IAR Embedded Workbench® IDE
User Guide

for Advanced RISC Machines Ltd’s
ARM Microprocessor Family
COPYRIGHT NOTICE
Copyright © 1999–2010 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.

ARM and Thumb are registered trademarks of Advanced RISC Machines Ltd.
EmbeddedICE is a trademark of Advanced RISC Machines Ltd. OCDemon is a trademark of Macraigor Systems LLC. μC/OS-II is a trademark of Micrium Inc. CMX-RTX is a trademark of CMX Systems, Inc. ThreadX is a trademark of Express Logic. RTXC is a trademark of Quadros Systems. Fusion is a trademark of Unicoi Systems.

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

CodeWright is a registered trademark of Starbase Corporation.

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

EDITION NOTICE
Seventeenth edition: July 2010
Part number: UARM-17b

This guide describes version 5.5x of the IAR Embedded Workbench® IDE for Advanced RISC Machines Ltd’s ARM cores family.

Internal reference: 5.7.x. tut2009.2, ISUD.
Brief contents

Tables ........................................................................................................................................ xxv
Figures ........................................................................................................................................ xxxi
Preface ......................................................................................................................................... xli

Part 1. Product overview ............................................................................................................. 1
Product introduction ...................................................................................................................... 3
Installed files ................................................................................................................................. 19

Part 2. Tutorials ............................................................................................................................ 27
Welcome to the tutorials .................................................................................................................. 29
Creating an application project ....................................................................................................... 33
Debugging using the IAR C-SPY® Debugger .............................................................................. 45
Mixing C and assembler modules .................................................................................................. 55
Using C++ ....................................................................................................................................... 59
Simulating an interrupt ..................................................................................................................... 63
Creating and using libraries ............................................................................................................. 75

Part 3. Project management and building .................................................................................... 79
The development environment ....................................................................................................... 81
Managing projects ............................................................................................................................. 87
Building .......................................................................................................................................... 97
Editing ............................................................................................................................................ 105
Assembler options ............................................................... 467
Converter options .............................................................. 475
Custom build options ......................................................... 477
Build actions options .......................................................... 479
Linker options ................................................................... 481
Library builder options ....................................................... 491
Debugger options ................................................................ 493
The C-SPY Command Line Utility—cspybat ...................... 499
C-SPY® macros reference .................................................... 527
Glossary ............................................................................. 569
Index .................................................................................. 585
Contents

Tables .................................................................................................................... xxv
Figures ................................................................................................................. xxxi
Preface .................................................................................................................. xli

Who should read this guide ............................................................................. xli
How to use this guide ....................................................................................... xli
What this guide contains ............................................................................... xlii
Other documentation ....................................................................................... xliv
Document conventions ...................................................................................... xlvi
  Typographic conventions ............................................................................. xlvi
  Naming conventions ..................................................................................... xlvii

Part 1. Product overview ................................................................................. 1
Product introduction ......................................................................................... 3

The IAR Embedded Workbench IDE ............................................................. 3
  An extensible and modular environment .................................................. 3
  Features ....................................................................................................... 4
  Documentation ............................................................................................. 5

IAR C-SPY Debugger ....................................................................................... 5
  The C-SPY driver ...................................................................................... 6
  General C-SPY debugger features ............................................................. 6
  C-SPY plugin modules ............................................................................. 8
  RTOS awareness ....................................................................................... 9
  Documentation ............................................................................................. 9

IAR C-SPY Debugger systems ..................................................................... 9
  IAR C-SPY Simulator ............................................................................. 10
  IAR C-SPY J-Link driver ........................................................................ 10
  IAR C-SPY LMI FTDI driver .................................................................. 11
  IAR C-SPY RDI driver ........................................................................... 11
  IAR C-SPY Macraigor driver ................................................................. 12
Using C++ .......................................................................................... 30
Simulating an interrupt ....................................................................... 30
Creating and using libraries ............................................................... 30
Getting started .................................................................................... 31

Creating an application project .......................................................... 33

Setting up a new project ..................................................................... 33
Creating a workspace .......................................................................... 33
Creating the new project .................................................................... 34
Adding files to the project .................................................................. 36
Setting project options ....................................................................... 37

Compiling and linking the application ............................................... 39
Compiling the source files .................................................................. 40
Viewing the list file ............................................................................ 41
Linking the application ...................................................................... 43
Viewing the map file .......................................................................... 44

Debugging using the IAR C-SPY® Debugger ..................................... 45

Debugging the application .................................................................. 45
Starting the debugger .......................................................................... 45
Organizing the windows ....................................................................... 45
Inspecting source statements ............................................................ 46
Inspecting variables .......................................................................... 48
Setting and monitoring breakpoints ................................................... 50
Debugging in disassembly mode ......................................................... 51
Monitoring memory ........................................................................... 51
Viewing terminal I/O .......................................................................... 53
Reaching program exit ....................................................................... 53

Mixing C and assembler modules .................................................... 55

Examining the calling convention ...................................................... 55
Adding an assembler module to the project ....................................... 57
Setting up the project ....................................................................... 57
Using C++ ............................................................................................................ 59

Creating a C++ application .................................................................................. 59
Compiling and linking the C++ application ......................................................... 59
Setting a breakpoint and executing to it ............................................................. 60
Printing the Fibonacci numbers .......................................................................... 62

Simulating an interrupt ........................................................................................ 63

Adding an interrupt handler ................................................................................ 63
The application—a brief description .................................................................. 63
Writing an interrupt handler for ARM7TDMI .................................................... 64
Writing an interrupt handler for Cortex-M3 ....................................................... 64
Setting up the project .......................................................................................... 64

Setting up the simulation environment .............................................................. 65
Defining a C-SPY setup macro file ..................................................................... 65
Setting C-SPY options ....................................................................................... 66
Building the project ............................................................................................ 68
Starting the simulator ......................................................................................... 68
Specifying a simulated interrupt ......................................................................... 68
Setting an immediate breakpoint ....................................................................... 69

Simulating the interrupt ...................................................................................... 70
Executing the application ................................................................................... 70

Using macros for interrupts and breakpoints .................................................... 72

Creating and using libraries .............................................................................. 75

Using libraries ..................................................................................................... 75
Creating a new project ....................................................................................... 76
Creating a library project .................................................................................. 76
Using the library in your application project ..................................................... 77

Part 3. Project management and building .......................................................... 79

The development environment .......................................................................... 81

The IAR Embedded Workbench IDE .................................................................. 81
The tool chain ....................................................................................................... 81
Running the IDE ................................................................. 82
Exiting ...................................................................................... 83

Customizing the environment ...................................................... 83
Organizing the windows on the screen .......................................... 83
Customizing the IDE ................................................................. 84
Invoking external tools ............................................................... 85

Managing projects ........................................................................ 87

The project model ........................................................................ 87
How projects are organized .......................................................... 87
Creating and managing workspaces .............................................. 89

Navigating project files ............................................................... 91
Viewing the workspace ................................................................. 92
Displaying browse information ...................................................... 93

Source code control ..................................................................... 94
Interacting with source code control systems ................................. 94

Building ...................................................................................... 97

Building your application ............................................................ 97
Setting options ............................................................................ 97
Building a project .......................................................................... 99
Building multiple configurations in a batch .................................... 99
Using pre- and post-build actions ................................................ 100
Correcting errors found during build ............................................ 100
Building from the command line .................................................. 101

Extending the tool chain .............................................................. 101
Tools that can be added to the tool chain ....................................... 102
Adding an external tool ............................................................... 102

Editing ....................................................................................... 105

Using the IAR Embedded Workbench editor ................................. 105
Editing a file ................................................................................ 105
Using and adding code templates ............................................... 109
Navigating in and between files .................................................. 111
Searching .................................................................................... 112
Customizing the editor environment ................................................. 112
Using an external editor .................................................................. 112

Part 4. Debugging ............................................................................ 115
The IAR C-SPY® Debugger ................................................................. 117

Debugger concepts ........................................................................... 117
C-SPY and target systems ............................................................... 117
Debugger ......................................................................................... 118
Target system .................................................................................. 118
User application .............................................................................. 118
C-SPY Debugger systems ................................................................. 118
ROM-monitor program ................................................................. 119
Third-party debuggers .................................................................... 119

The C-SPY environment ................................................................. 119
An integrated environment .............................................................. 119

Setting up C-SPY ............................................................................ 120
Choosing a debug driver ................................................................. 120
Executing from reset ...................................................................... 121
Using a setup macro file ............................................................... 121
Selecting a device description file .................................................. 121
Loading plugin modules .................................................................. 122

Starting C-SPY ............................................................................... 122
Executable files built outside of the IDE ........................................ 123
Starting a debug session with source files missing ......................... 123
Loading multiple images ............................................................... 124
Redirecting debugger output to a file ............................................. 125

Adapting C-SPY to target hardware .............................................. 125
Device description file ................................................................. 125
Remapping memory ....................................................................... 126

Executing your application .............................................................. 129

Source and disassembly mode debugging ........................................ 129
Executing .................................................................................................. 129
Step ........................................................................................................... 130
Go ............................................................................................................. 132
Run to Cursor .......................................................................................... 132
Highlighting ............................................................................................ 132
Using breakpoints to stop ........................................................................ 132
Using the Break button to stop .............................................................. 133
Stop at program exit ................................................................................ 133
Call stack information ............................................................................ 133
Terminal input and output ...................................................................... 134
Working with variables and expressions .................................................. 135
C-SPY expressions .................................................................................. 135
C symbols ............................................................................................... 135
Assembler symbols .................................................................................. 136
Macro functions ...................................................................................... 136
Macro variables ....................................................................................... 137
Limitations on variable information ....................................................... 137
Effects of optimizations .......................................................................... 137
Viewing variables and expressions ......................................................... 138
Working with the windows ..................................................................... 138
Viewing assembler variables ................................................................... 139
Using breakpoints ................................................................................... 141
The breakpoint system ........................................................................... 141
Defining breakpoints ................................................................................ 141
Breakpoint icons ...................................................................................... 142
Different ways to set a breakpoint ............................................................. 142
Toggling a simple code breakpoint ............................................................ 143
Defining breakpoints using the dialog box ............................................... 143
Setting a data breakpoint in the Memory window ..................................... 144
Defining breakpoints using system macros ................................................. 145
Useful breakpoint tips ............................................................................. 145
Viewing all breakpoints ......................................................................... 146
Using the Breakpoint Usage dialog box ..................................................... 147
Breakpoint consumers ................................................................. 148

Monitoring memory and registers .................................................. 149
  Memory addressing ........................................................................ 149
  Windows for monitoring memory and registers ............................ 150
    Using the Stack window ............................................................. 150
    Working with registers .............................................................. 152

Using the C-SPY® macro system .................................................. 155
  The macro system ........................................................................ 155
    The macro language ................................................................. 156
    The macro file .......................................................................... 156
    Setup macro functions .............................................................. 157
  Using C-SPY macros ................................................................. 158
    Using the Macro Configuration dialog box ................................. 158
    Registering and executing using setup macros and setup files ...... 159
    Executing macros using Quick Watch ......................................... 160
    Executing a macro by connecting it to a breakpoint .................... 161

Analyzing your application .......................................................... 163
  Function-level profiling .............................................................. 163
    Using the profiler ...................................................................... 164
  Code coverage ............................................................................ 166
    Using Code Coverage ............................................................... 166

Using trace ..................................................................................... 169
  Collecting and using trace data ................................................... 169
    Reasons for using trace ............................................................ 169
    Briefly about trace .................................................................... 170
    Requirements for using trace ..................................................... 171
    Getting started with trace in the C-SPY simulator ....................... 172
    Getting started with ETM trace .................................................. 172
    Getting started with SWO trace .................................................. 173
    Setting up concurrent use of ETM and SWO ............................... 173
    Trace data collection using breakpoints ..................................... 173
Searching in trace data ................................................................. 174
Browsing through trace data ........................................................... 174

**Trace-related reference information** ........................................ 175
ETM Trace Settings dialog box ......................................................... 176
SWO Trace Window Settings dialog box .......................................... 178
SWO Configuration dialog box ....................................................... 180
Trace window .............................................................................. 183
Trace Save dialog box .................................................................. 187
Function Trace window ................................................................. 188
Timeline window .......................................................................... 189
Trace Start breakpoints dialog box ................................................. 193
Trace Stop breakpoints dialog box ................................................. 194
Trace Expressions window ............................................................ 195
Find in Trace dialog box ............................................................... 197
Find in Trace window .................................................................. 198

**Part 5. IAR C-SPY Simulator** .................................................... 199

Simulator-specific debugging .......................................................... 201

**The C-SPY Simulator introduction** ....................................... 201
Features ....................................................................................... 201
Selecting the simulator driver ......................................................... 201

**Simulator-specific menus** ....................................................... 202
Simulator menu ........................................................................... 202

**Memory access checking** ....................................................... 203
Memory Access setup dialog box .................................................... 203
Edit Memory Access dialog box .................................................... 206

**Using breakpoints in the simulator** ........................................ 206
Data breakpoints ........................................................................ 207
Data breakpoints dialog box .......................................................... 207
Immediate breakpoints .................................................................. 209
Immediate breakpoints dialog box ................................................. 210
Breakpoint Usage dialog box ......................................................... 211
Simulating interrupts .............................................................. 213
  The C-SPY interrupt simulation system .................................. 213
Using the interrupt simulation system ...................................... 216
  Interrupt Setup dialog box .................................................... 217
  Edit Interrupt dialog box .................................................... 218
  Forced interrupt window ..................................................... 220
  C-SPY system macros for interrupts ....................................... 220
Simulating a simple interrupt .................................................. 222

Part 6. C-SPY hardware debugger systems ......................... 225
Introduction to C-SPY® hardware debugger systems .............. 227
  The C-SPY hardware debugger systems ............................... 227
    Differences between the C-SPY drivers ............................. 228
Getting started ................................................................. 229
  Running the demo program ................................................ 229
  The IAR C-SPY Angel debug monitor driver ......................... 230
The IAR C-SPY GDB Server driver ......................................... 231
  Configuring the OpenOCD Server ....................................... 232
The IAR C-SPY ROM-monitor driver ..................................... 233
The IAR C-SPY J-Link/J-Trace drivers .................................. 234
  Installing the J-Link USB driver ......................................... 235
The IAR C-SPY LMI FTDI driver ........................................... 236
  Installing the FTDI USB driver .......................................... 237
The IAR C-SPY Macraigor driver ......................................... 237
The IAR C-SPY RDI driver .................................................. 238
The IAR C-SPY ST-Link driver ............................................. 240
  An overview of the debugger startup ................................ 241
    Debugging code in flash ................................................ 241
    Debugging code in RAM ................................................ 242
Hardware-specific debugging .............................................. 243
  C-SPY options for debugging using hardware systems ........ 243
    Download ........................................................................ 245
Extra Options page ................................................................. 246

Debugging using the Angel debug monitor driver ............... 247
Angel ....................................................................................... 247

Debugging using the IAR C-SPY GDB Server driver .......... 248
GDB Server ............................................................................. 249
Breakpoints ............................................................................ 249
The GDB Server menu ........................................................... 250

Debugging using the IAR C-SPY ROM-monitor driver ........ 250
IAR ROM-monitor ................................................................. 251

Debugging using the IAR C-SPY J-Link/J-Trace driver ......... 252
Setup ....................................................................................... 252
Connection ............................................................................. 256
Breakpoints ............................................................................ 257
The J-Link menu ................................................................. 257
Live watch and use of DCC .................................................... 259
Terminal I/O and use of DCC ............................................... 259

Debugging using the IAR C-SPY LMI FTDI driver ............... 260
Setup ....................................................................................... 260
The LMI FTDI menu ............................................................ 261

Debugging using the IAR C-SPY Macraigor driver .......... 261
Breakpoints ............................................................................ 264
The Macraigor JTAG menu .................................................. 264

Debugging using the RDI driver .......................................... 265
RDI ....................................................................................... 265
RDI menu ............................................................................. 267

Debugging using the ST-Link driver .................................... 267
ST-Link .................................................................................. 268
ST-Link menu ....................................................................... 268

Debugging using a third-party driver .................................. 269
Third-Party Driver ............................................................... 269

Analyzing your program using driver-specific tools ............ 271

Using breakpoints in the hardware debugger systems ......... 271
Available number of breakpoints .................................... 272
Breakpoints options ................................................................. 272
Code breakpoints dialog box .................................................. 274
Data breakpoints dialog box .................................................... 275
Data Log breakpoints dialog box ............................................ 278
Breakpoint Usage dialog box ................................................ 280
Breakpoints on vectors .......................................................... 280
Setting breakpoints in __ramfunc declared functions ............ 281

Using JTAG watchpoints .......................................................... 282
JTAG watchpoints dialog box ................................................ 283

Using J-Link trace triggers and trace filters ................................. 285
Trace Start breakpoints dialog box ....................................... 287
Trace Stop breakpoints dialog box ....................................... 289
Trace Filter breakpoints dialog box .................................... 292

Using the data and interrupt logging systems .............................. 294
Data Log window ................................................................. 296
Data Log Summary window ................................................ 299
Interrupt Log window ......................................................... 300
Interrupt Log Summary window ....................................... 302

Using the profiler .................................................................. 303
Function Profiler window .................................................... 306

Using flash loaders .................................................................. 309

The flash loader ..................................................................... 309
Setting up the flash loader(s) ................................................ 309
The flash loading mechanism .............................................. 310
Build considerations ............................................................ 310
Flash Loader Overview dialog box ..................................... 310
Flash Loader Configuration dialog box ................................. 312

Part 7. Reference information .................................................. 315

IAR Embedded Workbench® IDE reference ............................... 317

Windows .............................................................................. 317
IAR Embedded Workbench IDE window ................................. 318
<table>
<thead>
<tr>
<th>Section</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>Configure Viewers dialog box</td>
<td>399</td>
</tr>
<tr>
<td>Edit Viewer Extensions dialog box</td>
<td>399</td>
</tr>
<tr>
<td>Window menu</td>
<td>400</td>
</tr>
<tr>
<td>Help menu</td>
<td>401</td>
</tr>
<tr>
<td><strong>C-SPY® reference</strong></td>
<td>403</td>
</tr>
<tr>
<td><strong>C-SPY windows</strong></td>
<td></td>
</tr>
<tr>
<td>Editing in C-SPY windows</td>
<td>404</td>
</tr>
<tr>
<td>C-SPY Debugger main window</td>
<td>404</td>
</tr>
<tr>
<td>Disassembly window</td>
<td>406</td>
</tr>
<tr>
<td>Resolve Symbol Ambiguity dialog box</td>
<td>409</td>
</tr>
<tr>
<td>Memory window</td>
<td>410</td>
</tr>
<tr>
<td>Fill dialog box</td>
<td>413</td>
</tr>
<tr>
<td>Memory Save dialog box</td>
<td>414</td>
</tr>
<tr>
<td>Memory Restore dialog box</td>
<td>415</td>
</tr>
<tr>
<td>Symbolic Memory window</td>
<td>416</td>
</tr>
<tr>
<td>Register window</td>
<td>418</td>
</tr>
<tr>
<td>Watch window</td>
<td>418</td>
</tr>
<tr>
<td>Locals window</td>
<td>420</td>
</tr>
<tr>
<td>Auto window</td>
<td>421</td>
</tr>
<tr>
<td>Live Watch window</td>
<td>421</td>
</tr>
<tr>
<td>Quick Watch window</td>
<td>422</td>
</tr>
<tr>
<td>Statics window</td>
<td>422</td>
</tr>
<tr>
<td>Select Statics dialog box</td>
<td>424</td>
</tr>
<tr>
<td>Call Stack window</td>
<td>425</td>
</tr>
<tr>
<td>Terminal I/O window</td>
<td>426</td>
</tr>
<tr>
<td>Code Coverage window</td>
<td>427</td>
</tr>
<tr>
<td>Profiling window</td>
<td>429</td>
</tr>
<tr>
<td>Images window</td>
<td>432</td>
</tr>
<tr>
<td>Stack window</td>
<td>433</td>
</tr>
<tr>
<td>Symbols window</td>
<td>436</td>
</tr>
<tr>
<td><strong>C-SPY menus</strong></td>
<td>437</td>
</tr>
<tr>
<td>Debug menu</td>
<td>438</td>
</tr>
<tr>
<td>Disassembly menu</td>
<td>442</td>
</tr>
</tbody>
</table>
General options ................................................................. 445
  Target ............................................................................. 445
  Output ............................................................................ 447
Library Configuration ...................................................... 448
Library Options .............................................................. 450
MISRA C ............................................................................. 451
Compiler options ............................................................ 453
  Multi-file compilation ................................................... 453
  Language ......................................................................... 453
  Code ................................................................................. 456
  Optimizations ............................................................... 457
    Optimizations ............................................................. 457
  Output ............................................................................. 458
  List .................................................................................. 460
    Output list file ............................................................. 460
    Output assembler file .................................................. 460
Preprocessor ................................................................. 461
  Diagnostics ..................................................................... 462
  MISRA C ......................................................................... 464
Extra Options ................................................................. 465
  Use command line options ............................................. 465
Assembler options .......................................................... 467
  Language ........................................................................ 467
  Output ............................................................................ 469
    Generate debug information ........................................ 469
  List .................................................................................. 470
Preprocessor ................................................................. 471
  Diagnostics ..................................................................... 473
    Max number of errors .................................................. 473
Extra Options ................................................................. 474
  Use command line options ............................................. 474
Converter options .............................................................................. 475
   Output .............................................................................................. 475
      Promable output format ................................................................. 475
      Output file .................................................................................. 475
Custom build options ........................................................................ 477
Custom Tool Configuration ................................................................. 477
Build actions options ........................................................................ 479
   Build Actions Configuration ............................................................ 479
      Pre-build command line ................................................................. 479
      Post-build command line .............................................................. 479
Linker options .................................................................................... 481
   Config ............................................................................................. 481
   Library ............................................................................................ 482
      Automatic runtime library selection ............................................. 482
      Additional libraries ...................................................................... 482
      Override default program entry .................................................. 482
   Input .................................................................................................. 483
      Keep symbols ............................................................................... 483
      Raw binary image ......................................................................... 484
   Output ............................................................................................. 484
      Output file .................................................................................. 484
      Output debug information ........................................................... 485
   List .................................................................................................... 485
      Generate linker map file ............................................................... 485
      Generate log .............................................................................. 485
   #define .......................................................................................... 486
      Define symbol .............................................................................. 486
   Diagnostics ..................................................................................... 486
   Checksum ....................................................................................... 488
      Fill unused code memory .............................................................. 489
   Extra Options .................................................................................. 490
      Use command line options ........................................................... 490
# Contents

Library builder options ................................................................. 491
   Output ........................................................................................ 491

Debugger options ........................................................................... 493
   Setup .......................................................................................... 493
   Download ................................................................................... 495
   Images ....................................................................................... 495
      Download extra image .............................................................. 495
   Extra Options .............................................................................. 496
      Use command line options ....................................................... 496
   Plugins ....................................................................................... 496

The C-SPY Command Line Utility—cspybat .................................. 499
   Using C-SPY in batch mode ......................................................... 499
   C-SPY command line options .................................................... 500
   Descriptions of C-SPY command line options ......................... 505

C-SPY® macros reference ............................................................... 527
   The macro language ................................................................... 527
      Macro functions ....................................................................... 527
      Predefined system macro functions ....................................... 527
      Macro variables ...................................................................... 528
      Macro statements ................................................................... 529
      Formatted output .................................................................... 530
   Setup macro functions summary ............................................... 532
   C-SPY system macros summary ................................................ 533
   Description of C-SPY system macros ....................................... 535

Glossary .......................................................................................... 569

Index .............................................................................................. 585
Tables

1: Typographic conventions used in this guide ....................................................... xlvi
2: Naming conventions used in this guide ............................................................ xlvii
3: The ARM directory ............................................................................................ 19
4: The common directory ....................................................................................... 21
5: File types ........................................................................................................... 21
6: General settings for project1 ............................................................................. 38
7: Compiler options for project1 ............................................................................ 39
8: Compiler options for project2 ............................................................................. 56
9: Project options for C++ tutorial ......................................................................... 59
10: Settings in the Edit Interrupt dialog box ......................................................... 68
11: Breakpoints dialog box ..................................................................................... 70
12: General options for a library project ................................................................ 76
13: Command shells .............................................................................................. 86
14: iarbuild.exe command line options ................................................................. 101
15: C-SPY assembler symbols expressions ......................................................... 136
16: Handling name conflicts between hardware registers and assembler labels .... 136
17: Project options for enabling profiling ............................................................. 164
18: Project options for enabling code coverage .................................................... 166
19: Trace toolbar buttons ...................................................................................... 184
20: Trace window columns for the C-SPY simulator ............................................ 185
21: Trace window columns for ETM trace ............................................................ 186
22: Trace window columns for SWO trace ........................................................... 186
23: Commands on the Timeline window context menu ......................................... 192
24: Toolbar buttons in the Trace Expressions window ........................................... 196
25: Trace Expressions window columns ............................................................... 196
26: Description of Simulator menu commands ..................................................... 202
27: Function buttons in the Memory Access Setup dialog box .............................. 205
28: Memory Access types ....................................................................................... 208
29: Breakpoint conditions ....................................................................................... 209
30: Memory Access types ....................................................................................... 210
31: Interrupt statuses ............................................................................................ 215
66: Description of source code control states .................................................. 326
67: Description of commands on the editor window tab context menu .......... 331
68: Description of commands on the editor window context menu ................ 332
69: Editor keyboard commands for insertion point navigation ....................... 335
70: Editor keyboard commands for scrolling .................................................. 335
71: Editor keyboard commands for selecting text .......................................... 335
72: Columns in Source Browser window ....................................................... 337
73: Information in Source Browser window .................................................. 337
74: Source Browser window context menu commands ..................................... 339
75: Breakpoints window context menu commands ......................................... 340
76: Breakpoint conditions ............................................................................... 343
77: Log breakpoint conditions ........................................................................ 344
78: Location types .......................................................................................... 345
79: File menu commands ................................................................................ 351
80: Edit menu commands ................................................................................ 353
81: Find dialog box options ............................................................................ 356
82: Replace dialog box options ....................................................................... 357
83: Incremental Search function buttons ....................................................... 360
84: View menu commands .............................................................................. 362
85: Project menu commands .......................................................................... 363
86: Argument variables .................................................................................. 366
87: Configurations for project dialog box options .......................................... 368
88: New Configuration dialog box options ..................................................... 368
89: Description of Create New Project dialog box .......................................... 369
90: Project option categories .......................................................................... 370
91: Description of the Batch Build dialog box ............................................... 371
92: Description of the Edit Batch Build dialog box ........................................ 372
93: Tools menu commands ............................................................................. 373
94: Project IDE options .................................................................................. 387
95: Register Filter options ............................................................................. 392
96: Configure Tools dialog box options .......................................................... 395
97: Command shells ....................................................................................... 396
98: Window menu commands ........................................................................ 400
99: Editing in C-SPY windows ...................................................................... 404
100: C-SPY menu .................................................................................................... 405
101: Disassembly window toolbar ........................................................................ 406
102: Disassembly context menu commands ......................................................... 407
103: Memory window operations ......................................................................... 410
104: Commands on the memory window context menu ....................................... 412
105: Fill dialog box options .................................................................................... 414
106: Memory fill operations ................................................................................... 414
107: Symbolic Memory window toolbar ............................................................... 416
108: Symbolic Memory window columns ............................................................. 417
109: Commands on the Symbolic Memory window context menu .................... 417
110: Watch window context menu commands ..................................................... 419
111: Effects of display format setting on different types of expressions ............ 420
112: Statics window columns ............................................................................... 423
113: Statics window context menu commands .................................................... 424
114: Code Coverage window toolbar .................................................................. 428
115: Code Coverage window context menu commands ........................................ 429
116: Profiling window columns .......................................................................... 431
117: Images window columns ............................................................................. 432
118: Images window context menu commands ................................................... 433
119: Stack window columns ................................................................................ 434
120: Symbols window columns .......................................................................... 436
121: Commands on the Symbols window context menu .................................... 437
122: Debug menu commands .............................................................................. 438
123: Log file options ............................................................................................ 441
124: Description of Disassembly menu commands ............................................. 443
125: Assembler list file options .......................................................................... 470
126: Linker checksum algorithms ....................................................................... 489
127: C-SPY driver options ................................................................................... 494
128: cspybat parameters ..................................................................................... 499
129: Examples of C-SPY macro variables .......................................................... 528
130: C-SPY setup macros .................................................................................... 532
131: Summary of system macros ........................................................................ 533
132: __cancelInterrupt return values ................................................................... 536
133: __disableInterrupts return values ............................................................... 537
134: __driverType return values ............................................................................. 538
135: __emulatorSpeed return values ....................................................................... 539
136: __enableInterrupts return values ..................................................................... 540
137: __evaluate return values ............................................................................. 540
138: __hwReset return values ............................................................................. 541
139: __hwResetRunToBp return values ................................................................... 542
140: __hwResetWithStrategy return values ............................................................ 543
141: __isBatchMode return values ......................................................................... 543
142: __jtagResetTRST return values ...................................................................... 548
143: __loadImage return values ............................................................................ 549
144: __openFile return values ............................................................................. 551
145: __readFile return values ............................................................................. 553
146: __setCodeBreak return values ........................................................................ 557
147: __setDataBreak return values ........................................................................ 558
148: __setLogBreak return values .......................................................................... 559
149: __setSimBreak return values .......................................................................... 560
150: __setTraceStartBreak return values ............................................................... 561
151: __setTraceStopBreak return values ............................................................... 562
152: __sourcePosition return values ...................................................................... 563
153: __unloadImage return values .......................................................................... 566
Figures

1: Create New Project dialog box ................................................................. 34
2: Workspace window ................................................................................. 35
3: Save Workspace As dialog box ............................................................... 36
4: Adding files to project 1 .......................................................................... 37
5: Setting general options ........................................................................... 38
6: Setting compiler options ......................................................................... 39
7: Compilation message .............................................................................. 40
8: Workspace window after compilation ..................................................... 41
9: Setting the option Scan for changed files ............................................... 42
10: Linker options dialog box for project 1 .................................................. 43
11: The C-SPY Debugger main window ...................................................... 46
12: Stepping in C-SPY ................................................................................ 47
13: Using Step Into in C-SPY ...................................................................... 48
14: Inspecting variables in the Auto window .............................................. 49
15: Watching variables in the Watch window ............................................ 49
16: Setting breakpoints ............................................................................... 50
17: Debugging in disassembly mode ............................................................ 51
18: Monitoring memory ............................................................................... 52
19: Displaying memory contents as 16-bit units ........................................ 52
20: Output from the I/O operations .............................................................. 53
21: Reaching program exit in C-SPY ............................................................ 54
22: Assembler settings for creating a list file .............................................. 57
23: Project 2 output in terminal I/O window .............................................. 58
24: Setting a breakpoint in CppTutor.cpp .................................................. 60
25: Setting a breakpoint with skip count .................................................... 61
26: Inspecting the function calls ................................................................. 62
27: Printing Fibonacci sequences .............................................................. 62
28: Specifying the setup macro file for ARM7TDI ..................................... 67
29: Inspecting the interrupt settings for ARM7TDI ..................................... 68
30: Register window for ARM7TDI ............................................................ 71
31: Printing the Fibonacci values in the Terminal I/O window ................. 71
66: Trace Save dialog box ................................................................. 187
67: Function Trace window ............................................................. 188
68: Timeline window ...................................................................... 189
69: Timeline window context menu .................................................. 191
70: Trace Start breakpoints dialog box .............................................. 193
71: Trace Stop breakpoints dialog box .............................................. 194
72: Trace Expressions window ......................................................... 195
73: Find in Trace dialog box ............................................................ 197
74: Find in Trace window ............................................................... 198
75: Simulator menu .......................................................................... 202
76: Memory Access Setup dialog box ................................................. 204
77: Edit Memory Access dialog box ................................................. 206
78: Data breakpoints dialog box ........................................................ 208
79: Immediate breakpoints page ....................................................... 210
80: Breakpoint Usage dialog box ....................................................... 211
81: Simulated interrupt configuration ............................................... 214
82: Simulation states - example 1 ...................................................... 215
83: Simulation states - example 2 ...................................................... 216
84: Interrupt Setup dialog box ........................................................ 217
85: Edit Interrupt dialog box ............................................................ 218
86: Forced Interrupt window ........................................................... 220
87: C-SPY Angel debug monitor communication overview .............. 231
88: C-SPY GDB Server communication overview ............................... 232
89: C-SPY ROM-monitor communication overview ........................... 234
90: C-SPY J-Link communication overview ...................................... 235
91: C-SPY Macraigor communication overview ................................. 238
92: C-SPY RDI communication overview ........................................... 239
93: C-SPY ST-Link communication overview .................................... 240
94: Debugger startup when debugging code in flash .......................... 241
95: Debugger startup when debugging code in RAM ........................... 242
96: C-SPY Download options .......................................................... 245
97: Extra Options page for C-SPY command line options .................. 246
98: C-SPY Angel options .............................................................. 247
99: GDB Server options ................................................................. 249
100: The GDB Server menu ................................................................. 250
101: IAR C-SPY ROM-monitor options .............................................. 251
102: C-SPY J-Link/J-Trace Setup options ........................................... 252
103: C-SPY J-Link/J-Trace Connection options .................................... 256
104: The J-Link menu ................................................................. 257
105: C-SPY LMI FTDI Setup options ..................................................... 260
106: The LMI FTDI menu ................................................................. 261
107: C-SPY Macraigor options ............................................................ 262
108: The Macraigor JTAG menu .......................................................... 264
109: C-SPY RDI options ................................................................. 265
110: The RDI menu ................................................................. 267
111: C-SPY ST-Link setup options ...................................................... 268
112: The ST-Link menu ................................................................. 268
113: C-SPY Third-Party Driver options .................................................. 269
114: Breakpoints options ................................................................. 272
115: Code breakpoints dialog box ......................................................... 274
116: Data breakpoints dialog box .......................................................... 276
117: Data Log breakpoints dialog box ..................................................... 278
118: Breakpoint Usage dialog box .......................................................... 280
119: The Vector Catch dialog box—for ARM9/Cortex-R4 versus for Cortex-M3 280
120: JTAG Watchpoints dialog box ....................................................... 283
121: Trace Start breakpoints dialog box ................................................... 287
122: Trace Stop breakpoints dialog box ................................................... 290
123: Trace Filter breakpoints dialog box ................................................. 292
124: Data Log window ................................................................. 296
125: Data Log window context menu ..................................................... 298
126: Data Log Summary window ........................................................... 299
127: Interrupt Log window ................................................................. 300
128: Interrupt Log Summary window ..................................................... 302
129: Instruction count in Disassembly window ......................................... 305
130: Function Profiler window .............................................................. 306
131: Function Profiler window context menu .......................................... 308
132: Flash Loader Overview dialog box .................................................. 310
133: Flash Loader Configuration dialog box ............................................. 312
134:  IAR Embedded Workbench IDE window ........................................................... 318
135:  IDE toolbar ..................................................................................................... 319
136:  IAR Embedded Workbench IDE window status bar ........................................ 320
137:  Workspace window ....................................................................................... 320
138:  Workspace window context menu ................................................................. 322
139:  Select Source Code Control Provider dialog box ........................................... 324
140:  Source Code Control menu ............................................................................ 324
141:  Select Source Code Control Provider dialog box ........................................... 327
142:  Check In Files dialog box ............................................................................... 327
143:  Check Out Files dialog box ............................................................................. 327
144:  Editor window ................................................................................................. 330
145:  Go to Function window ................................................................................... 331
146:  Editor window tab context menu ................................................................... 331
147:  Editor window context menu .......................................................................... 332
148:  Source Browser window ............................................................................... 336
149:  Source Browser window context menu ......................................................... 338
150:  Breakpoints window ....................................................................................... 340
151:  Breakpoints window context menu ................................................................. 340
152:  Code breakpoints page ................................................................................... 342
153:  Log breakpoints page ..................................................................................... 343
154:  Enter Location dialog box ............................................................................... 345
155:  Resolve Source Ambiguity dialog box ............................................................. 346
156:  Build window (message window) .................................................................. 347
157:  Build window context menu .......................................................................... 347
158:  Find in Files window (message window) ....................................................... 348
159:  Find in Files window context menu ............................................................... 348
160:  Tool Output window (message window) ......................................................... 349
161:  Tool Output window context menu ............................................................... 349
162:  Debug Log window (message window) ......................................................... 349
163:  Debug Log window context menu ................................................................ 350
164:  File menu ........................................................................................................ 351
165:  Edit menu ...................................................................................................... 353
166:  Find in Files dialog box .................................................................................. 358
167:  Incremental Search dialog box ...................................................................... 360
168: Template dialog box ................................................................. 361
169: View menu .................................................................................. 362
170: Project menu .............................................................................. 363
171: Erase Memory dialog box ................................................................ 366
172: Configurations for project dialog box ................................................ 367
173: New Configuration dialog box .................................................... 368
174: Create New Project dialog box .................................................... 369
175: Batch Build dialog box ............................................................... 371
176: Edit Batch Build dialog box ........................................................ 372
177: Tools menu .................................................................................. 373
178: Common Fonts options .............................................................. 374
179: Key Bindings options ................................................................. 375
180: Language options ....................................................................... 376
181: Editor options ............................................................................ 377
182: Configure Auto Indent dialog box ................................................. 380
183: External Editor options ............................................................... 381
184: Editor Setup Files options .......................................................... 382
185: Editor Colors and Fonts options .................................................. 383
186: Messages option ......................................................................... 385
187: Message dialog box containing a Don’t show again option .................................................. 386
188: Project options ........................................................................... 386
189: Source Code Control options ...................................................... 388
190: Debugger options ....................................................................... 389
191: Stack options ............................................................................. 390
192: Register Filter options ............................................................... 392
193: Terminal I/O options ................................................................... 393
194: Configure Tools dialog box ......................................................... 395
195: Customized Tools menu .............................................................. 396
196: Filename Extensions dialog box .................................................. 397
197: Filename Extension Overrides dialog box ..................................... 398
198: Edit Filename Extensions dialog box .......................................... 398
199: Configure Viewers dialog box ..................................................... 399
200: Edit Viewer Extensions dialog box ............................................. 399
201: Window menu ............................................................................ 400
202: C-SPY debug toolbar ................................................................. 405
203: C-SPY Disassembly window ....................................................... 406
204: Disassembly window context menu .......................................... 407
205: Resolve Symbol Ambiguity dialog box ....................................... 409
206: Memory window ....................................................................... 410
207: Memory window context menu .................................................. 412
208: Fill dialog box .......................................................................... 413
209: Memory Save dialog box ............................................................ 414
210: Memory Restore dialog box ......................................................... 415
211: Symbolic Memory window .......................................................... 416
212: Symbolic Memory window context menu .................................... 417
213: Register window ...................................................................... 418
214: Watch window ......................................................................... 419
215: Watch window context menu ...................................................... 419
216: Locals window .......................................................................... 420
217: Auto window ............................................................................ 421
218: Live Watch window ................................................................... 421
219: Quick Watch window ................................................................. 421
220: Statics window .......................................................................... 423
221: Statics window context menu ...................................................... 423
222: Select Statics dialog box ............................................................. 424
223: Call Stack window ..................................................................... 425
224: Call Stack window context menu .............................................. 425
225: Terminal I/O window ................................................................. 426
226: Ctrl codes menu ....................................................................... 426
227: Input Mode dialog box ............................................................... 427
228: Code Coverage window ............................................................. 427
229: Code coverage window context menu ........................................ 429
230: Profiling window ...................................................................... 430
231: Profiling window context menu ................................................ 430
232: Images window ........................................................................ 432
233: Images window context menu .................................................. 432
234: Stack window .......................................................................... 433
235: Stack window context menu ...................................................... 435
236: Symbols window ................................................................. 436
237: Symbols window context menu ........................................... 437
238: Debug menu ........................................................................... 438
239: Autostep settings dialog box ................................................ 439
240: Macro Configuration dialog box ........................................... 440
241: Log File dialog box .............................................................. 441
242: Terminal I/O Log File dialog box .......................................... 442
243: Disassembly menu ............................................................... 442
244: Target options ................................................................. 445
245: Output options ................................................................. 447
246: Library Configuration options ............................................. 448
247: Library Options page .......................................................... 450
248: Multi-file Compilation .......................................................... 453
249: Compiler language options ................................................. 454
250: Compiler code options ....................................................... 456
251: Compiler optimizations options ......................................... 457
252: Compiler output options .................................................... 458
253: Compiler list file options ..................................................... 460
254: Compiler preprocessor options .......................................... 461
255: Compiler diagnostics options ............................................. 463
256: Extra Options page for the compiler ................................. 465
257: Assembler language options ............................................. 467
258: Choosing macro quote characters ...................................... 468
259: Assembler output options .................................................. 469
260: Assembler list file options .................................................. 470
261: Assembler preprocessor options ....................................... 471
262: Assembler diagnostics options ......................................... 473
263: Extra Options page for the assembler ............................... 474
264: Converter output file options ............................................. 475
265: Custom tool options ........................................................ 477
266: Build actions options ....................................................... 479
267: Linker configuration options ............................................... 481
268: Library Usage page ........................................................... 482
269: ILINK input file options ...................................................... 483
270: ILINK output file options ......................................................... 484
271: Linker diagnostics options ........................................................ 485
272: Linker defined symbols options ................................................. 486
273: Linker diagnostics options ........................................................ 487
274: Linker checksum and fill options .............................................. 488
275: Extra Options page for the linker .............................................. 490
276: Library builder output options ................................................. 491
277: Generic C-SPY options .............................................................. 493
278: Images page for C-SPY .............................................................. 495
279: Extra Options page for C-SPY .................................................. 496
280: C-SPY plugin options ............................................................... 497
Preface

Welcome to the IAR Embedded Workbench® IDE User Guide for ARM®. The purpose of this guide is to help you fully use the features in IAR Embedded Workbench with its integrated Windows development tools for the ARM core. The IAR Embedded Workbench IDE is a very powerful Integrated Development Environment that allows you to develop and manage a complete embedded application project.

The user guide includes product overviews and reference information, as well as tutorials that will help you get started. It also describes the processes of editing, project managing, building, and debugging.

Who should read this guide

Read this guide if you want to get the most out of the features and tools available in the IDE. 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 ARM core (refer to the chip manufacturer’s documentation)
- The operating system of your host computer.

Refer to the IAR C/C++ Development Guide for ARM® and ARM® IAR Assembler Reference Guide for more information about the other development tools incorporated in the IDE.

How to use this guide

If you are new to using this product, we suggest that you first read Part 1. Product overview 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, Part 2. Tutorials is a good place to begin. The process of managing projects and building, as well as editing, is described in Part 3. Project management and building, page 79, whereas information about how to use C-SPY is described in Part 4. Debugging, page 115.
If you are an experienced user and need this guide only for reference information, see the reference chapters in Part 7. Reference information and the online help system available from the IAR Embedded Workbench IDE Help menu.

Finally, we recommend the Glossary in the IAR C/C++ Development Guide for ARM® 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.

