Edit Interrupt option) ................................. 201
hold time (interrupt property), definition of ............ 193
--hwmul_type (C-SPY command line option) ............. 262

I
icons, in this guide ................................................. 23
if else (macro statement) ........................................ 215
if (macro statement) ............................................. 215
Ignore (Report Assert option) .................................... 69
illegal memory accesses, checking for .................... 122
Images window ..................................................... 47
Images, loading multiple ........................................ 272
immediate breakpoints, overview ............................... 89
Include (Log File option) ........................................ 68
Init New Device (Emulator menu) ............................... 284
input formats, C-SPY ............................................. 27
Input Mode dialog box ............................................. 64
input, special characters in Terminal I/O window ........ 64
installation directory ............................................. 23
Instruction Profiling (Disassembly window context menu) 60
Intel-extended, C-SPY input format ......................... 27, 30
Interrupt Log window ............................................ 203
Interrupt Log (Simulator menu) ................................. 282
Interrupt Setup dialog box ...................................... 198
Interrupt Setup (Simulator menu) ............................... 282
interrupt system, using device description file ........ 195
Interrupt (Edit Interrupt option) ............................... 200
Interrupt (Timeline window context menu) ............... 152

J
JTAG password (C-SPY FET option) ......................... 278
JTAG pins, shared with port pins ............................... 286

L
labels (assembler), viewing ..................................... 75
LCD Settings dialog box ....................................... 66
LCD window ..................................................... 65
Leave Target Running (Emulator menu) .................. 285
Length (Fill option) ............................................. 129
lightbulb icon, in this guide ..................................... 23
Little Endian (Memory window context menu) ........... 125
Live Watch window ............................................... 79
__loadImage (C-SPY system macro).......................... 225
loading multiple images ..................................... 39
list of currently loaded .................................... 47
Locals window ............................................. 77
log breakpoints, overview ................................ 88
Log File dialog box ........................................ 67
Logging>Set Log file (Debug menu) ...................... 46
Logging>Set Terminal I/O Log file (Debug menu) ...... 46
loop statements, in C-SPY macros ....................... 215
--lpt1 (C-SPY command line option) .................... 263
--lpt2 (C-SPY command line option) .................... 263
--lpt3 (C-SPY command line option) .................... 263
mac (filename extension), using a macro file ........ 37
--macro (C-SPY command line option) .................. 263
Macro Configuration dialog box ......................... 209
macro files, specifying .................................... 37, 271
macro statements ......................................... 215
Macros (Debug menu) ..................................... 46
macros. See C-SPY macros
main function, C-SPY running to when starting .... 36, 271
--mapu (C-SPY command line option) ................. 263
Mask (Advanced Trigger breakpoints option) ......... 113
Mask (Conditional breakpoints option) ............... 111
Match Case (Text search setting) ....................... 156
Match whole word (Text search setting) .............. 156
memory access checking .................................. 122, 138
Memory access checking (Memory Access Setup option) 138
Memory Access Setup dialog box ....................... 137
Memory Access Setup (Simulator menu) ................ 282
memory accesses, illegal ................................ 122
memory contents, writing to a file .................... 140
Memory Dump dialog box ................................ 140
Memory Fill (Memory window context menu) .......... 126
memory map ............................................... 137
Memory range (Edit Memory Access option) .......... 139
Memory Restore dialog box .............................. 128
Memory Restore (Memory window context menu) ...... 126
Memory Save dialog box ................................. 127
Memory Save (Memory window context menu) ........ 126
Memory window .......................................... 123
memory zones ............................................ 120
in device description file ................................. 121
__memoryRestore (C-SPY system macro) ............... 226
__memorySave (C-SPY system macro) .................. 226
Memory>Restore (Debug menu) .......................... 45
Memory>Save (Debug menu) ............................. 45
menu bar, C-SPY-specific ............................... 44
Message (Log breakpoints option) ..................... 103
migration, from earlier IAR compilers ................. 22
Mixed Mode (Disassembly window context menu) .... 61
Mode (Cycle Counter 5xx option) ...................... 188
Motorola, C-SPY input format .......................... 27, 30
Move to PC (Disassembly window context menu) ..... 60
MSP430F5xx, extra cycle counter ....................... 183
N
naming conventions ....................................... 23
Navigate (Timeline window context menu) .......... 151
New Breakpoint (Breakpoints window context menu) 100
Next Statement (Debug menu) .......................... 45
Next Symbol (Symbolic Memory window context menu) 131
O
--odd_word_check (C-SPY command line option) ...... 264
Offset (Download Extra Image setting) ................ 272
Only search in one column (Text search setting) ... 156
__openFile (C-SPY system macro) ....................... 227
Operation (Fill option) .................................. 129
Operator (Advanced Trigger breakpoints option) .... 113
Operator (Conditional breakpoints option) .......... 110
operators, sizeof in C-SPY ......................... 74, 74
optimizations, effects on variables .................. 74
options
   in the IDE ...................................... 269
   on the command line ......................... 255
Options (Stack window context menu) ......... 134
OR (Operation setting) ......................... 129
__orderInterrupt (C-SPY system macro) ........ 229