Part 1. Product overview

This section provides a general overview of all the IAR Systems development tools so that you can become familiar with them:

- Product introduction provides a brief summary and lists the features offered in each of the IAR Systems development tools— IAR Embedded Workbench® IDE, IAR C/C++ Compiler, IAR Assembler, IAR ILINK Linker and its accompanying tools, and IAR C-SPY Debugger—for the ARM core.
- Installed files describes the directory structure and the types of files it contains. The chapter also includes an overview of the documentation supplied with the IAR Systems development tools.

Part 2. Tutorials

The tutorials give you hands-on training to help you get started with using the tools:

- Welcome to the tutorials gives an overview of the tutorials and helps you getting started.
- Creating an application project guides you through setting up a new project, compiling your application, examining the list file, and linking your application. The tutorial demonstrates a typical development cycle, which is continued with debugging in the next chapter.
- Debugging using the IAR C-SPY® Debugger explores the basic facilities of the debugger.
- Mixing C and assembler modules demonstrates how you can easily combine source modules written in C with assembler modules. The chapter also demonstrates how to use the compiler for examining the calling convention.
- Using C++ shows how to create a C++ class, which creates two independent objects. The application is then built and debugged.
Simulating an interrupt shows how to add an interrupt handler to the project and how to simulate this interrupt, using C-SPY facilities for simulated interrupts, breakpoints, and macros.

Creating and using libraries demonstrates how to create library modules.

Part 3. Project management and building
This section describes the process of editing and building your application:

- The development environment introduces you to the IAR Embedded Workbench development environment. The chapter also demonstrates the facilities available for customizing the environment to meet your requirements.
- Managing projects describes how you can create workspaces with multiple projects, build configurations, groups, source files, and options that help you handle different versions of your applications.
- Building discusses the process of building your application.
- Editing contains detailed descriptions of the IAR Embedded Workbench editor, how to use it, and the facilities related to its usage. The final section also contains information about how to integrate an external editor of your choice.

Part 4. Debugging
This section gives conceptual information about C-SPY functionality and how to use it:

- The IAR C-SPY® Debugger introduces some of the concepts that are related to debugging in general and to the IAR C-SPY Debugger in particular. It also introduces you to the C-SPY environment and how to set up, start, and configure the debugger to reflect the target hardware.
- Executing your application describes how to initialize C-SPY, 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 defines 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 define breakpoints.
- Monitoring memory and registers shows how you can examine memory and registers.
- Using the C-SPY® macro system describes the C-SPY macro system, its features, the purposes of these features, and how to use them.
What this guide contains

- **Analyzing your application** presents facilities for analyzing your application.
- **Using trace** gives information about collecting and using trace data as well as detailed reference information about the trace-related windows and dialog boxes. Information is provided for all C-SPY drivers.

**Part 5. IAR C-SPY Simulator**
- **Simulator-specific debugging** describes the functionality specific to the simulator.
- **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.

**Part 6. C-SPY hardware debugger systems**
- **Introduction to C-SPY® hardware debugger systems** introduces you to the available C-SPY debugger drivers other than the simulator driver. The chapter briefly shows the difference in functionality provided by the drivers and the simulator driver.
- **Hardware-specific debugging** describes the additional options, menus, and features provided by the debugger systems.
- **Analyzing your program using driver-specific tools** describes the additional features provided by the debugger systems.
- **Using flash loaders** describes the flash loader, what it is and how to use it.

**Part 7. Reference information**
- **IAR Embedded Workbench® IDE reference** contains detailed reference information about the development environment, such as details about the graphical user interface.
- **C-SPY® reference** provides detailed reference information about the graphical user interface of the IAR C-SPY Debugger.
- **General options** specifies the target, output, library, and MISRA C options.
- **Compiler options** specifies compiler options for language, optimizations, code, output, list file, preprocessor, diagnostics, and MISRA C.
- **Assembler options** describes the assembler options for language, output, list, preprocessor, and diagnostics.
- **Converter options** describes the options available for converting linker output files from the ELF format.
- **Custom build options** describes the options available for custom tool configuration.
- **Build actions options** describes the options available for pre-build and post-build actions.
● **Linker options** describes the ILINK options for output, input, defining symbols, diagnostics, list generation, and configuration.

● **Library builder options** describes the XAR options available in the Embedded Workbench.

● **Debugger options** gives reference information about generic C-SPY options.

● **The C-SPY Command Line Utility—cspybat** describes how to use the debugger in batch mode.

● **C-SPY® macros reference** gives reference information about C-SPY macros, such as a syntax description of the macro language, summaries of the available setup macro functions, and pre-defined system macros. Finally, a description of each system macro is provided.

### Other documentation

The complete set of IAR Systems development tools are described in a series of guides. For information about:

- Programming for the IAR C/C++ Compiler for ARM and linking using the IAR ILINK Linker, refer to the **IAR C/C++ Development Guide for ARM®**
- Programming for the IAR Assembler for ARM, refer to the **ARM® IAR Assembler Reference Guide**
- Using the IAR DLIB Library, refer to the **DLIB Library Reference information**, available in the IDE online help system.
- Porting application code and projects created with a previous version of the IAR Embedded Workbench for ARM, refer to **ARM® IAR Embedded Workbench® Migration Guide**.
- Developing safety-critical applications using the MISRA C guidelines, refer to the **IAR Embedded Workbench® MISRA C:2004 Reference Guide** or the **IAR Embedded Workbench® MISRA C:1998 Reference Guide**.

All of these guides are delivered in hypertext PDF or HTML format on the installation media. Note that additional documentation might be available on the Help menu depending on your product installation.

Recommended web sites:

- The Advanced RISC Machines Ltd web site, [www.arm.com](http://www.arm.com), contains information and news about the ARM cores.

- The IAR Systems web site, [www.iar.com](http://www.iar.com), holds application notes and other product information.
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 arm\doc, the full path to the location is assumed, for example c:\Program Files\IAR Systems\Embedded Workbench 5.n\arm\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 filename.h where filename represents the name of the file.</td>
</tr>
<tr>
<td>[option]</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>...</td>
<td>An ellipsis indicates that the previous item can be repeated an arbitrary number of times.</td>
</tr>
<tr>
<td><img src="icon.png" alt="Icon" /></td>
<td>Identifies instructions specific to the IAR Embedded Workbench® IDE interface.</td>
</tr>
<tr>
<td><img src="icon.png" alt="Icon" /></td>
<td>Identifies instructions specific to the command line interface.</td>
</tr>
<tr>
<td><img src="icon.png" alt="Icon" /></td>
<td>Identifies helpful tips and programming hints.</td>
</tr>
<tr>
<td><img src="icon.png" alt="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 ARM</td>
<td>IAR Embedded Workbench®</td>
</tr>
<tr>
<td>IAR Embedded Workbench® IDE for ARM</td>
<td>the IDE</td>
</tr>
<tr>
<td>IAR C-SPY® Debugger for ARM</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 ARM</td>
<td>the compiler</td>
</tr>
<tr>
<td>IAR Assembler™ for ARM</td>
<td>the assembler</td>
</tr>
<tr>
<td>IAR ILINK Linker™</td>
<td>ILINK, the linker</td>
</tr>
<tr>
<td>IAR DLIB Library™</td>
<td>the DLIB library</td>
</tr>
</tbody>
</table>

Table 2: Naming conventions used in this guide
Part I. Product overview

This part of the IAR Embedded Workbench® IDE User Guide includes the following chapters:

- Product introduction
- Installed files.
Product introduction

The IAR Embedded Workbench® IDE is a very powerful Integrated Development Environment, that allows you to develop and manage complete embedded application projects. It is a development platform, with all the features you would expect to find in your everyday working place.

This chapter describes the IDE and provides a general overview of all the tools that are integrated in this product.

The IAR Embedded Workbench IDE

The IDE is the framework where all necessary tools are seamlessly integrated:

- The highly optimizing IAR C/C++ Compiler
- The IAR Assembler
- The versatile IAR ILINK Linker, including accompanying tools
- A powerful editor
- A project manager
- A command line build utility
- The IAR C-SPY® Debugger, a state-of-the-art high-level language debugger.

IAR Embedded Workbench is available for many microprocessors and microcontrollers in the 8-, 16-, and 32-bit segments, allowing you to stay within a well-known development environment also for your next project. It provides an easy-to-learn and highly efficient development environment with maximum code inheritance capabilities, comprehensive and specific target support. IAR Embedded Workbench promotes a useful working methodology, and you can reduce your development time significantly by using the IAR Systems tools.

If you want detailed information about supported target processors, contact your software distributor or your IAR Systems representative, or visit the IAR Systems web site www.iar.com for information about recent product releases.

AN EXTENSIBLE AND MODULAR ENVIRONMENT

Although the IDE provides all the features required for a successful project, we also recognize the need to integrate other tools. Therefore the IDE is easily adapted to work with your favorite editor and source code control system. The IAR ILINK Linker can produce output files in the ELF format including DWARF for debug information,
allowing for debugging on most third-party emulators. Support for RTOS-aware
debugging and high-level debugging of state machines can also be added to the product.

The compiler, assembler, and linker can also be run from a command line environment,
if you want to use them as external tools in an already established project environment.

FEATURES

The IDE is a flexible integrated development environment, allowing you to develop
applications for a variety of target processors. It provides a convenient Windows
interface for rapid development and debugging.

Project management

The IDE comes with functions that will help you to stay in control of all project
modules, for example, C or C++ source code files, assembler files, include files, and
other related modules. You create workspaces and let them contain one or several
projects. Files can be grouped, and options can be set on all levels—project, group, or
file. Changes are tracked so that a request for rebuild will retranslate all required
modules, making sure that no executable files contain out-of-date modules. This list
shows some additional features:

● Project templates to create a project that can be built and executed out of the box for
  a smooth development startup
● Hierarchical project representation
● Source browser with an hierarchical symbol presentation
● Options can be set globally, on groups of source files, or on individual source files
● The Make command automatically detects changes and performs only the required
  operations
● Text-based project files
● Custom Build utility to expand the standard tool chain in an easy way
● Command line build with the project file as input.

Source code control

Source code control (SCC)—or revision control—is useful for keeping track of different
versions of your source code. IAR Embedded Workbench can identify and access any
third-party source code control system that conforms to the SCC interface published by
Microsoft.

Window management

To give you full and convenient control of the placement of the windows, each window
is dockable and you can optionally organize the windows in tab groups. The system of
dockable windows also provides a space-saving way to keep many windows open at the same time. It also makes it easy to rearrange the size of the windows.

The text editor

The integrated text editor allows editing of multiple files in parallel, and provides all basic editing features expected from a modern editor, including unlimited undo/redo and automatic completion. In addition, it provides functions specific to software development, like coloring of keywords (C/C++, assembler, and user-defined), block indent, and function navigation within source files. It also recognizes C language elements like matching brackets. This list shows some additional features:

- Context-sensitive help system that can display reference information for DLIB library functions
- Syntax of C or C++ programs and assembler directives shown using text styles and colors
- Powerful search and replace commands, including multi-file search
- Direct jump to context from error listing
- Multibyte character support
- Parenthesis matching
- Automatic indentation
- Bookmarks
- Unlimited undo and redo for each window.

DOCUMENTATION

The IDE is documented in the *IAR Embedded Workbench® IDE User Guide for ARM®* (this guide). Help and hypertext PDF versions of the user documentation are available online.

**IAR C-SPY Debugger**

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

- Editing while debugging. During a debug session, you can make corrections directly into the same source code window that is used to control the debugging. Changes will be included in the next project rebuild.
Setting source code breakpoints before starting the debugger. Breakpoints in source code will be associated with the same piece of source code even if additional code is inserted.

THE C-SPY DRIVER
C-SPY consists both of a general part which provides a basic set of debugger features, and of a driver. The C-SPY driver is the part that provides communication with and control of the target system. The driver also provides a user interface—special menus, windows, and dialog boxes—to the features that the target system provides, for instance, special breakpoints.

Contact your software distributor or IAR Systems representative for information about available C-SPY drivers. You can also find information on the IAR Systems website, www.iar.com.

Depending on your product installation, C-SPY is available with a simulator driver and optional drivers for hardware debugger systems.

For a brief overview of the available C-SPY drivers, see IAR C-SPY Debugger systems, page 9.

GENERAL C-SPY DEBUGGER FEATURES
Because IAR Systems provides an entire tool chain, 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 the general features described in this section.

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.

Debugging the C or C++ source code provides the quickest and easiest way of verifying the program logic of your application whereas disassembly debugging lets you focus on the critical sections of your application, and provides you with precise control over the hardware. In Mixed-Mode display, the debugger also displays the corresponding C/C++ source code interleaved with the disassembly listing.

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.

The debug information also presents inlined functions as if a call was made, making the source code of the inlined function available.

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. You can set a code breakpoint to investigate whether your program logic is correct. You can also set a data breakpoint, to investigate how and when the data changes. Finally, you can add conditions and connect actions to your breakpoints.

Monitoring variables and expressions
When you work with variables and expressions you are presented with 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 premium 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 registers available.

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 solely 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:

- A modular and extensible architecture allowing third-party extensions to the debugger, for example, real-time operating systems, peripheral simulation modules, and emulator drivers
- Threaded execution keeps the IDE responsive while running the target application
- Automatic stepping
- Source browser provides easy navigation to functions, types and variables
- Extensive type recognition of variables
- Configurable registers (CPU and peripherals) and memory windows
- Dedicated Stack window
- Support for code coverage and function-level profiling
- Target application can access files on the host computer using file I/O
- Optional terminal I/O emulation.

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, and can be supplied by third-party vendors. Examples of such modules are:

- Code Coverage, Profiling, and the Stack window, all well-integrated in the IDE.
- The various C-SPY drivers for debugging using certain debug systems.
- RTOS plugin modules for support for real-time OS awareness debugging.
- Peripheral simulation modules make C-SPY simulate peripheral units. Such plugin modules are not provided by IAR Systems, but can be developed and distributed by third-party suppliers.
- 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.
RTOS AWARENESS

C-SPY supports real-time OS awareness debugging. These operating systems are currently supported:

- IAR PowerPac RTOS
- CMX CMX-RTX and CMX-Tiny+ real-time operating systems
- Micrium μC/OS-II
- Express Logic ThreadX
- RTXC Quadros RTOS
- Fusion RTOS
- OSEK (ORTI)
- OSE Epsilon
- Micro Digital SMX RTOS
- NORTi MiSPO
- Segger embOS.

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.

DOCUMENTATION

C-SPY is documented in the IAR Embedded Workbench® IDE User Guide for ARM® (this guide). Generic debugger features are described in Part 4. Debugging, whereas features specific to each debugger driver are described in Part 5. IAR C-SPY Simulator, and Part 6. C-SPY hardware debugger systems. There are also help and hypertext PDF versions of the documentation available online.

IAR C-SPY Debugger systems

At the time of writing this guide, the IAR C-SPY Debugger for the ARM core is available with drivers for these target systems:

- Simulator
- RDI (Remote Debug Interface)
- J-Link / J-Trace JTAG interface
- Macraigor JTAG interface
- Angel debug monitor
- IAR ROM-monitor for Analog Devices ADuC7xxx boards, IAR Kickstart Card for Philips LPC210x, and OKI evaluation boards
IAR C-SPY Debugger systems

- Luminary FTDI JTAG interface (for Cortex devices only)
- ST-Link JTAG interface (for ST Cortex-M devices only).

Contact your software distributor or IAR Systems representative for information about available C-SPY drivers. You can also find information on the IAR Systems web site, www.iar.com.

For further details about the concepts that are related to the IAR C-SPY Debugger, see Debugger concepts, page 117. In the following sections you can find general descriptions of the various drivers.

IAR C-SPY SIMULATOR

The C-SPY simulator driver simulates the functions of the target processor entirely in software. With this driver, 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 of C-SPY, the simulator driver also provides:

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

For additional information about the IAR C-SPY Simulator, refer to Part 5. IAR C-SPY Simulator in this guide.

IAR C-SPY J-LINK DRIVER

Using the IAR ARM C-SPY J-Link driver, C-SPY can connect to the Segger J-Link/J-Trace JTAG interface.

Features

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

- Execution in real time
- Communication through USB
- Zero memory footprint on the target system
• Use of the two available hardware breakpoints in the ARM core to allow debugging code in non-volatile memory such as flash. Cortex devices have support for six hardware breakpoints
• Direct access to the ARM core watchpoint registers
• An unlimited number of breakpoints when debugging code in RAM
• A possibility for the debugger to attach to a running application without resetting the target system.

Note: Code coverage is supported by J-Trace. Live watch is supported by the C-SPY J-Link/J-Trace driver for Cortex devices. For ARM7/9 devices it is supported if a DCC handler is added to your application.

For additional information about the IAR C-SPY J-Link driver, see Part 6. C-SPY hardware debugger systems.

IAR C-SPY LMI FTDI DRIVER

Using the IAR ARM C-SPY LMI FTDI driver, C-SPY can connect to the Luminary FTDI JTAG interface.

Features

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

• Support for Luminary Cortex devices
• Execution in real time
• Communication through USB
• Zero memory footprint on the target system
• Use of the six available hardware breakpoints
• An unlimited number of breakpoints when debugging code in RAM.

Note: Code coverage is not supported by the LMI FTDI driver. Live watch is supported.

For additional information about the IAR C-SPY LMI FTDI driver, see Part 6. C-SPY hardware debugger systems.

IAR C-SPY RDI DRIVER

Using the IAR ARM C-SPY RDI driver, C-SPY can connect to an RDI-compliant debug system. This can, for example, be a simulator, a ROM-monitor, a JTAG interface, or an emulator. The IAR ARM C-SPY RDI driver is compliant with the RDI specification 1.5.1.
In the feature list below, an RDI-based connection to a JTAG interface is assumed.

**Features**

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

- Execution in real time
- High-speed communication through USB, Ethernet, or the parallel port depending on the RDI-compatible JTAG interface used
- Zero memory footprint on the target system
- Use of the two available hardware breakpoints in the ARM core to allow debugging code in non-volatile memory, such as flash
- An unlimited number of breakpoints when debugging code in RAM
- A possibility for the debugger to attach to a running application without resetting the target system.

**Note:** Code coverage and live watch are not supported by the C-SPY RDI driver.

For additional information about the IAR C-SPY RDI driver, see Part 6. *C-SPY hardware debugger systems*.

**IAR C-SPY MACRAIGOR DRIVER**

Using the IAR ARM C-SPY Macraigor JTAG driver, C-SPY can connect to the Macraigor mpDemon, USB2 Demon, and USB2 Sprite JTAG interfaces.

**Features**

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

- Execution in real time
- Communication through the parallel port, Ethernet, or USB
- Zero memory footprint on the target system
- Use of the two available hardware breakpoints in the ARM core to allow debugging code in non-volatile memory such as flash
- Direct access to the ARM core watchpoint registers
- An unlimited number of breakpoints when debugging code in RAM
- A possibility for the debugger to attach to a running application without resetting the target system.

**Note:** Code coverage and live watch are not supported by the C-SPY Macraigor JTAG driver.
For additional information about the IAR C-SPY Macraigor JTAG driver, see Part 6.

C-SPY hardware debugger systems.

**IAR C-SPY ROM-MONITOR DRIVER**

Using the IAR ROM-monitor driver, C-SPY can connect to the Analog Devices ADuC7xxx boards, the IAR Kickstart Card for Philips LPC210x, and OKI evaluation boards. The boards contain firmware (the ROM-monitor itself) that runs in parallel with your application software.

**Features for Analog Devices evaluation boards**

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

- Execution in real time
- Communication through serial port
- Support for the Analog Devices ADuC7xxx evaluation board
- Download and debug in flash memory
- An unlimited number of breakpoints in both flash memory and RAM.

**Note:** Code coverage and live watch are not supported by the C-SPY ROM-monitor driver.

For additional information about the IAR C-SPY ROM-monitor driver, see Part 6. C-SPY hardware debugger systems.

**Features for IAR Kickstart Card for Philips LPC210x**

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

- Execution in real time
- Communication through the RS232 serial port
- Support for the IAR Kickstart Card for Philips LPC210x.

**Note:** Code coverage and live watch are not supported by the C-SPY ROM-monitor driver.

**Features for OKI evaluation boards**

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

- Execution in real time
- Communication through serial port or USB connection
Support for the OKI JOB671000 evaluation board.

**Note:** Code coverage and live watch are not supported by the C-SPY OKI ROM-monitor driver.

For additional information about the IAR C-SPY ROM-monitor driver, see Part 6. *C-SPY hardware debugger systems.*

**IAR C-SPY ANGEL DEBUG MONITOR DRIVER**

Using the IAR Angel debug monitor driver, you can communicate with any device compliant with the Angel debug monitor protocol. In most cases these are evaluation boards. However, the EPI JEENI JTAG interface also uses this protocol. When connecting to an evaluation board the Angel firmware will run in parallel with your application software.

**Features**

In addition to the general features of the IAR C-SPY Debugger the ANGEL debug monitor driver also provides:

- Execution in real time
- Communication through the serial port or Ethernet
- Support for all Angel equipped evaluation boards
- Support for the EPI JEENI JTAG interface.

**Note:** Code coverage and live watch are not supported by the C-SPY Angel debug monitor driver.

For additional information about the IAR C-SPY Angel debug monitor driver, see Part 6. *C-SPY hardware debugger systems.*

**IAR C-SPY ST-LINK DRIVER**

Using the IAR C-SPY ST-Link driver, C-SPY can connect to the ST-Link JTAG interface.

**Features**

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

- Support for ST Cortex-M devices
- Execution in real time
- Communication through USB
- Zero memory footprint on the target system
• Use of the six available hardware breakpoints
• An unlimited number of breakpoints when debugging code in RAM.

Note: Code coverage and live watch are not supported by the ST-Link driver.

For additional information about the IAR C-SPY ST-Link driver, see Part 6, C-SPY hardware debugger systems.

IAR C/C++ Compiler

The IAR C/C++ Compiler for ARM is a state-of-the-art compiler that offers the standard features of the C or C++ languages, plus many extensions designed to take advantage of the ARM-specific facilities.

The compiler is integrated with other IAR Systems software in the IDE.

FEATURES

The compiler provides the following features:

Code generation
• Generic and ARM-specific optimization techniques produce very efficient machine code
• Comprehensive output options, including relocatable object code, assembler source code, and list files with optional assembler mnemonics
• Support for ARM EABI ELF/DWARF object format
• The object code can be linked together with assembler routines
• Generation of extensive debug information.

Language facilities
• Support for the C and C++ programming languages
• Support for IAR Extended EC++ with features such as full template support, namespace support, the cast operators static_cast, const_cast, and reinterpret_cast, as well as the Standard Template Library (STL).
• Conformance to the ISO/ANSI C standard for a free-standing environment
• Target-specific language extensions, such as special function types, extended keywords, pragma directives, predefined symbols, intrinsic functions, absolute allocation, and inline assembler
• Standard library of functions applicable to embedded systems
• IEEE-compatible floating-point arithmetic
Interrupt functions can be written in C or C++.

**Type checking**
- Extensive type checking at compile time
- External references are type checked at link time
- Link-time inter-module consistency checking of the application.

**RUNTIME ENVIRONMENT**
The IAR Embedded Workbench for ARM supports the IAR DLIB Library, which supports ISO/ANSI C and C++. This library also supports floating-point numbers in IEEE 754 format, multi-byte characters, and locales.

Several mechanisms for customizing the runtime environment and the runtime libraries are available. For the runtime library, library source code is included.

**DOCUMENTATION**
The compiler is documented in the *IAR C/C++ Development Guide for ARM®*.

---

**IAR Assembler**

The IAR Assembler is integrated with other IAR Systems software for the ARM core. It is a powerful relocating macro assembler (supporting the Intel/Motorola style) with a versatile set of directives and expression operators. The assembler features a built-in C language preprocessor and supports conditional assembly.

The assembler uses the same mnemonics and operand syntax as the Advanced RISC Machines Ltd Assembler for ARM, which simplifies the migration of existing code. For detailed information, see the *ARM® IAR Assembler Reference Guide*.

**FEATURES**
The IAR Assembler provides these features:
- C preprocessor
- List file with extensive cross-reference output
- Number of symbols and program size limited only by available memory
- Support for complex expressions with external references
- 255 significant characters in symbol names
- Support for ARM EABI ELF/DWARF object format.
IAR ILINK Linker and accompanying tools

The IAR ILINK Linker is a powerful, flexible software tool for use in the development of embedded applications. It is equally well suited for linking small, single-file, absolute assembler programs as it is for linking large, relocatable, multi-module, C/C++, or mixed C/C++ and assembler programs.

ILINK combines one or more relocatable object files—produced by the IAR Systems compiler or assembler—with selected parts of one or more object libraries to produce an executable image in the industry-standard format ELF (Executable and Linking Format).

ILINK will automatically load only those library modules—user libraries and standard C or C++ library variants—that are actually needed by the application you are linking. Further, ILINK eliminates duplicate sections and sections that are not required.

ILINK can link both ARM and Thumb code, as well as a combination of them. By automatically inserting additional instructions (veneers), ILINK will assure that the destination will be reached for any calls and branches, and that the processor state is switched when required.

ILINK uses a configuration file where you can specify separate locations for code and data areas of your target system memory map. This file also supports automatic handling of the application’s initialization phase, which means initializing global variable areas and code areas by copying initializers and possibly decompressing them as well.

The final output produced by ILINK is an absolute object file containing the executable image in the ELF (including DWARF for debug information) format. The file can be downloaded to C-SPY or any other debugger that supports ELF/DWARF, or it can be programmed into EPROM.

To handle ELF files, various tools are included, such as an archiver, an ELF reader, and a format converter.

FEATURES

- Flexible section commands allow detailed control of code and data placement
- Link-time symbol definition enables flexible configuration control
- Optional code checksum generation for runtime checking
- Removes unused code and data
- Support for ARM EABI ELF/DWARF object format.
DOCUMENTATION

The IAR ILINK Linker is documented in the IAR C/C++ Development Guide for ARM®.
Installed files

This chapter describes which directories are created during installation and what file types are used. At the end of the chapter, there is a section that describes what information you can find in the various guides and online documentation.

Refer to the QuickStart Card and the Installation and Licensing Guide, which are delivered with the product, for system requirements and information about how to install and register the IAR Systems products.

Directory structure

The installation procedure creates several directories to contain the various types of files used with the IAR Systems development tools. The following sections give a description of the files contained by default in each directory.

ROOT DIRECTORY

The root directory created by the default installation procedure is the \Program Files\IAR Systems\Embedded Workbench 5.n\ directory where x is the drive where Microsoft Windows is installed and 5.n is the version number of the IDE.

THE ARM DIRECTORY

The arm directory contains all product-specific subdirectories.

<table>
<thead>
<tr>
<th>Directory</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>arm\bin</td>
<td>The arm\bin subdirectory contains executable files for ARM-specific components, such as the compiler, the assembler, the linker and the library tools, and the C-SPY® drivers.</td>
</tr>
</tbody>
</table>

Table 3: The ARM directory
### Directory Structure

<table>
<thead>
<tr>
<th>Directory</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>arm\config</td>
<td>The arm\config subdirectory contains files used for configuring the development environment and projects, for example:</td>
</tr>
<tr>
<td></td>
<td>• Linker configuration files (*.icf)</td>
</tr>
<tr>
<td></td>
<td>• C-SPY device description files (*.ddf)</td>
</tr>
<tr>
<td></td>
<td>• Device selection files (*.i79, *.menu)</td>
</tr>
<tr>
<td></td>
<td>• Flash loader applications for various devices (*.out)</td>
</tr>
<tr>
<td></td>
<td>• Syntax coloring configuration files (*.cfg)</td>
</tr>
<tr>
<td></td>
<td>• Project templates for both application and library projects (*.ewp), and for the library projects, the corresponding library configuration files.</td>
</tr>
<tr>
<td>arm\doc</td>
<td>The arm\doc subdirectory contains release notes with recent additional information about the ARM tools. We recommend that you read all of these files.</td>
</tr>
<tr>
<td></td>
<td>The directory also contains online versions in hypertext PDF format of this user guide, and of the ARM reference guides, as well as online help files (*.chm).</td>
</tr>
<tr>
<td>arm\drivers</td>
<td>The arm\drivers subdirectory contains low-level device drivers, typically USB drivers required by the C-SPY drivers.</td>
</tr>
<tr>
<td>arm\examples</td>
<td>The arm\examples subdirectory contains files related to example projects, which can be opened from the Information Center.</td>
</tr>
<tr>
<td>arm\inc</td>
<td>The arm\inc subdirectory holds include files, such as the header files for the standard C or C++ library. There are also specific header files that define special function registers (SFRs); these files are used by both the compiler and the assembler.</td>
</tr>
<tr>
<td>arm\lib</td>
<td>The arm\lib subdirectory holds prebuilt libraries and the corresponding library configuration files, used by the compiler.</td>
</tr>
<tr>
<td>arm\plugins</td>
<td>The arm\plugins subdirectory contains executable files and description files for components that can be loaded as plugin modules.</td>
</tr>
<tr>
<td>arm\powerpac</td>
<td>The arm\powerpac subdirectory contains files related to the add-on product IAR PowerPac.</td>
</tr>
<tr>
<td>arm\src</td>
<td>The arm\src subdirectory holds source files for some configurable library functions. This directory also holds the library source code and the source code for ELF utilities.</td>
</tr>
<tr>
<td>arm\tutor</td>
<td>The arm\tutor subdirectory contains the files used for the tutorials in this guide.</td>
</tr>
</tbody>
</table>

*Table 3: The ARM directory (Continued)*
THE COMMON DIRECTORY

The common directory contains subdirectories for components shared by all IAR Embedded Workbench products.

<table>
<thead>
<tr>
<th>Directory</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>common\bin</td>
<td>The common\bin subdirectory contains executable files for components common to all IAR Embedded Workbench products, such as the editor and the graphical user interface components. The executable file for the IDE is also located here.</td>
</tr>
<tr>
<td>common\config</td>
<td>The common\config subdirectory contains files used by the IDE for settings in the development environment.</td>
</tr>
<tr>
<td>common\doc</td>
<td>The common\doc subdirectory contains release notes with recent additional information about the components common to all IAR Embedded Workbench products, such as the linker and library tools. We recommend that you read these files.</td>
</tr>
<tr>
<td>common\plugins</td>
<td>The common\plugins subdirectory contains executable files and description files for components that can be loaded as plugin modules, for example modules for Code coverage and Profiling.</td>
</tr>
</tbody>
</table>

Table 4: The common directory

THE INSTALL-INFO DIRECTORY

The install-info directory contains metadata (version number, name, etc.) about the installed product components. Do not modify these files.

File types

The ARM versions of the IAR Systems development tools use the following default filename extensions to identify the produced files and other recognized file types:

<table>
<thead>
<tr>
<th>Ext.</th>
<th>Type of file</th>
<th>Output from</th>
<th>Input to</th>
</tr>
</thead>
<tbody>
<tr>
<td>asm</td>
<td>Assembler source code</td>
<td>Text editor</td>
<td>Assembler</td>
</tr>
<tr>
<td>bat</td>
<td>Windows command batch file</td>
<td>C-SPY</td>
<td>Windows</td>
</tr>
<tr>
<td>c</td>
<td>C source code</td>
<td>Text editor</td>
<td>Compiler</td>
</tr>
<tr>
<td>cfg</td>
<td>Syntax coloring configuration</td>
<td>Text editor</td>
<td>IDE</td>
</tr>
<tr>
<td>chm</td>
<td>Online help system</td>
<td>--</td>
<td>IDE</td>
</tr>
<tr>
<td>cpp</td>
<td>C++ source code</td>
<td>Text editor</td>
<td>Compiler</td>
</tr>
<tr>
<td>dat</td>
<td>Macros for formatting of STL containers</td>
<td>IDE</td>
<td>IDE</td>
</tr>
<tr>
<td>dbgt</td>
<td>Debugger desktop settings</td>
<td>C-SPY</td>
<td>C-SPY</td>
</tr>
</tbody>
</table>

Table 5: File types
<table>
<thead>
<tr>
<th>Ext.</th>
<th>Type of file</th>
<th>Output from</th>
<th>Input to</th>
</tr>
</thead>
<tbody>
<tr>
<td>ddf</td>
<td>Device description file</td>
<td>Text editor</td>
<td>C-SPY</td>
</tr>
<tr>
<td>dep</td>
<td>Dependency information</td>
<td>IDE</td>
<td>IDE</td>
</tr>
<tr>
<td>dni</td>
<td>Debugger initialization file</td>
<td>C-SPY</td>
<td>IDE</td>
</tr>
<tr>
<td>ewd</td>
<td>Project settings for C-SPY</td>
<td>IDE</td>
<td>IDE</td>
</tr>
<tr>
<td>ewp</td>
<td>IAR Embedded Workbench project (current version)</td>
<td>IDE</td>
<td>IDE</td>
</tr>
<tr>
<td>ewplugin</td>
<td>IDE description file for plugin modules</td>
<td>--</td>
<td>IDE</td>
</tr>
<tr>
<td>eww</td>
<td>Workspace file</td>
<td>IDE</td>
<td>IDE</td>
</tr>
<tr>
<td>flash</td>
<td>Flash memory description file</td>
<td>Text editor</td>
<td>C-SPY</td>
</tr>
<tr>
<td>flashdict</td>
<td>Container for flash files</td>
<td>Text editor</td>
<td>C-SPY</td>
</tr>
<tr>
<td>fmt</td>
<td>Formatting information for the Locals and Watch windows</td>
<td>IDE</td>
<td>IDE</td>
</tr>
<tr>
<td>h</td>
<td>C/C++ or assembler header source</td>
<td>Text editor</td>
<td>Compiler or assembler #include</td>
</tr>
<tr>
<td>helpfiles</td>
<td>Help menu configuration file</td>
<td>Text editor</td>
<td>IDE</td>
</tr>
<tr>
<td>html.htm</td>
<td>HTML document</td>
<td>Text editor</td>
<td>IDE</td>
</tr>
<tr>
<td>i</td>
<td>Preprocessed source</td>
<td>Compiler</td>
<td>Compiler</td>
</tr>
<tr>
<td>i79</td>
<td>Device selection file</td>
<td>Text editor</td>
<td>IDE</td>
</tr>
<tr>
<td>icf</td>
<td>Linker configuration file</td>
<td>Text editor</td>
<td>ILINK linker</td>
</tr>
<tr>
<td>inc</td>
<td>Assembler header source</td>
<td>Text editor</td>
<td>Assembler #include</td>
</tr>
<tr>
<td>ini</td>
<td>Project configuration</td>
<td>IDE</td>
<td>–</td>
</tr>
<tr>
<td>log</td>
<td>Log information</td>
<td>IDE</td>
<td>–</td>
</tr>
<tr>
<td>lst</td>
<td>List output</td>
<td>Compiler and assembler</td>
<td>–</td>
</tr>
<tr>
<td>mac</td>
<td>C-SPY macro definition</td>
<td>Text editor</td>
<td>C-SPY</td>
</tr>
<tr>
<td>menu</td>
<td>Device selection file</td>
<td>Text editor</td>
<td>IDE</td>
</tr>
<tr>
<td>o</td>
<td>Object module</td>
<td>Compiler and assembler</td>
<td>ILINK</td>
</tr>
<tr>
<td>out</td>
<td>Target application</td>
<td>ILINK</td>
<td>EPROM, C-SPY, etc.</td>
</tr>
<tr>
<td>out</td>
<td>Target application with debug information</td>
<td>ILINK</td>
<td>C-SPY and other symbolic debuggers</td>
</tr>
<tr>
<td>pbd</td>
<td>Source browse information</td>
<td>IDE</td>
<td>IDE</td>
</tr>
</tbody>
</table>

Table 5: File types (Continued)
When you run the IDE, some files are created and located in dedicated directories under your project directory, by default $PROJ_DIR$\Debug, $PROJ_DIR$\Release, $PROJ_DIR$\settings, and the file *.*dep under the installation directory. None of these directories or files affect the execution of the IDE, which means you can safely remove them if required.

**FILES WITH NON-DEFAULT FILENAME EXTENSIONS**

In the IDE you can increase the number of recognized filename extensions using the Filename Extensions dialog box, available from the Tools menu. You can also connect your filename extension to a specific tool in the tool chain. See Filename Extensions dialog box, page 397.

To override the default filename extension from the command line, include an explicit extension when you specify a filename.

### Documentation

This section briefly describes the information that is available in the ARM user and reference guides, in the online help, and on the Internet.

You can access the ARM online documentation from the Help menu in the IDE. Help is also available via the F1 key in the IDE.

We recommend that you read the file readme.htm for recent information that might not be included in the user guides. It is located in the arm\doc directory.
THE USER AND REFERENCE GUIDES

The user and reference guides provided with IAR Embedded Workbench are as follows:

IAR Embedded Workbench® IDE User Guide for ARM®

This guide. For a brief overview, see What this guide contains, page xlii.

IAR C/C++ Development Guide for ARM®

This guide provides reference information about the IAR C/C++ Compiler and the IAR ILINK Linker for ARM. Refer to this guide for information about:

- How to configure the compiler to suit your target processor and application requirements
- How to write efficient code for your target processor
- The assembler language interface and the calling convention
- The available data types
- The runtime libraries
- The IAR Systems language extensions
- The IAR ILINK Linker reference sections provide information about ILINK invocation syntax, environment variables, diagnostics, options, and syntax for the linker configuration file.

ARM® IAR Assembler Reference Guide

This guide provides reference information about the IAR Assembler, including details of the assembler source format, and reference information about the assembler operators, directives, mnemonics, and diagnostics.

DLIB Library Reference information

This online documentation in HTML format provides reference information about the IAR DLIB library functions. It is available from the IAR Embedded Workbench® IDE online help system.

IAR Embedded Workbench® MISRA C Reference Guide

This online guide in hypertext PDF format describes how IAR Systems has interpreted and implemented the rules given in Guidelines for the Use of the C Language in Vehicle Based Software to enforce measures for stricter safety in the ISO standard for the C programming language [ISO/IEC 9899:1990].
ONLINE HELP
The context-sensitive online help contains reference information about the menus and dialog boxes in the IDE. It also contains 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.

IAR SYSTEMS ON THE WEB
You can find the latest news from IAR Systems at the web site www.iar.com, available from the Help menu in the IDE. Visit it for information about:

- Product announcements
- Updates and news about current versions
- Special offerings
- Evaluation copies of the IAR Systems products
- Technical Support, including technical notes
- Application notes
- Links to chip manufacturers and other interesting sites
- Distributors; the names and addresses of distributors in each country.
Part 2. Tutorials

This part of the IAR Embedded Workbench® IDE User Guide contains the following chapters:

- Welcome to the tutorials
- Creating an application project
- Debugging using the IAR C-SPY® Debugger
- Mixing C and assembler modules
- Using C++
- Simulating an interrupt
- Creating and using libraries.
Welcome to the tutorials

The tutorials give you hands-on training to help you get started using the IAR Embedded Workbench IDE and its tools.

Below you will get an overview of the tutorials.

Tutorials overview

The tutorials are divided into different parts. You can work through all tutorials as a suite or you can choose to go through the tutorials individually.

Note: The tutorials call the printf library function, which calls the low-level write function part of the DLIB library or the putchar function part of the CLIB library. This works in the C-SPY simulator, but if you want to run the tutorials in a release configuration on real hardware, you must provide your own version of these functions (depending on the library that you are using), adapted to your hardware.

CREATING AN APPLICATION PROJECT

This tutorial guides you through how to set up a new project, compiling your application, examining the list file, and linking your application. These are the related files:

Workspace: tutorials.eww
Project files: project1.ewp
Source files: Tutor.c, Tutor.h, Utilities.c, and Utilities.h

DEBUGGING USING THE IAR C-SPY® DEBUGGER

This tutorial explores the basic facilities of the debugger while debugging the application used in project1. These are the related files:

Workspace: tutorials.eww
Project files: project1.ewp
Source files: Tutor.c, Tutor.h, Utilities.c, and Utilities.h
MIXING C AND ASSEMBLER MODULES
This tutorial demonstrates how you can easily combine source modules written in C with assembler modules. The chapter also demonstrates how to use the compiler for examining the compiler calling convention. These are the related files:

Workspace: tutorials.eww
Project files: project2.ewp
Source files: Tutor.c, Tutor.h, Utilities.c, Utilities.h, and Write.s

USING C++
This tutorial demonstrates how to create a C++ class, which creates two independent objects. The application is then built and debugged. These are the related files:

Workspace: tutorials.eww
Project files: project3.ewp
Source files: CppTutor.cpp, Fibonacci.cpp, and Fibonacci.h

SIMULATING AN INTERRUPT
This tutorial demonstrates how you add an interrupt handler to the project and how you simulate this interrupt, using C-SPY facilities for simulated interrupts, breakpoints, and macros. These are the related files:

Workspace: tutorials.eww
Project files: project4.ewp (ARM7TDMI), project4CM3.ewp (Cortex-M3)
Source files (ARM7TDMI): Interrupt.c, Utilities.c, and Utilities.h
Source files (Cortex-M3): InterruptCM3.c, Utilities.c, Utilities.h, and CstartupCM3.s

CREATING AND USING LIBRARIES
This tutorial demonstrates how to create library modules. These are the related files:

Workspace: tutorials.eww
Project files: project5.ewp and tutor_library.ewp
Source files: Main.s, MaxMin.s, and Utilities.h
GETTING STARTED

You can access the tutorials from the Information Center available from the Help menu in the IDE. Click TUTORIALS and then click the Open the tutorial project hyperlink. This will create a copy of the workspace and its associated project files.

You can find all the files needed for the tutorials in the arm\tutor directory. Make a copy of the tutor directory in your projects directory.

Note: You can customize the amount of information to be displayed in the Build messages window. In the tutorial projects, the default setting is not used. Thus, the contents of the Build messages window on your screen might differ from the screenshots in the text.

Now you can start with the first tutorial project: Creating an application project, page 33.
Tutorials overview
Creating an application project

This chapter introduces you to the IAR Embedded Workbench® integrated development environment (IDE). The tutorial demonstrates a typical development cycle and shows how you use the compiler and the linker to create a small application for the ARM core. For instance, creating a workspace, setting up a project with C source files, and compiling and linking your application.

For a short overview of all tutorials and their related files, see Tutorials overview, page 29.

Setting up a new project

Using the IDE, you can design advanced project models. You create a workspace to which you add one or several projects. There are ready-made project templates for both application and library projects. Each project can contain a hierarchy of groups in which you collect your source files. For each project you can define one or several build configurations. For more details about designing project models, see the chapter Managing projects in this guide.

Because the application in this tutorial is a simple application with very few files, the tutorial does not need an advanced project model.

Before you can create your project, you must first create a workspace.

CREATING A WORKSPACE

The first step is to create a new workspace for the tutorial application. When you start the IDE for the first time, there is already a ready-made workspace, which you can use for the tutorial projects. If you are using that workspace, you can ignore the first step.

Choose File>New>Workspace. Now you are ready to create a project and add it to the workspace.
CREATING THE NEW PROJECT

Note: If you copied all files from the arm\tutor directory before you started with the tutorials, a project file will already be available in your projects directory. You can use that ready-made file, or create your own file.

1. To create a new project, choose Project>Create New Project. The Create New Project dialog box appears, which lets you base your new project on a project template.

2. From the Tool chain drop-down list, choose the tool chain you are using and click OK.

3. For this tutorial, select the project template Empty project, which simply creates an empty project that uses default project settings.

4. In the standard Save As dialog box that appears, specify where you want to place your project file, that is, in your newly created projects directory. Type project1 in the File name box, and click Save to create the new project.
The project will appear in the Workspace window.

![Workspace window](image)

*Figure 2: Workspace window*

By default, two build configurations are created: Debug and Release. In this tutorial only Debug will be used. You choose the build configuration from the drop-down menu at the top of the window. The asterisk in the project name indicates that there are changes that have not been saved.

**Note:** The tutorials call the `printf` library function, which calls the low-level `write` function that works in the C-SPY simulator. If you want to run the tutorials in a release configuration on real hardware, you must provide your own `write` function that has been adapted to your hardware.

A project file—with the filename extension `ewp`—will be created in the `projects` directory, not immediately, but later on when you save the workspace. This file contains information about your project-specific settings, such as build options.
Before you add any files to your project, you should save the workspace. Choose File>Save Workspace and specify where you want to place your workspace file. In this tutorial, you should place it in your newly created projects directory. Type tutorials in the File name box, and click Save to create the new workspace.

A workspace file—with the filename extension eww—has now been created in the projects directory. This file lists all projects that you will add to the workspace. Information related to the current session, such as the placement of windows and breakpoints is located in the files created in the projects\settings directory.

ADDITION FILES TO THE PROJECT

This tutorial uses the source files Tutor.c and Utilities.c.

- The Tutor.c application is a simple program using only standard features of the C language. It initializes an array with the ten first Fibonacci numbers and prints the result to stdout.
- The Utilities.c application contains utility routines for the Fibonacci calculations.

Creating several groups is a possibility for you to organize your source files logically according to your project needs. However, because this project only contains two files, you do not need to create a group. For more information about how to create complex project structures, see the chapter Managing projects.

In the Workspace window, select the destination to which you want to add a source file; a group or, as in this case, directly to the project.
2 Choose Project>Add Files to open a standard browse dialog box. Locate the files Tutor.c and Utilities.c, select them in the file selection list, and click Open to add them to the project1 project.

![Image: Adding files to project1](image)

**SETTING PROJECT OPTIONS**

Now you will set the project options. For application projects, options can be set on all levels of nodes. First you will set the general options to suit the processor configuration in this tutorial. Because these options must be the same for the whole build configuration, they must be set on the project node.

1 Select the project folder icon project1 - Debug in the Workspace window and choose Project>Options.
Setting up a new project

The **Target** options page in the **General Options** category is displayed.

![Options for mode "project1"](image)

**Figure 5: Setting general options**

Verify that these settings are used:

<table>
<thead>
<tr>
<th>Page</th>
<th>Setting</th>
</tr>
</thead>
<tbody>
<tr>
<td>Target</td>
<td>Core: Cortex-M3</td>
</tr>
<tr>
<td>Output</td>
<td>Output file: Executable</td>
</tr>
<tr>
<td>Library Configuration</td>
<td>Library: Normal</td>
</tr>
<tr>
<td>Library Configuration</td>
<td>Library low-level interface implementation: Semihosted</td>
</tr>
</tbody>
</table>

*Table 6: General settings for project1*
2 Select C/C++ Compiler in the Category list to display the compiler option pages.

![Figure 6: Setting compiler options](image)

3 Verify that these settings are used:

<table>
<thead>
<tr>
<th>Page</th>
<th>Setting</th>
</tr>
</thead>
<tbody>
<tr>
<td>Optimizations</td>
<td>Level: None (Best debug support)</td>
</tr>
<tr>
<td>Output</td>
<td>Generate debug information</td>
</tr>
<tr>
<td>List</td>
<td>Output list file</td>
</tr>
<tr>
<td></td>
<td>Assembler mnemonics</td>
</tr>
</tbody>
</table>

Table 7: Compiler options for project1

4 Click OK to set the options you have specified.

The project is now ready to be built.

### Compiling and linking the application

You can now compile and link the application. You will also view the compiler list file and the linker map file.
COMPILING THE SOURCE FILES

1 To compile the file Utilities.c, select it in the Workspace window.
2 Choose Project>Compile.
Alternatively, click the Compile button on the toolbar or choose the Compile command from the context menu that appears when you right-click on the selected file in the Workspace window.

The progress is displayed in the Build messages window.

3 Compile the file Tutor.c in the same manner.

The IDE has now created new directories in your project directory. Because you are using the build configuration Debug, a Debug directory has been created containing the subdirectories List, Obj, and Exe:

- The List directory is the destination directory for the list files. The list files have the extension lst.
- The Obj directory is the destination directory for the object files from the compiler and the assembler. These files have the extension o and are used as input to the IAR ILINK Linker.
- The Exe directory is the destination directory for the executable file. It has the extension out and is used as input to the IAR C-SPY® Debugger. Note that this directory is empty until you have linked the object files.

Click on the plus signs in the Workspace window to expand the view. As you can see, the IDE has also created an output folder icon in the Workspace window containing any
generated output files. All included header files are displayed as well, showing the dependencies between the files.

![Workspace window after compilation](image)

**VIEWING THE LIST FILE**

Now examine the compiler list file and notice how it is automatically updated when you, as in this case, will investigate how different optimization levels affect the generated code size.

Open the list file `Utilities.lst` by double-clicking it in the Workspace window. Examine the list file, which contains the following information:

- The **header** shows the product version, information about when the file was created, and the command line version of the compiler options that were used.
- The **body** of the list file shows the assembler code and binary code generated for each statement. It also shows how the variables are assigned to sections.
- The **end** of the list file shows the amount of stack, code, and data memory required, and contains information about error and warning messages that might have been generated.

Notice the amount of generated code at the end of the file and keep the file open.
Compiling and linking the application

2 Choose Tools>Options to open the IDE Options dialog box and click the Editor tab. Select the option Scan for changed files. This option turns on the automatic update of any file open in an editor window, such as a list file.

![IDE Options dialog box](image)

Click the OK button.

3 Select the file Utilities.c in the Workspace window, right-click and choose Options from the context menu to open the C/C++ Compiler options dialog box. Select the Override inherited settings option. Click the Optimizations tab and choose High level of optimization. Click OK.

Notice that the options override on the file node is indicated with a red dot in the Workspace window.

4 Compile the file Utilities.c. Now you will notice two things. First, note the automatic updating of the open list file due to the selected option Scan for changed files. Second, look at the end of the list file and notice the effect on the code size due to the increased optimization.

5 For this tutorial, the optimization level None should be used, so before linking the application, restore the default optimization level. Open the C/C++ Compiler options dialog box by right-clicking on the selected file in the Workspace window. Deselect the Override inherited settings option and click OK. Recompile the file Utilities.c.
LINKING THE APPLICATION

Now you should set up the options for the IAR ILINK Linker.

Select the project folder icon **project1 - Debug** in the Workspace window and choose **Project>Options**, or right-click and choose **Options** from the context menu. Then select **Linker** in the **Category** list to display the linker option pages.

![Linker options dialog box for project1](image)

For this tutorial, default factory settings are used. However, pay attention to the choice of linker configuration file.

**Output format**

The linker produces an output file in the ELF format, including DWARF for debug information. If you need to have a file in the Motorola or Intel-standard formats instead, for example to load the file to a PROM memory, you must convert the file. You can use the converter provided with IAR Embedded Workbench, see **Converter options**, page 475.
Compiling and linking the application

**Linker configuration file**

Program code and data are placed in memory according to the configuration specified in the linker configuration file (filename extension \texttt{icf}). It is important to be familiar with its syntax for how sections are placed in memory. Read more about this in the *IAR C/C++ Development Guide for ARM®*.

The definitions in the supplied linker configuration files are not related to any particular hardware. The linker configuration file template supplied with the product can be used as is in the simulator, but when using it for your target system you must adapt it to your actual hardware memory layout. You can find linker configuration files for some evaluation boards in \texttt{src\examples}.

In this tutorial you will use the default linker configuration file, which you can see on the \texttt{Config} page.

If you want to examine the linker configuration file, use a suitable text editor, such as the IAR Embedded Workbench editor, or print a copy of the file, and verify that the definitions match your requirements. Alternatively, click the \texttt{Edit} button to open the linker configuration file editor.

**Linker map file**

By default, no linker map file is generated. To generate a linker map file, click the \texttt{List} tab and select the option \texttt{Generate linker map file}.

2 Click \texttt{OK} to save the linker options. Now you should link the object file, to generate code that can be debugged.

3 Choose \texttt{Project>Make}. The progress will as usual be displayed in the Build messages window. The result of the linking is a code file \texttt{project1.out} with debug information located in the \texttt{Debug\Exe} directory and a map file \texttt{project1.map} located in the \texttt{Debug\List} directory.

**VIEWING THE MAP FILE**

Examine the file \texttt{project1.map} to see how the sections were placed in memory.

Read more about the map file in the *IAR C/C++ Development Guide for ARM®*. The \texttt{project1.out} application is now ready to be run in C-SPY.
Debugging using the IAR C-SPY® Debugger

This chapter continues the development cycle started in the previous chapter and explores the basic features of C-SPY.

Note that, depending on what IAR Systems product package you have installed, C-SPY might or might not be included. The tutorials assume that you are using the C-SPY Simulator.

For a short overview of all tutorials and their related files, see Tutorials overview, page 29.

Debugging the application

The project1.out application, created in the previous chapter, is now ready to be run in C-SPY where you can watch variables, set breakpoints, view code in disassembly mode, monitor registers and memory, and print the program output in the Terminal I/O window, etc.

STARTING THE DEBUGGER

Before starting C-SPY, you must set a few options.

1. Choose Project>Options and then the Debugger category. On the Setup page, make sure that you have chosen Simulator from the Driver drop-down list and that Run to main is selected. Click OK.

2. Choose Project>Download and Debug. Alternatively, click the Download and Debug button in the toolbar. C-SPY starts with the project1.out application loaded. In addition to the windows already opened in the IDE, a set of C-SPY-specific windows are now available.

ORGANIZING THE WINDOWS

In the IDE, you can dock windows at specific places, and organize them in tab groups. You can also make a window floating, which means it is always on top of other windows. If you change the size or position of a floating window, other currently open windows are not affected.
Debugging the application

The status bar, located at the bottom of the Embedded Workbench main window, contains useful help about how to arrange windows. For further details, see Organizing the windows on the screen, page 83.

Make sure the following windows and window contents are open and visible on the screen: the Workspace window with the active build configuration tutorials → project1, the editor window with the source files Tutor.c and Utilities.c, and the Debug Log window.

Figure 11: The C-SPY Debugger main window

INSPECTING SOURCE STATEMENTS

1. To inspect the source statements, double-click the file Tutor.c in the Workspace window.

2. With the file Tutor.c displayed in the editor window, first step over with the Debug→Step Over command.

IAR Embedded Workbench® IDE
User Guide for ARM
Alternatively, click the Step Over button on the toolbar.

Step until the call to the InitFib function.

```c
/* Increment the 'callCount' variable. */
30 void SetFibProc( void )
31 { 
32 unsigned int fib;
33 Reentrant();
34 fib = SetFib( callCount );
35 PutFib( fib );
36
37 /* Main program. */
38 /* Prints the Fibonacci sequence. */
39 void main( void )
40 { 
41 callCount = 0;
42
43 InitFib();
44
45 while ( callCount < MAX_FIB )
46 { 
47 DoForegroundProc();
48
49 Figure 12: Stepping in C-SPY

3 Choose Debug→Step Into to step into the function InitFib.

Alternatively, click the Step Into button on the toolbar.

At source level, the Step Over and Step Into commands allow you to execute your application a statement at a time. Step Into continues stepping inside function or subroutine calls, whereas Step Over executes each function call in a single step. For further details, see Step, page 130.

When Step Into is executed you will notice that the active window changes to Utilities.c as the function InitFib is located in this file.
4 Use the **Step Into** command until you reach the `for` loop.

```c
31:   unsigned int Fib[MAX_FIB];
32:
33: /* Initializes MAX_FIB Fibonacci numbers. */
34: void InitFib (void)
35: {
36:     i = 0;
37:     Fib[0] = Fib[1] = 1;
38:     for (i = 2; i < MAX_FIB; i++)
39:     {
40:         Fib[i] = GetFib(i) + GetFib(i-1);
41:     }
42: }
```

**Figure 13: Using Step Into in C-SPY**

5 Use **Step Over** until you are back in the header of the `for` loop. Notice that the step points are on a function call level, not on a statement level.

You can also step on a statement level. Choose **Debug>Next statement** to execute one statement at a time. Alternatively, click the **Next statement** button on the toolbar.

Notice how this command differs from the **Step Over** and the **Step Into** commands.

### INSPECTING VARIABLES

C-SPY allows you to watch variables or expressions in the source code, so that you can keep track of their values as you execute your application. You can look at a variable in several ways. For example, point at it in the source window with the mouse pointer, or open one of the Auto, Locals, Live Watch, Statics, or Watch windows. In this tutorial, we will look into some of these methods. For more information about inspecting variables, see the chapter *Working with variables and expressions*.

**Note:** When optimization level **None** is used, all non-static variables will live during their entire scope and thus, the variables are fully debuggable. When higher levels of optimizations are used, variables might not be fully debuggable.

### Using the Auto window

1 Choose **View>Auto** to open the Auto window.
The Auto window will show the current value of recently modified expressions.

<table>
<thead>
<tr>
<th>Expression</th>
<th>Value</th>
<th>Location</th>
<th>Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>i</td>
<td>5</td>
<td>P4</td>
<td>short</td>
</tr>
<tr>
<td>Fib[0]</td>
<td>0</td>
<td>0x40102214</td>
<td>unsigned int</td>
</tr>
<tr>
<td>InitFib</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>GetFib[i]</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Figure 14: Inspecting variables in the Auto window

2 Keep stepping to see how the values change.

**Setting a watchpoint**

Next you will use the Watch window to inspect variables.

3 Choose View>Watch to open the Watch window. Notice that it is, by default, grouped together with the currently open Auto window; the windows are located as a tab group.

4 Set a watchpoint on the variable i using this procedure: Click the dotted rectangle in the Watch window. In the entry field that appears, type i and press the Enter key.

You can also drag a variable from the editor window to the Watch window.

5 Select the Fib array in the InitFib function, then drag it to the Watch window.

The Watch window will show the current value of i and Fib. You can expand the Fib array to watch it in more detail.

<table>
<thead>
<tr>
<th>Expression</th>
<th>Value</th>
<th>Location</th>
<th>Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>i</td>
<td>5</td>
<td>P4</td>
<td>short</td>
</tr>
<tr>
<td>Fib</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>[0]</td>
<td>1</td>
<td>0x40102260</td>
<td>unsigned int</td>
</tr>
<tr>
<td>[1]</td>
<td>1</td>
<td>0x40102264</td>
<td>unsigned int</td>
</tr>
<tr>
<td>[2]</td>
<td>2</td>
<td>0x40102268</td>
<td>unsigned int</td>
</tr>
<tr>
<td>[3]</td>
<td>3</td>
<td>0x40102270</td>
<td>unsigned int</td>
</tr>
<tr>
<td>[4]</td>
<td>5</td>
<td>0x40102270</td>
<td>unsigned int</td>
</tr>
<tr>
<td>[5]</td>
<td>0</td>
<td>0x40102274</td>
<td>unsigned int</td>
</tr>
<tr>
<td>[6]</td>
<td>0</td>
<td>0x40102276</td>
<td>unsigned int</td>
</tr>
<tr>
<td>[7]</td>
<td>0</td>
<td>0x40102278</td>
<td>unsigned int</td>
</tr>
<tr>
<td>[8]</td>
<td>0</td>
<td>0x40102280</td>
<td>unsigned int</td>
</tr>
</tbody>
</table>

Figure 15: Watching variables in the Watch window

6 Execute some more steps to see how the values of i and Fib change.
To remove a variable from the Watch window, select it and press Delete.

**SETTING AND MONITORING BREAKPOINTS**

C-SPY contains a powerful breakpoint system with many features. For detailed information about the breakpoints, see *The breakpoint system*, page 141.

The most convenient way is usually to set breakpoints interactively, simply by positioning the insertion point in or near a statement and then choosing the Toggle Breakpoint command.

1. Set a breakpoint on the function call `GetFib(i)` using this procedure: First, click the `Utilities.c` tab in the editor window and click in the statement to position the insertion point. Then choose *Edit>Toggle Breakpoint*.

   Alternatively, click the Toggle Breakpoint button on the toolbar.

   A breakpoint will be set at this function call. The function call will be highlighted and there will be a red dot in the margin to show that there is a breakpoint there.

   ![Figure 16: Setting breakpoints](image)

To view all defined breakpoints, choose *View>Breakpoints* to open the Breakpoints window. You can find information about the breakpoint execution in the Debug Log window.

**Executing up to a breakpoint**

2. To execute your application until it reaches the breakpoint, choose *Debug>Go*.

   Alternatively, click the Go button on the toolbar.

   The application will execute up to the breakpoint you set. The Watch window will display the value of the `Fib` expression and the Debug Log window will contain information about the breakpoint.
Select the breakpoint, right-click and choose **Toggle Breakpoint (Code)** from the context menu, alternatively choose **Toggle Breakpoint** from the **Edit** menu to remove the breakpoint.

**DEBUGGING IN DISASSEMBLY MODE**

Debugging with C-SPY is usually quicker and more straightforward in C/C++ source mode. However, if you want to have full control over low-level routines, you can debug in disassembly mode where each step corresponds to one assembler instruction. C-SPY lets you switch freely between the two modes.

1. First reset your application by clicking the **Reset** button on the toolbar.
2. Choose **View>Disassembly** to open the Disassembly window, if it is not already open. You will see the assembler code corresponding to the current C statement.

To view code coverage information, right-click in the Disassembly window and choose **Code Coverage>Enable** and then **Code Coverage>Show** from the context menu.

Try the step commands also in the Disassembly window and note the code coverage information indicated by green diamonds.

**MONITORING MEMORY**

The Memory window lets you monitor selected areas of memory. In the following example, the memory corresponding to the variable **Fib** will be monitored.

1. Choose **View>Memory** to open the Memory window.
2 Make the Utilities.c window active and select `Fib`. Then drag it from the C source window to the Memory window. The memory contents in the Memory window corresponding to `Fib` will be selected.

3 To display the memory contents as 16-bit data units, choose the **x2 Units** command from the drop-down arrow menu on the Memory window toolbar. If not all of the memory units have been initialized by the `InitFib` function of the C application yet, continue to step over and you will notice how the memory contents are updated.

To change the memory contents, edit the values in the Memory window. Just place the insertion point at the memory content that you want to edit and type the desired value.

Close the Memory window.
VIEWING TERMINAL I/O

Sometimes you might have to debug constructions in your application that make use of `stdin` and `stdout` without the possibility of having hardware support. C-SPY lets you simulate `stdin` and `stdout` by using the Terminal I/O window.

**Note:** The Terminal I/O window is only available in C-SPY if you have linked your project using the output option **Semihosted** or the **IAR breakpoint** option. This means that some low-level routines are linked that direct `stdin` and `stdout` to the Terminal I/O window, see [Linking the application](Linking the application), page 43.

Choose **View>Terminal I/O** to display the output from the I/O operations.

![Figure 20: Output from the I/O operations](Figure 20: Output from the I/O operations)

The contents of the window depends on how far you have executed the application.

REACHING PROGRAM EXIT

To complete the execution of your application, choose **Debug>Go**.

Alternatively, click the **Go** button on the toolbar.
As no more breakpoints are encountered, C-SPY reaches the end of the application and a Program exit reached message is printed in the Debug Log window.

![Figure 21: Reaching program exit in C-SPY](image)

All output from the application has now been displayed in the Terminal I/O window.

If you want to start again with the existing application, choose Debug>Reset, or click the Reset button on the toolbar.

2 To exit from C-SPY, choose Debug>Stop Debugging. Alternatively, click the Stop Debugging button on the toolbar. The Embedded Workbench workspace is displayed. C-SPY also provides many other debugging facilities. Some of these—for example macros and interrupt simulation—are described in the following tutorial chapters.

For further details about how to use C-SPY, see Part 4. Debugging. For reference information about the features of C-SPY, see Part 7. Reference information and the online help system.
Mixing C and assembler modules

In some projects it might be necessary to write certain pieces of source code in assembler language. The chapter first demonstrates how the compiler can be helpful in examining the calling convention, which you must be familiar with when calling assembler modules from C/C++ modules or vice versa. Furthermore, this chapter demonstrates how you can easily combine source modules written in C with assembler modules, but the procedure is applicable to projects containing source modules written in C++, too.

This tutorial assumes that you are familiar with the basics of the IAR Embedded Workbench® IDE described in the previous tutorial chapters.

For a short overview of all tutorials and their related files, see Tutorials overview, page 29.

Examining the calling convention

When you write an assembler routine that is called from a C routine, you must be aware of the calling convention that the compiler uses. If you create skeleton code in C and let the compiler produce an assembler output file from it, you can study the produced assembler output file and find the details of the calling convention.

In this example you will make the compiler create an assembler output file from the file Utilities.c.

1 Create a new project in the same workspace tutorials as used in the previous tutorial project, and name the project project2.

2 Add the files Tutor.c and Utilities.c to the project.

To display an overview of the workspace, click the Overview tab available at the bottom of the Workspace window. To view only the newly created project, click the project2 tab. For now, the project2 view should be visible.

3 To set options on file level node, in the Workspace window, select the file Utilities.c.
Choose Project>Options. You will notice that only the C/C++ Compiler and Custom Build categories are available.

4 In the C/C++ Compiler category, select Override inherited settings and verify these settings:

<table>
<thead>
<tr>
<th>Page</th>
<th>Option</th>
</tr>
</thead>
<tbody>
<tr>
<td>Optimizations</td>
<td>Level: None (Best debug support)</td>
</tr>
<tr>
<td>List</td>
<td>Output assembler file</td>
</tr>
<tr>
<td></td>
<td>Include source</td>
</tr>
<tr>
<td></td>
<td>Include call frame information (must be deselected).</td>
</tr>
</tbody>
</table>

Table 8: Compiler options for project2

Note: In this example you must use a low optimization level when you compile the code, to show local and global variable accesses. If you use a higher level of optimization, the required references to local variables might be removed. However, the actual function declaration is not changed by the optimization level.

5 Click OK and return to the Workspace window.

6 Compile the file Utilities.c. You can find the output file Utilities.s in the subdirectory projects\debug\list.

Note: If you have a limited product installation, it might not be possible to generate an assembler file. In that case you can instead study the pre-generated file, located in the tutor directory.

7 To examine the calling convention and to see how the C or C++ code is represented in assembler language, open the file Utilities.s.

You can now study where and how parameters are passed, how to return to the program location from where a function was called, and how to return a resulting value. You can also see which registers an assembler-level routine must preserve.

Note: The generated assembler source file might contain compiler-internal information, for example CPI directives. These directives are available for debugging purpose and you should ignore these details.

To obtain the correct interface for your own application functions, you should create skeleton code for each function that you need.

For more information about the calling convention used in the compiler, see the IAR C/C++ Development Guide for ARM®.
Adding an assembler module to the project

This tutorial demonstrates how you can easily create a project containing both assembler modules and C modules. You will also compile the project and view the assembler output list file.

You will add an assembler module containing a `__write` function. This function is a primitive output function, which is used by `putchar` to handle all character output. The standard implementation of `__write` redirects character output to the C-SPY® Terminal I/O window. Your own version of `__write` also does this, with the extra feature that it outputs the sign ‘*’ before every character written.

**SETTING UP THE PROJECT**

1. Modify `project2` by adding the `Write.s` file.
   
   **Note:** To view assembler files in the Add files dialog box, choose Project>Add Files and choose Assembler Files from the Files of type drop-down list.

2. Select the project level node in the Workspace window, choose Project>Options. Use the default settings in the General Options, C/C++ Compiler, and Linker categories. Select the Assembler category, click the List tab, and select the option Output list file.

![Options for node "project1"](image)

*Figure 22: Assembler settings for creating a list file*
Adding an assembler module to the project

Click OK.

3 Select the file Write.s in the Workspace window and choose Project>Compile to assemble it.

Assuming that the source file was assembled successfully, the file Write.o is created, containing the linkable object code.

Viewing the assembler list file

4 Open the list file by double-clicking the file Write.lst available in the Output folder icon in the Workspace window.

The end of the file contains a summary of errors and warnings that were generated.

For further details of the list file format, see the ARM® IAR Assembler Reference Guide.

5 Choose Project>Make to relink project2.

6 Start C-SPY to run the project2.out application and see that it behaves like the application in the previous tutorial, but with an * before every character written.

Exit the debugger when you are done.
Using C++

In this chapter, C++ is used to create a C++ class. The class is then used for creating two independent objects, and the application is built and debugged. We also show an example of how to set a conditional breakpoint.

This tutorial assumes that you are familiar with the basics of the IAR Embedded Workbench® IDE described in the previous tutorial chapters.

For a short overview of all tutorials and their related files, see Tutorials overview, page 29.

Creating a C++ application

This tutorial demonstrates how to use the C++ features. The tutorial consists of two files:

- Fibonacci.h and Fibonacci.cpp define a class Fibonacci that can be used to extract a series of Fibonacci numbers
- CppTutor.cpp creates two objects, fib1 and fib2, from the class Fibonacci and extracts two sequences of Fibonacci numbers using the Fibonacci class.

To demonstrate that the two objects are independent of each other, the numbers are extracted at different speeds. A number is extracted from fib1 each turn in the loop while a number is extracted from fib2 only every second turn.

The object fib1 is created using the default constructor while the definition of fib2 uses the constructor that takes an integer as its argument.

**COMPILING AND LINKING THE C++ APPLICATION**

1. In the workspace tutorials used in the previous chapters, create a new project, project3.
2. Add the files Fibonacci.cpp and CppTutor.cpp to project3.
3. Choose Project>Options and make sure these options are selected:

<table>
<thead>
<tr>
<th>Category</th>
<th>Page</th>
<th>Option</th>
</tr>
</thead>
<tbody>
<tr>
<td>General Options</td>
<td>Target</td>
<td>Cortex-M3</td>
</tr>
<tr>
<td>C/C++ Compiler</td>
<td>Language</td>
<td>Embedded C++</td>
</tr>
</tbody>
</table>

Table 9: Project options for C++ tutorial
Creating a C++ application

All you have to do to switch to the C++ programming language, is to choose the language option **Embedded C++**.

4 Choose **Project>Make** to compile and link your application.

Alternatively, click the **Make** button on the toolbar. The **Make** command compiles and links those files that have been modified.

5 Choose **Project>Debug** to start C-SPY.

**SETTING A BREAKPOINT AND EXECUTING TO IT**

1 Open the CppTutor.cpp window if it is not already open.

2 To see how the object is constructed, set a breakpoint on the C++ object `fib1` on this line:

\[
\text{Fibonacci fib1;}
\]

3 Choose **Debug>Go**, or click the **Go** button on the toolbar.

The cursor should now be placed at the breakpoint.

4 To step into the constructor, choose **Debug>Step Into** or click the **Step Into** button in the toolbar. Then click **Step Out** again.

5 **Step Over** until the line:

\[
\text{cout << fib1.next();}
\]

**Step Into** until you are in the function `next` in the file `Fibonacci.cpp`.

![Figure 24: Setting a breakpoint in CppTutor.cpp](image)
6 Use the Go to function button in the lower left corner of the editor window and double-click the function name \texttt{nth} to find and go to the function. Set a breakpoint on the function call \texttt{nth(n-1)} at the line 

\begin{verbatim}
    value = nth(n-1) + nth(n-2);
\end{verbatim}

7 It can be interesting to backtrace the function calls a few levels down and to examine the value of the parameter for each function call. If you add a condition to the breakpoint, the break will not be triggered until the condition is true, and you will be able to see each function call in the Call Stack window.

To open the Breakpoints window, choose View\textgreater Breakpoints. Select the breakpoint in the Breakpoints window, right-click to open the context menu, and choose Edit to open the Edit Breakpoints dialog box.

\begin{figure}[h]
\centering
\includegraphics[width=0.5\textwidth]{fig25.png}
\caption{Setting a breakpoint with skip count}
\end{figure}

Set the value in the Skip count text box to 4 and click OK.

**Looking at the function calls**

8 Choose Debug\textgreater Go to execute the application until the breakpoint condition is fulfilled.
When C-SPY stops at the breakpoint, choose View>Call Stack to open the Call Stack window.

Five instances of the function \texttt{nth} are displayed on the call stack. Because the Call Stack window displays the values of the function parameters, you can see the different values of \texttt{n} in the different function instances.

You can also open the Register window to see how it is updated as you trace the function calls by double-clicking on the function instances.

**PRINTING THE FIBONACCI NUMBERS**

1. Open the Terminal I/O window from the View menu.
2. Remove the breakpoints and run the application to the end and verify the Fibonacci sequences being printed.
Simulating an interrupt

In this tutorial an interrupt handler for a serial port is added to the project. The Fibonacci numbers are read from an on-chip communication peripheral device (UART).

This tutorial will show how to write an interrupt function for ARM7TDMI and for Cortex-M3. The tutorial will also show how to simulate an interrupt, using the features that support interrupts, breakpoints, and macros. Notice that this example does not describe an exact simulation; the purpose is to illustrate a situation where C-SPY® macros, breakpoints, and the interrupt system can be useful to simulate hardware.

This tutorial assumes that you are familiar with the basics of the IAR Embedded Workbench® IDE described in the previous tutorial chapters.

Note that interrupt simulation is possible only when you are using the IAR C-SPY Simulator.

For a short overview of all tutorials and their related files, see Tutorials overview, page 29.

Adding an interrupt handler

This section will demonstrate how to write an interrupt in an easy way. It starts with a brief description of the application used in this project, followed by a description of how to set up the project.

THE APPLICATION—A BRIEF DESCRIPTION

The interrupt handler will read values from the serial communication port receive register (UART), UARTRBRTHR/UART0DR. It will then print the value. The main program enables interrupts and starts printing periods (.) in the foreground process while waiting for interrupts.

Note: In this tutorial, UARTRBRTHR/UART0DR refers to the receive register on the ARM7TDI-based device m1674001 and the Cortex-M3-based device lm3s101, respectively. To follow this tutorial and simulate the interrupt in the C-SPY simulator, you should use the register that is suitable for the core and device you are using.
WRITING AN INTERRUPT HANDLER FOR ARM7TDMI

The following lines define the interrupt handler used in this tutorial (the complete source code can be found in the file Interrupt.c in project4 supplied in the arm\tutor directory):

/* define the IRQ handler */
__irq __arm void IRQ_Handler( void )

The __irq keyword is used for directing the compiler to use the calling convention needed for an interrupt function. The __arm keyword is used for making sure that the IRQ handler is compiled in ARM mode. In this example only UART receive interrupts are used, so there is no need to check the interrupt source. In the general case however, when several interrupt sources are used, an interrupt service routine must check the source of an interrupt before action is taken.

For detailed information about the extended keywords used in this tutorial, see the IAR C/C++ Development Guide for ARM®.

WRITING AN INTERRUPT HANDLER FOR CORTEX-M3

On Cortex-M3, an interrupt service routine enters and returns in the same way as a normal function, which means no special keywords are required.

In the InterruptCM3.c file in Project4CM3, the interrupt function UART0_Handler is provided. Note that when you add an interrupt function for Cortex-M devices, you must also add the name of that function in the interrupt vector table. You do this in the system startup code cstartup.s. For this tutorial, a reference to the UART0_Handler function is already provided in _vector_table, which you can find in CstartupCM3.s.

For more information about how to write device-specific interrupt functions for Cortex-M, see the IAR C/C++ Development Guide for ARM®.

SETTING UP THE PROJECT

1. Add a new project—project4 for ARM7TDMI and project4CM3 for Cortex-M3—to the workspace tutorials used in previous tutorials.

2. Add these files to the project:

   ARM7TDMI: Utilities.c and Interrupt.c
   Cortex-M3: Utilities.c and InterruptCM3.c

3. In the Workspace window, select the project level node and choose Project>Options. Select the General Options category, and click the Target tab. Choose ARM7TDMI or Cortex-M3 from the Core drop-down menu.
4 For ARM7TDMI, select the C/C++ Compiler category, and click the Code tab. Select the option Generate interwork code.

5 In addition, make sure the factory settings are used in the C/C++ Compiler and Linker categories.

Next you will set up the simulation environment.

Setting up the simulation environment

The C-SPY interrupt system is based on the cycle counter. You can specify the amount of cycles to pass before C-SPY generates an interrupt.

To simulate the input to UART, values are read from the file InputData.txt, which contains the Fibonacci series. You will set an immediate read breakpoint on the UART receive register, UARTRBTHR/UART0DR, and connect a user-defined macro function to it (in this example the Access macro function). The macro reads the Fibonacci values from the text file.

Whenever an interrupt is generated, the interrupt routine reads UARTRBTHR/UART0DR and the breakpoint is triggered, the Access macro function is executed and the Fibonacci values are fed into the UART receive register.

The immediate read breakpoint will trigger the break before the processor reads the UARTRBTHR/UART0DR register, allowing the macro to store a new value in the register that is immediately read by the instruction.

This section will demonstrate the steps involved in setting up the simulator for simulating a serial port interrupt. The steps involved are:

● Defining a C-SPY setup file which will open the file InputData.txt and define the Access macro function
● Specifying debugger options
● Building the project
● Starting the simulator
● Specifying the interrupt request
● Setting the breakpoint and associating the Access macro function to it.

Note: For a simple example of a system timer interrupt simulation, see Simulating a simple interrupt, page 222.

DEFINING A C-SPY SETUP MACRO FILE

In C-SPY, you can define setup macros that will be registered during the C-SPY startup sequence. In this tutorial you will use the C-SPY macro file SetupSimple.mac for
Setting up the simulation environment

ARM7TDMI or SetupSimpleCM3.mac for Cortex-M3, available in the arm\tutor directory. It is structured as follows:

First the setup macro function execUserSetup is defined, which is automatically executed during C-SPY setup. Thus, it can be used to set up the simulation environment automatically. A message is printed in the Log window to confirm that this macro has been executed:

```
execUserSetup()
{
   __message "execUserSetup() called\n";
}
```

Then the file InputData.txt, which contains the Fibonacci series to be fed into UART, is opened:

```
_fileHandle = __openFile( "$PROJ_DIR$\InputData.txt", "r" );
```

After that, the macro function Access is defined. It will read the Fibonacci values from the file InputData.txt, and assign them to the receive register address:

```
Access()
{
   __message "Access() called\n";
   __var _fibValue;
   if( 0 == __readFile( _fileHandle, &_fibValue ) )
   {
      UART0DR = _fibValue;
   }
}
```

**Note:** For ARM7TDMI, UART0DR must be changed to UARTRBRTHR.

You must connect the Access macro to an immediate read breakpoint. However, this will be done at a later stage in this tutorial.

Finally, the file contains two macro functions for managing correct file handling at reset and exit.

For detailed information about macros, see the chapters Using the C-SPY® macro system and C-SPY® macros reference.

Next you will specify the macro file and set the other debugger options needed.

**SETTING C-SPY OPTIONS**

1. To set debugger options, choose Project>Options. In the Debugger category, click the Setup tab.
2 Use the **Use macro file** browse button to specify the macro file to be used: 
SetupSimple.mac or SetupSimpleCM3.mac

Alternatively, use an argument variable to specify the path:
$TOOLKIT_DIR\tutor\SetupSimple.mac

See **Argument variables summary**, page 366, for details.

3 Next, you will specify the device description file. This file makes it possible to view the value of UARTBRTHR/UART0DR in the Register window and provides the interrupt definitions that are needed by the interrupt system.

For ARM7TDMI set the **Device description file** option to ioml674001.ddf.

For Cortex-M3 set the **Device description file** option to iolm3s101ddf ($TOOLKIT_DIR\CONFIG\debugger\TexasInstruments\iolm3s101ddf).

4 Select **Run to main** and click **OK**. This will ensure that the debug session will start by running to the **main function**.

The project is now ready to be built.
BUILDING THE PROJECT

1. Compile and link the project by choosing **Project**->**Make**.

   Alternatively, click the **Make** button on the toolbar. The **Make** command compiles and links those files that have been modified.

STARTING THE SIMULATOR

1. Start C-SPY to run the `project4` or `project4CM3` project.

   The Interrupt.c or InterruptCM3.c window is displayed (among other windows). Click in it to make it the active window.

2. Examine the Log window. Note that the macro file has been loaded and that the `execUserSetup` function has been called.

SPECIFYING A SIMULATED INTERRUPT

Now you will specify your interrupt to make it simulate an interrupt every 2000 cycles.

1. Choose **Simulator**->**Interrupt Setup** to display the **Interrupt Setup** dialog box. Click **New** to display the **Edit Interrupt** dialog box:

   ![Figure 29: Inspecting the interrupt settings for ARM7TDMI](image)

   Make these settings for your interrupt:

<table>
<thead>
<tr>
<th>Setting</th>
<th>Value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Interrupt</td>
<td>IRQ</td>
<td>Specifies which interrupt to use for ARM7TDMI.</td>
</tr>
<tr>
<td></td>
<td>UART0</td>
<td>Specifies which interrupt to use for Cortex-M3 (the LM3S101 device).</td>
</tr>
</tbody>
</table>

   **Table 10: Settings in the Edit Interrupt dialog box**
Simulating an interrupt

During execution, C-SPY will wait until the cycle counter has passed the activation time. When the current assembler instruction is executed, C-SPY will generate an interrupt which is repeated approximately every 2000 cycles.

When you have specified the settings, click **OK** to close the **Edit Interrupt** dialog box, and then click **OK** to close the **Interrupt Setup** dialog box.

For information about how you can use the system macro `__orderInterrupt` in a C-SPY setup file to automate the procedure of defining the interrupt, see Using macros for interrupts and breakpoints, page 72.

### SETTING AN IMMEDIATE BREAKPOINT

By defining a macro and connecting it to an immediate breakpoint, you can make the macro simulate the behavior of a hardware device, for instance an I/O port, as in this tutorial. The immediate breakpoint will not halt the execution, only temporarily suspend it to check the conditions and execute any connected macro.

In this example, the input to the UART is simulated by setting an immediate read breakpoint on the UARTRXRTHR/UART0DR address and connecting the defined Access macro to it. The macro will simulate the input to the UART. These are the steps involved:

1. Choose **View>Breakpoints** to open the Breakpoints window, right-click to open the context menu, choose **New Breakpoint>Immediate** to open the **Immediate** tab.

<table>
<thead>
<tr>
<th>Setting</th>
<th>Value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Description</td>
<td>As is</td>
<td>The interrupt definition that the simulator uses to be able to simulate the interrupt correctly.</td>
</tr>
<tr>
<td>First activation</td>
<td>4000</td>
<td>Specifies the first activation moment for the interrupt. The interrupt is activated when the cycle counter has passed this value.</td>
</tr>
<tr>
<td>Repeat interval</td>
<td>2000</td>
<td>Specifies the repeat interval for the interrupt, measured in clock cycles.</td>
</tr>
<tr>
<td>Hold time</td>
<td>Infinite</td>
<td>Hold time, not used here.</td>
</tr>
<tr>
<td>Probability %</td>
<td>100</td>
<td>Specifies probability. 100% specifies that the interrupt will occur at the given frequency. Another percentage might be used for simulating a more random interrupt behavior.</td>
</tr>
<tr>
<td>Variance %</td>
<td>0</td>
<td>Time variance, not used here.</td>
</tr>
</tbody>
</table>

Table 10: Settings in the Edit Interrupt dialog box (Continued)
Simulating the interrupt

In this section you will execute your application and simulate the serial port interrupt.

**EXECUTING THE APPLICATION**

1. In the Interrupt.c or InterruptCM3.c source window, step through the application and stop when it reaches the `while` loop, where the application waits for input.

2. In the Interrupt.c or InterruptCM3.c source window, locate the function `IRQ_Handler` or `UART0_Handler`.

3. Place the insertion point on the `++callCount;` statement in this function and set a breakpoint by choosing `Edit>Toggle Breakpoint`, or click the `Toggle Breakpoint` button on the toolbar. Alternatively, use the context menu.

   If you want to inspect the details of the breakpoint, choose `View>Breakpoints`.

4. Open the Terminal I/O window and run your application by choosing `Debug>Go` or clicking the `Go` button on the toolbar.

   The application should stop in the interrupt function.

5. In the Register window you can monitor and modify the contents of the processor registers.

---

**Simulating the interrupt**

In this section you will execute your application and simulate the serial port interrupt.

**EXECUTING THE APPLICATION**

1. In the Interrupt.c or InterruptCM3.c source window, step through the application and stop when it reaches the `while` loop, where the application waits for input.

2. In the Interrupt.c or InterruptCM3.c source window, locate the function `IRQ_Handler` or `UART0_Handler`.

3. Place the insertion point on the `++callCount;` statement in this function and set a breakpoint by choosing `Edit>Toggle Breakpoint`, or click the `Toggle Breakpoint` button on the toolbar. Alternatively, use the context menu.

   If you want to inspect the details of the breakpoint, choose `View>Breakpoints`.

4. Open the Terminal I/O window and run your application by choosing `Debug>Go` or clicking the `Go` button on the toolbar.

   The application should stop in the interrupt function.

5. In the Register window you can monitor and modify the contents of the processor registers.

---

**Table 11: Breakpoints dialog box**

<table>
<thead>
<tr>
<th>Setting</th>
<th>Value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Break at</td>
<td>UARTRBRTHR/UART0DR</td>
<td>Receive buffer address.</td>
</tr>
<tr>
<td>Access Type</td>
<td>Read</td>
<td>The breakpoint type (Read or Write)</td>
</tr>
<tr>
<td>Action</td>
<td><code>Access()</code></td>
<td>The macro connected to the breakpoint.</td>
</tr>
</tbody>
</table>

During execution, when C-SPY detects a read access from the `UARTRBRTHR/UART0DR` address, C-SPY will temporarily suspend the simulation and execute the `Access` macro. The macro will read a value from the file `InputData.txt` and write it to `UARTRBRTHR/UART0DR`. C-SPY will then resume the simulation by reading the receive buffer value in `UARTRBRTHR/UART0DR`.

3. Click OK to close the breakpoints dialog box.

   For information about how you can use the system macro `__setSimBreak` in a C-SPY setup file to automate the breakpoint setting, see Using macros for interrupts and breakpoints, page 72.
To inspect the contents of the serial communication port receive register \texttt{UARTRBRTHR/UART0DR}, choose \texttt{View/Register} to open the Register window. Choose UART from the drop-down list.

![Figure 30: Register window for ARM7TDMI](image)

6 Run your application by choosing \texttt{Debug>Go} or clicking the \texttt{Go} button on the toolbar. The application should stop in the interrupt function. Note how the contents of \texttt{UARTRBRTHR/UART0DR} has been updated.

7 Click \texttt{Go} again to see the next number being printed in the Terminal I/O window. Because the main program has an upper limit on the Fibonacci value counter, the tutorial application will soon reach the \texttt{exit} label and stop.

The Terminal I/O window will display the Fibonacci series.

![Figure 31: Printing the Fibonacci values in the Terminal I/O window](image)
Using macros for interrupts and breakpoints

To automate the setting of breakpoints and the procedure of defining interrupts, the system macros __setSimBreak and __orderInterrupt, respectively, can be executed by the setup macro execUserSetup.

The file SetupAdvanced.mac for ARM7TDMI and SetupAdvancedCM3.mac for Cortex-M3 are extended with system macro calls for setting the breakpoint and specifying the interrupt.

For ARM7TDMI:

```
simulationSetup()
{
  _interruptID = __orderInterrupt( "IRQ", 4000,
                                   2000, 0, 1, 0, 100 );

  if( -1 == _interruptID )
  {
    __message "ERROR: failed to order interrupt";
  }

  _breakID = __setSimBreak( "UARTRBTHR", "R", "Access()" );
}
```

For Cortex-M3 (the lm3s101 device):

```
simulationSetup()
{
  _interruptID = __orderInterrupt( "UART0", 4000,
                                   2000, 0, 1, 0, 100 );

  if( -1 == _interruptID )
  {
    __message "ERROR: failed to order interrupt";
  }

  _breakID = __setSimBreak( "UART0DR", "R", "Access()" );
}
```

If you replace the file SetupSimple.mac or SetupSimpleCM3.mac, used in the previous tutorial, with the file SetupAdvanced.mac or SetupAdvancedCM3.mac, C-SPY will automatically set the breakpoint and define the interrupt at startup. Thus, you do not need to start the simulation by manually filling in the values in the Interrupts and Breakpoints dialog boxes.
Note: Before you load the file SetupAdvanced.mac or SetupAdvancedCM3.mac you should remove the previously defined breakpoint and interrupt.
Using macros for interrupts and breakpoints
Creating and using libraries

This tutorial demonstrates how to create a library project and how you can combine it with an application project.

This tutorial assumes that you are familiar with the basics of the IAR Embedded Workbench® IDE described in the previous tutorial chapters.

For a short overview of all tutorials and their related files, see Tutorials overview, page 29.

Using libraries

If you are working on a large project, you will soon accumulate a collection of useful modules that contain one or more routines to be used by several of your applications. To avoid having to assemble or compile a module each time it is needed, you can store such modules as object files, that is, assembled or compiled but not linked.

You can collect many modules in a single object file which then is referred to as a library. It is recommended that you use library files to create collections of related routines, such as a device driver.

Use the IAR Archive Tool iarchive to build libraries.

The Main.s program

The Main.s program uses a routine called max to set the contents of the register R1 to the maximum value of the word registers R1 and R2. The EXTERN directive declares max as an external symbol, to be resolved at link time.

A copy of the program is provided in the arm\tutor directory.

The library routines

The two library routines will form a separately assembled library. It consists of the max routine called by main, and a corresponding min routine, both of which operate on the contents of the registers R1 and R2 and return the result in R1. The files containing these library routines are called Max.s and Min.s and copies are provided in the arm\tutor directory.
The **PUBLIC** directive makes the `max` and `min` symbols public to other modules.

For detailed information about the **PUBLIC** directive, see the *ARM® IAR Assembler Reference Guide*.

**CREATING A NEW PROJECT**

1. In the workspace **tutorials** used in previous chapters, add a new project called `project5`.
2. Add the file `Main.s` to the new project.
3. To set options, choose **Project>Options**. Select the **General Options** category and click the **Library Configuration** tab. Choose **None** from the **Library** drop-down list, which means that a standard C/C++ library will not be linked.

On the **General Options>Target** page, choose Cortex-M3 from the **Core** drop-down list.

The default options are used for the other option categories.

4. To assemble the file `Main.s`, choose **Project>Compile**. You can also click the **Compile** button on the toolbar.

**CREATING A LIBRARY PROJECT**

Now you are ready to create a library project.

1. In the same workspace **tutorials**, add a new project called `tutor_library`.
2. Add the files `Max.s` and `Min.s` to the project.
3. To set options, choose **Project>Options**. In the **General Options** category, verify these settings:

<table>
<thead>
<tr>
<th>Page</th>
<th>Option</th>
</tr>
</thead>
<tbody>
<tr>
<td>Output</td>
<td>Output file: Library</td>
</tr>
<tr>
<td>Library Configuration</td>
<td>Library: None</td>
</tr>
</tbody>
</table>

*Table 12: General options for a library project*

Note that **Library Builder** appears in the list of categories, which means that the IAR Archive Tool `iarchive` is added to the build tool chain. You do not have to set any `iarchive`-specific options for this tutorial.

Click **OK**.

4. Choose **Project>Make**.
The library output file `tutor_library.a` has now been created in the `projects\Debug\Exe` directory.

**USING THE LIBRARY IN YOUR APPLICATION PROJECT**

Now add your library containing the maxmin routine to `project5`.

1. In the Workspace window, click the `project5` tab. Choose `Project> Add Files` and add the file `tutor_library.a` located in the `projects\Debug\Exe` directory. Click `Open`.

2. Click `Make` to build your project.

3. You have now combined a library with an executable project, and the application is ready to be executed. For information about how to manipulate the library, see the IAR Archive Tool documentation available in the *IAR C/C++ Development Guide for ARM®*. 
Using libraries
Part 3. Project management and building

This part of the IAR Embedded Workbench® IDE User Guide contains the following chapters:

- The development environment
- Managing projects
- Building
- Editing.
The development environment

This chapter introduces you to the IAR Embedded Workbench® development environment (IDE). The chapter also demonstrates how you can customize the environment to suit your requirements.

The IAR Embedded Workbench IDE

THE TOOL CHAIN

The IDE is the framework where all necessary tools—the tool chain—are seamlessly integrated: a C/C++ compiler, an assembler, the IAR ILINK Linker and its accompanying tools, an editor, a project manager with Make utility, and the IAR C-SPY® Debugger, which is a high-level language debugger. The tools used specifically for building your source code are referred to as the build tools.

The tool chain that comes with your product installation is adapted for a certain microcontroller. However, the IDE can simultaneously manage multiple tool chains for various microcontrollers.

You can also add IAR visualSTATE to the tool chain, which means that you can add state machine diagrams directly to your project in the IDE.

You can use the Custom Build mechanism to incorporate also other tools to the tool chain, see Extending the tool chain, page 101.

The compiler, assembler, and linker can also be run from a command line environment, if you want to use them as external tools in an already established project environment.
This illustration shows the IAR Embedded Workbench IDE window with various components.

![IAR Embedded Workbench IDE window]

The window might look different depending on what additional tools you are using.

**RUNNING THE IDE**

Click the **Start** button on the Windows taskbar and choose **All Programs**>**IAR Systems**>**IAR Embedded Workbench for ARM**>**IAR Embedded Workbench**.

The file `IarIdePm.exe` is located in the `common\bin` directory under your IAR Systems installation, in case you want to start the program from the command line or from within Windows Explorer.
Double-clicking the workspace filename

The workspace file has the filename extension .eww. If you double-click a workspace filename, the IDE starts. If you have several versions of IAR Embedded Workbench installed, the workspace file is opened by the most recently used version of your IAR Embedded Workbench that uses that file type.

EXITING

To exit the IDE, choose File>Exit. You will be asked whether you want to save any changes to editor windows, the projects, and the workspace before closing them.

Customizing the environment

The IDE is a highly customizable environment. This section demonstrates how you can work with and organize the windows on the screen, the possibilities for customizing the IDE, and how you can set up the environment to communicate with external tools.

ORGANIZING THE WINDOWS ON THE SCREEN

In the IDE, you can position the windows and arrange a layout according to your preferences. You can dock windows at specific places, and organize them in tab groups. You can also make a window floating, which means it is always on top of other windows. If you change the size or position of a floating window, other currently open windows are not affected.

Each time you open a previously saved workspace, the same windows are open, and they have the same sizes and positions.

For every project that is executed in the C-SPY environment, a separate layout is saved. In addition to the information saved for the workspace, information about all open debugger-specific windows is also saved.

Using docked versus floating windows

Each window that you open has a default location, which depends on other currently open windows. To give you full and convenient control of window placement, each window can either be docked or floating.

A docked window is locked to a specific area in the Embedded Workbench main window, which you can decide. To keep many windows open at the same time, you can organize the windows in tab groups. This means one area of the screen is used for several concurrently open windows. The system also makes it easy to rearrange the size of the windows. If you rearrange the size of one docked window, the sizes of any other docked windows are adjusted accordingly.
A floating window is always on top of other windows. Its location and size does not affect other currently open windows. You can move a floating window to any place on your screen, also outside of the IAR Embedded Workbench IDE main window.

**Note:** The editor window is always docked. When you open the editor window, its placement is decided automatically depending on other currently open windows. For more information about how to work with the editor window, see [*Using the IAR Embedded Workbench editor*](#), page 105.

**Organizing windows**

To place a window as a separate window, drag it next to another open window.

To place a window in the same tab group as another open window, drag the window you want to locate to the middle of the area and drop the window.

To make a window floating, double-click on the window’s title bar.

The status bar, located at the bottom of the IAR Embedded Workbench IDE main window, contains useful help about how to arrange windows.

**CUSTOMIZING THE IDE**

To customize the IDE, choose *Tools>*Options to get access to a wide variety of commands for:

- Configuring the editor
- Configuring the editor colors and fonts
- Configuring the project build command
- Organizing the windows in C-SPY
- Using an external editor
- Changing common fonts
- Changing key bindings
- Configuring the amount of output to the Messages window.

In addition, you can increase the number of recognized filename extensions. By default, each tool in the build tool chain accepts a set of standard filename extensions. If you have source files with a different filename extension, you can modify the set of accepted filename extensions. Choose *Tools>*Filename Extensions to get access to the necessary commands.

For reference information about the commands for customizing the IDE, see [*Tools menu*](#), page 373. You can also find further information related to customizing the editor in the section [*Customizing the editor environment*](#), page 112. For further information about customizations related to C-SPY, see [*Part 4. Debugging*](#).
INVOKING EXTERNAL TOOLS

The Tools menu is a configurable menu to which you can add external tools for convenient access to these tools from within the IDE. For this reason, the menu might look different depending on which tools you have preconfigured to appear as menu commands.

To add an external tool to the menu, choose Tools>Configure Tools to open the Configure Tools dialog box.

![Configure Tools dialog box](image)

For reference information about this dialog box, see Configure Tools dialog box, page 395.

**Note:** You cannot use the Configure Tools dialog box to extend the tool chain in the IDE, see The tool chain, page 81.
After you have entered the appropriate information and clicked OK, the menu command you have specified is displayed on the Tools menu.

Figure 34: Customized Tools menu

Note: If you intend to add an external tool to the standard build tool chain, see Extending the tool chain, page 101.

Adding command line commands

Command line commands and calls to batch files must be run from a command shell.

You can add command line commands to the Tools menu and execute them from there.

To add commands to the Tools menu, you must specify an appropriate command shell.

1 Type one of these command shells in the Command text box:

<table>
<thead>
<tr>
<th>Command shell</th>
<th>System</th>
</tr>
</thead>
<tbody>
<tr>
<td>cmd.exe</td>
<td>Windows XP/Vista/7</td>
</tr>
</tbody>
</table>

Table 13: Command shells

2 Specify the command line command or batch file name in the Argument text box.

The Argument text should be specified as:

/C name

where name is the name of the command or batch file you want to run.

The /C option terminates the shell after execution, to allow the IDE to detect when the tool has finished.

Example

To add the command Backup to the Tools menu to make a copy of the entire project directory to a network drive, you would specify Command either as command.cmd or as cmd.exe depending on your host environment, and Argument as:

/C copy c:\project\*. * F:

Alternatively, to use a variable for the argument to allow relocatable paths:

/C copy $PROJ_DIR$\*. * F:
Managing projects

This chapter discusses the project model used by the IAR Embedded Workbench IDE. It covers how projects are organized and how you can specify workspaces with multiple projects, build configurations, groups, source files, and options that help you handle different versions of your applications. The chapter also describes the steps involved in interacting with an external third-party source code control system.

The project model

In a large-scale development project, with hundreds of files, you must be able to organize the files in a structure that is easily navigated and maintained by perhaps several engineers involved.

The IDE is a flexible environment for developing projects also with several different target processors in the same project, and a selection of tools for each target processor.

HOW PROJECTS ARE ORGANIZED

The IDE has been designed to suit the way that software development projects are typically organized. For example, perhaps you need to develop related versions of an application for different versions of the target hardware, and you might also want to include debugging routines into the early versions, but not in the final application.

Versions of your applications for different target hardware will often have source files in common, and you might want to be able to maintain only one unique copy of these files, so that improvements are automatically carried through to each version of the application. Perhaps you also have source files that differ between different versions of the application, such as those dealing with hardware-dependent aspects of the application.

The IDE allows you to organize projects in a hierarchical tree structure showing the logical structure at a glance. In the following sections the various levels of the hierarchy are described.

Projects and workspaces

Typically you create a project which contains the source files needed for your embedded systems application. If you have several related projects, you can access and work with them simultaneously. To achieve this, you can organize related projects in workspaces.
Each workspace you define can contain one or more projects, and each project must be part of at least one workspace.

Consider this example: two related applications—for instance A and B—are developed, requiring one development team each (team A and B). Because the two applications are related, they can share parts of the source code between them. The following project model can be applied:

- Three projects—one for each application, and one for the common source code
- Two workspaces—one for team A and one for team B.

Collecting the common sources in a library project (compiled but not linked object code) is both convenient and efficient, to avoid having to compile it unnecessarily.

For an example where a library project has been combined with an application project, see the chapter Creating and using libraries in Part 2. Tutorials.

**Projects and build configurations**

Often, you need to build several versions of your project. The Embedded Workbench lets you define multiple build configurations for each project. In a simple case, you might need just two, called Debug and Release, where the only differences are the options used for optimization, debug information, and output format. In the Release
configuration, the preprocessor symbol \texttt{NDEBUG} is defined, which means the application will not contain any asserts.

Additional build configurations might be useful, for instance, if you intend to use the application on different target devices. The application is the same, but hardware-related parts of the code differ. Thus, depending on which target device you intend to build for, you can exclude some source files from the build configuration. These build configurations might fulfil these requirements for Project A:

- Project A - Device 1: Release
- Project A - Device 1: Debug
- Project A - Device 2: Release
- Project A - Device 2: Debug

Groups

Normally, projects contain hundreds of files that are logically related. You can define each project to contain one or more groups, in which you can collect related source files. You can also define multiple levels of subgroups to achieve a logical hierarchy. By default, each group is present in all build configurations of the project, but you can also specify a group to be excluded from a particular build configuration.

Source files

Source files can be located directly under the project node or in a hierarchy of groups. The latter is convenient if the amount of files makes the project difficult to survey. By default, each file is present in all build configurations of the project, but you can also specify a file to be excluded from a particular build configuration.

Only the files that are part of a build configuration will actually be built and linked into the output code.

Once a project has been successfully built, all include files and output files are displayed in the structure below the source file that included or generated them.

\textbf{Note:} The settings for a build configuration can affect which include files that are used during the compilation of a source file. This means that the set of include files associated with the source file after compilation can differ between the build configurations.

\textbf{CREATING AND MANAGING WORKSPACES}

This section describes the overall procedure for creating the workspace, projects, groups, files, and build configurations. The \textbf{File} menu provides the commands for creating workspaces. The \textbf{Project} menu provides commands for creating projects, adding files to a project, creating groups, specifying project options, and running the IAR Systems development tools on the current projects.
For reference information about these menus, menu commands, and dialog boxes, see the chapter IAR Embedded Workbench® IDE reference.

The steps involved for creating and managing a workspace and its contents are:

- Creating a workspace.
  An empty Workspace window appears, which is the place where you can view your projects, groups, and files.
- Adding new or existing projects to the workspace.
  When creating a new project, you can base it on a *template project* with preconfigured project settings. Template projects are available for C applications, C++ applications, assembler applications, and library projects.
- Creating groups.
  A group can be added either to the project’s top node or to another group within the project.
- Adding files to the project.
  A file can be added either to the project’s top node or to a group within the project.
- Creating new build configurations.
  By default, each project you add to a workspace will have two build configurations called Debug and Release.
  You can base a new configuration on an already existing configuration. Alternatively, you can choose to create a default build configuration.
  Note that you do not have to use the same tool chain for the new build configuration as for other build configurations in the same project.
- Excluding groups and files from a build configuration.
  Note that the icon indicating the excluded group or file will change to white in the Workspace window.
- Removing items from a project.

For a detailed example, see *Creating an application project*, page 33.

**Note:** It might not be necessary for you to perform all of these steps.

**Drag and drop**

You can easily drag individual source files and project files from the Windows file explorer to the Workspace window. Source files dropped on a *group* are added to that group. Source files dropped outside the project tree—on the Workspace window background—are added to the active project.
**Source file paths**

The IDE supports relative source file paths to a certain degree, for:

- **Project file**
  Paths to files part of the project file is relative if they are located on the same drive. The path is relative either to $PROJ_DIR$ or $SEM_DIR$. The argument variable $SEM_DIR$ is only used if the path refers to a file located in subdirectory to $SEM_DIR$ and the distance from $SEM_DIR$ is shorter than the distance from $PROJ_DIR$.
  Paths to files that are part of the project file are absolute if the files are located on different drives.

- **Workspace file**
  For files located on the same drive as the workspace file, the path is relative to $PROJ_DIR$.
  For files located on another drive as the workspace file, the path is absolute.

- **Debug files**
  The path is absolute if the file is built with IAR Systems compilation tools.

**Starting the IAR C-SPY® Debugger**

When you start the IAR C-SPY Debugger, the current project is loaded. It is also possible to load C-SPY with a project that was built outside IAR Embedded Workbench, for example projects built on the command line. For more information, see *Starting C-SPY*, page 122.

**Navigating project files**

There are two main different ways to navigate your project files: using the Workspace window or the Source Browser window. The Workspace window displays an hierarchical view of the source files, dependency files, and output files and how they are logically grouped. The Source Browser window, on the other hand, displays information about the build configuration that is currently active in the Workspace window. For that configuration, the Source Browser window displays a hierarchical view of all globally defined symbols, such as variables, functions, and type definitions. For classes, information about any base classes is also displayed.
VIEWING THE WORKSPACE

The Workspace window is where you access your projects and files during the application development.

1 To choose which project you want to view, click its tab at the bottom of the Workspace window.

2 To display the project with a different build configuration, choose that build configuration from the drop-down list at the top of the Workspace window.

For each file that has been built, an Output folder icon appears, containing generated files, such as object files and list files. The latter is generated only if the list file option is enabled. There is also an Output folder related to the project node that contains generated files related to the whole project, such as the executable file and the linker map file (if the list file option is enabled).

Also, any included header files will appear, showing dependencies at a glance.
The project and build configuration you have selected are displayed highlighted in the Workspace window. It is the project and build configuration that you select from the drop-down list that is built when you build your application.

3 To display an overview of all projects in the workspace, click the **Overview** tab at the bottom of the Workspace window.

An overview of all project members is displayed.

![Workspace window—an overview](image)

The current selection in the **Build Configuration** drop-down list is also highlighted when an overview of the workspace is displayed.

**DISPLAYING BROWSE INFORMATION**

To display browse information in the Source Browser window, choose **Tools** > **Options** > **Project** and select the option **Generate browse information**.

To open the Source Browser window, choose **View** > **Source Browser**. The Source Browser window is, by default, docked with the Workspace window. Source browse information is displayed for the active build configuration. For reference information, see **Source Browser window**, page 336.

Note that you can choose a file filter and a type filter from the context menu that appears when you right-click in the top pane of the window.

To see the definition of a global symbol or a function, you can use three alternative methods:

- In the Source Browser window, right-click on a symbol, or function, and choose the **Go to definition** command from the context menu that appears.
In the Source Browser window, double-click on a row
In the editor window, right-click on a symbol, or function, and choose the Go to definition command from the context menu that appears.

The definition of the symbol or function is displayed in the editor window.

The source browse information is continuously updated in the background. While you are editing source files, or when you open a new project, there will be a short delay before the information is up-to-date.

Source code control

IAR Embedded Workbench can identify and access any installed third-party source code control (SCC) system that conforms to the SCC interface published by Microsoft corporation. From within the IDE you can connect an IAR Embedded Workbench project to an external SCC project, and perform some of the most commonly used operations.

To connect your IAR Embedded Workbench project to a source code control system you should be familiar with the source code control client application you are using. Note that some of the windows and dialog boxes that appear when you work with source code control in the IAR Embedded Workbench IDE originate from the SCC system and are not described in the documentation from IAR Systems. For information about details in the client application, refer to the documentation supplied with that application.

Note: Different SCC systems use very different terminology even for some of the most basic concepts involved. You must keep this in mind when you read the following description.

INTERACTING WITH SOURCE CODE CONTROL SYSTEMS

In any SCC system, you use a client application to maintain a central archive. In this archive you keep the working copies of the files of your project. The SCC integration in IAR Embedded Workbench allows you to conveniently perform a few of the most common SCC operations directly from within the IDE. However, several tasks must still be performed in the client application.

To connect an IAR Embedded Workbench project to a source code control system, you should:

- In the SCC client application, set up an SCC project
- In IAR Embedded Workbench, connect your project to the SCC project.
Setting up an SCC project in the SCC client application

Use your SCC client tools to set up a working directory for the files in your IAR Embedded Workbench project that you want to control using your SCC system. The files can be placed in one or more nested subdirectories, all located under a common root. Specifically, all the source files must reside in the same directory as the ewp project file, or nested in subdirectories of this directory.

For information about the steps involved, refer to the documentation supplied with the SCC client application.

Connecting projects in IAR Embedded Workbench

In IAR Embedded Workbench, connect your application project to the SCC project.

1 In the Workspace window, select the project for which you have created an SCC project. From the Project menu, choose Source Code Control>Add Project To Source Control. This command is also available from the context menu that appears when you right-click in the Workspace window.

Note: The commands on the Source Code Control submenu are available when at least one SCC client application is available.

2 If you have source code control systems from different vendors installed, a dialog box will appear to let you choose which system you want to connect to.

3 An SCC-specific dialog box will appear where you can navigate to the proper SCC project that you have set up.

Viewing the SCC states

When your IAR Embedded Workbench project has been connected to the SCC project, a column that contains status information for source code control will appear in the Workspace window. Different icons are displayed depending on whether:

- a file is checked out to you
- a file is checked out to someone else
- a file is checked in
- a file has been modified
- a new version of a file is in the archive.

There are also icons for some combinations of these states. Note that the interpretation of these states depends on the SCC client application you are using. For reference information about the icons and the different states they represent, see Source code control states, page 326.
For reference information about the commands available for accessing the SCC system, see Source Code Control menu, page 324.

**Configuring the source code control system**

To customize the source code control system, choose Tools>Options and click the Source Code Control tab. For reference information about the available commands, see Source Code Control options, page 388.
Building

This chapter briefly discusses the process of building your application, and describes how you can extend the chain of build tools with tools from third-party suppliers.

Building your application

The building process consists of these steps:

- Setting project options
- Building the project
- Correcting any errors detected during the build procedure.

To make the build process more efficient, you can use the Batch Build command. This gives you the possibility to perform several builds in one operation. If necessary, you can also specify pre-build and post-build actions.

In addition to using the IAR Embedded Workbench IDE to build projects, you can also use the command line utility iarbuild.exe.

For examples of building application and library projects, see Part 2. Tutorials in this guide. For further information about building library projects, see the IAR C/C++ Development Guide for ARM®.

SETTING OPTIONS

To specify how your application should be built, you must define one or several build configurations. Every build configuration has its own settings, which are independent of the other configurations. All settings are indicated in a separate column in the Workspace window.

For example, a configuration that is used for debugging would not be highly optimized, and would produce output that suits the debugging. Conversely, a configuration for building the final application would be highly optimized, and produce output that suits a flash or PROM programmer.

For each build configuration, you can set options on the project level, group level, and file level. Many options can only be set on the project level because they affect the entire build configuration. Examples of such options are General Options (for example, processor variant and library object file), linker settings, and debug settings. Other options, such as compiler and assembler options, that you set on project level are default for the entire build configuration.
To override project level settings, select the required item—for instance a specific group of files—and then select the option **Override inherited settings**. The new settings will affect all members of that group, that is, files and any groups of files. To restore all settings to the default factory settings, click the **Factory Settings** button.

**Note:** There is one important restriction on setting options. If you set an option on group or file level (group or file level override), no options on higher levels that operate on files will affect that group or file.

### Using the Options dialog box

The **Options** dialog box—available by choosing **Project>Options**—provides options for the build tools. You set these options for the selected item in the Workspace window. Options in the **General Options**, **Linker**, and **Debugger** categories can only be set for the entire build configuration, and not for individual groups and files. However, the options in the other categories can be set for the entire build configuration, a group of files, or an individual file.

**Figure 38: General options**

The **Category** list allows you to select which building tool to set options for. Which tools that are available in the **Category** list depends on which tools are included in your product. If you select **Library** as output file on the **Output** page, **Linker** is replaced by...
**Library Builder** in the category list. When you select a category, one or more pages containing options for that component are displayed.

Click the tab corresponding to the type of options you want to view or change. To restore all settings to the default factory settings, click the **Factory Settings** button, which is available for all categories except **General Options** and **Custom Build**. Note that two sets of factory settings are available: Debug and Release. Which one that is used depends on your build configuration; see **New Configuration dialog box**, page 368.

For information about each option and how to set options, see the chapters **General options**, **Compiler options**, **Assembler options**, **Linker options**, **Library builder options**, **Custom build options**, and **Debugger options** in Part 7. **Reference information** in this guide. For information about options specific to the C-SPY driver you are using, see the part of this book that corresponds to your driver.

**Note:** If you add to your project a source file with a non-recognized filename extension, you cannot set options on that source file. However, you can add support for additional filename extensions. For reference information, see **Filename Extensions dialog box**, page 397.

**BUILDING A PROJECT**

You have access to the build commands both from the **Project** menu and from the context menu that appears if you right-click an item in the Workspace window.

The three build commands **Make**, **Compile**, and **Rebuild All** run in the background, so you can continue editing or working with the IDE while your project is being built.

For further reference information, see **Project menu**, page 363.

**BUILDING MULTIPLE CONFIGURATIONS IN A BATCH**

Use the batch build feature when you want to build more than one configuration at once. A batch is an ordered list of build configurations. The **Batch Build** dialog box—available from the **Project** menu—lets you create, modify, and build batches of configurations.

For workspaces that contain several configurations, it is convenient to define one or more different batches. Instead of building the entire workspace, you can build only the appropriate build configurations, for instance Release or Debug configurations.

For detailed information about the **Batch Build** dialog box, see **Batch Build dialog box**, page 371.
USING PRE- AND POST-BUILD ACTIONS

If necessary, you can specify pre-build and post-build actions that you want to occur before or after the build. The Build Actions dialog box—available from the Project menu—lets you specify the actions required.

For detailed information about the Build Actions dialog box, see Build actions options, page 479.

Using pre-build actions for time stamping

You can use pre-build actions to embed a time stamp for the build in the resulting binary file. Follow these steps:

1. Create a dedicated time stamp file, for example, timestamp.c and add it to your project.
2. In this source file, use the preprocessor macros __TIME__ and __DATE__ to initialize a string variable.
3. Choose Project>Options>Build Actions to open the Build Actions dialog box.
4. In the Pre-build command line text field, specify for example this pre-build action:
   "touch $PROJ_DIR\timestamp.c"
   You can use the open source command line utility touch for this purpose or any other suitable utility which updates the modification time of the source file.
5. If the project is not entirely up-to-date, the next time you use the Make command, the pre-build action will be invoked before the regular build process. Then the regular build process must always recompile timestamp.c and the correct timestamp will end up in the binary file.
   If the project already is up-to-date, the pre-build action will not be invoked. This means that nothing is built, and the binary file still contains the timestamp for when it was last built.

CORRECTING ERRORS FOUND DURING BUILD

The compiler, assembler, and debugger are fully integrated with the development environment. If your source code contains errors, you can jump directly to the correct position in the appropriate source file by double-clicking the error message in the error listing in the Build message window, or selecting the error and pressing Enter.

After you have resolved any problems reported during the build process and rebuilt the project, you can directly start debugging the resulting code at the source level.

To specify the level of output to the Build message window, choose Tools>Options to open the IDE Options dialog box. Click the Messages tab and select the level of output
in the Show build messages drop-down list. Alternatively, you can right-click in the Build Messages window and select Options from the context menu.

For reference information about the Build messages window, see Build window, page 347.

**BUILDING FROM THE COMMAND LINE**

To build the project from the command line, use the IAR Command Line Build Utility ( iarbuild.exe ) located in the common\bin directory. As input you use the project file, and the invocation syntax is:

```
 iarbuild project.ewp [-clean|-build|-make] <configuration>
 [-log errors|warnings|info|all]
```

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>project.ewp</td>
<td>Your IAR Embedded Workbench project file.</td>
</tr>
<tr>
<td>-clean</td>
<td>Removes any intermediate and output files.</td>
</tr>
<tr>
<td>-build</td>
<td>Rebuilds and relinks all files in the current build configuration.</td>
</tr>
<tr>
<td>-make</td>
<td>Brings the current build configuration up to date by compiling, assembling,</td>
</tr>
<tr>
<td></td>
<td>and linking only the files that have changed since the last build.</td>
</tr>
<tr>
<td>configuration</td>
<td>The name of the configuration you want to build, which can either be one of</td>
</tr>
<tr>
<td></td>
<td>the predefined configurations Debug or Release, or a name that you define</td>
</tr>
<tr>
<td></td>
<td>yourself. For more information about build configurations, see Projects</td>
</tr>
<tr>
<td></td>
<td>and build configurations, page 88.</td>
</tr>
<tr>
<td>-log errors</td>
<td>Displays build error messages.</td>
</tr>
<tr>
<td>-log warnings</td>
<td>Displays build warning and error messages.</td>
</tr>
<tr>
<td>-log info</td>
<td>Displays build warning and error messages, and messages issued by the</td>
</tr>
<tr>
<td></td>
<td>#pragma message preprocessor directive.</td>
</tr>
<tr>
<td>-log all</td>
<td>Displays all messages generated from the build, for example compiler</td>
</tr>
<tr>
<td></td>
<td>sign-on information and the full command line.</td>
</tr>
</tbody>
</table>

Table 14: iarbuild.exe command line options

If you run the application from a command shell without specifying a project file, you will get a sign-on message describing available parameters and their syntax.

**Extending the tool chain**

IAR Embedded Workbench provides a feature—Custom Build—which lets you extend the standard tool chain. This feature is used for executing external tools (not provided
Extending the tool chain

by IAR Systems). You can make these tools execute each time specific files in your project have changed.

If you specify custom build options on the Custom tool configuration page, the build commands treat the external tool and its associated files in the same way as the standard tools within the IAR Embedded Workbench IDE and their associated files. The relation between the external tool and its input files and generated output files is similar to the relation between the C/C++ Compiler, c files, h files, and o files. See Custom build options, page 477, for details about available custom build options.

You specify filename extensions of the files used as input to the external tool. If the input file has changed since you last built your project, the external tool is executed; just as the compiler executes if a c file has changed. In the same way, any changes in additional input files (for instance include files) are detected.

You must specify the name of the external tool. You can also specify any necessary command line options needed by the external tool, and the name of the output files generated by the external tool. Note that you can use argument variables for substituting file paths.

For some of the file information, you can use argument variables.

You can specify custom build options to any level in the project tree. The options you specify are inherited by any sublevel in the project tree.

TOOLS THAT CAN BE ADDED TO THE TOOL CHAIN

Some examples of external tools, or types of tools, that you can add to the IAR Embedded Workbench tool chain are:

- Tools that generate files from a specification, such as Lex and YACC
- Tools that convert binary files—for example files that contain bitmap images or audio data—to a table of data in an assembler or C source file. This data can then be compiled and linked together with the rest of your application.

ADDING AN EXTERNAL TOOL

The following example demonstrates how to add the tool Flex to the tool chain. The same procedure can be used also for other tools.

In the example, Flex takes the file foo.lex as input. The two files foo.c and foo.h are generated as output.

1. Add the file you want to work with to your project, for example foo.lex.

2. Select this file in the Workspace window and choose Project>Options. Select Custom Build from the list of categories.
3 In the **Filename extensions** field, type the filename extension `.lex`. Remember to specify the leading period (`.`).

4 In the **Command line** field, type the command line for executing the external tool, for example

```
flex $FILE_PATH$ -o$FILE_BPATH$.c
```

During the build process, this command line is expanded to:

```
flex foo.lex -ofoo.c
```

Note the usage of *argument variables*. For further details of these variables, see *Argument variables summary*, page 366.

Take special note of the use of `$FILE_BNAME$` which gives the base name of the input file, in this example appended with the `c` extension to provide a C source file in the same directory as the input file `foo.lex`.

5 In the **Output files** field, describe the output files that are relevant for the build. In this example, the tool Flex would generate two files—one source file and one header file. The text in the **Output files** text box for these two files would look like this:

```
$FILE_BPATH$.c  
$FILE_BPATH$.h
```

6 If the external tool uses any additional files during the build, these should be added in the **Additional input files** field, for instance:

```
$TOOLKIT_DIR\inc\stdio.h
```

This is important, because if the dependency files change, the conditions will no longer be the same and the need for a rebuild is detected.

7 Click **OK**.

8 To build your application, choose **Project>Make**.
Extending the tool chain
Editing

This chapter describes in detail how to use the IAR Embedded Workbench editor. The final section describes how to customize the editor and how to use an external editor of your choice.

Using the IAR Embedded Workbench editor

The integrated text editor allows editing of multiple files in parallel, and provides all basic editing features expected from a modern editor. In addition, it provides features specific to software development. It also recognizes C or C++ language elements.

EDITING A FILE

The editor window is where you write, view, and modify your source code. You can open one or several text files, either from the File menu, or by double-clicking a file in the Workspace window. If you open several files, they are organized in a tab group. Several editor windows can be open at the same time.
Using the IAR Embedded Workbench editor

Click the tab for the file that you want to display. All open files are also available from the drop-down menu at the upper right corner of the editor window.

The name of the open source file is displayed on the tab. If a file is read-only, a padlock is visible at the bottom left corner of the editor window. If a file has been modified after it was last saved, an asterisk appears on the tab after the filename, for example Utilities.c *.

The commands on the Window menu allow you to split the editor window into panes. On the Window menu you also find commands for opening multiple editor windows, and commands for moving files between editor windows. For reference information about each command on the menu, see Window menu, page 400. For reference information about the editor window, see Editor window, page 330.

Note: When you want to print a source file, it can be useful to enable the option Show line numbers—available by choosing Tools>Options>Editor.

Accessing reference information for DLIB library functions

When you need to know the syntax for any C or Embedded C++ library function, select the function name in the editor window and press F1. The library documentation for the selected function appears in a help window.

IAR Embedded Workbench® IDE
106 User Guide for ARM
Using and customizing editor commands and shortcut keys

The Edit menu provides commands for editing and searching in editor windows, for instance, unlimited undo/redo (the Edit>Undo and Edit>Redo commands, respectively). You can also find some of these commands on the context menu that appears when you right-click in the editor window. For reference information about each command, see Edit menu, page 353.

There are also editor shortcut keys for:

- moving the insertion point
- scrolling text
- selecting text.

For detailed information about these shortcut keys, see Editor key summary, page 335.

To change the default shortcut key bindings, choose Tools>Options, and click the Key Bindings tab. For further details, see Key Bindings options, page 375.

Splitting the editor window into panes

You can split the editor window horizontally or vertically into multiple panes, to look at different parts of the same source file at once, or to move text between two different panes.

To split the window, double-click the appropriate splitter bar, or drag it to the middle of the window. Alternatively, you can split a window into panes using the Window>Split command.

To revert to a single pane, double-click the splitter control or drag it back to the end of the scroll bar.

Dragging and dropping of text

You can easily move text within an editor window or between editor windows. Select the text and drag it to the new location.

Syntax coloring

If the Tools>Options>Editor>Syntax highlighting option is enabled, the IAR Embedded Workbench editor automatically recognizes the syntax of:

- C and C++ keywords
- C and C++ comments
- Assembler directives and comments
- Preprocessor directives
- Strings.
Using the IAR Embedded Workbench editor

The different parts of source code are displayed in different text styles.

To change these styles, choose Tools>Options, and use the Editor>Colors and Fonts options. For additional information, see Editor Colors and Fonts options, page 383.

In addition, you can define your own set of keywords that should be syntax-colored automatically:

1. In a text file, list all the keywords that you want to be automatically syntax-colored. Separate each keyword with either a space or a new line.

2. Choose Tools>Options and select Editor>Setup Files.

3. Select the Use Custom Keyword File option and specify your newly created text file. A browse button is available for your convenience.

4. Select Editor>Colors and Fonts and choose User Keyword from the Syntax Coloring list. Specify the font, color, and type style of your choice. For additional information, see Editor Colors and Fonts options, page 383.

5. In the editor window, type any of the keywords you listed in your keyword file; see how the keyword is syntax-colored according to your specification.

Automatic text indentation

The text editor can perform various kinds of indentation. For assembler source files and normal text files, the editor automatically indents a line to match the previous line. If you want to indent several lines, select the lines and press the Tab key. Press Shift+Tab to move a whole block of lines to the left.

For C/C++ source files, the editor indents lines according to the syntax of the C/C++ source code. This is performed whenever you:

- Press the Return key
- Type any of the special characters {, }, ::, and #
- Have selected one or several lines, and choose the Edit>Auto Indent command.

To enable or disable the indentation:

1. Choose Tools>Options and select Editor.

2. Select or deselect the Auto indent option.

To customize the C/C++ automatic indentation, click the Configure button.

For additional information, see Configure Auto Indent dialog box, page 379.
Matching brackets and parentheses

When the insertion point is located next to a parenthesis, the matching parenthesis is highlighted with a light gray color:

```c
for(int i = 0; i < 10; i++)
{
}
```

Figure 40: Parentheses matching in editor window

The highlight remains in place as long as the insertion point is located next to the parenthesis.

To select all text between the brackets surrounding the insertion point, choose Edit>Match Brackets. Every time you choose Match Brackets after that, the selection will increase to the next hierarchic pair of brackets.

Note: Both of these functions—automatic matching of corresponding parentheses and selection of text between brackets—apply to (), [], and {}.

Displaying status information

As you are editing, the status bar—available by choosing View>Status Bar—shows the current line and column number containing the insertion point, and the Caps Lock, Num Lock, and Overwrite status:

Figure 41: Editor window status bar

USING AND ADDING CODE TEMPLATES

Code templates is a method for conveniently inserting frequently used source code sequences, for example for loops and if statements. The code templates are defined in a normal text file. By default, a few example templates are provided. In addition, you can easily add your own code templates.

Enabling code templates

By default, code templates are enabled. To enable and disable the use of code templates:

1. Choose Tools>Options.
2. Go to the Editor Setup Files page.
3 Select or deselect the Use Code Templates option.

4 In the text field, specify which template file you want to use; either the default file or one of your own template files. A browse button is available for your convenience.

**Inserting a code template into your source code**

To insert a code template into your source code, place the insertion point at the location where you want the template to be inserted, right-click, and choose Insert Template and the appropriate code template from the menu that appears.

If the code template you choose requires any type of field input, as in the for loop example which needs an end value and a count variable, an input dialog box appears.

**Adding your own code templates**

The source code templates are defined in a normal text file. The original template file CodeTemplates.txt is located in the common\config installation directory. The first time you use IAR Embedded Workbench, the original template file is copied to a directory for local settings, and this is the file that is used by default if code templates are enabled. To use your own template file, follow the procedure described in Enabling code templates, page 109.

To open the template file and define your own code templates, choose Edit>Code Templates>Edit Templates.

The syntax for defining templates is described in the default template file.
Selecting the correct language version of the code template file

When you start the IAR Embedded Workbench IDE for the very first time, you are asked to select a language version. This only applies if you are using an IDE that is available in other languages than English.

Selecting a language creates a corresponding language version of the default code template file in the Application Data\IAR Embedded Workbench subdirectory of the current Windows user (for example CodeTemplates.ENU.txt for English and CodeTemplates.JPN.txt for Japanese). The default code template file does not change automatically if you change the language version of the IDE afterwards.

To change the code template:

1. Choose Tools>Options>IDE Options>Editor>Setup Files.
2. Click the browse button of the Use Code Templates option and select a different template file.
   
   If the code template file you want to select is not in the browsed directory, you must:
3. Delete the file name in the Use Code Templates text box.
4. Deselect the Use Code Templates option and click OK.
5. Restart the IAR Embedded Workbench IDE.
6. Then choose Tools>Options>IDE Options>Editor>Setup Files again.

   The default code template file for the selected language version of the IDE should now be displayed in the Use Code Templates text box. Select the check box to enable the template.

NAVIGATING IN AND BETWEEN FILES

The editor provides several functions for easy navigation within the files and between files:

- Switching between source and header files
  
  If the insertion point is located on an #include line, you can choose the Open "header.h" command from the context menu, which opens the header file in an editor window. You can also choose the command Open Header/Source File, which opens the header or source file that corresponds to the current file, or activates it if it is already open. This command is available if the insertion point is located on any line except an #include line.

Part 3. Project management and building

111
Customizing the editor environment

- Function navigation
  Click the Go to function button in the bottom left corner in an editor window to list all functions defined in the source file displayed in the window. You can then choose to go directly to one of the functions by double-clicking it in the list.

- Adding bookmarks
  Use the Edit>Navigate>Toggle Bookmark command to add and remove bookmarks. To switch between the marked locations, choose Edit>Navigate>Go to Bookmark.

SEARCHING
There are several standard search functions available in the editor:
- Quick search text box
- Find dialog box
- Replace dialog box
- Find in files dialog box
- Incremental Search dialog box.

To use the Quick search text box on the toolbar, type the text you want to search for and press Enter. Press Esc to cancel the search. This is a quick method for searching for text in the active editor window.

To use the Find, Replace, Find in Files, and Incremental Search functions, choose the corresponding command from the Edit menu. For reference information about each search function, see Edit menu, page 353.

Customizing the editor environment
The IDE editor can be configured on the IDE Options pages Editor and Editor Colors and Fonts. Choose Tools>Options to access the pages.
For details about these pages, see Tools menu, page 373.

USING AN EXTERNAL EDITOR
The External Editor options—available by choosing Tools>Options>Editor—let you specify an external editor of your choice.

Note: While debugging using C-SPY, C-SPY will not use the external editor for displaying the current debug state. Instead, the built-in editor will be used.

To specify an external editor of your choice, follow this procedure:
1. Select the option Use External Editor.
2. An external editor can be called in one of two ways, using the **Type** drop-down menu. **Command Line** calls the external editor by passing command line parameters. **DDE** calls the external editor by using DDE (Windows Dynamic Data Exchange).

3. If you use the command line, specify the command line to pass to the editor, that is, the name of the editor and its path, for instance:

```
C:\WINNT\NOTEPAD.EXE.
```

To send an argument to the external editor, type the argument in the **Arguments** field. For example, type `$FILE_PATH$` to start the editor with the active file (in editor, project, or Messages window).

4. If you use DDE, specify the editor’s DDE service name in the **Service** field. In the **Command** field, specify a sequence of command strings to send to the editor.

The service name and command strings depend on the external editor that you are using. Refer to the user documentation of your external editor to find the appropriate settings.

The command strings should be entered as:

```
DDE-Topic CommandString1
DDE-Topic CommandString2
```
as in this example, which applies to Codewright®:

The command strings used in this example will open the external editor with a dedicated file activated. The cursor will be located on the current line as defined in the context from where the file is open, for instance when searching for a string in a file, or when double-clicking an error message in the Message window.

5 Click OK.

When you double-click a file in the Workspace window, the file is opened by the external editor.

Variables can be used in the arguments. For more information about the argument variables that are available, see Argument variables summary, page 366.
Part 4. Debugging

This part of the IAR Embedded Workbench® IDE User Guide contains the following chapters:

● The IAR C-SPY® Debugger
● Executing your application
● Working with variables and expressions
● Using breakpoints
● Monitoring memory and registers
● Using the C-SPY® macro system
● Analyzing your application
● Using trace.
The IAR C-SPY® Debugger

This chapter introduces you to the IAR C-SPY Debugger. First some of the concepts are introduced that are related to debugging in general and to C-SPY in particular. Then the C-SPY environment is presented, followed by a description of how to set up, start, and finally adapt C-SPY to target hardware.

Debugger concepts

This section introduces some of the concepts that are related to debugging in general and to C-SPY in particular. This section does not contain specific conceptual information related to the functionality of C-SPY. Instead, you will find such information in each chapter of this part of the guide. The IAR Systems user documentation uses the following terms 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 shows an overview of C-SPY and possible target systems.

**DEBUGGER**

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

**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.

**USER 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. There are three main types of C-SPY drivers:

- Simulator driver
- ROM-monitor driver
- Emulator driver

If you have more than one C-SPY driver installed on your computer, you can switch between them from within the IDE.

For an overview of the general features of C-SPY, see IAR C-SPY Debugger, page 5. In that chapter you can also find an overview of the functionality provided by each driver. Contact your software distributor or IAR Systems representative for information about available C-SPY drivers. You can also find information on the IAR Systems website, www.iar.com.

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 tool chain as long as the third-party debugger can read ELF/DWARF, Intel-extended, or Motorola. For information about which format to use with third-party debuggers, see the user documentation supplied with that tool.

The C-SPY environment

AN INTEGRATED ENVIRONMENT
C-SPY is a high-level-language debugger for embedded applications. It is designed for use with the IAR Systems compiler and assembler for ARM, and is completely integrated in the IDE, providing development and debugging within the same application.

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.
You can modify your source code in an editor window during the debug session, but changes will not take effect until you exit from the debugger and rebuild your application.

The integration also makes it possible to set breakpoints in the text editor 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 remain between your debug sessions. When the debugger is running, breakpoints are highlighted in the editor windows.

In addition to the features available in the IDE, the C-SPY environment consists of a set of C-SPY-specific items, such as a debugging toolbar, menus, windows, and dialog boxes.

For reference information about each item specific to C-SPY, see the chapter C-SPY® reference, page 403.

For specific information about a C-SPY driver, see the part of the book corresponding to the driver.

### Setting up C-SPY

Before you start C-SPY, you should set options to set up the debugger system. These options are available on the Setup page of the Debugger category, available with the Project>Options command. On the Plugins page you can find options for loading plug-in modules.

In addition to the options for setting up the debugger system, you can also set debugger-specific IDE options. These options are available with the Tools>Options command. For further information about these options, see Debugger options, page 389.

For information about how to configure the debugger to reflect the target hardware, see Adapting C-SPY to target hardware, page 125.

#### CHOOSING A DEBUG DRIVER

Before starting C-SPY, you must choose a driver for the debugger system from the Driver drop-down list on the Setup page.

If you choose a driver for a hardware debugger system, you must also set hardware-specific options. For information about these options, see Part 5. C-SPY hardware debugger systems, page 203 in this guide.

**Note:** You can only choose a driver you have installed on your computer.
EXECUTING FROM RESET

Using the Run to option, you can specify 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 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 standard 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.

To register a setup macro file, 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. A browse button is available for your convenience.

For detailed information about setup macro files and functions, see The macro file, page 156. For an example about how to use a setup macro file, see the chapter Simulating an interrupt in Part 2. Tutorials.

SELECTING A DEVICE DESCRIPTION FILE

C-SPY uses device description files to handle several of the target-specific adaptations. They contain device-specific information about for example, definitions of peripheral units and CPU registers, and groups of these.

If you want to use the device-specific information provided in the device description file during your debug session, you must select the appropriate device description file. Device description files are provided in the arm\config directory and they have the filename extension .ddf.
To load a device description file that suits your device, you must, before you start C-SPY, choose Project>Options and select the Debugger category. On the Setup page, enable the use of a description file and select a file using the Device description file browse button.

For more information about device description files, see Adapting C-SPY to target hardware, page 125. For an example about how to use a setup macro file, see Simulating an interrupt in Part 2. Tutorials.

For an example about how to use a setup macro file, see Simulating an interrupt in Part 2. Tutorials.

LOADING PLUGIN MODULES

On the Plugins page you can specify C-SPY plugin modules that are to be loaded and made 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 information about how to load plugin modules, see Plugins, page 496.

The C-SPY RTOS awareness plugin modules

C-SPY RTOS awareness plugin modules give 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 set of windows and buttons when a debug session is started (provided that the RTOS is linked with the application). For links to the RTOS documentation, see the release notes that are available from the Help menu.

Starting C-SPY

When you have set up the debugger, you are ready to start a debug session.

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.

For information about how to execute your application and how to use the C-SPY features, see the remaining chapters in Part 4. Debugging.
EXECUTABLE FILES BUILT OUTSIDE OF THE IDE

You can also load C-SPY with a project that was built outside the IDE, for example projects built on the command line. To be able to set debugger options for the externally built project, you must create a project within the IDE.

To load an externally built executable file, you must first create a project for it in your workspace. Choose Project>Create New Project, and specify a project name. 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 (filename extension .out). To start the executable file, select the project in the Workspace window and click the 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.

To flash an externally generated application, a corresponding sim file must be available in the same directory as the .out file. The sim file is automatically generated by C-SPY.

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)

At this point you can choose from these actions:

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

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 code 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.

**LOADING MULTIPLE IMAGES**

Normally, a debuggable application consists of exactly one file that you debug. However, you can also load additional debug files (images) after a debug session has started. 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.

Follow these steps:

1. To load an additional debug file in the IDE, choose **Project>Options>Debugger>Images**. For more information, see **Images**, page 495.
To load an additional debug file from the command line, use the `__loadImage` system macro. For more information, see `__loadImage`, page 549.

To display a list of loaded debug files, choose **Images** from the **View** menu. The Images window is displayed, see **Images window**, page 432.

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

**REDIRECTING DEBUGGER OUTPUT TO A FILE**

The Debug Log window—available from the **View** menu—displays debugger output, such as diagnostic messages and trace information. It can sometimes be convenient to log the information to a file where you can easily inspect it. The **Log Files** dialog box—available from the **Debug** menu—allows you to log output from C-SPY to a file. 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, what breakpoints have been triggered etc.

By default, the information printed in the file is the same as the information listed in the Log window. However, you can choose what you want to log in the file: errors, warnings, system information, user messages, or all of these. For reference information about the Log File options, see **Log File dialog box**, page 441.

---

**Adapting C-SPY to target hardware**

This section describes how to configure the debugger to reflect the target hardware. The C-SPY device description file and its contents is described, as well as how you can use C-SPY macro functions to remap memory before your application is downloaded.

**DEVICE DESCRIPTION FILE**

C-SPY uses device description files provided with the product to handle several of the target-specific adaptations. They contain device-specific information such as definitions of peripheral units, and groups of these.

You can find device description files for each ARM device in the `arm\config` directory.

For information about how to load a device description file, see **Selecting a device description file**, page 121.
Registers
Each device has a hardwired group of CPU registers. Their contents can be displayed and edited in the Register window. 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 ARM cores.

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. By default, there is one register group in the ARM debugger, namely CPU Registers.

For details about how to work with the Register window, view various register groups, and how to configure your own register groups to better suit the use of registers in your application, see the section Working with registers, page 152.

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.

Modifying a device description file
Normally, you do not need to modify the device description file. However, if the predefineds are not sufficient for some reason, you can edit the file. The syntax of the device descriptions is described in the files. 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.

Note: The syntax of the device description files is described in the Writing device header files guide (EWARM_DDFFormat.pdf) located in the arm\doc directory.

REMAPPING MEMORY
A common feature of many ARM-based processors is the ability to remap memory. After a reset, the memory controller typically maps address zero to non-volatile memory, such as flash. By configuring the memory controller, the system memory can be remapped to place RAM at zero and non-volatile memory higher up in the address map. By doing this the exception table will reside in RAM and can be easily modified when you download code to the evaluation board.
You must configure the memory controller before you download your application code. You can do this best by using a C-SPY macro function that is executed before the code download takes place—execUserPreload(). The macro function __writeMemory32() will perform the necessary initialization of the memory controller.

The following example illustrates a macro used to set up the memory controller and remap memory on the Atmel AT91EB55 chip, similar mechanisms exist in processors from other ARM vendors.

```
execUserPreload()
{
    __message "Setup memory controller, do remap command\n";

    // Flash at 0x01000000, 16MB, 2 hold, 16 bits, 3 WS
    __writeMemory32(0x01002529, 0xffe00000, "Memory");

    // RAM at 0x02000000, 16MB, 0 hold, 16 bits, 1 WS
    __writeMemory32(0x02002121, 0xffe00004, "Memory");

    // unused
    __writeMemory32(0x20000000, 0xffe00008, "Memory");

    // unused
    __writeMemory32(0x30000000, 0xffe00010, "Memory");

    // unused
    __writeMemory32(0x40000000, 0xffe00014, "Memory");

    // unused
    __writeMemory32(0x50000000, 0xffe00018, "Memory");

    // unused
    __writeMemory32(0x60000000, 0xffe0001c, "Memory");

    // REMAP command
    __writeMemory32(0x00000001, 0xffe00020, "Memory");

    // standard read
    __writeMemory32(0x00000000, 0xffe00024, "Memory");
}
```
Adapting C-SPY to target hardware

Note that the setup macro `execUserReset()` might have to be defined in the same way to reinitialize the memory mapping after a C-SPY reset. This can be needed if you have set up your hardware debugger system to do a hardware reset on C-SPY reset, for example by adding `__hwReset()` to the `execUserReset()` macro.

For instructions on how to install a macro file in C-SPY, see *Registering and executing using setup macros and setup files*, page 159. For details about the macro functions used, see the chapter *C-SPY® macros reference*. 
Executing your application

The IAR C-SPY® Debugger provides a flexible range of facilities for executing your application during debugging. This chapter contains information about:

- The conceptual differences between source mode and disassembly mode debugging
- Executing your application
- The call stack
- Handling terminal input and output.

Source and disassembly mode debugging

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

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

Disassembly mode debugging lets you focus on the critical sections of your application, and provides you with precise control over the hardware. 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 instruction at a time. In Mixed-Mode display, the debugger also displays the corresponding C/C++ source code interleaved with the disassembly listing.

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

For an example of a debug session both in C source mode and disassembly mode, see Debugging the application, page 45.

Executing

C-SPY provides a flexible range of features for executing your application. You can find commands for executing on the Debug menu and on the toolbar.
STEP

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

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

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

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, f(n-1):

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

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 f(n-2) function call, which is not a statement on its own but part of the same statement as f(n-1). Thus,
you can skip uninteresting calls which are parts of statements and instead focus on critical parts:

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

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

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

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
int f(int n)
{
    value = f(n-1) + f(n-2) + f(n-3);
    return value;
}
...
    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 Embedded 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.
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 window is currently placed over the other window.

![Figure 47: C-SPY highlighting source location]

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

Occasionally, you will notice that a statement in the source window is highlighted using a pale variant of the normal highlight color. This happens when the program counter is at an assembly 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.

USING BREAKPOINTS TO STOP

You can set breakpoints in the application to stop at locations of particular interest. These locations can be either at code sections where you want to investigate whether your program logic is correct, or at data accesses to investigate when and how the data
is changed. Depending on which debugger system you are using you might also have access to additional types of breakpoints. For instance, if you are using the C-SPY Simulator, a special kind of breakpoint facilitates simulation of simple hardware devices. See the chapter Simulator-specific debugging for further details.

For a more advanced simulation, you can stop under certain conditions, which you specify. You can also connect a C-SPY macro to the breakpoint. The macro can be defined to perform actions, which for instance can simulate specific hardware behavior.

All these possibilities provide you with a flexible tool for investigating the status of, for example, variables and registers at different stages during the application execution.

For detailed information about the breakpoint system and how to use the breakpoint types, see the chapter Using breakpoints.

**USING THE BREAK BUTTON TO STOP**

While your application is executing, the Break button on the debug toolbar is highlighted in red. To stop the execution, click the Break button or choose the Debug>Break command.

**STOP AT PROGRAM EXIT**

Typically, the execution of an embedded application is not intended to end, which means that the application will not make use of a traditional exit. However, in some situations a controlled exit is necessary, such as during debug sessions. You can link your application with a special library that contains an exit label. A breakpoint will be automatically set on that label to stop execution when it gets there. Before you start C-SPY, choose Project>Options, and select the Linker category. On the Output page, select the General Options category. On the Library Configuration page, select the option Semihosted.

**Call stack information**

The compiler generates extensive backtrace information. This allows C-SPY to show, without any runtime penalty, the complete 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 incorrect values in parameters, thus locating the function in the call chain where the problem occurred.

The Call Stack window—available from the View menu—shows a list of function calls, with the current function at the top. When you inspect a function in the call chain, by double-clicking on any function call frame, 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 (---). For reference information about the Call Stack window, see Call Stack window, page 425.

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—available on the Debug menu, or alternatively on the context menu—to execute to that function.

Assembler source code does not automatically contain any backtrace information. To be able to see the call chain also for your assembler modules, you can add the appropriate CFI assembler directives to the source code. For further information, see the ARM® 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—available on the View menu—lets you enter input to your application, and display output from it.

This facility is useful in two different contexts:

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

To use this window, you must build your application with the option Semihosted or the IAR breakpoint option. C-SPY will then direct stdin, stdout, and stderr to this window.

For reference information, see Terminal I/O window, page 426.

Directing stdin and stdout to a file

You can also direct stdin and stdout directly to a file. You can then open the file in another tool, for instance an editor, to navigate and search within the file for particularly interesting parts. The Terminal I/O Log Files dialog box—available by choosing Debug>Logging—allows you to select a destination log file, and to log terminal I/O input and output from C-SPY to this file.

For reference information, see Terminal I/O Log File dialog box, page 442.
Working with variables and expressions

This chapter defines the variables and expressions used in C-SPY®. It also demonstrates the methods for examining variables and expressions.

C-SPY expressions

C-SPY lets you examine the C variables, C expressions, and assembler symbols that you have defined in your application code. In addition, C-SPY allows you to define C-SPY macro variables and macro functions and use them when evaluating expressions. Expressions that are built with these components are called C-SPY expressions and there are several methods for monitoring these in C-SPY.

C-SPY expressions can include any type of C expression, except function calls. 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

Examples of valid C-SPY expressions are:

```c
i + j
i = 42
#asm_label
#R2
#PC
my_macro_func(19)
```

C SYMBOLS

C symbols are symbols that you have defined in the C source code of your application, for instance variables, constants, and functions. C symbols can be referenced by their names.
Using sizeof

According to the ISO/ANSI C standard, there are two syntactical forms of `sizeof`:

\[
\begin{align*}
\text{sizeof (type)} \\
\text{sizeof expr}
\end{align*}
\]

The former is for types and the latter for expressions.

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)`.

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 Device description file, page 125.

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

<table>
<thead>
<tr>
<th>Example</th>
<th>What it does</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>#pc++</code></td>
<td>Increments the value of the program counter.</td>
</tr>
<tr>
<td><code>myptr = #label7</code></td>
<td>Sets <code>myptr</code> to the integral address of <code>label7</code> within its zone.</td>
</tr>
</tbody>
</table>

Table 15: C-SPY assembler symbols expressions

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

<table>
<thead>
<tr>
<th>Example</th>
<th>What it does</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>#pc</code></td>
<td>Refers to the program counter.</td>
</tr>
<tr>
<td><code>#pc</code></td>
<td>Refers to the assembler label <code>pc</code>.</td>
</tr>
</tbody>
</table>

Table 16: Handling name conflicts between hardware registers and assembler labels

Which processor-specific symbols are available by default can be seen in the Register window, using the CPU Registers register group. See Register groups, page 152.

MACRO FUNCTIONS

Macro functions consist of C-SPY 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 The macro language, page 156.
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 The macro language, page 527.

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. Depending on your project settings, a high level of optimization results in smaller or faster code, but also in increased compile time. 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
foo()
{
  int i = 42;
  ...
  x = bar(i); //Not until 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 C-SPY will not be able to display the value until it is actually used. If you try to view a 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 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 pointer. The value is displayed next to the variable.
- **The Auto window**—available from the View menu—automatically displays a useful selection of variables and expressions in, or near, the current statement.
- **The Locals window**—available from the View menu—automatically displays the local variables, that is, auto variables and function parameters for the active function.
- **The Watch window**—available from the View menu—allows you to monitor the values of C-SPY expressions and variables.
- **The Live Watch window**—available from the View menu—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.
- **The Statics window**—available from the View menu—automatically displays the values of variables with static storage duration.
- **The Quick Watch window**, see Using the Quick Watch window, page 139.
- **The Trace system**, see Using trace, page 169.

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

For reference information about the windows, see C-SPY windows, page 403.

**WORKING WITH THE WINDOWS**

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

A context menu containing useful commands is available in all windows if you right-click in each window. Convenient drag-and-drop between windows is supported, except for in the Locals window and the Quick Watch window where it is not applicable.
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 in the Value field and modify its content. To remove an expression, select it and press the Delete key.

**Using the Quick Watch window**

The Quick Watch window—available from the View menu—lets you watch the value of a variable or expression and evaluate expressions.

The Quick Watch window is different from the Watch window in the following ways:

- The Quick Watch window offers a fast method for inspecting and evaluating expressions. Right-click on the expression you want to examine and choose Quick Watch from the context menu that appears. The expression will automatically appear in the Quick Watch window.
- 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 side effects, such as assignments and C-SPY macro functions, it allows you to perform evaluations under controlled conditions.

**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:

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.
Using breakpoints

This chapter describes the breakpoint system and various ways to create and monitor breakpoints. The chapter also gives some useful breakpoint tips and information about breakpoint consumers.

The breakpoint system

The C-SPY® breakpoint system lets you set various kinds 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. If you are using the simulator driver you can also set immediate breakpoints. C-SPY also provides several ways of defining the breakpoints.

All your breakpoints are listed in the Breakpoints window where you can conveniently monitor, enable, and disable them.

You can let the execution stop only under certain conditions, which you specify. You can also let the breakpoint trigger a side effect, for instance executing a C-SPY macro function, without stopping the execution. The macro function can be defined to perform a wide variety of actions, for instance, simulating hardware behavior.

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.

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

Defining breakpoints

You can set breakpoints in many various ways and 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.

Breakpoints are set with a higher precision than single lines, using the same mechanism as when stepping; for more details about the precision, see Step, page 130.
Defining breakpoints

For reference information about code and log breakpoints, see Code breakpoints dialog box, page 341 and Log breakpoints dialog box, page 343, respectively. For details about any additional breakpoint types, see the driver-specific documentation.

BREAKPOINT ICONS

A breakpoint is marked with an icon in the left margin of the editor window, and the icon is different for code and for log breakpoints:

If the breakpoint icon does not appear, make sure the option Show bookmarks is selected, see Editor options, page 377.

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. For more information about breakpoint icons, see the driver-specific documentation.

DIFFERENT 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
Using breakpoints

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 allow 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.

DEFINING BREAKPOINTS USING THE DIALOG BOX

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

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

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

To define a new breakpoint

1. Choose View->Breakpoints to open the Breakpoints window.
2. In the Breakpoints window, right-click to open the context menu.
3. On the context menu, choose New Breakpoint.
4. On the submenu, choose the breakpoint type you want to set. Depending on the C-SPY driver you are using, different breakpoint types might be available.
A breakpoint dialog box appears. Specify the breakpoint settings and click **OK**. The breakpoint is displayed in the Breakpoints window, see *Viewing all breakpoints*, page 146.

**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.

A breakpoint dialog box appears. Specify the breakpoint settings and click **OK**. The breakpoint is displayed in the Breakpoints window, see *Viewing all breakpoints*, page 146.

**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; you can see, edit, and remove it in 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.

Setting different types of breakpoints in the Memory window is only supported if the driver you use supports these types of breakpoints.

**DEFINING BREAKPOINTS USING SYSTEM MACROS**

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

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:

- __setCodeBreak
- __setDataBreak
- __setSimBreak
- __clearBreak

For details of each breakpoint macro, see the chapter *C-SPY® macros reference*.

**Defining 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 159.

**USEFUL BREAKPOINT TIPS**

Below are some useful tips related to setting breakpoints.

**Tracing incorrect function arguments**

If a function with a pointer argument is sometimes incorrectly called with a NULL argument, it is useful to put a breakpoint on the first line of the function with a condition
that is true only when the parameter is 0. The breakpoint will then not be triggered until the problematic situation actually occurs.

**Performing a task with or without stopping execution**

You can perform a task when a breakpoint is triggered with or without stopping the 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.

If you instead want to perform a task without stopping the execution, you can set a condition which returns 0 (false). When the breakpoint is triggered, the condition 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.

---

**Viewing all breakpoints**

To view breakpoints, you can use the Breakpoints window and the **Breakpoints Usage** dialog box.

For information about the Breakpoints window, see *Breakpoints window*, page 340.
USING THE BREAKPOINT USAGE DIALOG BOX

The Breakpoint Usage dialog box—available from C-SPY driver-specific menus, for example the Simulator menu—lists all active breakpoints.

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. For each breakpoint in the list, the address and access type are shown. Each breakpoint can also be expanded to show its originator. The format of the items in this dialog box depends on which 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 shown in the breakpoint dialog box.

Exceeding the number of available low-level 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 breakpoints, you can use the Breakpoint Usage dialog box for:

- Identifying all consumers of breakpoints
- 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.

For information about the number of available breakpoints in the debugger system you are using and how to use the available breakpoints in a better way, see the section about breakpoints in the part of this book that corresponds to the debugger system you are using.
BREAKPOINT CONSUMERS

A debugger system includes several consumers of breakpoints.

User breakpoints—the breakpoints you define in the breakpoint dialog box or by toggling breakpoints in the editor window—often consume one low-level breakpoint each, but this can vary greatly. Some user breakpoints consume several low-level breakpoints and conversely, several user breakpoints can share one low-level 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 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 Breakpoint Usage window.
- the Semihosted or the IAR breakpoint option 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.

C-SPY plugin modules, for example modules for real-time operating systems, can consume additional breakpoints. Specifically, by default, the Stack window consumes a breakpoint. To disable the breakpoint used by the Stack window:

- Choose Tools>Options>Stack.
- Deselect the Stack pointer(s) not valid until program reaches: label option.
Monitoring memory and registers

This chapter describes how to use the features available in the IAR C-SPY® Debugger for examining memory and registers.

Memory addressing

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. The ARM architecture has only one zone, *Memory*, which covers the whole ARM memory range.

![Default zone Memory](image)

*Figure 52: 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.

By using different memory zones, you can control the access width used for reading and writing in, for example, the Memory window. For normal memory, the default zone *Memory* can be used, but certain I/O registers may require to be accessed as 8, 16, or 32 bits to give correct results.
Windows for 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. For more information, see Memory window, page 410. See also Setting a data breakpoint in the Memory window, page 144.

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

- **The Stack window**
  Displays the contents of the stack, including how stack variables are laid out in memory. In addition, some integrity checks of the stack can be performed to detect and warn about problems with stack overflow. For example, the Stack window is useful for determining the optimal size of the stack.

- **The Register window**
  Gives an up-to-date display of the contents of the processor registers and SFRs, and allows you to edit them.

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.

**USING THE STACK WINDOW**

Before you can open the Stack window you must make sure it is enabled; Choose Project>Options>Debugger>Plugins and select Stack from the list of plugins. In C-SPY, you can then open a Stack window by choosing View>Stack. You can open
several instances of the Stack window, each showing a different stack—if several stacks are available—or the same stack with different display settings.

![Stack window diagram]

For detailed reference information about the Stack window, and the method used for computing the stack usage and its limitations, see Stack window, page 433. For reference information about the options specific to the window, see Stack options, page 390.

Place the mouse pointer over the stack bar to get tool tip information about stack usage.

**Detecting stack overflows**

If you have selected the option **Enable stack checks**, available by choosing Tools>Options>Stack, you have also enabled the functionality needed to detect stack overflows. This means that C-SPY can issue warnings for stack overflow when the application stops executing. Warnings are issued either when the stack usage exceeds a threshold that you can specify, or when the stack pointer is outside the stack memory range.

**Viewing the stack contents**

The display area of the Stack window shows the contents of the stack, which can be useful in many contexts. Some examples are:

- Investigating the stack usage when assembler modules are called from C modules and vice versa
- Investigating whether the correct elements are located on the stack
- Investigating whether the stack is restored properly.
WORKING WITH REGISTERS

The Register window gives an up-to-date display of the contents of the processor registers and special function registers, and allows you to edit them.

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. You can expand some registers to show individual bits or subgroups of bits.

To change the display format, change the Base setting on the Register Filter page—available by choosing Tools>Options.

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.

Register groups

Due to the large amount of registers—memory-mapped peripheral unit registers and CPU registers—it is inconvenient to show all registers concurrently in the Register window. Instead you can divide registers into register groups. By default, there are two register groups in the debugger: Current CPU Registers and CPU Registers. The Current CPU Registers group contains the registers that are available in the current processor mode. The CPU Registers group contains both the current registers and their banked counterparts available in other processor modes.

In addition to the CPU Registers, additional register groups are predefined in the device description files—available in the arm\config directory—that make all SFR registers available in the register window. The device description file contains a section that defines the special function registers and their groups.
You can select which register group to display in the Register window using the drop-down list. You can conveniently keep track of different register groups simultaneously, as you can open several instances of the Register window.

**Enabling predefined register groups**

To use any of the predefined register groups, select a device description file that suits your device, see *Selecting a device description file*, page 121.

The available register groups are listed on the Register Filter page, available if you choose the **Tools>Options** command when C-SPY is running.

**Defining application-specific groups**

In addition to the predefined register groups, you can create your own register groups that better suit the use of registers in your application.

To define new register groups, choose **Tools>Options** and click the Register Filter tab. This page is only available when the debugger is running.

For reference information about this dialog box, see *Register Filter options*, page 392.
Windows for monitoring memory and registers
Using the C-SPY® macro system

C-SPY includes a comprehensive macro system which allows you to automate the debugging process and to simulate peripheral devices. Macros can be used in conjunction with complex breakpoints and interrupt simulation to perform a wide variety of tasks.

This chapter describes the macro system, its features, for what purpose these features can be used, and how to use them.

The macro system

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.
- Developing small debug utility functions, for instance calculating the stack depth, see the provided example stack.mac located in the directory \arm\src\sim.
- Simulating peripheral devices, see the chapter Simulating interrupts. This only applies if you are using the simulator driver.

The macro system has several features:

- The similarity between the macro language and the C language, which lets you write your own macro functions.
- Predefined system macros which perform useful tasks such as opening and closing files, setting breakpoints, and defining simulated interrupts.
- Reserved setup macro functions which can be used for defining at which stage the macro function should be executed. You define the function yourself, in a setup macro file.
- The option of collecting your macro functions in one or several macro files.
- A dialog box where you can view, register, and edit your macro functions and files. Alternatively, you can register and execute your macro files and functions using either the setup functionality or system macros.
Many C-SPY tasks can be performed either in a dialog box or by using macro functions. The advantage of using a dialog box is that it provides you with a graphical interface where you can interactively fine-tune the characteristics of the task you want to perform, for instance setting a breakpoint. You can add parameters and quickly test whether the breakpoint works according to your intentions.

Macros, on the other hand, are useful when you already have specified your breakpoints so that they fully meet your requirements. To set up your simulator environment automatically, write a macro file and execute it, for instance, when you start C-SPY. Another advantage is that the debug session will be documented, and if several engineers are involved in the development project, you can share the macro files within the group.

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. You can define macro functions, with or without parameters and return values. You can use built-in system macros, similar to C library functions. Finally, you can define global and local macro variables. For a detailed description of the macro language components, see The macro language, page 527.

Example

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

```c
CheckLatest(value)
{
    oldvalue;
    if (oldvalue != value)
    {
        __message "Message: Changed from ", oldvalue, " to ", value;
        oldvalue = value;
    }
}
```

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

THE MACRO FILE

You collect your macro variables and functions in one or several macro files. To define a macro variable or macro function, first create a text file containing the definition. You can use any suitable text editor, such as the editor supplied with the IDE. Save the file with a suitable name using the filename extension `.mac`.

Setup macro file
You can load a macro file at C-SPY startup; such a file is called a setup macro file. This is especially convenient if you want to make C-SPY perform actions before you load your application software, for instance to initialize some CPU registers or memory-mapped peripheral units. Other reasons might be if you want to automate the initialization of C-SPY, or if you want to register multiple setup macro files. You will find an example of a C-SPY setup macro file, SetupSimple.mac, in the arm\tutor directory.

For information about how to load a setup macro file, see Registering and executing using setup macros and setup files, page 159. For an example of how to use setup macro files, see the chapter Simulating an interrupt in Part 2. Tutorials.

SETUP MACRO FUNCTIONS
The setup macro functions are reserved macro function names that are called by C-SPY at specific stages during execution. The stages to choose between are:

- 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 the name of a setup macro function. For instance, if you want to clear a specific memory area before you load your application software, the macro setup function execUserPreload is suitable. This function is also suitable if you want to initialize some CPU registers or memory-mapped peripheral units before you load your application software. For detailed information about each setup macro function, see Setup macro functions summary, page 532.

As with any macro function, you collect your setup macro functions in a macro file. Because many of the setup macro functions execute before main is reached, you should define these functions in a setup macro file.

Remapping memory
A common feature of many ARM-based processors is the ability to remap memory. After a reset, the memory controller typically maps address zero to non-volatile memory, such as flash. By configuring the memory controller, the system memory can be remapped to place RAM at zero and non-volatile memory higher up in the address map. By doing this, the exception table will reside in RAM and can be easily modified when you download code to the evaluation board. To handle this in C-SPY, the setup
Using C-SPY macros

If you decide to use C-SPY macros, you must first create a macro file in which you define your macro functions. C-SPY must know that you intend to use your defined macro functions, and thus you must register (load) your macro file. During the debug session, you might have to list all available macro functions and execute them.

To list the registered macro functions, you can use the Macro Configuration dialog box. There are various ways to both register and execute macro functions:

- You can register a macro interactively in the Macro Configuration dialog box.
- You can register and execute macro functions at the C-SPY startup sequence by defining setup macro functions in a setup macro file.
- 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 555.
- The Quick Watch window lets you evaluate expressions, and can thus be used for executing macro functions.
- A macro can be connected to a breakpoint; when the breakpoint is triggered the macro is executed.

USING THE MACRO CONFIGURATION DIALOG BOX

The Macro Configuration dialog box—available by choosing Debug>Macros—lets you 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.

REGISTERING AND EXECUTING USING SETUP MACROS AND SETUP FILES

It can be convenient to register a macro file during the C-SPY startup sequence, especially if you have several ready-made macro functions. C-SPY can then execute the macros before main is reached. 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 C-SPY.

If you use the setup macro function names to define the macro functions, you can define exactly at which stage you want the macro function to be executed.
Follow these steps:

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 macro files MyMacroUtils.mac and MyDeviceSimulation.mac. Because the macro function is defined with the execUserSetup function name, it will be executed directly after your application has been downloaded.
   
2. Save the file using the filename extension .mac.
   
3. Before you start C-SPY, choose Project>Options and click the Setup tab in the Debugger category. Select the check box Use Setup file and choose the macro file you just created.

   The interrupt macro will now be loaded during the C-SPY startup sequence.

EXECUTING MACROS USING QUICK WATCH

The Quick Watch window—available from the View menu—lets you watch the value of any variables or expressions and evaluate them. For macros, the Quick Watch window is especially useful because it is a method which lets you dynamically choose when to execute a macro function.

Consider this simple macro function which checks the status of a watchdog timer interrupt enable bit:

```c
WDTstatus()
{
   if (#WD_SR & 0x01 != 0) /* Checks the status of WDOVF */
      return "Watchdog triggered"; /* C-SPY macro string used */
   else
      return "Watchdog not triggered"; /* C-SPY macro string used*/
}
```

1. Save the macro function using the filename extension .mac. Keep the file open.

2. To register the macro file, choose Debug>Macros. The Macro Configuration dialog box appears. Locate the file, click Add and then Register. The macro function appears in the list of registered macros.
3. In the macro file editor window, select the macro function name `WDTstatus`. Right-click, and choose `Quick Watch` from the context menu that appears.

![Quick Watch window](image)

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

Click `Close` to close the window.

**EXECUTING A MACRO BY CONNECTING IT TO A BREAKPOINT**

You can connect a macro to a breakpoint. The macro will then be executed at the time 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 changes. 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.

For an example of how to create a log macro and connect it to a breakpoint, follow these steps:

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.

   Save the macro function in a macro file, with the filename extension `.mac`. 
3 Before you can execute the macro it must be registered. Open the Macro Configuration dialog box—available by choosing Debug>Macros—and add your macro file to the list Selected Macro Files. Click Register and your macro function will appear in the list Registered Macros. Close the dialog box.

4 Next, you should toggle a code breakpoint—using the Toggle Breakpoint button—on the first statement within the function fact in your application source code. Open the Breakpoint dialog box—available by choosing Edit>Breakpoints—your breakpoint will appear in the list of breakpoints at the bottom of the dialog box. Select the breakpoint.

5 Connect the log macro function to the breakpoint by typing the name of the macro function, logfact(), in the Action field and clicking Apply. Close the dialog box.

6 Now you can execute your application source code. When the breakpoint has been triggered, the macro function will be executed. You can see the result in the Log window.

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 530.

For a complete example where a serial port input buffer is simulated using the method of connecting a macro to a breakpoint, see the chapter Simulating an interrupt in Part 2. Tutorials.
Analyzing your application

It is important to locate an application's bottle-necks and to verify that all parts of an application have been tested. This chapter presents facilities available in the IAR C-SPY® Debugger for analyzing your application so that you can efficiently spend time and effort on optimizations.

Code coverage and profiling are not supported by all C-SPY drivers. For information about the driver you are using, see the driver-specific documentation. Code coverage and profiling are supported by the C-SPY Simulator.

Note that the profiler described in this chapter is a C-SPY plugin module and should not be mixed up with the profiler that is part of the C-SPY driver, see Using the profiler, page 303.

Function-level profiling

The profiler will help you find the functions where most time is spent during execution, for a given stimulus. Those functions are the parts you should focus on when spending time and effort on optimizing your code. A simple method of optimizing a function is to compile it using speed optimization. Alternatively, you can move the function into the memory which uses the most efficient addressing mode. For detailed information about efficient memory usage, see the IAR C/C++ Development Guide for ARM®.

The Profiling window displays profiling information, 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 active until it is turned off.

The profiler measures the time between the entry and return of a function. This means that time consumed in a function is not added until the function returns or another function is called. You will only notice this if you are stepping into a function.

For reference information about the Profiling window, see Profiling window, page 429.
USING THE PROFILER

Before you can use the Profiler window, 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>Output&gt;Include debug information in output</td>
</tr>
<tr>
<td>Debugger</td>
<td>Plugins&gt;Profiling</td>
</tr>
</tbody>
</table>

Table 17: Project options for enabling profiling

1. After you have built your application and started C-SPY, choose View>Profiling to open the window, and click the Activate button to turn on the profiler.
2. Click the Clear button, alternatively use the context menu available when you right-click in the window, when you want to start a new sampling.
3. Start the execution. When the execution stops, for instance because the program exit is reached or a breakpoint is triggered, click the Refresh button.

Profiling information is displayed in the window.

Viewing the figures

Clicking on a column header sorts the entire list according to that column.

A dimmed item in the list indicates that the function has been called by a function which does not contain source code (compiled without debug information). When a function is called by functions that do not have their source code available, such as library functions, no measurement in time is made.

There is always an item in the list called Outside main. This is time that cannot be placed in any of the functions in the list. That is, code compiled without debug information, for instance, all startup and exit code, and C/C++ library code.
Clicking the **Graph** button toggles the percentage columns to be displayed either as numbers or as bar charts.

![Graphs in Profiling window](image)

**Figure 59: Graphs in Profiling window**

Clicking the **Show details** button displays more detailed information about the function selected in the list. A window is opened showing information about callers and callees for the selected function:

![Function details window](image)

**Figure 60: Function details window**

**Producing reports**

To produce a report, right-click in the window and choose the **Save As** command on the context menu. The contents of the Profiling window are saved to a file.
The code coverage functionality helps you verify whether all parts of your code have been executed. This is useful when you design your test procedure to make sure that all parts of the code have been executed. It also helps you identify parts of your code that are not reachable.

**USING CODE COVERAGE**

The Code Coverage window—available from the View menu—reports the status of the current code coverage analysis, that is, what parts of the code have been executed at least once since the start of the analysis. The compiler generates detailed stepping information in the form of step points at each statement, and at each function call. The report includes information about all modules and functions. It reports the amount of all step points, in percentage, that have been executed and lists all step points that have not been executed up to the point where the application has been stopped. The coverage will continue until turned off.

For reference information about the Code Coverage window, see *Code Coverage window*, page 427.

Before using the Code Coverage window 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>Output&gt;Include debug information in output</td>
</tr>
<tr>
<td>Debugger</td>
<td>Plugins&gt;Code Coverage</td>
</tr>
</tbody>
</table>

*Table 18: 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. This window is displayed:

```
Figure 61: Code Coverage window
```

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

2. 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.

### Viewing the figures

The code coverage information is displayed in a tree structure, showing the program, module, function and step point levels. The plus sign and minus sign icons allow you to expand and collapse the structure.

These icons are used to give you an overview of the current status on all levels:

- A red diamond signifies that 0% of the code has been executed
- A green diamond signifies that 100% of the code has been executed
- A red and green diamond signifies that some of the code has been executed
- A yellow diamond signifies a step point that has not been executed.

The percentage displayed at the end of every program, module and function line shows the amount of code that has been covered so far, that is, the number of executed step points divided with the total number of step points.

For step point lines, the information displayed is the column number range and the row number of the step point in the source window, followed by the address of the step point.