P
   -p (C-SPY command line option) ............. 264
   parameters
      tracing incorrect values of ............... 55
      typographic convention .................. 23
      part number, of this guide ............... 2
Path (Download Extra Image setting) .......... 272
Peripheral simulation (C-SPY simulator option) 280
   peripheral units
      device-specific ............................ 40
      in Register window ..................... 135
      simulating ................................ 280
   Please select one symbol
      (Resolve Symbol Ambiguity option) ....... 85
      --plugin (C-SPY command line option) ... 264
   plugin modules (C-SPY) ..................... 30
      loading ................................... 37
   Plugins (C-SPY options) .................... 273
      __popSimulatorInterruptExecutingStack (C-SPY
         system macro) ......................... 229
      --port (C-SPY command line option) ....... 265
      port pins, shared with JTAG pins .......... 286
      prerequisites, programming experience ... 19
   Previous Symbol
      (Symbolic Memory window context menu) .... 131
      probability (interrupt property) ......... 201
      definition of ................................ 193
   Probability % (Edit Interrupt option) ...... 201
   profiling
      on function level ........................... 161
      on instruction level ....................... 161
      using MSP430F5xx cycle counter ........... 185
   profiling information, on functions and instructions ... 159
   program execution, in C-SPY .................. 51
   program flow, monitoring .................... 171
   program states, monitoring .................. 171
   programming experience ...................... 19
   projects, for debugging externally built applications ... 38
   --protocol (C-SPY command line option) ....... 265
   publication date, of this guide ............. 2

Q
   Quick Watch window ........................... 83
   executing C-SPY macros ....................... 211