```
<column start>-<column end>:row
```
A step point is considered to be executed when one of its instructions has been executed. When a step point has been executed, it is removed from the window.

Double-clicking a step point or a function in the Code Coverage window displays that step point 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.

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.

**What parts of the code are displayed?**

The window displays only statements that were compiled with debug information. Thus, startup code, exit code and library code are 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.

**Producing reports**

To produce a report, right-click in the window and choose the Save As command on the context menu. The contents of the Code Coverage window are saved to a file.
Using trace

This chapter gives you information about collecting and using trace data, reference information for trace-related windows and dialog boxes.

Collecting and using trace data

Before you start using the trace-related functionality in C-SPY, read this section to get a good understanding of how to take advantage of the trace data for debugging.

More specifically, these topics are covered:

- Reasons for using trace, page 169
- Briefly about trace, page 170
- Requirements for using trace, page 171
- Getting started with trace in the C-SPY simulator, page 172
- Getting started with ETM trace, page 172
- Getting started with SWO trace, page 173
- Setting up concurrent use of ETM and SWO, page 173
- Trace data collection using breakpoints, page 173
- Searching in trace data, page 174
- Browsing through trace data, page 174.

When you need detailed information about a specific dialog box or window, see Trace-related reference information, page 175.

For related information, see also:

- Using J-Link trace triggers and trace filters, page 285
- Using the data and interrupt logging systems, page 294
- Using the profiler, page 303.

REASONS FOR USING TRACE

By using trace, you can inspect the program flow up to a specific state, for instance an application crash, and use the trace data to locate the origin of the problem. Trace data can be useful for locating programming errors that have irregular symptoms and occur sporadically.
BRIEFLY ABOUT TRACE

Your target system must be able to generate trace data. Once generated, C-SPY can collect it and you can visualize and analyze the data in various windows and dialog boxes.

C-SPY supports collecting trace data from these target systems:

- Devices with support for ETM (Embedded Trace Macrocell)—ETM trace
- Devices with support for the SWD (Serial Wire Debug) interface using the SWO (Serial Wire Output) communication channel—SWO trace
- The C-SPY simulator.

Depending on your target system, different types of trace data can be generated.

ETM trace

ETM trace (also known as full trace) is a continuously collected sequence of every executed instruction for a selected portion of the execution. Usually, it is not possible to collect very long sequences of data in real time without saturating the communication channel that transmits the data to C-SPY.

The debug probe contains a trace buffer that collects trace data in real time, but the data is not displayed in the C-SPY windows until after the execution has stopped.

SWO trace

SWO trace is a sequence of events of various kinds, generated by the on-chip debug hardware. The events are transmitted in real time from the target system over the SWO communication channel. This means that the C-SPY windows are continuously updated while the target system is executing. The most important events are:

- PC sampling
  The hardware can sample and transmit the value of the program counter at regular intervals. This is not a continuous sequence of executed instructions (like ETM trace), but a sparse regular sampling of the PC. A modern ARM CPU typically executes millions of instructions per second, while the PC sampling rate is usually counted in thousands per second.

- Interrupt logs
  The hardware can generate and transmit data related to the execution of interrupts, generating events when entering and leaving an interrupt handler routine.

- Data logs
  Using Data Log breakpoints, the hardware can be configured to generate and transmit events whenever a certain variable, or simply an address range, is accessed by the CPU.
The SWO channel does not have unlimited throughput, so it is usually not possible to use all the above features at the same time, at least not if either the frequency of PC sampling, of interrupts, or of accesses to the designated variables is high.

**Trace-related features in C-SPY**

In C-SPY, you can use the trace-related windows Trace, Function Trace, Timeline, and Find in Trace. In the C-SPY simulator, you can also use the Trace Expressions window. Depending on your C-SPY driver, you can set various types of trace breakpoints and triggers to control the collection of trace data.

In addition, several other features in C-SPY also use trace data, features such as the Profiler, Code coverage, and Instruction profiling.

If you use the C-SPY J-Link/J-Trace driver, you have access to windows such as the Interrupt Log, Interrupt Log Summary, Data Log, and Data Log Summary windows.

When you are debugging, two buttons labeled **ETM** and **SWO**, respectively, are visible on the IDE main window toolbar. If any of these buttons is green, it means that the corresponding trace hardware is generating trace data. Just point at the button with the mouse pointer to get detailed tooltip information about which C-SPY features that have requested trace data generation. This is useful, for example, if your SWO communication channel often overflows because too many of the C-SPY features are currently using trace data.

**REQUIREMENTS FOR USING TRACE**

To use trace-related functionality in C-SPY, you need debug components (hardware, a debug probe, and a C-SPY driver) that all support trace. Alternatively, you can use the trace features provided by the C-SPY simulator.

**Note:** The specific set of debug components you are using determine which of the trace features in C-SPY that are supported.

**Requirements for using ETM trace**

ETM trace is available for some ARM devices.

To use ETM trace you need one of these combinations:

- A J-Trace debug probe and a device that supports ETM. Make sure to use the C-SPY J-Link/J-Trace driver.
- A J-Link debug probe and a device that supports ETM with ETB (Embedded Trace Buffer). Make sure to use the C-SPY J-Link/J-Trace driver.
- The C-SPY RDI driver, and a debug probe and a device that both support ETM.
Collecting and using trace data

Requirements for using SWO trace
To use SWO trace you need a J-Link or J-Trace debug probe that supports the SWO communication channel and a device that supports the SWD/SWO interface.

GETTING STARTED WITH TRACE IN THE C-SPY SIMULATOR
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 183.

GETTING STARTED WITH ETM TRACE
To set up ETM trace, follow these steps:

1 Before you start C-SPY:
   - For the C-SPY RDI driver, choose Project>Options>RDI>ETM trace
   - For J-Trace no specific settings are required before starting C-SPY
   - The trace port must be set up. For some devices this is done automatically when the trace logic is enabled. However, for some devices, typically Atmel and ST devices based on ARM 7 or ARM 9, you need to set up the trace port explicitly. You do this by means of a C-SPY macro file. You can find examples of such files (ETM_init*.mac) in the example projects. To use a macro file, choose Project>Options>Debugger>Setup>Use macro files(s). Specify your macro file; a browse button is available for your convenience.
   
   Note that the pins used on the hardware for the trace signals cannot be used by your application.

2 After you have started C-SPY, choose ETM Trace Settings from the driver-specific menu on the menu bar. In the ETM Trace Settings dialog box that appears, check if you need to change any of the default settings. For details, see ETM Trace Settings dialog box, page 176.

3 Open the Trace window—available from the driver-specific menu—and click the Activate button to enable trace data collection.

4 Start the execution. When the execution stops, for instance because a breakpoint is triggered, trace data is displayed in the Trace window. For more information about the window, see Trace window, page 183.
GETTING STARTED WITH SWO TRACE

To set up SWO trace, follow these steps:

1. Before you start C-SPY, choose Project>Options>J-Link/J-Trace and click the Connection tab. Choose Interface>SWD.

2. After you have started C-SPY, choose SWO Trace Windows Settings from the J-Link menu. In the SWO Trace Windows Settings dialog box that appears, make your settings for controlling the output in the Trace window. For details, see SWO Trace Window Settings dialog box, page 178.

3. To configure the hardware’s generation of trace data, click the SWO Configuration button available in the SWO Configuration dialog box. For details, see SWO Configuration dialog box, page 180.

   Note specifically these settings:
   ● The value of the CPU clock option must reflect the frequency of the CPU clock speed at which the application executes. Note also that the settings you make are preserved between debug sessions.
   ● To decrease the amount of transmissions on the communication channel, you can disable the Timestamp option. Alternatively, set a lower rate for PC Sampling or use a higher SWO clock frequency.

4. Open the Trace window—available from the J-Link/J-Trace menu—and click the Activate button to enable trace data collection.

5. Start the execution. The Trace window is continuously updated with trace data. For more information about this window, see Trace window, page 183.

SETTING UP CONCURRENT USE OF ETM AND SWO

If you have a J-Trace debug probe for Cortex-M3, you can use ETM trace and SWO trace concurrently.

In this case, if you activate the ETM trace and the SWO trace, SWO trace data will also be collected in the ETM trace buffer, instead of being streamed via the SWO channel. This means that the SWO trace data will not be displayed until the execution has stopped, instead of being continuously updated live in the Trace window.

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. 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. Alternatively, in the Breakpoints window, choose...
**Collecting and using trace data**

**Trace Start, Trace Stop, or Trace Filter.** In the C-SPY simulator, the C-SPY system macros __setTraceStartBreak and __setTraceStopBreak can also be used.

For details about these breakpoints, see *Trace Start breakpoints dialog box*, page 193 and *Trace Stop breakpoints dialog box*, page 194, 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 select 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 197.

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 198.

**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.

---

**Trace-related reference information**

To use the trace system, you might need reference information about these windows and dialog boxes:

- ETM Trace Settings dialog box, page 176
- SWO Trace Window Settings dialog box, page 178
- SWO Configuration dialog box, page 180
- Trace window, page 183 (ETM Trace, SWO Trace, and Trace for the C-SPY simulator)
- Function Trace window, page 188
- Timeline window, page 189
- Trace Start breakpoints dialog box, page 193 (for the C-SPY simulator only)
- Trace Start breakpoints dialog box, page 287 (for the C-SPY J-Link/J-Trace driver only)
- Trace Stop breakpoints dialog box, page 194 (for the C-SPY simulator only)
- Trace Stop breakpoints dialog box, page 289 (for the C-SPY J-Link/J-Trace driver only)
- Trace Filter breakpoints dialog box, page 292 (for the C-SPY J-Link/J-Trace driver only)
- Trace Expressions window, page 195 (for the C-SPY simulator only)
- Find in Trace window, page 198
- Find in Trace dialog box, page 197.
ETM TRACE SETTINGS DIALOG BOX

The ETM Trace Settings dialog box is available from the driver-specific menu.

Figure 62: ETM Trace Settings dialog box

Note: This dialog box looks slightly different for the RDI drivers.

Using this dialog box

Use the ETM Trace Settings dialog box to configure ETM trace generation and collection.

See also Getting started with ETM trace, page 172.

Trace port width

The trace bus width can be set to 1, 2, 4, 8, or 16 bits. The value must correspond with what is supported by the hardware and the debug probe. For Cortex-M3, 1, 2, and 4 bits are supported by the J-Trace debug probe. For ARM7/9, only 4 bits are supported by the J-Trace debug probe.

Trace port mode

Use these options to set the trace clock rate; choose between:

- Normal, full-rate clocking
- Normal, half-rate clocking
- Multiplexed
- Demultiplexed
- Demultiplexed, half-rate clocking.

Note: For RDI drivers, only the two first alternatives are available. For the J-Trace driver, the available alternatives depend on the device you are using.
Trace buffer size

Use the text box to specify the size of the trace buffer. By default, the number of trace frames is 0x10000. For ARM7/9 the maximum number is 0x100000, and for Cortex-M3 the maximum number is 0x400000.

For ARM7/9, one trace frame corresponds to 2 bytes of the physical J-Trace buffer size. For Cortex-M3, one trace frame corresponds to approximately 1 byte of the buffer size.

Note: The Trace buffer size option is only available for the J-Trace driver.

Cycle accurate tracing

Select this option to emit trace frames synchronous to the processor clock even when no trace data is available. This makes it possible to use the trace data for real-time timing calculations. However, if you select this option, the risk for FIFO buffer overflow increases.

Note: The Cycle accurate tracing option is only available for ARM7/9 devices.

Broadcast all branch addresses

Use this option to make the processor send more detailed address trace information. However, if you select this option, the risk for FIFO buffer overflow increases.

Note: The Broadcast all branch addresses option is only available for ARM7/9 devices. For Cortex, this option is always enabled.

Stall processor on FIFO full

The trace FIFO buffer might in some situations become full—FIFO buffer overflow—which means trace data will be lost. If you use this option, the processor will be stalled in case the FIFO buffer fills up.

Show timestamp

Use this option to make the Trace window display seconds instead of cycles in the Index column. To make this possible you must also specify the appropriate speed for your CPU in the Trace port (CPU core) speed text box.

Note: The Show timestamp option is only available when you use the J-Trace driver with ARM7/9 devices.
SWO TRACE WINDOW SETTINGS DIALOG BOX

The SWO Trace Window Settings dialog box is available from the J-Link menu, alternatively from the SWO Trace window toolbar.

![Image of SWO Trace Window Settings dialog box]

**Using this dialog box**

Use the SWO Trace Window Settings dialog box to specify what you want to display in the Trace window.

Note that you also need to configure the generation of trace data, click SWO Configuration. For more information, see *SWO Configuration dialog box*, page 180.

**Force**

Use this option to enable trace data generation, if it is not already enabled by other features using SWO trace data. The Trace window displays all generated SWO data. Other features in C-SPY, for example Profiling, can also enable SWO trace data generation. If no other feature has enabled the generation, use the **Force** options to generate SWO trace data.

The generated data will be displayed in the Trace window. Choose between:

- **Time Stamps**
  Enables timestamps for various SWO trace packets, that is sent over the SWO communication channel. Use the resolution drop-down list to choose the resolution of the timestamp value. For example, 1 to count every cycle, or 16 to count every 16th cycle. Note that the lowest resolution is only useful if the time between each event packet is long enough. 16 is useful if using a low SWO clock frequency.

- **PC samples**
  Enables sampling the program counter register, PC, at regular intervals. To choose the sampling rate, see *PC Sampling*, page 181.
Interrupt Logs

Generate

Use this option to enable trace data generation for these events. The generated data will be displayed in the Trace window. The value of the counters are displayed in the Comment column in the SWO Trace window. Choose between:

- **CPI** Enables generation of trace data for the CPI counter.
- **EXC** Enables generation of trace data for the EXC counter.
- **SLEEP** Enables generation of trace data for the SLEEP counter.
- **LSU** Enables generation of trace data for the LSU counter.
- **FOLD** Enables generation of trace data for the FOLD counter.

**SWO Configuration button**

Displays the SWO Configuration dialog box where you can configure the hardware's generation of trace data. See SWO Configuration dialog box, page 180.
SWO CONFIGURATION DIALOG BOX

The SWO Configuration dialog box is available from the J-Link menu, alternatively from the SWO Trace Window Settings dialog box.

![SWO Configuration dialog box](image)

*Figure 64: SWO Configuration dialog box*
Using this dialog box

Use the SWO Configuration dialog box to configure the serial-wire output communication channel and the hardware’s generation of trace data.

See also *Getting started with SWO trace*, page 173.

**PC Sampling**

These items are available:

- **In use by**
  Lists the features in C-SPY that can use trace data for PC Sampling. ON indicates features currently using trace data. OFF indicates features currently not using trace data.

- **Rate**
  Use the drop-down list to choose the sampling rate, that is, the number of samples per second. The highest possible sampling rate depends on the SWO clock value and on how much other data that is sent over the SWO communication channel. The higher values in the list will not work if the SWO communication channel is not fast enough to handle that much data.

**Data Log Events**

Use this option to specify what to log when a Data Log breakpoint is triggered. These items are available:

- **In use by**
  Lists the features in C-SPY that can use trace data for Data Log Evens. ON indicates features currently using trace data. OFF indicates features currently not using trace data.

- **PC only**
  Logs the value of the program counter.

- **PC + data value + base addr**
  Logs the value of the program counter, the value of the data object, and its base address.

- **Data value + exact addr**
  Logs the value of the data object and the exact address of the data object that was accessed.
Interrupt Log

Interrupt Log lists the features in C-SPY that can use trace data for Interrupt Logs. ON indicates features currently using trace data. OFF indicates features currently not using trace data.

For more information about interrupt logging, see Using the data and interrupt logging systems, page 294.

CPU clock

Use this option to specify the exact clock frequency used by the internal processor clock, \textit{HCLK}, in MHz. The value can have decimals.

This value is used for configuring the SWO communication speed and for calculating time stamps.

SWO clock

Use this option to specify the clock frequency of the SWO communication channel in kHz. Choose between these options:

- Autodetect: Automatically uses the highest possible frequency that the J-Link debug probe can handle.
- Wanted: Manually selects the frequency to be used. The value can have decimals. Use this option if data packets are lost during transmission.

The clock frequency that is actually used is displayed in the Actual text box.

Timestamps

Use this drop-down list to choose the resolution of the timestamp value. For example, 1 to count every cycle, or 16 to count every 16th cycle. Note that the lowest resolution is only useful if the time between each event packet is long enough.

ITM Stimulus Ports

The ITM Stimulus Ports are used for sending data from your application to the debugger host without stopping the program execution. There are 32 such ports.
Use the check boxes to select which ports you want to redirect and to where:

**Enabled ports**
- Enables the ports to be used. Only enabled ports will actually send any data over the SWO communication channel to the debugger.

**To Terminal I/O window**
- Specifies the ports to use for routing data to the Terminal I/O window.

**To Log File**
- Specifies the ports to use for routing data to a log file. To use a different log file than the default one, use the browse button.

The stdout and stderr of your application can be rerouted via SWO and this means that stdout/stderr will appear in the C-SPY Terminal I/O window. To achieve this, choose **Project>Options>General Options>Library Configuration>Library low-level interface implementation>stdout/stderr>Via SWO**.

This can be disabled if you deselect the port settings in the **Enabled ports** and **To Terminal I/O** options.

**TRACE WINDOW**

The Trace window is available from the driver-specific menu.

![Figure 65: The Trace window in the simulator](image)

**Note:** There are three different Trace windows—ETM Trace, SWO Trace, and just Trace for the C-SPY simulator. The windows look slightly different.
Using this window

The Trace window displays the collected trace data, where the content differs depending on the C-SPY driver you are using and the trace support of your debug probe:

| C-SPY simulator | The window displays a collected sequence of executed machine instructions. In addition, the window can display trace data for expressions. |
| ETM trace | The window displays the sequence of executed instructions—optionally with embedded source—which has been continuously collected during application execution, that is full trace. The data has been collected in the ETM trace buffer. The collected data is displayed after the execution has stopped. For information about the requirements for using ETM trace, see Requirements for using ETM trace, page 171. |
| SWO trace | The window displays all events transmitted on the SWO channel. The data is streamed from the target system, via the SWO communication channel, and continuously updated live in the Trace window. Note that if you use the SWO communication channel on a trace probe, the data will be collected in the trace buffer and displayed after the execution has stopped. For information about the requirements for using SWO trace, see Requirements for using SWO trace, page 172. |

Trace toolbar

The Trace toolbar at the top of the Trace window and in the Function trace window provides these toolbar buttons:

<table>
<thead>
<tr>
<th>Toolbar button</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Enable/Disable</td>
<td>Enables and disables collecting and viewing trace data in this window. This button is not available in the Function trace window.</td>
</tr>
<tr>
<td>Clear trace data</td>
<td>Clears the trace buffer. Both the Trace window and the Function trace window are cleared.</td>
</tr>
<tr>
<td>Toggle Source</td>
<td>Toggles the Trace column between showing only disassembly or disassembly together with the corresponding source code.</td>
</tr>
<tr>
<td>Browse</td>
<td>Toggles browse mode on and off for a selected item in the Trace window. For more information about browse mode, see Browsing through trace data, page 174.</td>
</tr>
</tbody>
</table>

Table 19: Trace toolbar buttons
Using trace

The Trace window contains the following columns for the C-SPY simulator:

<table>
<thead>
<tr>
<th>Trace window column</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>#</td>
<td>A serial number for each row in the trace buffer. Simplifies the navigation within the buffer.</td>
</tr>
<tr>
<td>Cycles</td>
<td>The number of cycles elapsed to this point.</td>
</tr>
<tr>
<td>Trace</td>
<td>The collected sequence of executed machine instructions. Optionally, the corresponding source code can also be displayed.</td>
</tr>
<tr>
<td>Expression</td>
<td>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 195.</td>
</tr>
</tbody>
</table>

Table 20: Trace window columns for the C-SPY simulator
Display area for ETM trace

The Trace window contains the following columns for ETM trace:

<table>
<thead>
<tr>
<th>Trace window column</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Index</td>
<td>A number that corresponds to each ETM packet. Examples of ETM packets are instructions, synchronization points, and exception markers.</td>
</tr>
<tr>
<td>Frame/Time</td>
<td>When collecting trace data in cycle-accurate mode (requires ARM7/9)—enable <strong>Cycle accurate tracing</strong> in the ETM Trace Settings dialog box—the value corresponds to the number of elapsed cycles since the start of the execution. This column is only available for the J-Link/J-Trace driver. When collecting trace data in non-cycle-accurate mode, the value corresponds to an approximate amount of cycles. For Cortex-M devices, the value is repeatedly calibrated with the actual number of cycles. When the <strong>Show timestamp</strong> option is selected in the ETM Trace Settings dialog box, the value displays the time instead of cycles. To display the value as time requires collecting data in cycle-accurate mode, see <strong>Cycle accurate tracing</strong>, page 177, and the J-Link/J-Trace driver.</td>
</tr>
<tr>
<td>Address</td>
<td>The address of the executed instruction.</td>
</tr>
<tr>
<td>Opcode</td>
<td>The operation code of the executed instruction.</td>
</tr>
<tr>
<td>Trace</td>
<td>The collected sequence of executed machine instructions. Optionally, the corresponding source code can also be displayed.</td>
</tr>
<tr>
<td>Comment</td>
<td>This column is only available for the J-Link/J-Trace driver.</td>
</tr>
</tbody>
</table>

Table 21: Trace window columns for ETM trace

**Note:** For RDI drivers, this window looks slightly different.

Display area for SWO trace

The SWO Trace window displays all data from the SWO communication channel in these columns:

<table>
<thead>
<tr>
<th>Trace window column</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Index</td>
<td>An index number for each row in the trace buffer. Simplifies the navigation within the buffer.</td>
</tr>
<tr>
<td>SWO Packet</td>
<td>The contents of the captured SWO packet.</td>
</tr>
</tbody>
</table>

Table 22: Trace window columns for SWO trace
### TRACE SAVE DIALOG BOX

Use the Trace Save dialog box—available from the driver-specific menu, and from the Trace window and the SWO Trace window—to save the collected trace data, as it is displayed in the window, to a file.

![Trace Save dialog box](image)

**Table 22: Trace window columns for SWO trace (Continued)**

<table>
<thead>
<tr>
<th>Trace window column</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Cycles</td>
<td>The approximate number of cycles from the start of the execution until the event.</td>
</tr>
<tr>
<td>Event</td>
<td>The event type of the captured SWO packet. If the column displays <strong>Overflow</strong>, the data packet could not be sent, because too many SWO features use the SWO channel at the same time. To decrease the amount of transmissions on the communication channel, point at the SWO button—on the IDE main window toolbar—with the mouse pointer to get detailed tooltip information about which C-SPY features that have requested trace data generation. Disable some of the features.</td>
</tr>
<tr>
<td>Value</td>
<td>The event value, if any.</td>
</tr>
<tr>
<td>Trace</td>
<td>If the event is a sampled PC value, the instruction is displayed in the Trace column. Optionally, the corresponding source code can also be displayed.</td>
</tr>
<tr>
<td>Comment</td>
<td>Additional information. This includes the values of the selected Trace Events counters, or the number of the comparator (hardware breakpoint) used for the Data Log breakpoint.</td>
</tr>
</tbody>
</table>

If the display area seems to show garbage, make sure you specified correct value for the **CPU clock** option in the SWO Configuration dialog box.
Index Range
Use this option to save a range of frames to a file. Specify a start index and an end index (as numbered in the index column in the Trace window).

Append to file
Appends the trace data to an existing file.

Use tab-separated format
Saves the content in columns that are tab separated, instead of separated by white spaces.

File
Use this text box to locate a file for the trace data.

FUNCTION TRACE WINDOW
The Function Trace window is available from the driver-specific menu when you are using the C-SPY simulator or any driver that support ETM trace.

Using this window
The Function Trace 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 184.
The display area

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

- Trace display area in the C-SPY simulator, page 185
- Display area for ETM trace, page 186.

TIMELINE WINDOW

The Timeline window is available from the Simulator menu and from the J-Link menu during a debug session.

Using this window

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

Display area

The display area can be populated with up to three different graphs:

- Interrupt Log Graph
- Data 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 SWO trace or 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 300.

**Data Log Graph**

The Data Log Graph displays the data logs generated by SWO trace or by the C-SPY simulator, for up to four different variables or address ranges specified as Data Log breakpoints, where:

- Each graph is labeled with—in the left-side area—the name or address for which you have specified the Data Log breakpoint.
- The graph itself displays how the value of the variable changes over time. The label area also displays the limits, or range, of the y-axis for a variable. You can use the context menu to change these limits. The graph can be displayed either as a thin line or as a color-filled solid graph. The graph is a graphical representation of the information in the Data Log window, see *Data Log window*, page 296.
- A red vertical line indicates overflow, which means that the communication channel failed to transmit all data logs from the target system.

**Call Stack Graph**

The Call Stack Graph displays the sequence of calls and returns collected by ETM 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
The context menu in the Timeline window contains some commands that are common to all three graphs and some commands that are specific to each graph.

![Timeline window context menu]

Note: This is the context menu for the Data Log Graph, which means that the menu looks slightly different for the Interrupt Log Graph and for the Call Stack Graph.
These commands are available on the context menu:

<table>
<thead>
<tr>
<th>Menu command</th>
<th>Applies to</th>
<th>Description</th>
</tr>
</thead>
</table>
| 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 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** enlarges the time scale. Shortcut key: +.  
  **Zoom Out** shrinks the time scale. Shortcut key: -.  
  **10ns, 100ns, 1us**, etc makes an interval of 10 nanoseconds, 100 nanoseconds, 1 microseconds, respectively, fit the window.  
  **1ms, 10ms**, etc makes an interval of 1 milliseconds or 10 milliseconds, respectively, fit the window.  
  **10m, 1h**, etc makes an interval of 10 minutes or 1 hour, respectively, fit the window. |
| Data Log     | Data Log Graph | A label that shows that the following Data Log-specific commands are available. |
| Enable       | All graphs   | Toggles the display of the graph. If you disable a graph, that graph will be indicated as OFF in the Timeline window. If no trace data has been collected for a graph, **no data** will appear instead of the graph. |

*Table 23: Commands on the Timeline window context menu*
The **Trace Start** dialog box is available from the context menu that appears when you right-click in the Breakpoints window.

**Using this dialog box**

This dialog box is available for the C-SPY simulator. See also *Using J-Link trace triggers and trace filters*, page 285.
To set a Trace Start breakpoint:

1. In the editor or Disassembly window, right-click and choose **Trace Start** from the context menu.

2. Alternatively, open the Breakpoints window by choosing **View>Breakpoints**.

3. 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.

4. In the **Trigger At** text box specify an expression, an absolute address, or a source location. Click **OK**.

5. When the breakpoint is triggered, the trace data collection starts.

**Trigger At**

Specify the location for the breakpoint in the **Trigger At** text box. Alternatively, click the **Edit** browse button to open the **Enter Location** dialog box; see **Enter Location dialog box**, page 345.

**TRACE STOP BREAKPOINTS DIALOG BOX**

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

![Trace Stop breakpoints dialog box](image)

**Using this dialog box**

This dialog box is available for the C-SPY simulator. See also **Using J-Link trace triggers and trace filters**, page 285.
To set a Trace Stop breakpoint:

1. In the editor or Disassembly window, right-click and choose **Trace Stop** from the context menu.

2. Alternatively, open the Breakpoints window by choosing **View>Breakpoints**.

3. 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.

4. In the **Trigger At** text box specify an expression, an absolute address, or a source location. Click **OK**.

5. When the breakpoint is triggered, the trace data collection stops.

**Trigger At**

Specify the location for the breakpoint in the **Trigger At** text box. Alternatively, click the **Edit** browse button to open the **Enter Location** dialog box; see **Enter Location dialog box**, page 345.

**TRACE EXPRESSIONS WINDOW**

The Trace Expressions window is available from the Trace window toolbar when you are using the C-SPY simulator.

**Using this window**

In the Trace Expressions window you can specify, for example, a specific variable (or an expression) for which you want to collect trace data.
Toolbar

Use the toolbar buttons to change the order between the expressions:

<table>
<thead>
<tr>
<th>Toolbar button</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Arrow up</td>
<td>Moves the selected row up.</td>
</tr>
<tr>
<td>Arrow down</td>
<td>Moves the selected row down.</td>
</tr>
</tbody>
</table>

Table 24: Toolbar buttons in the Trace Expressions window

Display area

In the display area you can specify expressions for which you want to collect trace data:

<table>
<thead>
<tr>
<th>Column</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Expression</td>
<td>Specify any expression that you want to collect data from. You can specify any expression that can be evaluated, such as variables and registers.</td>
</tr>
<tr>
<td>Format</td>
<td>Shows which display format that is used for each expression. Note that you can change display format via the context menu.</td>
</tr>
</tbody>
</table>

Table 25: Trace Expressions window columns

Each row in this window 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.

![Find in Trace dialog box](image)

Using this window

Use the Find in Trace 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 198.

See also, Searching in trace data, page 174.

You specify the search criteria with the following options.

**Text search**

A text field where you type the string you want to search for. Use these options to fine-tune the search:

- **Match Case**  
  Searches only for occurrences that exactly match the case of the specified text. Otherwise specifying int will also find INT and Int.

- **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.
**Address Range**

Use the text fields to specify an address range. 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.

![Find in Trace window](image)

**Using this window**

The Find in Trace 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 197.

See also, **Searching in trace data**, page 174.

**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.
Part 5. IAR C-SPY Simulator

This part of the IAR Embedded Workbench® IDE User Guide contains the following chapters:

- Simulator-specific debugging
- Simulating interrupts.
Simulator-specific debugging

In addition to the general C-SPY® features, the C-SPY Simulator provides some simulator-specific features, which are described in this chapter.

You will get reference information, and information about driver-specific characteristics, such as memory access checking and breakpoints.

The C-SPY Simulator introduction

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 listed in the chapter Product introduction, the C-SPY Simulator also provides:

- Instruction-accurate simulated execution
- Memory configuration and validation
- Interrupt simulation
- Immediate breakpoints with resume functionality
- Peripheral simulation (using the C-SPY macro system).

SELECTING THE SIMULATOR DRIVER

Before starting C-SPY, you must choose the simulator driver. In the IDE, choose Project>Options and click the Setup tab in the Debugger category. Choose Simulator from the Driver drop-down list.
Simulator-specific menus

When you use the simulator driver, the Simulator menu is added to the menu bar.

**SIMULATOR MENU**

<table>
<thead>
<tr>
<th>Menu command</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Interrupt Setup</td>
<td>Displays a dialog box to allow you to configure C-SPY interrupt simulation; see Interrupt Setup dialog box, page 217.</td>
</tr>
<tr>
<td>Forced Interrupts</td>
<td>Displays a window from which you can trigger an interrupt; see Forced interrupt window, page 220.</td>
</tr>
<tr>
<td>Interrupt Log</td>
<td>Displays a window which shows the status of all defined interrupts; see Interrupt Log window, page 300.</td>
</tr>
<tr>
<td>Interrupt Log Summary</td>
<td>Displays a window which shows the status of all defined interrupts; see Interrupt Log Summary window, page 302.</td>
</tr>
<tr>
<td>Memory Access Setup</td>
<td>Displays a dialog box to simulate memory access checking by specifying memory areas with different access types; see Memory Access setup dialog box, page 203.</td>
</tr>
<tr>
<td>Trace</td>
<td>Displays the Trace window which displays the recorded trace data; see Trace window, page 183.</td>
</tr>
<tr>
<td>Function Trace</td>
<td>Displays the Function Trace window which displays the trace data for which functions were called or returned from; see Function Trace window, page 188.</td>
</tr>
<tr>
<td>Function Profiler</td>
<td>Opens the Function Profiler window; see Function Profiler window, page 306.</td>
</tr>
<tr>
<td>Timeline</td>
<td>Opens the Timeline window; see Timeline window, page 189.</td>
</tr>
</tbody>
</table>

*Table 26: Description of Simulator menu commands*
Memory access checking

C-SPY 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. 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 section 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.

Choose Simulator>Memory Access Setup to open the Memory Access Setup dialog box.

**MEMORY ACCESS SETUP DIALOG BOX**

The Memory Access Setup dialog box—available from the Simulator menu—lists all defined memory areas, where each column in the list specifies the properties of the area.

<table>
<thead>
<tr>
<th>Menu command</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Breakpoint Usage</td>
<td>Displays the Breakpoint Usage dialog box which lists all active breakpoints; see Breakpoint Usage dialog box, page 211.</td>
</tr>
</tbody>
</table>

Table 26: Description of Simulator menu commands
In other words, the dialog box displays the memory access setup that will be used during the simulation.

![Memory Access Setup dialog box](image)

**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 206.

**Use ranges based on**

Use the **Use ranges based on** option to choose any of the predefined alternatives for the memory access setup. You can choose between:

- **Device description file**, which means the properties are loaded from the device description file
- **Debug file segment information**, which means the properties are based on the section information available in the debug file. This information is only available while debugging. The advantage of using this option, is that the simulator can catch memory accesses outside the linked application.
Use manual ranges

Use the Use manual ranges option to 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 206.

The ranges you define manually are saved between debug sessions.

Memory access checking

Use the Check for options to specify what to check for:

- Access type violation
- Access to unspecified ranges.

Use the Action options to specify 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

The Memory Access Setup dialog box contains these buttons:

<table>
<thead>
<tr>
<th>Button</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>OK</td>
<td>Standard OK.</td>
</tr>
<tr>
<td>Cancel</td>
<td>Standard Cancel.</td>
</tr>
<tr>
<td>New</td>
<td>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 206.</td>
</tr>
<tr>
<td>Edit</td>
<td>Opens the Edit Memory Access dialog box, where you can edit the selected memory area. See Edit Memory Access dialog box, page 206.</td>
</tr>
<tr>
<td>Delete</td>
<td>Deletes the selected memory area definition.</td>
</tr>
<tr>
<td>Delete All</td>
<td>Deletes all defined memory area definitions.</td>
</tr>
</tbody>
</table>

Table 27: Function buttons in the Memory Access Setup dialog box

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

In the Edit Memory Access dialog box—available from the Memory Access Setup dialog box—you can specify the memory ranges, and assign an access type to each memory range, for which you want to detect illegal accesses during the simulation.

Figure 77: Edit Memory Access dialog box

For each memory range you can define the following properties:

Memory range

Use these settings to define the memory area for which you want to check the memory accesses:

Zone

The memory zone; see Memory addressing, page 149.

Start address

The start address for the address range, in hexadecimal notation.

End address

The end address for the address range, in hexadecimal notation.

Access type

Use one of these options to assign an access type to the memory range; the access type can be one of Read and write, Read only, or Write only. You cannot assign two different access types to the same memory area.

Using breakpoints in the simulator

Using the C-SPY Simulator, you can set an unlimited amount of breakpoints. For code and data breakpoints you can define a size attribute, that is, you can set the breakpoint on a range. You can also set immediate breakpoints.
DATA BREAKPOINTS

Data breakpoints are triggered when data is accessed at the specified location. 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. The execution will usually stop directly after the instruction that accessed the data has been executed.

You can set a data breakpoint in various ways, using:

- A dialog box, see Data breakpoints dialog box, page 207
- A system macro, see __setDataBreak, page 558
- The Memory window, see Setting a data breakpoint in the Memory window, page 144
- The editor window, see Editor window, page 330.

DATA BREAKPOINTS DIALOG BOX

The options for setting data breakpoints are available from the context menu that appears when you right-click in the Breakpoints window. On the context menu, choose New Breakpoint>Data to set a new breakpoint. Alternatively, to modify an existing breakpoint, select a breakpoint in the Breakpoint window and choose Edit on the context menu.
The **Data** breakpoints dialog box appears.

![Data breakpoints dialog box](image)

**Break At**

Specify the location for the breakpoint in the **Break At** text box. Alternatively, click the **Edit** browse button to open the **Enter Location** dialog box; see *Enter Location dialog box*, page 345.

**Access Type**

Use the options in the **Access Type** area to specify the type of memory access that triggers data or immediate breakpoints.

<table>
<thead>
<tr>
<th>Memory Access type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Read/Write</td>
<td>Read or write from location.</td>
</tr>
<tr>
<td>Read</td>
<td>Read from location.</td>
</tr>
<tr>
<td>Write</td>
<td>Write to location.</td>
</tr>
</tbody>
</table>

*Table 28: Memory Access types*

**Note:** Data breakpoints never stop execution within a single instruction. They are recorded and reported after the instruction is executed. (Immediate breakpoints do not stop execution at all, they only suspend it temporarily. See *Immediate breakpoints*, page 209.)

**Size**

Optionally, you can specify a size—in practice, a *range* of locations. Each read and write 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.

There are 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**, you specify the size of the breakpoint manually in the **Size** text box.

**Action**

You can optionally connect an action to a breakpoint. You specify an expression, for instance a C-SPY macro function, which is evaluated when the breakpoint is triggered and the condition is true.

**Conditions**

You can specify simple and complex conditions.

<table>
<thead>
<tr>
<th>Conditions</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Expression</td>
<td>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 must be fulfilled before a break occurs (integer).</td>
</tr>
</tbody>
</table>

### IMMEDIATE BREAKPOINTS

In addition to generic 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 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 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.

The two different methods of setting an immediate breakpoint are by using:

- A dialog box, see *Immediate breakpoints dialog box*, page 210
A system macro, see __setSimBreak, page 560.

**IMMEDIATE BREAKPOINTS DIALOG BOX**

The options for setting immediate breakpoints are available from the context menu that appears when you right-click in the Breakpoints window. On the context menu, choose New Breakpoint>Immediate to set a new breakpoint. Alternatively, to modify an existing breakpoint, select a breakpoint in the Breakpoint window and choose Edit on the context menu.

The Immediate breakpoints dialog box appears.

![Immediate breakpoints page](image)

**Break At**

Specify the location for the breakpoint in the Break At text box. Alternatively, click the Edit browse button to open the Enter Location dialog box; see Enter Location dialog box, page 345.

**Access Type**

Use the options in the Access Type area to specify the type of memory access that triggers data or immediate breakpoints.

<table>
<thead>
<tr>
<th>Memory Access type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Read</td>
<td>Read from location.</td>
</tr>
<tr>
<td>Write</td>
<td>Write to location.</td>
</tr>
</tbody>
</table>

*Table 30: Memory Access types*
Note: Immediate breakpoints do not stop execution at all; they only suspend it temporarily. See Using breakpoints in the simulator, page 206.

Action
You should connect an action 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.

BREAKPOINT USAGE DIALOG BOX
The Breakpoint Usage dialog box—available from the Simulator menu—lists all active breakpoints.

Figure 80: Breakpoint Usage dialog box
In addition to listing all breakpoints that you have defined, this dialog box also lists the internal breakpoints that the debugger is using.

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

For more information, see Viewing all breakpoints, page 146.
Using breakpoints in the simulator
Simulating interrupts

By being able to simulate 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. Finally, reference information about each interrupt system macro is provided.

For information about the interrupt-specific facilities useful when writing interrupt service routines, see the IAR C/C++ Development Guide for ARM®.

The C-SPY interrupt simulation system

The C-SPY Simulator includes an interrupt simulation system that allows you to simulate the execution of interrupts during debugging. You can configure the interrupt simulation system so that it resembles your hardware interrupt 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. Simulated interrupts also let you test the logic of your interrupt service routines.

The interrupt system has the following features:

- Simulated interrupt support for the ARM core
- 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
- Two interfaces for configuring the simulated interrupts—a dialog box and a C-SPY system macro—that is, one interactive and one automating interface
- Activation of interrupts either instantly or based on parameters you define
- A log window which continuously displays the status for each defined interrupt.

The interrupt system is activated by default, but if it is not required you can turn it off to speed up the simulation. You can turn the interrupt system on or off as required either in the Interrupt Setup dialog box, or using a system macro. Defined interrupts will be preserved until you remove them. All interrupts you define using the Interrupt Setup dialog box are preserved between debug sessions.
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, and a variance.

![Simulated interrupt configuration](image)

* If probability is less than 100%; some interrupts may be omitted.

A = Activation time
R = Repeat interval
H = Hold time
V = Variance

Figure 81: 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 statuses can be displayed: *Idle*, *Pending*, *Executing*, *Executed*, *Removed*, or *Expired*.

<table>
<thead>
<tr>
<th>Status</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Idle</td>
<td>Interrupt activation signal is low (deactivated).</td>
</tr>
<tr>
<td>Pending</td>
<td>Interrupt activation signal is active, but the interrupt has not been</td>
</tr>
<tr>
<td></td>
<td>acknowledged yet by the interrupt handler.</td>
</tr>
<tr>
<td>Executing</td>
<td>The interrupt is currently being serviced, that is the interrupt handler</td>
</tr>
<tr>
<td></td>
<td>function is executing.</td>
</tr>
<tr>
<td>Executed</td>
<td>This is a single-occasion interrupt and it has been serviced.</td>
</tr>
<tr>
<td>Removed</td>
<td>The interrupt has been removed by the user, but because the interrupt is</td>
</tr>
<tr>
<td></td>
<td>currently executing it is visible in the Interrupt Setup dialog box until</td>
</tr>
<tr>
<td></td>
<td>it is finished.</td>
</tr>
<tr>
<td>Expired</td>
<td>This is a single-occasion interrupt which was not serviced while the</td>
</tr>
<tr>
<td></td>
<td>interrupt activation signal was active.</td>
</tr>
</tbody>
</table>

Table 31: Interrupt statuses

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:

![Diagram of interrupt status](image)

**Figure 82: 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.
Using the interrupt simulation system

If the interrupt repeat interval is shorter than the execution time, and the interrupt is re-entrant (or non-maskable), the status information at different times can look like this:

![Diagram of interrupt simulation states](image)

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.

**Using 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, and know how to use:

- The Forced Interrupt window
- The **Interrupts** and **Interrupt Setup** dialog boxes
- The C-SPY system macros for interrupts
- The Interrupt Log window, see *Interrupt Log window*, page 300
- Interrupt Log Summary window, see *Interrupt Log Summary window*, page 302.

**TARGET-ADAPTING THE INTERRUPT SIMULATION SYSTEM**

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 be able 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. You can find preconfigured .ddf files in the arm\config\debugger directory. The default settings are used if no device description file has been specified.

1 To load a device description file before you start C-SPY, choose Project>Options and click the Setup tab of the Debugger category.

2 Choose a device description file that suits your target.

Note: In case you do not find a preconfigured device description file that resembles your device, you can define one according to your needs. For details of device description files, see Device description file, page 125.

**INTERRUPT SETUP DIALOG BOX**

The Interrupt Setup dialog box—available by choosing Simulator>Interrupt Setup—lists all defined interrupts.

![Interrupt Setup dialog box](image)

The option Enable interrupt simulation enables or disables interrupt simulation. If the interrupt simulation is disabled, the definitions remain but no interrupts are generated. 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.

The columns contain this information:

- **Interrupt** Lists all interrupts.
Using the interrupt simulation system

<table>
<thead>
<tr>
<th>Type</th>
<th>Shows the type of the interrupt. The type can be <strong>Forced</strong>, <strong>Single</strong>, or <strong>Repeat</strong>.</th>
</tr>
</thead>
<tbody>
<tr>
<td>Status</td>
<td>Shows the status of the interrupt. The status can be <strong>Idle</strong>, <strong>Removed</strong>, <strong>Pending</strong>, <strong>Executing</strong>, or <strong>Expired</strong>.</td>
</tr>
<tr>
<td>Next Activation</td>
<td>Shows the next activation time in cycles.</td>
</tr>
</tbody>
</table>

**Note:** For repeatable interrupts there might be additional information in the Type column 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.

You can only edit or remove non-forced interrupts.

Click **New** or **Edit** to open the **Edit Interrupt** dialog box.

**EDIT INTERRUPT DIALOG BOX**

Use the **Edit Interrupt** dialog box—available from the **Interrupt Setup** dialog box—to add and modify interrupts. This dialog box provides you with a graphical interface where you can interactively fine-tune the interrupt simulation parameters. You can add the parameters and quickly test that the interrupt is generated according to your needs.

![Edit Interrupt dialog box](image)

*Figure 85: Edit Interrupt dialog box*
For each interrupt you can set these options:

**Interrupt**  
A drop-down list containing all available interrupts. Your selection will automatically update the Description box. For Cortex-M devices, the list is populated with entries from the device description file that you have selected. For other devices, only two interrupts are available: IRQ and FIQ.

**Description**  
Contains the description of the selected interrupt, if available. The description consists of a string describing the priority, vector offset, enable bit, and pending bit, separated by space characters. The enable bit and pending bit are optional. It is possible to have none, only the enable bit, or both.

For Cortex-M devices, the description is retrieved from the selected device description file and is editable. Enable bit and pending bit are not available from the ddf file; they must be manually edited if wanted. The priority is as in the hardware: the lower the number, the higher the priority. NMI and HardFault are special, and their descriptions should not be edited. Cortex-M interrupts are also affected by the PRIMASK, FAULTMASK, and BASEPRI registers, as described in the ARM documentation.

For other devices, the description strings for IRQ and FIQ are hardcoded and cannot be edited. In those descriptions, a higher priority number means a higher priority.

For interrupts specified using the system macro __orderInterrupt, the Description box is empty.

**First activation**  
The value of the cycle counter after which the specified type of interrupt will be generated.

**Repeat interval**  
The periodicity of the interrupt in cycles.

**Variance %**  
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**  
Describes 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.
Using the interrupt simulation system

**Probability %** The probability, in percent, that the interrupt will actually occur within the specified period.

**FORCED INTERRUPT WINDOW**

From the Forced Interrupt window—available from the Simulator menu—you can force an interrupt instantly. This is useful when you want to check your interrupt logistics and interrupt routines.

![Forced Interrupt Window](image)

**Figure 86: Forced Interrupt window**

To force an interrupt, the interrupt simulation system must be enabled. To enable the interrupt simulation system, see Interrupt Setup dialog box, page 217.

The Forced Interrupt window lists all available interrupts and their definitions. The information is retrieved from the selected device description file. See this file for a detailed description.

If you select an interrupt and click the **Trigger** button, an interrupt of the selected type is generated.

A triggered interrupt will have these characteristics:

<table>
<thead>
<tr>
<th>Characteristics</th>
<th>Settings</th>
</tr>
</thead>
<tbody>
<tr>
<td>First Activation</td>
<td>As soon as possible (0)</td>
</tr>
<tr>
<td>Repeat interval</td>
<td>0</td>
</tr>
<tr>
<td>Hold time</td>
<td>Infinite</td>
</tr>
<tr>
<td>Variance</td>
<td>0%</td>
</tr>
<tr>
<td>Probability</td>
<td>100%</td>
</tr>
</tbody>
</table>

*Table 32: Characteristics of a forced interrupt*

**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
Simulating interrupts

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 a set of predefined system macros for the interrupt simulation system. The advantage of using the system macros for specifying the simulated interrupts is that it lets you automate the procedure.

These are the available 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. To read more about how to use the __popSimulatorInterruptExecutingStack macro, see Interrupt simulation in a multi-task system, page 221.

For detailed reference information about each macro, see Description of C-SPY system macros, page 535.

Defining simulated interrupts at C-SPY startup using a setup file

If you want to use a setup file to define simulated interrupts at C-SPY startup, follow the procedure described in Registering and executing using setup macros and setup files, page 159.

Interrupt simulation 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 avoid these problems, you can use the __popSimulatorInterruptExecutingStack macro to inform the interrupt simulation system that the interrupt handler has finished executing, as if the normal
Simulating a simple interrupt

In this example you will simulate a system timer interrupt for OKI ML674001. However, the procedure can also be used for other types of interrupts.

This simple application contains an IRQ handler routine that handles system timer interrupts. It increments a tick variable. The main function sets the necessary status registers. The application exits when 100 interrupts have been generated.

Read more about how to write your own interrupt handler in the IAR C/C++ Development Guide for ARM®.

/* Enables use of extended keywords */
#pragma language=extended

#include <intrinsics.h>
#include <oki/ioml674001.h>
#include <stdio.h>

unsigned int ticks = 0;

/* IRQ handler */
__irq __arm void IRQ_Handler(void)
{
    /* We use only system timer interrupts, so we do not need to check the interrupt source. */
    ticks += 1;
    TM0VFR.bit.OVF = 1; /* Clear system timer overflow flag */
}

int main( void )
{
    __enable_interrupt();
    /* Timer setup code */
    ILC0_bit.ILR0 = 4; /* System timer interrupt priority */
    TMRLR.bit.TMRLR = 1B5; /* System timer reload value */
Simulating interrupts