R
   Range breakpoints dialog box ............... 107
   range breakpoints, overview ................ 89
   Reaction trigger (Cycle Counter 5xx option) ... 189
      __readFile (C-SPY system macro) .......... 230
      __readFileByte (C-SPY system macro) ..... 231
      reading guidelines ....................... 19
      __readMemoryByte (C-SPY system macro) ... 231
      __readMemory8 (C-SPY system macro) ...... 231
      __readMemory16 (C-SPY system macro) .... 231
      __readMemory32 (C-SPY system macro) .... 232
      reference information, typographic convention ... 23
   Refresh (Debug menu) ....................... 45
   register groups (predefined), enabling ........ 135
   Register window ............................. 135
   registered trademarks ....................... 2
      __registerMacroFile (C-SPY system macro) ... 232
   registers, displayed in Register window ...... 135
   Release JTAG on Go (Emulator menu) .......... 283
   Remove (Watch window context menu) .......... 79
   repeat interval (interrupt property), definition of ... 193
   Replace (Memory window context menu) ........ 126
   Report Assert dialog box ........................ 68
   Reset Counter (Cycle Counter 5xx option) ...... 189
stack usage, computing ........................................ 122
Stack window .................................................... 132
standard C, sizeof operator in C-SPY ........................ 74
Start address (Edit Memory Access option).............. 139
Start address (Fill option) .................................... 129
Start address (Memory Save option) ....................... 127
Start (Cycle Counter 5xx option) .......................... 189
state storage
    setting up .................................................. 172
    starting using the sequencer ......................... 177
State Storage Control window ............................... 174
State Storage Control (Emulator menu) .................... 284
State Storage window ......................................... 175
State Storage Window (Emulator menu) ..................... 284
Statics window ................................................ 80
stdin and stdout, redirecting to C-SPY window .......... 63
Step Into (Debug menu) ...................................... 45
Step Into, description ....................................... 53
Step Out (Debug menu) ....................................... 45
Step Out, description ........................................ 54
Step Over (Debug menu) ...................................... 45
Step Over, description ...................................... 53
step points, definition of ................................... 52
Stop address (Memory Save option) ....................... 127
Stop Debugging (Debug menu) ............................... 45
Stop (Cycle Counter 5xx option) .......................... 189
stopping execution using the sequencer .................. 177
__strFind (C-SPY system macro) ............................ 244
__subString (C-SPY system macro) ......................... 245
Symbolic Memory window .................................... 130
Symbols window ............................................... 84
symbols, using in C-SPY expressions ..................... 72
System breakpoints on (C-SPY FET option) .............. 279
Terminal IO Log Files (Terminal IO Log Files option) .... 65
Terminal IO Log Files dialog box .......................... 64
Terminal I/O window ......................................... 56, 63
terminology .................................................... 22
Text search (Find in Trace option) ........................ 156
Time Axis Unit (Timeline window context menu) ....... 152
Timeline window (trace) ...................................... 149
Timeline (Simulator menu) .................................. 282
--timeout (C-SPY command line option) .................. 267
timer interrupt, example .................................... 196
Toggle Breakpoint (Code)
    (Call Stack window context menu) .................... 63
Toggle Breakpoint (Code)
    (Disassembly window context menu) ................. 61
Toggle Breakpoint (Log)
    (Call Stack window context menu) .................... 63
Toggle Breakpoint (Log)
    (Disassembly window context menu) ................. 61
Toggle Breakpoint (Trace Start)
    (Disassembly window context menu) ................. 61
Toggle Breakpoint (Trace Stop)
    (Disassembly window context menu) ................. 61
toggle source (Trace toolbar) .............................. 147
toLower (C-SPY system macro) .............................. 245
tools icon, in this guide .................................... 23
toString (C-SPY system macro) .............................. 246
toUpper (C-SPY system macro) .............................. 246
trace data
    definition of ............................................ 143
    searching in ............................................ 145
Trace Expressions window .................................. 154
trace start and stop breakpoints, overview ............. 88
Trace Start breakpoints dialog box ...................... 152
Trace Stop breakpoints dialog box ....................... 153
Trace window .................................................. 146
Trace (Simulator menu) ..................................... 282
trace, state storage being variant of .................... 171
trademarks ...................................................... 2
Type (Advanced Trigger breakpoints option) ............ 112
Type (Conditional breakpoints option) .................... 110

C-SPY® Debugging Guide
for MSP430

UCS430-1
typographic conventions ........................................... 23

U
UBROF ................................................................. 27
Unavailable, C-SPY message ................................. 75
Universal Binary Relocatable Object Format. See UBROF
__unloadImage (C-SPY system macro) ..................... 247
Use command line options (debugger option) ........... 273
Use manual ranges (Memory Access Setup option) ....... 138
Use ranges based on (Memory Access Setup option) .... 137
Use software breakpoints (C-SPY FET option) .......... 279
user application, definition of .................................. 29
--use_emulated_breakpoints
(C-SPY command line option) ................................ 267
--use_virtual_breakpoints (C-SPY command line option) 268

V
Value (Advanced Trigger breakpoints option) .......... 113
Value (Fill option) .................................................... 129
variables
  effects of optimizations .................................... 74
  information, limitation on .................................. 74
  using in C-SPY expressions ............................... 73
variance (interrupt property), definition of ......... 193
Variance % (Edit Interrupt option) ......................... 201
--vccvoltage (C-SPY command line option) ............. 268
Verify download (C-SPY FET option) ....................... 277
--verify_all (C-SPY command line option) ............... 268
version, of IAR Embedded Workbench ......................... 2
virtual breakpoints ................................................. 91
  configuring ...................................................... 279
visualSTATE, C-SPY plugin module for ........................ 30

W
warnings icon, in this guide ................................... 23
Watch window ....................................................... 78
  using ............................................................. 71
web sites, recommended ....................................... 22
while (macro statement) ....................................... 215
windows, specific to C-SPY ................................. 46
With I/O emulation modules (linker option), using .... 63
__writeFile (C-SPY system macro) ......................... 247
__writeFileByte (C-SPY system macro) .................... 248
__writeMemoryByte (C-SPY system macro) ............... 248
__writeMemory8 (C-SPY system macro) .................... 248
__writeMemory16 (C-SPY system macro) ................... 249
__writeMemory32 (C-SPY system macro) ................... 249

X
XOR (Operation setting) .......................................... 129

Z
zone
  defined in device description file ....................... 121
  in C-SPY ....................................................... 120
Zone (Edit Memory Access option) ........................... 139
Zone (Fill option) ................................................. 129
Zone (Memory Restore option) ............................... 128
Zone (Memory Save option) .................................... 127
Zone (Memory window context menu) ....................... 125
Zoom (Timeline window context menu) ...................... 151

Symbols
__cancelAllInterrupts (C-SPY system macro) .............. 221
__cancelInterrupt (C-SPY system macro) ................... 221
__clearBreak (C-SPY system macro) .......................... 221
__closeFile (C-SPY system macro) ............................ 222
__delay (C-SPY system macro) ................................ 222
__disableInterrupts (C-SPY system macro) ................. 222
__driverType (C-SPY system macro) .......................... 223
__enableInterrupts (C-SPY system macro) ................... 223
__evaluate (C-SPY system macro) . . . . . . . . . . . . . . . . . . 224
__fmessage (C-SPY macro statement) . . . . . . . . . . . . . . . 216
__isBatchMode (C-SPY system macro) . . . . . . . . . . . . . . 224
__loadImage (C-SPY system macro) . . . . . . . . . . . . . . . . 225
__memoryRestore (C-SPY system macro) . . . . . . . . . . . . 226
__memorySave (C-SPY system macro) . . . . . . . . . . . . . . 226
__message (C-SPY macro statement) . . . . . . . . . . . . . . . . 216
__openFile (C-SPY system macro) . . . . . . . . . . . . . . . . . . 227
__orderInterrupt (C-SPY system macro). . . . . . . . . . . . . . 229
__popSimulatorInterruptExecutingStack (C-SPY
system macro) . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 229
__readFile (C-SPY system macro) . . . . . . . . . . . . . . . . . . 230
__readFileByte (C-SPY system macro) . . . . . . . . . . . . . . 231
__readMemoryByte (C-SPY system macro) . . . . . . . . . . . 231
__readMemory8 (C-SPY system macro) . . . . . . . . . . . . . 231
__readMemory16 (C-SPY system macro) . . . . . . . . . . . . 231
__readMemory32 (C-SPY system macro) . . . . . . . . . . . . 232
__registerMacroFile (C-SPY system macro). . . . . . . . . . . 232
__resetFile (C-SPY system macro) . . . . . . . . . . . . . . . . . . 233
__setAdvancedTriggerBreak (C-SPY system macro) . . . . 233
__setCodeBreak (C-SPY system macro). . . . . . . . . . . . . . 235
__setConditionalBreak (C-SPY system macro) . . . . . . . . 236
__setDataBreak (C-SPY system macro) . . . . . . . . . . . . . . 237
__setLogBreak (C-SPY system macro) . . . . . . . . . . . . . . 238
__setRangeBreak (C-SPY system macro) . . . . . . . . . . . . . 240
__setSimBreak (C-SPY system macro) . . . . . . . . . . . . . . 241
__setTraceStartBreak (C-SPY system macro). . . . . . . . . . 242
__setTraceStopBreak (C-SPY system macro) . . . . . . . . . . 243
__smessage (C-SPY macro statement) . . . . . . . . . . . . . . . 216
__sourcePosition (C-SPY system macro) . . . . . . . . . . . . . 244
__strFind (C-SPY system macro) . . . . . . . . . . . . . . . . . . . 244
__subString (C-SPY system macro) . . . . . . . . . . . . . . . . . 245
__targetDebuggerVersion (C-SPY system macro) . . . . . . 245
__toLower (C-SPY system macro) . . . . . . . . . . . . . . . . . . 245
__toString (C-SPY system macro) . . . . . . . . . . . . . . . . . . 246
__toUpper (C-SPY system macro) . . . . . . . . . . . . . . . . . . 246
__unloadImage (C-SPY system macro) . . . . . . . . . . . . . . 247
__writeFile (C-SPY system macro) . . . . . . . . . . . . . . . . . 247
__writeFileByte (C-SPY system macro) . . . . . . . . . . . . . . 248
__writeMemoryByte (C-SPY system macro) . . . . . . . . . . 248

C-SPY® Debugging Guide

300

for MSP430

UCS430-1

__writeMemory8 (C-SPY system macro) . . . . . . . . . . . . . 248
__writeMemory16 (C-SPY system macro) . . . . . . . . . . . . 249
__writeMemory32 (C-SPY system macro) . . . . . . . . . . . . 249
-B (C-SPY command line option) . . . . . . . . . . . . . . . . . . . 256
-d (C-SPY command line option) . . . . . . . . . . . . . . . . . . . 258
-p (C-SPY command line option) . . . . . . . . . . . . . . . . . . . 264
--allow_access_to_BSL (C-SPY command line option) . . 255
--allow_locked_flash_access
(C-SPY command line option) . . . . . . . . . . . . . . . . . . . . . 255
--attach (C-SPY command line option) . . . . . . . . . . . . . . . 256
--backend (C-SPY command line option) . . . . . . . . . . . . . 256
--code_coverage (C-SPY command line option) . . . . . . . . 257
--connection (C-SPY command line option) . . . . . . . . . . . 256
--core (C-SPY command line option) . . . . . . . . . . . . . . . . 257
--cycles (C-SPY command line option) . . . . . . . . . . . . . . 258
--derivative (C-SPY command line option) . . . . . . . . . . . . 258
--disable_interrupts (C-SPY command line option) . . . . . 259
--disable_memory_cache (C-SPY command line option) . 259
--download_only (C-SPY command line option) . . . . . . . 259
--eem (C-SPY command line option) . . . . . . . . . . . . . . . . 260
--erase_main (C-SPY command line option) . . . . . . . . . . 260
--erase_main_and_info (C-SPY command line option) . . 260
--erase_retain (C-SPY command line option) . . . . . . . . . . 261
--hardware_multiplier (C-SPY command line option) . . . 261
--hwmult_type (C-SPY command line option) . . . . . . . . . 262
--lpt1 (C-SPY command line option) . . . . . . . . . . . . . . . . 263
--lpt2 (C-SPY command line option) . . . . . . . . . . . . . . . . 263
--lpt3 (C-SPY command line option) . . . . . . . . . . . . . . . . 263
--macro (C-SPY command line option) . . . . . . . . . . . . . . 263
--mapu (C-SPY command line option) . . . . . . . . . . . . . . . 263
--odd_word_check (C-SPY command line option) . . . . . . 264
--plugin (C-SPY command line option) . . . . . . . . . . . . . . 264
--port (C-SPY command line option) . . . . . . . . . . . . . . . . 265
--protocol (C-SPY command line option) . . . . . . . . . . . . . 265
--settlingtime (C-SPY command line option) . . . . . . . . . . 266
--set_exit_breakpoint (C-SPY command line option) . . . . 266
--set_getchar_breakpoint (C-SPY command line option) . 266
--set_putchar_breakpoint (C-SPY command line option) . 266
--silent (C-SPY command line option) . . . . . . . . . . . . . . . 267
--timeout (C-SPY command line option) . . . . . . . . . . . . . 267


--use_emulated_breakpoints
(C-SPY command line option) .................................. 267
--use_virtual_breakpoints (C-SPY command line option). 268
--vcc_voltage (C-SPY command line option) ............ 268
--verify_all (C-SPY command line option) ............... 268
“Power on” Reset (Emulator menu) ......................... 285

Numerics

1x Units (Memory window context menu) ............... 125
1x Units (Stack window context menu) ................. 134
1x Units (Symbolic Memory window context menu) . 131
2x Units (Memory window context menu) ............... 125
2x Units (Stack window context menu) ................. 134
2x Units (Symbolic Memory window context menu) . 131
4x Units (Memory window context menu) ............... 125
4x Units (Stack window context menu) ................. 134
4x Units (Symbolic Memory window context menu) . 132