```c
TMEN_bit.TCEN = 1; /* Enable system timer */
while (ticks < 100);
printf("Done\n");
}
```

To simulate and debug an interrupt, do these steps:

1. Add your interrupt service routine to your application source code and add the file to your project.
2. Build your project and start the simulator.
3. 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 IRQ</td>
<td>4000</td>
</tr>
<tr>
<td>First Activation</td>
<td>2000</td>
</tr>
<tr>
<td>Repeat interval</td>
<td>0</td>
</tr>
<tr>
<td>Hold time</td>
<td>0</td>
</tr>
<tr>
<td>Probability %</td>
<td>100</td>
</tr>
<tr>
<td>Variance %</td>
<td>0</td>
</tr>
</tbody>
</table>

*Table 33: Timer interrupt settings*

Click OK.

4. 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.

To watch the interrupt in action, open the Interrupt Log window by choosing Simulator>Interrupt Log.
Simulating a simple interrupt
Part 6. C-SPY hardware debugger systems

This part of the IAR Embedded Workbench® IDE User Guide contains the following chapters:

- Introduction to C-SPY® hardware debugger systems
- Hardware-specific debugging
- Analyzing your program using driver-specific tools
- Using flash loaders.
Introduction to C-SPY® hardware debugger systems

This chapter introduces you to the C-SPY hardware debugger systems and to how they differ from the C-SPY Simulator.

The chapters specific to C-SPY hardware debugger systems assume that you already have some working knowledge of the target system you are using, as well as of the IAR C-SPY Debugger.

The C-SPY hardware debugger systems

C-SPY consists of both a general part which provides a basic set of C-SPY features, and a driver. The C-SPY driver is the part that provides communication with and control of the target system. The driver also provides a user interface—special menus, windows, and dialog boxes—to the functions provided by the target system, for instance special breakpoints. This driver is automatically installed during the installation of IAR Embedded Workbench.

At the time of writing this guide, the IAR C-SPY Debugger for the ARM core is available with drivers for these target systems:

- Simulator
- RDI (Remote Debug Interface)
- J-Link/J-Trace JTAG probe
- GDB Server for STR9-comStick
- Macraigor JTAG interface
- Angel debug monitor
- IAR ROM-monitor for Analog Devices ADuC7xxx boards, IAR Kickstart Card for Philips LPC210x, and OKI evaluation boards
- Luminary FTDI JTAG interface (for Cortex devices only)
- ST-Link JTAG probe (for ST STM32 devices only).
Note: In addition to the drivers supplied with the IAR Embedded Workbench, you can also load debugger drivers supplied by a third-party vendor; see Debugging using a third-party driver, page 269.

For further details about the concepts that are related to C-SPY, see Debugger concepts, page 117.

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>Angel</th>
<th>GDB Server</th>
<th>IAR ROM-monitor</th>
<th>J-Link/ J-Trace</th>
<th>LMI</th>
<th>Macraigor</th>
<th>RDI</th>
<th>ST-Link</th>
</tr>
</thead>
<tbody>
<tr>
<td>Data breakpoints</td>
<td>x</td>
<td>x 2)</td>
<td>x 2), 4)</td>
<td>x 2)</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Code breakpoints</td>
<td>x</td>
<td>x</td>
<td>x 2), 4)</td>
<td>x 2)</td>
<td>x 2)</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Execution in real time</td>
<td>x</td>
<td>x</td>
<td>x 2), 4)</td>
<td>x 2)</td>
<td>x 2)</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Zero memory footprint</td>
<td>x</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Simulated interrupts</td>
<td>x</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Real interrupts</td>
<td>x</td>
<td>x</td>
<td>x 2), 4)</td>
<td>x 2)</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Interrupt logging</td>
<td>x</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Trace</td>
<td>x</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Data logging</td>
<td>x</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Live watch</td>
<td>x</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Cycle counter</td>
<td>x</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Code coverage</td>
<td>x</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Data coverage</td>
<td>x</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Function/ Instruction profiler</td>
<td>x</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Profiling</td>
<td>x</td>
<td>x 1)</td>
<td>x 1) 3)</td>
<td>x 1) 3)</td>
<td>x 1) 3)</td>
<td>x 1) 3)</td>
<td>x 1) 3)</td>
<td>x 1) 3)</td>
<td>x 1) 3)</td>
</tr>
</tbody>
</table>

Table 34: Differences between available C-SPY drivers

1) Cycle counter statistics are not available.
2) Limited number, implemented using the ARM EmbeddedICE™ macrocell.
3) Profiling works provided that enough breakpoints are available. That is, the application is executed in RAM.
4) Limited number, implemented using the Data watchpoint and trigger unit (for data breakpoints) and the Flash patch and breakpoint unit (for code breakpoints).
5) Supported by J-Trace only. For detailed information about code coverage, see Code coverage, page 144.
6) Supported by Cortex devices. For ARM7/9 devices Live watch is supported if you add a DCC.
handler to your application. See Live watch and use of DCC, page 259.
7) Requires either SWD/SWO interface or ETM trace.
8) Cortex with SWD/SWO.

Contact your software distributor or IAR representative for information about available C-SPY drivers. General descriptions of the different drivers follow.

---

### Getting started

The following documents containing information about how to set up various debugging systems are available in the `arm\doc` subdirectory:

<table>
<thead>
<tr>
<th>File</th>
<th>Debugger system</th>
</tr>
</thead>
<tbody>
<tr>
<td>rdi_quickstart.htm</td>
<td>Quickstart reference for RDI-controlled JTAG debug interfaces</td>
</tr>
<tr>
<td>gdbserver_quickstart.htm</td>
<td>Quickstart reference for a GDB Server using OpenOCD together with STR9-comStick</td>
</tr>
<tr>
<td>angel_quickstart.htm</td>
<td>Quickstart reference for Angel ROM-monitors and JTAG interfaces</td>
</tr>
<tr>
<td>iar_rom_quickstart.htm</td>
<td>Quickstart reference for IAR and OKI ROM-monitor</td>
</tr>
</tbody>
</table>

Table 35: Available quickstart reference information

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 `arm\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 linker command files.

---

### RUNNING THE DEMO PROGRAM

1. To use an example application, 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. Click Open Project.
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 of the options Device and Debugger>Driver. As for other settings, the project is set up to suit the target system you selected.

For further details about the C-SPY options for the hardware target system and how to configure C-SPY to interact with the target board, see C-SPY options for debugging using hardware systems, page 243.

Click OK to close the Options dialog box.

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

7 To start C-SPY, choose Project>Debug or click the Debug button.

8 Choose Execute>Go or click the Go button to start the application program.

Click the Stop button to stop execution.

---

The IAR C-SPY Angel debug monitor driver

Using the C-SPY Angel debug monitor driver, C-SPY can connect to any devices conforming to the Angel debug monitor protocol. In most cases these are evaluation boards. However, the EPI JEENI JTAG interface also uses this protocol.

The rest of this section assumes the Angel connection is made to an evaluation board.

The evaluation board contains firmware (the Angel debug monitor itself) that runs in parallel with your application software. The firmware receives commands from the IAR C-SPY debugger over a serial port or Ethernet connection, and controls the execution of your application.
Except for the EPI JEENI JTAG interface, all the parts of your code that you want to debug must be located in RAM. The only way you can set breakpoints and step in your application code is to download it into RAM.

*For the EPI JEENI JTAG interface, the user application can be located in flash memory.

For further information, see the *angel_quickstart.htm* file, or refer to the manufacturer’s documentation.

### The IAR C-SPY GDB Server driver

Using the IAR GDB Server driver, C-SPY can connect to the available GDB Server-based JTAG solutions, currently Open OCD with STR9-comStick. JTAG is a standard on-chip debug connection available on most ARM processors.

To use any of the GDB server-based JTAG solutions, you must configure the hardware and the software drivers involved; see *Configuring the OpenOCD Server*, page 232.

Starting a debug session with the C-SPY GDB Server driver will add the **GDB Server** menu to the debugger menu bar. For further information about the menu commands, see *The GDB Server menu*, page 250.
The C-SPY GDB Server driver communicates with the GDB Server via an Ethernet connection, and the GDB Server communicates with the JTAG interface module over a USB connection. The JTAG interface module, in turn, communicates with the JTAG module on the hardware.

**Figure 88: C-SPY GDB Server communication overview**

**CONFIGURING THE OPENOCD SERVER**

Follow these instructions to configure the OpenOCD Server:

1. Install IAR Embedded Workbench for ARM.

Insert the STR9-comStick device into a USB port on your host computer. Windows will find the new hardware and ask for its driver. The USB driver is available in the arm\drivers\STComstickFTDI directory in your IAR Embedded Workbench installation.
Introduction to C-SPY® hardware debugger systems

3 Start the OpenOCD server from a command line window and specify the configuration file `str912_comStick.cfg` available in the `arm\examples\ST\STR91x\STR9-comStick` directory in your IAR Embedded Workbench installation. For example:

```
"C:\Program Files\openocd-2007re204\bin\openocd-ftd2xx" --file "C:\Program Files\IAR Systems\Embedded Workbench 5.0\arm\examples\ST\STR91x\STR9-comStick\str912_comStick.cfg"
```

4 Start the IAR Embedded Workbench IDE and open the USB demo example project for STR9-comStick, `usb.eww`.

5 Choose `Project>Options>Debugger>GDB Server` and specify the location of the OpenOCD server. If the server is located on the host computer, specify `localhost` with port `3333`, otherwise specify the host name or IP address.

6 Start the debug session and click the `Run` button when downloading has finished. The example application emulates a USB mouse. By connecting the secondary USB connector to a PC, the mouse pointer on the PC screen will start moving.

**The IAR C-SPY ROM-monitor driver**

Using the C-SPY ROM-monitor driver, C-SPY can connect to the IAR Kickstart Card for Philips LPC210x and for Analog Devices ADuC7xxx, and to OKI evaluation boards. The evaluation board contains firmware (the ROM-monitor itself) that runs in parallel with your application software. The firmware receives commands from the IAR C-SPY debugger over a serial port or USB connection (for OKI evaluation boards only), and controls the execution of your application.
Most ROM-monitors require that the code that you want to debug is located in RAM, because the only way you can set breakpoints and step in your application code is to download it to RAM. For some ROM-monitors, for example Analog Devices ADuC7xxx, the code that you want to debug can be located in flash memory. To maintain debug functionality, the ROM-monitor might simulate some instructions, for example when single stepping.

![Diagram of C-SPY ROM-monitor communication overview](iar_rom_quickstart.htm)

* For some ROM-monitors, the user application can be located in flash memory.

For further information, see the `iar_rom_quickstart.htm` file, or refer to the manufacturer’s documentation.

---

**The IAR C-SPY J-Link/J-Trace drivers**

Using the IAR J-Link/J-Trace driver, C-SPY can connect to the IAR J-Link JTAG debug probe and the IAR J-Trace JTAG debug probe. JTAG is a standard on-chip debug connection available on most ARM processors.
Before you can use the J-Link/J-Trace JTAG probe over the USB port, the Segger J-Link/J-Trace USB driver must be installed; see *Installing the J-Link USB driver*, page 235. You can find the driver on the IAR Embedded Workbench for ARM installation CD.

Starting a debug session with the J-Link driver will add the **J-Link** menu to the debugger menu bar. For further information about the menu commands, see *The J-Link menu*, page 257.

The C-SPY J-Link driver communicates with the JTAG interface module over a USB connection. The JTAG interface module, in turn, communicates with the JTAG module on the hardware.

**INSTALLING THE J-LINK USB DRIVER**

Before you can use the J-Link JTAG interface over the USB port, the Segger J-Link USB driver must be installed.

1. Install IAR Embedded Workbench for ARM.
2. Use the USB cable to connect the computer and J-Link. Do not connect J-Link to the target board yet. The green LED on the front panel of J-Link will blink for a few seconds while Windows searches for a USB driver.
Because this is the first time J-Link and the computer are connected, Windows will open a dialog box and ask you to locate the USB driver. The USB driver can be found in the product installation in the `arm\drivers\JLink` directory:

- jlink.inf, jlinkx64.inf
- jlink.sys, jlinkx64.sys

Once the initial setup is completed, you will not have to install the driver again.

Note that J-Link will continuously blink until the USB driver has established contact with the J-Link probe. When contact has been established, J-Link will start with a slower blink to indicate that it is alive.

---

**The IAR C-SPY LMI FTDI driver**

Using the IAR C-SPY LMI FTDI driver, C-SPY can connect to the Luminary FTDI onboard JTAG interface for Cortex devices.

Before you can use the FTDI JTAG interface over the USB port, the FTDI USB driver must be installed. You can find the driver on the IAR Embedded Workbench for ARM installation CD.
Starting a debug session with the FTDI driver will add the LMI FTDI menu to the debugger menu bar. For further information about the menu commands, see *The LMI FTDI menu*, page 261.

**INSTALLING THE FTDI USB DRIVER**

Before you can use the LMI FTDI JTAG interface over the USB port, the FTDI USB driver must be installed.

1. Install IAR Embedded Workbench for ARM.
2. Use the USB cable to connect the computer to the Luminary board.

Because this is the first time FTDI and the computer are connected, Windows will open a dialog box and ask you to locate the USB driver. The USB driver can be found in the product installation in the `arm\drivers\LuminaryFTDI` directory.

Once the initial setup is completed, you will not have to install the driver again.

**The IAR C-SPY Macraigor driver**

Using the IAR Macraigor driver, C-SPY can connect to the Macraigor mpDemon, USB2 Demon, and USB2 Sprite JTAG interfaces. JTAG is a standard on-chip debug connection available on most ARM processors.
Before you can use Macraigor JTAG interfaces over the parallel port or the USB port, the Macraigor OCDemon drivers must be installed. You can find the drivers on the IAR Embedded Workbench CD for ARM. This is not needed for serial and Ethernet connections.

Starting a debug session with the Macraigor driver will add the JTAG menu to the debugger menu bar. This menu provides commands for configuring JTAG watchpoints, and setting breakpoints on exception vectors (also known as vector catch). For further information about the menu commands, see The Macraigor JTAG menu, page 264.

The C-SPY Macraigor driver communicates with the JTAG interface module over a serial, USB, or Ethernet connection. The JTAG interface module, in turn, communicates with the JTAG module on the hardware.

**Figure 91: C-SPY Macraigor communication overview**

---

**The IAR C-SPY RDI driver**

Using the C-SPY RDI driver, C-SPY can connect to an RDI-compliant debug system. This can be a simulator, a ROM-monitor, a JTAG interface, or an emulator. For the remainder of this section, an RDI-based connection to a JTAG interface is assumed. JTAG is a standard on-chip debug connection available on most ARM processors.
Before you can use an RDI-based JTAG interface, you must install the RDI driver DLL provided by the JTAG interface vendor.

In the Embedded Workbench IDE, you must then locate the RDI driver DLL file. To do this, choose **Project>Options** and select the **C-SPY Debugger** category. On the **Setup** page, choose **RDI** from the **Driver** drop-down list. On the **RDI** page, locate the RDI driver DLL file using the **Manufacturer RDI Driver** browse button. For more information about the other options available, see *Debugging using the RDI driver*, page 265. When you have loaded the RDI driver DLL, the **RDI** menu will appear on the Embedded Workbench IDE menu bar. This menu provides a configuration dialog box associated with the selected RDI driver DLL. Note that this dialog box is unique to each RDI driver DLL.

The RDI driver DLL communicates with the JTAG interface module over a parallel, serial, Ethernet, or USB connection. The JTAG interface module, in turn, communicates with the JTAG module on the hardware.

![C-SPY RDI communication overview](diagram.png)

*Figure 92: C-SPY RDI communication overview*

For further information, see the *rdi_quickstart.htm* file, or refer to the manufacturer’s documentation.
The IAR C-SPY ST-Link driver

Using the IAR ST-Link driver, C-SPY can connect to the ST-Link JTAG probe. JTAG is a standard on-chip debug connection available on most ARM processors.

USB drivers for the ST-Link JTAG probe are automatically installed on the host computer when you connect the ST-Link JTAG probe for the first time. Normally, you do not have to do anything else.

The C-SPY ST-Link driver communicates with the JTAG interface module over a USB connection. The JTAG interface module, in turn, communicates with the JTAG module on the hardware.

For more information about the C-SPY environment when using the ST-Link driver, see Debugging using the ST-Link driver, page 267.

---

Figure 93: C-SPY ST-Link communication overview

For more information about the C-SPY environment when using the ST-Link driver, see Debugging using the ST-Link driver, page 267.
An overview of the debugger startup

To make it easier to understand and follow the startup flow, the following figures show the flow of actions performed by the C-SPY debugger, and by the target hardware, as well as the execution of any predefined C-SPY setup macros. There is one figure for debugging code located in flash and one for debugging code located in RAM.

To read more about C-SPY system macros, see the chapters Using the C-SPY® macro system and C-SPY® macros reference available in this guide.

**DEBUGGING CODE IN FLASH**

![Diagram of debugger startup when debugging code in flash](image-url)

*Figure 94: Debugger startup when debugging code in flash*
An overview of the debugger startup

**DEBUGGING CODE IN RAM**

![Diagram](image)

**Figure 95: Debugger startup when debugging code in RAM**

---

*IAR Embedded Workbench® IDE*

242   User Guide for ARM
Hardware-specific debugging

This chapter describes the additional options and menus provided by the C-SPY® hardware debugger systems. The chapter contains the following sections:

- C-SPY options for debugging using hardware systems
- Debugging using the Angel debug monitor driver
- Debugging using the IAR C-SPY GDB Server driver
- Debugging using the IAR C-SPY ROM-monitor driver
- Debugging using the IAR C-SPY J-Link/J-Trace driver
- Debugging using the IAR C-SPY LMI FTDI driver
- Debugging using the IAR C-SPY Macraigor driver
- Debugging using the RDI driver
- Debugging using the ST-Link driver
- Debugging using a third-party driver.

C-SPY options for debugging using hardware systems

Before you start any C-SPY hardware debugger you must set some options for the debugger system—both C-SPY generic options and options required for the hardware system (C-SPY driver-specific options).

Follow this procedure:

1. To open the Options dialog box, choose Project>Options.
2. To set C-SPY generic options and select a C-SPY driver:
   - Select Debugger from the Category list.
C-SPY options for debugging using hardware systems

- On the Setup page, select the appropriate C-SPY driver from the Driver list.

For information about the settings Setup macros, Run to, and Device descriptions, as well as for information about the pages Images and Plugins, see Debugger options, page 493.

Note that a default device description file and linker configuration file is automatically selected depending on your selection of a device on the General Options>Target page.

3 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 available option pages appears.

For details about each page, see:
- Download, page 245
- Angel, page 247
- GDB Server, page 249
- IAR ROM-monitor, page 251
- For J-Link/J-Trace, see Setup, page 252 and Connection, page 256
- Setup, page 260 for LMI FTDI
- Macraigor, page 262
- RDI, page 265
- ST-Link, page 268
- Third-Party Driver, page 269.

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

By default, C-SPY downloads the application into RAM or flash when a debug session starts. The Download options lets you modify the behavior of the download.

**Attach to program**

Use this option to make the debugger attach to a running application at its current location, without resetting and halting (for J-Link only) the target system. To avoid unexpected behavior when using this option, the Debugger>Setup option Run to should be deselected.

**Verify download**

Use this option to verify that the downloaded code image can be read back from target memory with the correct contents.

**Suppress download**

Use this option to debug an application that already resides in target memory. When this option is selected, the code download is disabled, while preserving the present content of the flash.

If this option is combined with the Verify download option, the debugger will read back the code image from non-volatile memory and verify that it is identical to the debugged program.

**Note:** It is important that the image that resides in target memory is linked consistently with how you use C-SPY for debugging. This applies, for example, if you first link your application using an output format without debug information, such as Intel-hex, and then load the application separately from C-SPY. If you then use C-SPY only for
debugging without downloading, you cannot build the debugged application with any of the options Semihosted or IAR breakpoint—on the General Options>Library Configuration page—as that would add extra code, resulting in two different code images.

**Use flash loader(s)**

Use the Use flash loader(s) option to use one or several flash loaders for downloading your application to flash memory. If a flash loader is available for the selected chip, it is used by default. To read more about flash loaders, see The flash loader, page 309.

**Override default .board file**

A default flash loader is selected based on your choice of device on the General Options>Target page. To override the default flash loader, select Override default .board file and specify the path to the flash loader you want to use. A browse button is available for your convenience. Press the Edit button to open the Flash Loader Overview dialog box. For more information, see Flash Loader Overview dialog box, page 310.

**EXTRA OPTIONS PAGE**

The Extra Options page provides you with a command line interface to C-SPY.

**Use command line options**

Additional command line arguments (not supported by the GUI) for C-SPY can be specified here.
Debugging using the Angel debug monitor driver

For detailed information about the Angel debug monitor interface and how to get started, see the angel_quickstart.htm file, available in the arm\doc subdirectory.

Using the Angel protocol, C-SPY can connect to a target system equipped with an Angel boot flash. This is an inexpensive solution to debug a target, because only a serial cable is needed.

The Angel protocol is also used by certain ICE hardware. For example, the EPI JEENI JTAG interface uses the Angel protocol.

ANGEL

This section describes the options that specify the C-SPY Angel debug monitor interface. In the IAR Embedded Workbench IDE, choose Project>Options and click the Angel tab in the Debugger category.

Send heartbeat

Use this option to make C-SPY poll the target system periodically while your application is running. That way, the debugger can detect if the target application is still running or has terminated abnormally. Enabling the heartbeat will consume some extra CPU cycles from the running program.

Communication

Use this option to select the Angel communication link. RS232 serial port connection and TCP/IP via an Ethernet connection are supported.
TCP/IP
Type the IP address of the target device in the text box.

Serial port settings
Use the Port drop-down list to select which serial port on the host computer to use as the Angel communication link, and set the communication speed using the Baud rate drop-down list.

The initial Angel serial speed is always 9600 baud. After the initial handshake, the link speed is changed to the specified speed. Communication problems can occur at very high speeds; some Angel-based evaluation boards will not work above 38,400 baud.

Log communication
Use this option to log the communication between C-SPY and the target system to a file. To interpret the result, a detailed knowledge of the Angel monitor protocol is required.

Debugging using the IAR C-SPY GDB Server driver
To use C-SPY for the GDB Server, you should be familiar with the following details:

● The C-SPY options specific to the GDB Server, see GDB Server, page 249
● Using breakpoints in the hardware debugger systems, page 271
● The GDB Server menu, page 250.
GDB SERVER

This section describes the options that specify the GDB Server for the STR9-comStick evaluation board. In the IAR Embedded Workbench IDE, choose Project>Options, select the GDB Server category, and click the GDB Server tab.

TCP/IP address or hostname

Use the text box to specify the IP address and port number of a GDB server; by default the port number 3333 is used. The TCP/IP connection is used for connecting to a J-Link server running on a remote computer.

Log communication

Use this option to log the communication between C-SPY and the target system to a file. To interpret the result, detailed knowledge of the JTAG interface is required.

BREAKPOINTS

In the IAR Embedded Workbench IDE, choose Project>Options, select the GDB Server category, and click the Breakpoints tab.

For details, see Breakpoints options, page 272.
Debugging using the IAR C-SPY ROM-monitor driver

The GDB Server menu

When you are using the C-SPY GDB Server driver, the additional menu **GDB Server** appears in C-SPY.

The following command is available on the **GDB Server** menu:

<table>
<thead>
<tr>
<th>Menu command</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Breakpoint Usage</td>
<td>Opens the Breakpoint Usage dialog box to list all active breakpoints; see</td>
</tr>
<tr>
<td></td>
<td>Breakpoint Usage dialog box, page 280.</td>
</tr>
</tbody>
</table>

Table 36: Commands on the GDB Server menu

Debugging using the IAR C-SPY ROM-monitor driver

For detailed information about the IAR ROM-monitor interface and how to get started using it together with the IAR Kickstart Card for Philips LPC210x or for Analog Devices ADuC7xxx, see the documentation that comes with the Kickstart product package.

For detailed information about the IAR ROM-monitor interface and how to get started using it together with the OKI JOB671000 evaluation board, see the iar_rom_quickstart.htm file, available in the arm\doc subdirectory.

The ROM-monitor protocol is an IAR Systems proprietary protocol used by some ARM-based evaluation boards.
IAR ROM-MONITOR

This section describes the options that specify the C-SPY IAR ROM-monitor interface. In the IAR Embedded Workbench IDE, choose **Project>Options** and click the **IAR ROM-monitor** tab in the **Debugger** category.

![IAR ROM-monitor options](image)

**Communication**

Use this option to select the ROM-monitor communication link. USB connection and RS232 serial port connection are supported.

**Serial port settings**

Use the **Port** drop-down list to select which serial port on the host computer to use as the ROM-monitor communication link, and set the communication speed using the **Baud rate** drop-down list. The serial port communication link speed must match the speed selected on the target board.

**Log communication**

Use this option to log the communication between C-SPY and the target system to a file. To interpret the result, a detailed knowledge of the ROM-monitor protocol is required.
Debugging using the IAR C-SPY J-Link/J-Trace driver

To use C-SPY for the J-Link/J-Trace debug probe, you should be familiar with the following details:

- The C-SPY options specific to the J-Link/J-Trace JTAG interface, see Setup, page 252, Connection, page 256
- The J-Link menu, page 257
- Live watch and use of DCC, page 259
- Using breakpoints in the hardware debugger systems, page 271
- Using JTAG watchpoints, page 282
- Using trace, page 169

SETUP

This section describes the options that specify the J-Link/J-Trace probe. In the IAR Embedded Workbench IDE, choose Project>Options, select the J-Link/J-Trace category, and click the Setup tab.

Figure 102: C-SPY J-Link/J-Trace Setup options
**Reset**

Use this option to select the reset strategy to be used when the debugger starts. Note that Cortex-M uses a different set of strategies than other devices.

For Cortex-M devices, choose between these strategies:

- **Normal (default)** Tries to reset the core via the reset strategy **Core and peripherals** first. If this fails, the reset strategy **Core only** is used. It is recommended that you use this strategy to reset the target.

- **Core** The core is reset via the VECTRESET bit; the peripheral units are not affected.

- **Reset Pin** J-Link pulls its RESET pin low to reset the core and the peripheral units. Normally, this causes the CPU RESET pin of the target device to go low as well, which results in a reset of both the CPU and the peripheral units.

- **Connect during reset** J-Link connects to the target while keeping Reset active (reset is pulled low and remains low while connecting to the target). This is the recommended reset strategy for STM32 devices. This strategy is available for STM32 devices only.

- **Halt after bootloader** This is the same strategy as the Normal strategy, but the target is halted when the bootloader has finished executing. This is the recommended reset strategy for LPC11xx and LPC13xx devices. This strategy is available for LPC11xx and LPC13xx devices only.

- **Halt before bootloader** This is the same strategy as the Normal strategy, but the target is halted before the bootloader has started executing. This strategy is normally not used, except in situations where the bootloader needs to be debugged. This strategy is available for LPC11xx and LPC13xx devices only.

All of these strategies are available for both the JTAG and the SWD interface, and all strategies halt the CPU after the reset.
For other cores, choose between these strategies:

<table>
<thead>
<tr>
<th>Strategy</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Hardware, halt after delay (ms)</strong></td>
<td>Hardware reset. Use the text box to specify the delay between the hardware reset and the halt of the processor. This is used for making sure that the chip is in a fully operational state when C-SPY starts to access it. By default, the delay is set to zero to halt the processor as quickly as possible.</td>
</tr>
<tr>
<td><strong>Hardware, halt using Breakpoint</strong></td>
<td>Hardware reset. After reset, J-Link continuously tries to halt the CPU using a breakpoint. Typically, this halts the CPU shortly after reset; the CPU can in most systems execute some instructions before it is halted.</td>
</tr>
<tr>
<td><strong>Hardware, halt at 0</strong></td>
<td>Hardware reset. The processor is halted by placing a breakpoint at the address zero. Note that this is not supported by all ARM microcontrollers.</td>
</tr>
<tr>
<td><strong>Hardware, halt using DBGRQ</strong></td>
<td>Hardware reset. After reset, J-Link continuously tries to halt the CPU using DBGRQ. Typically, this halts the CPU shortly after reset; the CPU can in most systems execute some instructions before it is halted.</td>
</tr>
<tr>
<td><strong>Software</strong></td>
<td>Software reset. Sets PC to the program entry address.</td>
</tr>
<tr>
<td><strong>Software, Analog devices</strong></td>
<td>Software reset. Uses a reset sequence specific for the Analog Devices ADuC7xxx family. This strategy is only available if you have selected such a device from the Device drop-down list on the General Options&gt;Target page.</td>
</tr>
<tr>
<td><strong>Hardware, NXP LPC</strong></td>
<td>Hardware reset specific to NXP LPC devices. This strategy is only available if you have selected such a device from the Device drop-down list on the General Options&gt;Target page.</td>
</tr>
<tr>
<td><strong>Hardware, Atmel AT91SAM7</strong></td>
<td>Hardware reset specific for the Atmel AT91SAM7 family. This strategy is only available if you have selected such a device from the Device drop-down list on the General Options&gt;Target page.</td>
</tr>
</tbody>
</table>

For more details about the different reset strategies, see the J-Link / J-Trace User’s Guide available in the arm\doc directory.
A software reset of the target does not change the settings of the target system; it only resets the program counter and the mode register `CPSR` to its reset state. Normally, a C-SPY reset is a software reset only. If you use the **Hardware reset** option, C-SPY will generate an initial hardware reset when the debugger is started. This is performed once before download, and if the option **Use flash loader(s)** is selected, also once after flash download, see Figure 94, *Debugger startup when debugging code in flash*, page 241, and Figure 95, *Debugger startup when debugging code in RAM*, page 242.

Hardware resets can be a problem if the low-level setup of your application is not complete. If the low-level setup does not set up memory configuration and clocks, the application will not work after a hardware reset. To handle this in C-SPY, the setup macro function `execUserReset()` is suitable. For a similar example where `execUserPreload()` is used, see *Remapping memory*, page 126.

**JTAG/SWD speed**

Use the JTAG speed options to set the JTAG communication speed in kHz.

**Auto**

If you use the **Auto** option, the J-Link probe will automatically use the highest possible frequency for reliable operation. The initial speed is the fixed frequency used until the highest possible frequency is found. The default initial frequency—32 kHz—can normally be used, but in cases where it is necessary to halt the CPU after the initial reset, in as short time as possible, the initial frequency should be increased.

A high initial speed is necessary, for example, when the CPU starts to execute unwanted instructions—for example power down instructions—from flash or RAM after a reset. A high initial speed would in such cases ensure that the debugger can quickly halt the CPU after the reset.

The initial value must be in the range 1–12000 kHz.

**Fixed**

Use the **Fixed** text box to set the JTAG communication speed in kHz. The value must be in the range 1–12000 kHz.

If there are JTAG communication problems or problems in writing to target memory (for example during program download), these problems might be avoided if the speed is set to a lower frequency.

**Adaptive**

The adaptive speed only works with ARM devices that have the `RTCK` JTAG signal available. For more information about adaptive speed, see the *J-Link / J-Trace User's Guide* available in the `arm\doc` directory.
CONNECTION

In the IAR Embedded Workbench IDE, choose Project>Options, select the J-Link/J-Trace category, and click the Connection tab.

Communication

Use this option to select the communication channel between C-SPY and the debug probe. Choose between USB and TCP/IP. If you choose TCP/IP, use the text box to specify the IP address of a J-Link server. The TCP/IP connection is used for connecting to a J-Link server running on a remote computer.

Interface

Use this option to specify communication interface between the J-Link debug probe and the target system. Choose between:

- JTAG (default)
- SWD; uses fewer pins than JTAG. Select SWD if you want to use the serial-wire output (SWO) communication channel. Note that if you select stdout/stderr via SWO on the General Options>Library Configuration page, SWD is selected automatically. For more information about SWO settings, see SWO Trace Window Settings dialog box, page 178.

JTAG scan chain

If there is more than one device on the JTAG scan chain, enable the JTAG scan chain with multiple targets option, and specify the TAP number option, which is the TAP (Test Access Port) position of the device you want to connect to. The TAP numbers start from zero.
For JTAG scan chains that mix ARM devices with other devices like, for example, FPGA, enable the **Scan chain contains non-ARM devices** option and specify the number of IR bits before the ARM device to be debugged in the **Preceding bits** text field.

**Log communication**

Use this option to log the communication between C-SPY and the target system to a file. To interpret the result, detailed knowledge of the JTAG interface is required.

**BREAKPOINTS**

In the IAR Embedded Workbench IDE, choose **Project>Options**, select the **J-Link/J-Trace** category, and click the **Breakpoints** tab.

For details, see *Breakpoints options*, page 272.

**THE J-LINK MENU**

When you are using the C-SPY J-Link driver, the additional menu **J-Link** appears in C-SPY.

![Figure 104: The J-Link menu](image)
The following commands are available on the **J-Link** menu:

<table>
<thead>
<tr>
<th>Menu command</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Watchpoints</td>
<td>Opens a dialog box for setting watchpoints, see <em>JTAG watchpoints dialog box</em>, page 283.</td>
</tr>
<tr>
<td>Vector Catch</td>
<td>Opens a dialog box for setting a breakpoint directly on a vector in the interrupt vector table, see <em>Breakpoints on vectors</em>, page 280. Note that this command is not available for all ARM cores.</td>
</tr>
<tr>
<td>Disable Interrupts</td>
<td>Ensures that only the stepped instructions will be executed. Interrupts will not be executed. This command can be used when not running at full speed and some interrupts interfere with the debugging process.</td>
</tr>
<tr>
<td>ETM Trace Settings</td>
<td>Opens the <strong>Trace Settings</strong> dialog box to configure ETM trace data generation and collection; see <em>ETM Trace Settings dialog box</em>, page 176.</td>
</tr>
<tr>
<td>ETM Trace Save</td>
<td>Opens the <strong>Trace Save</strong> dialog box to save the collected trace data to a file; see <em>Trace Save dialog box</em>, page 187.</td>
</tr>
<tr>
<td>ETM Trace Window</td>
<td>Opens the <strong>Trace window</strong> to display the collected trace data; see <strong>Trace window</strong>, page 183.</td>
</tr>
<tr>
<td>Function Trace Window</td>
<td>Opens the Function Trace window to display a subset of the trace data displayed in the Trace window; see <strong>Function Trace window</strong>, page 188.</td>
</tr>
<tr>
<td>SWO Configuration</td>
<td>Opens the <strong>SWO Configuration</strong> dialog box; see <strong>SWO Configuration dialog box</strong>, page 180.</td>
</tr>
<tr>
<td>SWO Trace Window Settings</td>
<td>Opens the <strong>SWO Trace Window Settings</strong> dialog box; see <strong>SWO Trace Window Settings dialog box</strong>, page 178.</td>
</tr>
<tr>
<td>SWO Trace Save</td>
<td>Opens the <strong>Trace Save</strong> dialog box to save the collected trace data to a file; see <strong>Trace Save dialog box</strong>, page 187.</td>
</tr>
<tr>
<td>SWO Trace Window</td>
<td>Opens the <strong>SWO Trace window</strong>; see <strong>Trace window</strong>, page 183.</td>
</tr>
<tr>
<td>Interrupt Log</td>
<td>Opens the Interrupt Log window; see <strong>Interrupt Log window</strong>, page 300.</td>
</tr>
<tr>
<td>Interrupt Log Summary</td>
<td>Opens the Interrupt Log Summary window; see <strong>Interrupt Log Summary window</strong>, page 302.</td>
</tr>
<tr>
<td>Data Log</td>
<td>Opens the Data Log window; see <strong>Data Log window</strong>, page 296.</td>
</tr>
<tr>
<td>Data Log Summary</td>
<td>Opens the Data Log Summary window; see <strong>Data Log Summary window</strong>, page 299.</td>
</tr>
<tr>
<td>Timeline</td>
<td>Opens the <strong>Timeline</strong> window; see <strong>Timeline window</strong>, page 189.</td>
</tr>
<tr>
<td>Function Profiler</td>
<td>Opens the Function Profiler window; see <strong>Function Profiler window</strong>, page 306.</td>
</tr>
<tr>
<td>Breakpoint Usage</td>
<td>Opens the <strong>Breakpoint Usage</strong> dialog box to list all active breakpoints; see <strong>Breakpoint Usage dialog box</strong>, page 280.</td>
</tr>
</tbody>
</table>

*Table 37: Commands on the J-Link menu*
1 Only available when the SWD/SWO interface is used.
2 Only available when using either ETM or J-Link with ETB.
3 Available when using either ETM or SWD/SWO.

LIVE WATCH AND USE OF DCC

The following possibilities for using live watch apply:

For Cortex-M

Access to memory or setting breakpoints is always possible during execution. The DCC (Debug Communications Channel) unit is not available.

For ARMxxx-S devices

Setting hardware breakpoints is always possible during execution.

For ARM7/ARM9 devices, including ARMxxx-S

Memory accesses must be made by your application. By adding a small program—a DCC handler—that communicates with the debugger through the DCC unit to your application, memory can be read/written during execution. Software breakpoints can also be set by the DCC handler.

Just add the files JLINKDCC_Process.c and JLINKDCC_HandleDataAbort.s located in arm\src\debugger\dcc to your project and call the JLINKDCC_Process function regularly, for example every millisecond.

In your local copy of the cstartup file, modify the interrupt vector table so that data aborts will call the JLINKDCC_HandleDataAbort handler.

TERMINAL I/O AND USE OF DCC

The following possibilities for using Terminal I/O apply:

For Cortex-M

See ITM Stimulus Ports, page 182.

For ARM7/ARM9 devices, including ARMxxx-S

DCC can be used for Terminal I/O output by adding the file arm\src\debugger\dcc\DCC_Write.c to your project. DCC_write.c overrides the library function write. Functions such as printf can then be used to output text in real time to the C-SPY Terminal I/O window.
In this case, you can disable semihosting which means that the breakpoint it uses is freed for other purposes. To disable semihosting, choose General Options>Library Configuration>Library low-level interface implementation>None.

### Debugging using the IAR C-SPY LMI FTDI driver

To use C-SPY for the FTDI JTAG interface, you should be familiar with the following details:

- The C-SPY options specific to the Luminary FTDI JTAG interface, see Setup, page 260
- The LMI FTDI menu, see The LMI FTDI menu, page 261
- The breakpoint system, see Using breakpoints in the hardware debugger systems, page 271.

### SETUP

This section describes the options that specify the LMI FTDI interface. In the IAR Embedded Workbench IDE, choose Project>Options, select the LMI FTDI category, and click the Setup tab.

![Setup](image)

**Interface**

Use this option to specify communication interface between the J-Link debug probe and the target system. Choose between:

- **JTAG** (default)
• **SWD**: uses fewer pins than JTAG. Select SWD if you want to use the serial-wire output (SWO) communication channel. Note that if you select stdout/stderr via SWO on the General Options>Library Configuration page, SWD is selected automatically. For more information about SWO settings, see SWO Trace Window Settings dialog box, page 178.

**JTAG/SWD speed**

Use the JTAG speed option to set the JTAG communication speed in kHz.

**Log communication**

Use this option to log the communication between C-SPY and the target system to a file. To interpret the result, a detailed knowledge about the communication protocol is required.

**THE LMI FTDI MENU**

When you are using the C-SPY LMI FTDI driver, the additional menu LMI FTDI appears in C-SPY.

![Figure 106: The LMI FTDI menu](image)

The following command is available on the LMI FTDI menu:

<table>
<thead>
<tr>
<th>Menu command</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Breakpoint Usage</td>
<td>Opens the Breakpoint Usage dialog box to list all active breakpoints; see Breakpoint Usage dialog box, page 280.</td>
</tr>
</tbody>
</table>

**Table 38: Commands on the LMI FTDI menu**

**Debugging using the IAR C-SPY Macraigor driver**

To use C-SPY for the Macraigor JTAG interface, you should be familiar with the following details:

• The C-SPY options specific to the Macraigor JTAG interface, see Macraigor, page 262
• The Macraigor JTAG menu, page 264
• Using breakpoints in the hardware debugger systems, page 271
• Using JTAG watchpoints, page 282.
Debugging using the IAR C-SPY Macraigor driver

**MACRAIGOR**

In the IAR Embedded Workbench IDE, choose **Project>Options** and click the **Macraigor** tab in the **Debugger** category.

**OCD interface device**

Select the device corresponding to the hardware interface you are using. Supported Macraigor JTAG interface is Macraigor **mpDemon**.

**Interface**

Use this option to specify communication interface between the J-Link debug probe and the target system. Choose between:

- **JTAG** (default)
- **SWD**; uses fewer pins than JTAG. Select SWD if you want to use the serial-wire output (SWO) communication channel. Note that if you select stdout/stderr via SWO on the **General Options>Library Configuration** page, SWD is selected automatically. For more information about SWO settings, see **SWO Trace Window Settings dialog box**, page 178.

**JTAG speed**

This option sets the JTAG speed between the JTAG interface and the ARM JTAG ICE port. The number must be in the range 1–8 and sets the factor by which the JTAG interface clock is divided when generating the scan clock.

The mpDemon interface might require a higher setting such as 2 or 3, that is, a lower speed.
TCP/IP
Use this option to set the IP address of a JTAG interface connected to the Ethernet/LAN port.

Port
Use the Port drop-down list to select which serial port or parallel port on the host computer to use as communication link. Select the host port to which the JTAG interface is connected.

In the case of parallel ports, you should normally use LPT1 if the computer is equipped with a single parallel port. Note that a laptop computer might in some cases map its single parallel port to LPT2 or LPT3. If possible, configure the parallel port in EPP mode because this mode is fastest; bidirectional and compatible modes will work but are slower.

Baud rate
Set the serial communication speed using the Baud rate drop-down list.

Hardware reset
A software reset of the target does not change the settings of the target system; it only resets the program counter to its reset state. Normally, a C-SPY reset is a software reset only. If you use the Hardware reset option, C-SPY will generate an initial hardware reset when the debugger is started. This is performed once before download, and if the option Use flash loader(s) is selected, also once after flash download, see Figure 94, Debugger startup when debugging code in flash, page 241, and Figure 95, Debugger startup when debugging code in RAM, page 242.

Hardware resets can be a problem if the low-level setup of your application is not complete. If low-level setup does not set up memory configuration and clocks, the application will not work after a hardware reset. To handle this in C-SPY, the setup macro function execUserReset() is suitable. For a similar example where execUserPreload() is used, see Remapping memory, page 126.

JTAG scan chain with multiple targets
If there is more than one device on the JTAG scan chain, each device has to be defined, and you have to state which device you want to connect to. The syntax is:

<0>@dev0, dev1, dev2, dev3, ...

where 0 is the TAP number of the device to connect to, and dev0 is the nearest TDO pin on the Macraigor JTAG interface.
Debug handler address
Use this option to specify the location—the memory address—of the debug handler used by Intel XScale devices. To save memory space, you should specify an address where a small portion of cache RAM can be mapped, which means the location should not contain any physical memory. Preferably, find an unused area in the lower 16-Mbyte memory and place the handler address there.

Log communication
Use this option to log the communication between C-SPY and the target system to a file. To interpret the result, detailed knowledge of the JTAG interface is required.

BREAKPOINTS
In the IAR Embedded Workbench IDE, choose Project>Options, select the Macraigor category, and click the Breakpoints tab.
For details, see Breakpoints options, page 272.

THE MACRAIGOR JTAG MENU
When you are using the Macraigor driver, the additional menu JTAG appears in C-SPY.

<table>
<thead>
<tr>
<th>Menu command</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Watchpoints</td>
<td>Opens a dialog box for setting watchpoints, see JTAG watchpoints dialog box, page 283.</td>
</tr>
<tr>
<td>Vector Catch</td>
<td>Opens a dialog box for setting a breakpoint directly on a vector in the interrupt vector table, see Breakpoints on vectors, page 280. Note that this command is not available for all ARM cores.</td>
</tr>
<tr>
<td>Breakpoint Usage</td>
<td>Opens the Breakpoint Usage dialog box to list all active breakpoints; see Breakpoint Usage dialog box, page 280.</td>
</tr>
</tbody>
</table>

Table 39: Commands on the Macraigor JTAG menu
Debugging using the RDI driver

To use C-SPY for the RDI interface, you should be familiar with these details:

- The C-SPY options that specify the RDI interface, see RDI, page 265
- The RDI menu, see RDI menu, page 267
- The ETM trace mechanism, see Using trace, page 169.

For detailed information about the RDI interface and how to get started, see the rdi_quickstart.htm file, available in the arm\doc subdirectory.

RDI

To set RDI options, choose Project>Options and click the RDI tab in the Debugger category.

With the options on the RDI page you can use JTAG interfaces compliant with the ARM Ltd RDI 1.5.1 specification. One example of such an interface is the ARM RealView Multi-ICE JTAG interface.

Manufacturer RDI driver

This is the file path to the RDI driver DLL file provided with the JTAG pod.

Allow hardware reset

A software reset of the target does not change the settings of the target system; it only resets the program counter to its reset state.

Use the Allow Hardware Reset option to allow the emulator to perform a hardware reset of the target.
You should only allow hardware resets if the low-level setup of your application is complete. If the low-level setup does not set up memory configuration and clocks, the application will not work after a hardware reset. To handle this in C-SPY, the setup macro function execUserReset() is suitable. For a similar example where execUserPreload() is used, see Remapping memory, page 126.

Note: This option requires that hardware resets are supported by the RDI driver you are using.

ETM trace

Use this option to enable the debugger to use and display ETM trace. When the option is selected, the debugger will check that the connected JTAG interface supports RDI ETM and that the target processor supports ETM. If the connected hardware supports ETM, the RDI menu will contain the following commands:

- ETM Trace Window, see Trace window, page 183
- Trace Settings, see ETM Trace Settings dialog box, page 176
- Trace Save, see Trace Save dialog box, page 187.

Catch exceptions

Enabling the catch of an exception will cause the exception to be treated as a breakpoint. Instead of handling the exception as defined by the running program, the debugger will stop.

The ARM core exceptions that can be caught are:

<table>
<thead>
<tr>
<th>Exception</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Reset</td>
<td>Reset</td>
</tr>
<tr>
<td>Undef</td>
<td>Undefined instruction</td>
</tr>
<tr>
<td>SWI</td>
<td>Software interrupt</td>
</tr>
<tr>
<td>Data</td>
<td>Data abort (data access memory fault)</td>
</tr>
<tr>
<td>Prefetch</td>
<td>Prefetch abort (instruction fetch memory fault)</td>
</tr>
<tr>
<td>IRQ</td>
<td>Normal interrupt</td>
</tr>
<tr>
<td>FIQ</td>
<td>Fast interrupt</td>
</tr>
</tbody>
</table>

Table 40: Catching exceptions

Log RDI communication

Use this option to log the communication between C-SPY and the target system to a file. To interpret the result, detailed knowledge of the RDI interface is required.
RDI MENU

When you are using the C-SPY J-Link driver, the additional menu RDI appears in C-SPY.

![RDI menu](image)

These commands are available on the RDI menu:

<table>
<thead>
<tr>
<th>Menu command</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Configure</td>
<td>Opens a dialog box that originates from the RDI driver vendor. For information about details in this dialog box, refer to the driver documentation.</td>
</tr>
<tr>
<td>ETM Trace Window</td>
<td>Opens the Trace window to display the captured trace data; see Trace window, page 183.</td>
</tr>
<tr>
<td>Trace Settings</td>
<td>Opens the Trace Settings dialog box to configure the ETM trace; see ETM Trace Settings dialog box, page 176.</td>
</tr>
<tr>
<td>Trace Save</td>
<td>Opens the Trace Save dialog box to save the captured trace data to a file; see Trace Save dialog box, page 187.</td>
</tr>
<tr>
<td>Breakpoint Usage</td>
<td>Opens the Breakpoint Usage dialog box to list all active breakpoints; see Breakpoint Usage dialog box, page 280.</td>
</tr>
</tbody>
</table>

Table 41: Commands on the RDI menu

Note: To get the default settings in the configuration dialog box, it is for some RDI drivers necessary to just open and close the dialog box even though you do no need any specific settings for your project.

Debugging using the ST-Link driver

To use C-SPY for the ST-Link JTAG probe, you should be familiar with these details:

- The options specific to the C-SPY ST-Link driver, see ST-Link, page 268
- The ST-Link menu, see ST-Link menu, page 268
- The breakpoint system, see Using breakpoints in the hardware debugger systems, page 271.
Debugging using the ST-Link driver

**ST-Link**

This section describes the options that specify the ST-Link probe. In the IAR Embedded Workbench IDE, choose **Project>Options**, select the **ST-Link** category.

![ST-Link Interface Options](image)

**Interface**

Use this option to specify the communication interface between the ST-Link debug probe and the target system. Choose between:

- **JTAG** (default)
- **SWD**; uses fewer pins than JTAG.

**ST-Link Menu**

When you are using the C-SPY ST-Link driver, the additional menu **ST-Link** appears in C-SPY.

![ST-Link Menu](image)

**Table 42: Commands on the ST-Link menu**

<table>
<thead>
<tr>
<th>Menu command</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Breakpoint Usage</td>
<td>Opens the Breakpoint Usage dialog box to list all active breakpoints; see <strong>Breakpoint Usage</strong> dialog box, page 280.</td>
</tr>
</tbody>
</table>

This command is available on the **ST-Link** menu:
Debugging using a third-party driver

You can load other debugger drivers than those supplied with the IAR Embedded Workbench.

**THIRD-PARTY DRIVER**

To set options for the third-party driver, choose **Project>Options** and click the **Third-party Driver** tab in the **Debugger** category.

The **Third-Party Driver** options are used for loading any driver plugin provided by a third-party vendor. These drivers must be compatible with the IAR debugger driver specification.

**IAR debugger driver plugin**

Enter the file path to the third-party driver plugin DLL file in this text box, or browse to the driver DLL file using the browse button.

**Log communication**

Use this option to log the communication between C-SPY and the target system to a file. To interpret the result, detailed knowledge of the interface is required. This option can be used if it is supported by the third-party driver.
Debugging using a third-party driver
Analyzing your program using driver-specific tools

This chapter describes the additional features provided by the C-SPY® hardware debugger systems. The chapter contains these sections:

- Using breakpoints in the hardware debugger systems
- Using JTAG watchpoints
- Using J-Link trace triggers and trace filters
- Using the data and interrupt logging systems
- Using the profiler.

For related information, see also Using trace, page 169.

Using breakpoints in the hardware debugger systems

This section provides details about breakpoints that are specific to the different C-SPY drivers. The following is described:

- Available number of breakpoints, page 272
- Breakpoints options, page 272
- Code breakpoints dialog box, page 274
- Data breakpoints dialog box, page 275
- Data Log breakpoints dialog box, page 278
- Breakpoint Usage dialog box, page 280
- Breakpoints on vectors, page 280
- Setting breakpoints in __ramfunc declared functions, page 281.

For information about the different methods for setting breakpoints and the facilities for monitoring breakpoints, see Using breakpoints, page 141.
AVAILABLE NUMBER OF BREAKPOINTS

Normally when you set a breakpoint, C-SPY sets two breakpoints for internal use. This can be unacceptable if you debug on hardware where a limited number of hardware breakpoints are available. For more information about breakpoint consumers, see Breakpoint consumers, page 148.

Note: Cortex devices support additional hardware breakpoints.

Exceeding the number of available hardware breakpoints will cause the debugger to single step. This will significantly reduce the execution speed.

You can prevent the debugger from using breakpoints in these situations. In the first case, by deselecting the C-SPY option Run to. In the second case, you can deselect the Semihosted or the IAR breakpoint option.

When you use the Stack window, it requires one hardware breakpoint in some situations, see Stack pointer(s) not valid until reaching, page 391.

BREAKPOINTS OPTIONS

For the following hardware debugger systems it is possible to set some driver-specific breakpoint options before you start C-SPY:

- GDB Server
- J-Link/J-Trace JTAG interface
- Macraigor JTAG interface.

In the IAR Embedded Workbench IDE, choose Project>Options, select the category specific to the debugger system you are using, and click the Breakpoints tab.

![Figure 114: Breakpoints options](image_url)
Analyzing your program using driver-specific tools

Default breakpoint type
Use this option to select the type of breakpoint resource to be used when setting a breakpoint. Choose between:

- **Auto**
  The C-SPY debugger will use a software breakpoint; if this is not possible, a hardware breakpoint will be used. The debugger will use read/write sequences to test for RAM; in that case, a software breakpoint will be used. The Auto option works for most applications. However, there are cases when the performed read/write sequence will make the flash memory malfunction. In that case, use the **Hardware** option.

- **Hardware**
  Hardware breakpoints will be used. If it is not possible to use a hardware breakpoint, no breakpoint will be set.

- **Software**
  Software breakpoints will be used. If it is not possible to use a software breakpoint, no breakpoint will be set.

Restore software breakpoints at
Use this option to restore automatically any breakpoints that were destroyed during system startup.

This can be useful if you have an application that is copied to RAM during startup and is then executing in RAM. This can, for example, be the case if you use the `__ramfunc` declared functions in your application.

In this case, all breakpoints will be destroyed during the RAM copying when the C-SPY debugger starts. By using the **Restore software breakpoints at** option, C-SPY will restore the destroyed breakpoints.

Use the text field to specify the location in your application at which point you want C-SPY to restore the breakpoints.

Catch exceptions
This option is supported by the C-SPY J-Link/J-Trace driver only.

Use this option to set a breakpoint directly on a vector in the interrupt vector table, without using a hardware breakpoint. This option is available for ARM9, Cortex-R4, and Cortex-M3 devices. The settings you make will work as default settings for the project. However, you can override these default settings during the debug session by using the **Vector Catch** dialog box, see *Breakpoints on vectors*, page 280.

The settings you make will be preserved during debug sessions.
Using breakpoints in the hardware debugger systems

**CODE BREAKPOINTS DIALOG BOX**

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.

To set a code breakpoint, right-click in the Breakpoints window and choose **New Breakpoint>Code** on the context menu. To modify an existing breakpoint, select it in the Breakpoints window and choose **Edit** on the context menu.

The **Code** breakpoints dialog box appears.

![Figure 115: Code breakpoints dialog box](image)

**Break At**

Specify the location of the breakpoint in the **Break At** text box. Alternatively, click the **Edit browse** button to open the **Enter Location** dialog box; see **Enter Location dialog box**, page 345.

**Breakpoint type**

Use the **Breakpoint type** options to override the default breakpoint type. Select the **Override default** check box and choose between the **Software** and **Hardware** options.

You can specify the breakpoint type for these C-SPY drivers:

- GDB Server
- J-Link/J-Trace JTAG interface
Macraigor JTAG interface.

**Action**

You can optionally connect an action to a 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**

You can specify simple and complex conditions.

<table>
<thead>
<tr>
<th>Conditions</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Expression</td>
<td>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 must be fulfilled before a break occurs (integer).</td>
</tr>
</tbody>
</table>

*Table 43: Breakpoint conditions*

**DATA BREAKPOINTS DIALOG BOX**

The options for setting data breakpoints are available from the context menu that appears when you right-click in the Breakpoints window. On the context menu, choose **New Breakpoint>Data** to set a new breakpoint. Alternatively, to modify an existing breakpoint, select a breakpoint in the Breakpoint window and choose **Edit** on the context menu.
The **Data** breakpoints dialog box appears.

![Data breakpoints dialog box](image.png)

**Figure 116: Data breakpoints dialog box**

**Note:** Setting data breakpoints is possible for the:
- GDB Server
- J-Link/J-Trace JTAG interface
- Macraigor JTAG interface
- Luminary FTDI JTAG interface
- RDI drivers.

**Break At**
Specify the location for the breakpoint in the **Break At** text box. Alternatively, click the **Edit** browse button to open the **Enter Location** dialog box; see *Enter Location dialog box*, page 345.

**Access Type**
Use the options in the **Access Type** area to specify the type of memory access that triggers data breakpoints.

<table>
<thead>
<tr>
<th>Memory Access type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Read/Write</td>
<td>Read from or write to location.</td>
</tr>
</tbody>
</table>

*Table 44: Memory Access types*
Analyzing your program using driver-specific tools

Size

The Size option controls the size of the address range to be traced. There are two different ways to specify the size:

- **Auto**, the size will be set automatically. This can be useful if Break at contains a variable.
- **Manual**, you specify the size of the breakpoint range manually in the Size text box.

Trigger range

Trigger range shows the requested range and the effective range to be covered by the trace. The range suggested is either within or exactly the area specified by the Break at and the Size options.

Extend to cover requested range

For data structures that do not fit the size of the possible breakpoint ranges supplied by the hardware breakpoint unit, for example three bytes, the breakpoint range will not cover the whole data structure. In this case, use this option to make the breakpoint be extended so that the whole data structure is covered. Note that the breakpoint range will be extended beyond the size of the data structure.

Match data

To match the data accessed, click Enable in the Match data area. Use the Match data options in combination with the Read/Write, Read, or Write access types for data. This option can be useful when you want a trigger when a variable has a certain value. Use the Mask option to specify which part of the value to match (word, halfword, or byte).

The Match data options are only available for J-Link/J-Trace and when using an ARM7/9 or a Cortex-M device.

**Note**: For Cortex-M devices, only one breakpoint with Match data can be set. Such a breakpoint uses two breakpoint resources.

<table>
<thead>
<tr>
<th>Memory Access type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Read</td>
<td>Read from location.</td>
</tr>
<tr>
<td>Write</td>
<td>Write to location.</td>
</tr>
</tbody>
</table>

Note: Data breakpoints never stop execution within a single instruction. They are recorded and reported after the instruction is executed.
DATA LOG BREAKPOINTS DIALOG BOX

Data Log breakpoints are triggered when data is accessed at the specified location. If you have set a log breakpoint on a specific address or a range, a log message is displayed in the SWO Trace window for each access to that location. A log message can also be displayed in the Data Log window, if that window is enabled. However, these log messages require that you have set up trace data in the SWO Settings dialog box, see SWO Trace Window Settings dialog box, page 178.

The options for setting data log breakpoints are available from the context menu that appears when you right-click in the Breakpoints window or in the Memory window. On the context menu, choose New Breakpoint>Data Log to set a new breakpoint. Alternatively, to modify an existing breakpoint, select a breakpoint in the Breakpoint window and choose Edit on the context menu.

The Data Log breakpoints dialog box appears.

![Data Log breakpoints dialog box](image)

Note: Setting Data Log breakpoints is possible only for Cortex-M with SWO using the J-Link debug probe.
Access type

Use these options to specify the type of memory access that triggers a data log breakpoint.

<table>
<thead>
<tr>
<th>Memory Access type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Read/Write</td>
<td>Read from or write to location.</td>
</tr>
<tr>
<td>Read</td>
<td>Read from location; for Cortex-M3 revision 2 devices only.</td>
</tr>
<tr>
<td>Write</td>
<td>Write to location; for Cortex-M3 revision 2 devices only.</td>
</tr>
</tbody>
</table>

Table 45: Memory Access types for Data Log breakpoints

Size

The Size option controls the size of the address range to be traced. There are two different ways to specify the size:

- Auto, the size will be set automatically. This can be useful if Trigger at contains a variable.
- Manual, you specify the size of the breakpoint range manually in the Size text box.

Trigger range

Trigger range shows the requested range and the effective range to be covered by the trace. The range suggested is either within or exactly the area specified by the Trigger at and the Size options.

Extend to cover requested range

For data structures that do not fit the size of the possible breakpoint ranges supplied by the hardware breakpoint unit, for example three bytes, the breakpoint range will not cover the whole data structure. In this case, use the option Extend to cover requested range to make the range be extended so that the whole data structure is covered. Note that the breakpoint range will be extended beyond the size of the data structure, which might cause false triggers at adjacent data.
BREAKPOINT USAGE DIALOG BOX

The Breakpoint Usage dialog box—available from the driver-specific menu—lists all active breakpoints.

In addition to listing all breakpoints that you have defined, this dialog box also lists the internal breakpoints that the debugger is using.

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

For more information, see Viewing all breakpoints, page 146.

BREAKPOINTS ON VECTORS

To set the breakpoint directly on a vector in the interrupt vector table, choose the Vector Catch command from the J-Link menu.

Figure 119: The Vector Catch dialog box—for ARM®/Cortex-R4 versus for Cortex-M3
Usage

Use the Vector Catch dialog box to set a breakpoint directly on a vector in the interrupt vector table, without using a hardware breakpoint. You can set breakpoints on vectors for ARM9, Cortex-R4, and Cortex-M3 devices. Note that the settings you make here will not be preserved between debug sessions.

Follow these steps:

1. Select the correct device. Before starting C-SPY, choose Project>Options and select the General Options category. Choose the appropriate device from the Processor variant drop-down list available on the Target page.

2. Start C-SPY.

3. Choose J-Link>Vector Catch. By default, vectors are selected according to your settings on the Breakpoints options page, see Breakpoints, page 257.

4. In the Vector Catch dialog box, select the vector you want to set a breakpoint on, and click OK. The breakpoint will only be triggered at the beginning of the exception.

Note: The Vector Catch dialog box is only available for the:

- J-Link/J-Trace JTAG interface
- Macraigor JTAG interface.

Note: For the J-Link/J-Trace driver and for RDI drivers, it is also possible to set breakpoints directly on a vector already in the options dialog box, see Setup, page 252 and RDI, page 265.

SETTING BREAKPOINTS IN __ramfunc DECLARED FUNCTIONS

To set a breakpoint in a __ramfunc declared function, the program execution must have reached the main function. The system startup code moves all __ramfunc declared functions from their stored location—normally flash memory—to their RAM location, which means the __ramfunc declared functions are not in their proper place and breakpoints cannot be set until you have executed up to the main function. Use the Restore software breakpoints option to solve this problem, see Restore software breakpoints at, page 273.

In addition, breakpoints in __ramfunc declared functions added from the editor have to be disabled prior to invoking C-SPY and prior to exiting a debug session.
Using JTAG watchpoints

The C-SPY J-Link/J-Trace driver and the C-SPY Macraigor driver can take advantage of the JTAG watchpoint mechanism in ARM7/9 cores. The watchpoints are defined using the J-Link>Watchpoints and the JTAG>Watchpoints menu commands, respectively.

THE WATCHPOINT MECHANISM

The watchpoints are implemented using the functionality provided by the ARM EmbeddedICE™ macrocell. The macrocell is part of every ARM core that supports the JTAG interface.

The EmbeddedICE watchpoint comparator compares the address bus, data bus, CPU control signals and external input signals with the defined watchpoint in real time. When all defined conditions are true the program will break.

The watchpoints are implicitly used by C-SPY to set code breakpoints in the application. When setting breakpoints in read/write memory, only one watchpoint is needed by the debugger. When setting breakpoints in read-only memory, one watchpoint is needed for each breakpoint. Because the macrocell only implements two hardware watchpoints, the maximum number of breakpoints in read-only memory is two.

For a more detailed description of the ARM JTAG watchpoint mechanism, refer to the following documents from Advanced RISC Machines Ltd:

- Application Note 28, The ARM7TDMI Debug Architecture.
JTAG WATCHPOINTS DIALOG BOX

The JTAG Watchpoints dialog box is opened from the driver-specific menu:

![JTAG Watchpoints dialog box](image)

The JTAG Watchpoints dialog box makes it possible to directly control the two hardware watchpoint units. If the number of needed watchpoints (including implicit watchpoints used by the breakpoint system) exceeds two, an error message will be displayed when you click the OK button. This check is also performed for the C-SPY GO button.

**Address**

Use these options to specify the address to watch for. In the Value text box, enter an address or a C-SPY expression that evaluates to an address. Alternatively, you can select an address you have previously watched for from the drop-down list. For detailed information about C-SPY expressions, see C-SPY expressions, page 135.

Use the Mask box to qualify each bit in the value. A zero bit in the mask will cause the corresponding bit in the value to be ignored in the comparison.

The Address Bus Pattern field shows the bit pattern to be used by the address comparator. Ignored bits as specified in the mask are shown as x.

To match any address, enter 0 in the mask. Note that the mask values are inverted with respect to the notation used in the ARM hardware manuals.
Using JTAG watchpoints

### Access Type

Use these options to define the access type of the data to watch for:

<table>
<thead>
<tr>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Any</td>
<td>Matches any access type</td>
</tr>
<tr>
<td>OP Fetch</td>
<td>Operation code (instruction) fetch</td>
</tr>
<tr>
<td>Read</td>
<td>Data read</td>
</tr>
<tr>
<td>Write</td>
<td>Data write</td>
</tr>
<tr>
<td>R/W</td>
<td>Data read or write</td>
</tr>
</tbody>
</table>

Table 46: Data access types

### Data

Use these options to specify the data to watch for. Data accesses can be made as **Byte**, **Halfword** or **Word**. If the **Any Size** option is used the mask should be set in the interval **0 to 0xFF** since higher bits on the data bus may contain random data depending on the instruction.

Enter a value or a C-SPY expression in the **Value** box. Alternatively, you can select a value you have previously watched for from the drop-down list. For detailed information about C-SPY expressions, see **C-SPY expressions**, page 135.

Use the **Mask** box to qualify each bit in the value. A zero bit in the mask will cause the corresponding bit in the value to be ignored in the comparison.

The **Data Bus Pattern** field shows the bit pattern to be used by the data comparator. Ignored bits as specified in the mask are shown as **x**.

To match any address, enter **0** in the mask. Note that the mask values are inverted with respect to the notation used in the ARM hardware manuals.

### Extern

Use these options to define the state of the external input:

<table>
<thead>
<tr>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Any</td>
<td>The state is ignored.</td>
</tr>
<tr>
<td>0</td>
<td>The state is low.</td>
</tr>
<tr>
<td>1</td>
<td>The state is high.</td>
</tr>
</tbody>
</table>
Analyzing your program using driver-specific tools

Mode
Use these options to define the CPU mode that must be active for a match:

<table>
<thead>
<tr>
<th>Mode</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>User</td>
<td>The CPU must run in USER mode</td>
</tr>
<tr>
<td>Non User</td>
<td>The CPU must run in one of the SYSTEM SVC, UND, ABORT, IRQ or FIQ modes</td>
</tr>
<tr>
<td>Any</td>
<td>The CPU mode is ignored</td>
</tr>
</tbody>
</table>

Table 47: CPU modes

Break Condition
Use these options to specify how to use the defined watchpoints:

<table>
<thead>
<tr>
<th>Break condition</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Normal</td>
<td>The two watchpoints are used individually (OR).</td>
</tr>
<tr>
<td>Range</td>
<td>Both watchpoints are combined to cover a range where watchpoint 0 defines the start of the range and watchpoint 1 the end of the range. Selectable ranges are restricted to being powers of 2.</td>
</tr>
<tr>
<td>Chain</td>
<td>A trigger of watchpoint 1 will arm watchpoint 0. A program break will then occur when watchpoint 0 is triggered.</td>
</tr>
</tbody>
</table>

Table 48: Break conditions

For example, to cause a trigger for accesses in the range 0x20–0xFF:

1. Set Break Condition to Range.
2. Set the address value of watchpoint 0 to 0 and the mask to 0xFF.
3. Set the address value of watchpoint 1 to 0 and the mask to 0x1F.

Using J-Link trace triggers and trace filters
This section gives you information about using the J-Link trace triggers and trace filters. More specifically, these topics are covered:

- Requirements for using the J-Link trace triggers and trace filters, page 286
- Reasons for using the J-Link trace triggers and trace filters, page 286
- How to use the J-Link trace triggers and trace filters, page 286
- Related reference information, page 287.
REQUIREMENTS FOR USING THE J-LINK TRACE TRIGGERS AND TRACE FILTERS

The trace triggering and trace filtering features are available only for J-Trace and when using an ARM7/9 or Cortex-M device.

REASONS FOR USING THE J-LINK TRACE TRIGGERS AND TRACE FILTERS

By using trace trigger and trace filter conditions, you can select the interesting parts of your source code and use the trace buffer in J-Trace more efficiently. Trace triggers—Trace Start and Trace Stop breakpoints—specify for example a code section for which you want to collect trace data. A trace filter specifies conditions that, when fulfilled, activate the trace data collection during execution.

For ARM7/9 devices, you can specify up to 16 trace triggers and trace filters in total, of which 8 can be trace filters.

For Cortex-M devices, you can specify up to 4 trace triggers and trace filters in total.

HOW TO USE THE J-LINK TRACE TRIGGERS AND TRACE FILTERS

1. Use the Trace Start dialog box to set a start condition—a start trigger—to start collecting trace data.

2. Use the Trace Stop dialog box to set a stop condition—a stop trigger—to stop collecting trace data.

3. Optionally, set additional conditions for the trace data collection to continue. Then set one or more trace filters, using the Trace Filter dialog box.

4. If needed, set additional trace start or trace stop conditions.

5. Enable the Trace window and start the execution.

6. Stop the execution.

7. You can view the trace data in the Trace window and in browse mode also in the Disassembly window, where also the trace marks for your trace triggers and trace filters are visible.

   If you have set a trace filter, the trace data collection is performed while the condition is true plus some further instructions. When viewing the trace data and looking for a certain data access, remember that the access took place one instruction earlier.
RELATED REFERENCE INFORMATION

To use trace triggers and trace filters, you might need reference information about these dialog boxes:

- Trace Start breakpoints dialog box, page 287
- Trace Stop breakpoints dialog box, page 289
- Trace Filter breakpoints dialog box, page 292.

TRACE START BREAKPOINTS DIALOG BOX

To set the conditions that determine when to start collecting trace data, right-click in the Breakpoints window and choose New Breakpoint > Trace Start on the context menu. You can also right-click in the editor window or the Disassembly window, and then choose Toggle Breakpoint (Trace Start). The Trace Start dialog box appears:

![Trace Start breakpoints dialog box](image)

When the trace condition is triggered, the trace data collection is started.

**Trigger at**

Use this option to specify the starting point of the code section for which you want to collect trace data. You can specify a variable name, an address, or a cycle counter value in the text box.
Using J-Link trace triggers and trace filters

Size
The **Size** option controls the size of the address range, that when reached, will trigger the start of the trace data collection. There are two different ways to specify the size:

- **Auto**, the size will be set automatically. This can be useful if **Trigger at** contains a variable.
- **Manual**, you specify the size of the breakpoint range manually in the text box.

Trigger range
**Trigger range** shows the requested range and the effective range to be covered by the trace data collection. The range suggested is either within or exactly the area specified by the **Trigger at** and the **Size** options.

**Extend to cover requested range**
For data structures that do not fit the size of the possible ranges supplied by the hardware breakpoint unit, for example three bytes, the range will not cover the whole data structure. In this case, use the option **Extend to cover requested range** to make the range be extended so that the whole data structure is covered. Note that the range will be extended beyond the size of the data structure, which might cause false triggers at adjacent data.

This option is not enabled for ARM7/9 devices because the range for such devices will always cover the whole data structure.

Access type
Use the options in the **Access type** area to specify the type of memory access that triggers the trace data collection.

<table>
<thead>
<tr>
<th>Memory Access type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Read/Write</td>
<td>Read from or write to location.</td>
</tr>
<tr>
<td>Read</td>
<td>Read from location.</td>
</tr>
<tr>
<td>Write</td>
<td>Write to location.</td>
</tr>
<tr>
<td>OP-Fetch</td>
<td>At execution address.</td>
</tr>
<tr>
<td>Cycle</td>
<td>The number of cycle counter cycles at a specific point in time, counted from where the execution started. This option is only available for Cortex-M devices.</td>
</tr>
</tbody>
</table>

*Table 49: Trace Start access types*
**Match data**

To match the data accessed, click **Enable** in the **Match data** area. Use the **Match data** options in combination with the Read/Write, Read, or Write access types for data. This option can be useful when you want a trigger when a variable access has a certain value. Use the **Mask** option to specify which part of the value to match (word, halfword, or byte).

The **Match data** options are only available for J-Link/J-Trace and when using a Cortex-M device.

**Note:** For Cortex-M devices, only one breakpoint with **Match data** can be set. Such a breakpoint uses two breakpoint resources.

**Link condition**

Use the **Link condition** options **AND** and **OR** to specify how trace conditions are combined. When combining a condition that has the link condition **AND** with a condition that has the link condition **OR**, **AND** has precedence. The option **Inverse** inverts the trace condition and is individual for each trace filter condition. If one trace start or stop condition is inverted, all others will be too. An inverted trace start or stop condition means that the trace data collection is performed everywhere except for this section of the application code.

For ARM7/9 devices, trace filters are combined using the OR algorithm. Use the **Inverse** option to invert the trace filter; all trace filters are affected. The trace filter will be combined with the start and stop triggers, if any, using the AND algorithm.

**TRACE STOP BREAKPOINTS DIALOG BOX**

To set the conditions that determine when to stop collecting trace data, right-click in the Breakpoints window and choose **New Breakpoint**>**Trace Stop** on the context menu.
You can also right-click in the editor window or the Disassembly window, and then choose **Toggle Breakpoint (Trace Stop)**. The **Trace Stop** dialog box appears:

![Trace Stop dialog box](image)

When the trace condition is triggered, the trace data collection is performed for some further instructions, and then the collection is stopped.

**Trigger at**

Use this option to specify the stopping point of the code section for which you want to collect trace data. You can specify a variable name, an address, or a cycle counter value in the text box.

**Size**

The **Size** option controls the size of the address range, that when reached, will trigger the stop of the trace data collection. There are two different ways to specify the size:

- **Auto**, the size will be set automatically. This can be useful if **Trigger at** contains a variable.
- **Manual**, you specify the size of the breakpoint range manually in the text box.

**Trigger range**

**Trigger range** shows the requested range and the effective range to be covered by the trace data collection. The range suggested is either within or exactly the area specified by the **Trigger at** and the **Size** options.
Extend to cover requested range

For data structures that do not fit the size of the possible ranges supplied by the hardware breakpoint unit, for example three bytes, the range will not cover the whole data structure. In this case, use the option Extend to cover requested range to make the range be extended so that the whole data structure is covered. Note that the range will be extended beyond the size of the data structure, which might cause false triggers at adjacent data.

This option is not enabled for ARM7/9 devices because the range will always cover the whole data structure.

Access type

Use the options in the Access type area to specify the type of memory access that triggers the trace data collection.

<table>
<thead>
<tr>
<th>Memory Access type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Read/Write</td>
<td>Read from or write to location.</td>
</tr>
<tr>
<td>Read</td>
<td>Read from location.</td>
</tr>
<tr>
<td>Write</td>
<td>Write to location.</td>
</tr>
<tr>
<td>OP-Fetch</td>
<td>At execution address.</td>
</tr>
<tr>
<td>Cycle</td>
<td>The number of cycle counter cycles at a specific point in time, counted from where the execution started. This option is only available for Cortex-M devices.</td>
</tr>
</tbody>
</table>

Table 50: Memory Access types

Match data

To match the data accessed, click Enable in the Match data area. Use the Match data options in combination with the Read/Write, Read, or Write access types for data. This option can be useful when you want a trigger when a variable access has a certain value. Use the Mask option to specify which part of the value to match (word, halfword, or byte).

The Match data options are only available for J-Link/J-Trace and when using a Cortex-M device.

Note: For Cortex-M devices, only one breakpoint with Match data can be set. Such a breakpoint uses two breakpoint resources.

Link condition

Use the Link condition options AND and OR to specify how trace conditions are combined. When combining a condition that has the link condition AND with a
condition that has the link condition OR, AND has precedence. The option Inverse inverts the trace condition and is individual for each trace filter condition. If one start or stop condition is inverted, all others will be too. An inverted trace start or stop condition means that trace data collection is performed everywhere except for this section of the application code.

For ARM7/9 devices, trace filters are combined using the OR algorithm. Use the Inverse option to invert the trace filter; all trace filters are affected. The trace filter will be combined with the start and stop triggers, if any, using the AND algorithm.

**TRACE FILTER BREAKPOINTS DIALOG BOX**

To set a trace filter breakpoint, right-click in the Breakpoints window and choose New Breakpoint>Trace Filter on the context menu. You can also right-click in the editor window or the Disassembly window, and then choose Toggle Breakpoint (Trace Filter). The Trace Filter dialog box appears:

![Figure 123: Trace Filter breakpoints dialog box](image)

**Trigger at**

Use this option to specify the code section for which you want to collect trace data. You can specify a variable name, an address, or a cycle counter value in the text box.
Size
The Size option controls the size of the address range where filtered trace is active.
There are two different ways to specify the size:

- **Auto**, the size will be set automatically. This can be useful if Trigger at contains a variable.
- **Manual**, you specify the size of the range manually in the text box.

Trigger range
Trigger range shows the requested range and the effective range to be covered by the filtered trace data collection. The range suggested is either within or exactly the area specified by the Trigger at and the Size options.

Extend to cover requested range
For data structures that do not fit the size of the possible ranges supplied by the hardware breakpoint unit, for example three bytes, the range will not cover the whole data structure. In this case, use the option **Extend to cover requested range** to make the range be extended so that the whole data structure is covered. Note that the range will be extended beyond the size of the data structure, which might cause false triggers at adjacent data.

This option is not enabled for ARM7/9 devices because the range will always cover the whole data structure.

Access type
Use the options in the Access type area to specify the type of memory access that activates the trace data collection.

<table>
<thead>
<tr>
<th>Memory Access type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Read/Write</td>
<td>Read from or write to location.</td>
</tr>
<tr>
<td>Read</td>
<td>Read from location.</td>
</tr>
<tr>
<td>Write</td>
<td>Write to location.</td>
</tr>
<tr>
<td>OP-Fetch</td>
<td>At execution address.</td>
</tr>
<tr>
<td>Cycle</td>
<td>The number of cycle counter cycles at a specific point in time, counted from where the execution started. This option is only available for Cortex-M devices.</td>
</tr>
</tbody>
</table>

Table 51: Memory Access types
Match data
To match the data accessed, click Enable in the Match data area. Use the Match data options in combination with the Read/Write, Read, or Write access types for data. This option can be useful when you want the trace to be active when a variable access has a certain value. Use the Mask option to specify which part of the value to match (word, halfword, or byte).

The Match data options are only available for J-Link/J-Trace and when using a Cortex-M device.

Note: For Cortex-M devices, only one Trace Filter breakpoint with Match data can be set. Such a breakpoint uses two breakpoint resources.

Link condition
For Cortex-M devices, use the Link condition options AND and OR to specify how trace conditions are combined. When combining a condition that has the link condition AND with a condition that has the link condition OR, AND has precedence. The option Inverse inverts the trace condition and is individual for each trace filter condition. An inverted link condition means that the trace data collection is active everywhere except for when the trace conditions are fulfilled.

For ARM7/9 devices, trace filters are combined using the OR algorithm. Use the Inverse option to invert the trace filter; all trace filters are affected. The trace filter will be combined with the start and stop triggers, if any, using the AND algorithm.

Using the data and interrupt logging systems
This section gives you information about using the data and interrupt logging systems. More specifically, these topics are covered:

- Requirements for using the logging systems, page 294
- Reasons for using the logging systems, page 295
- How to use the logging systems, page 295
- Related reference information, page 296.

Requirements for using the logging systems
To use the data and interrupt logging system, you need:

- A J-Link debug probe
- An SWD interface between the J-Link debug probe and the target system.
REASONS FOR USING THE LOGGING SYSTEMS

You can use these systems to log:

- Accesses to up to four different memory locations or areas. The logs are displayed in the Data Log window and a summary is available in the Data Log Summary window.
- Entrances to and exits from interrupts. The logs are displayed in the Interrupt Log window and a summary is available in the Interrupt Log Summary window. The Interrupt Graph window provides a graphical view of the interrupt events during the execution of your application program.

These windows are repeatedly updated and display the latest information read from the target:

- Data Log window
- Function Profiler window
- Interrupt Log window
- Timeline window
- Terminal I/O via SWO.

The data logging system can help you locate frequently accessed data and the code that is accessing it. You can then consider whether you should place that data in more efficient memory. The interrupt logging system provides you with comprehensive information about the interrupt events. This might be useful, for example, to help you locate which interrupts you can fine-tune to become faster.

Thus, the logging systems can help you both to make your application program more efficient and to debug it.

HOW TO USE THE LOGGING SYSTEMS

1. Use the Data Log breakpoints dialog box to set a Data Log breakpoint on the data you want to collect log information for.
2. In the SWO Configuration dialog box, set up the serial-wire output communication channel for trace data. Note specifically the CPU clock option.
3. From the context menu, available in the Data Log and Interrupt Log windows, respectively, choose Enable to enable the logging system.
4. Start executing your application program to collect the log information.
5. To view the data log information, choose J-Link>Data Log. For the summary, choose J-Link>Data Log Summary.
Using the data and interrupt logging systems

6 To view the interrupt log information, choose J-Link>Interrupt Log. For the summary, choose J-Link>Interrupt Log Summary.

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

8 To disable the logging system, choose Disable from the context menu in each window where you have enabled it.

RELATED REFERENCE INFORMATION
To use the logging system, you might need reference information about these windows and dialog boxes:

- Data Log breakpoints dialog box, page 278
- SWO Trace Window Settings dialog box, page 178
- Data Log window, page 296
- Data Log Summary window, page 299
- Interrupt Log window, page 300
- Interrupt Log Summary window, page 302
- Timeline window, page 189.

DATA LOG WINDOW
The Data Log window—available from the J-Link menu—logs accesses to up to four different memory locations or areas.

![Data Log window](image)

Figure 124: Data Log window

White rows indicate read accesses

Grey rows indicate write accesses

IAR Embedded Workbench® IDE
User Guide for ARM

296
Analyzing your program using driver-specific tools

Using this window
To use this window you must:

- Enable data logging on the context menu
- Set Data Log breakpoints for the memory locations or areas you want to log accesses to, see Data Log breakpoints dialog box, page 278
- Make sure to set the relevant Data Log Events option and optionally make sure that timestamps are enabled, see SWO Trace Window Settings dialog box, page 178.

The display area
Each row in the display area shows the time, the program counter, and, for every tracked data object, its value and address. More specifically, this information is provided:

<table>
<thead>
<tr>
<th>Column</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Time</td>
<td>The time for the data access, based on the clock frequency specified in the SWO Configuration dialog box.</td>
</tr>
<tr>
<td></td>
<td>If the time is displayed in italics, the target system has not been able to collect a correct time, but instead had to approximate it.</td>
</tr>
<tr>
<td></td>
<td>This column is available when you have selected Show cycles from the context menu.</td>
</tr>
<tr>
<td>Cycles</td>
<td>The number of cycles from the start of the execution until the event. This information is cleared at reset.</td>
</tr>
<tr>
<td></td>
<td>If a cycle is displayed in italics, the target system has not been able to collect a correct time, but instead had to approximate it.</td>
</tr>
<tr>
<td></td>
<td>This column is available when you have selected Show cycles from the context menu.</td>
</tr>
<tr>
<td>Program Counter*</td>
<td>The content of the PC, that is, the address of the instruction that performed the memory access.</td>
</tr>
<tr>
<td></td>
<td>If the column displays ---, the target system failed to provide the debugger with any information. If the column displays Overflow in red, the communication channel failed to transmit all data from the target system.</td>
</tr>
<tr>
<td>Value</td>
<td>Displays the access type and the value (using the access size) for the location or area you want to log accesses to. For example, if zero is read using a byte access it will be displayed as 0x00, and for a long access it will be displayed as 0x00000000.</td>
</tr>
<tr>
<td></td>
<td>To specify what data you want to log accesses to, use the Data Log breakpoint dialog box.</td>
</tr>
</tbody>
</table>

Table 52: Data Log window columns
Using the data and interrupt logging systems

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

**Data Log window context menu**

This context menu is available in the Data Log window, Data Log Summary window, Interrupt Log window, and in the Interrupt Log Summary window:

Figure 125: Data Log window context menu

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

**Table 53: Commands on the Data Log window context menu**

<table>
<thead>
<tr>
<th>Menu command</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Enable</td>
<td>Enables the logging system. The system will log information also when the window is closed.</td>
</tr>
<tr>
<td>Show time</td>
<td>Displays the Time column in the Data Log window and in the Interrupt Log window, respectively.</td>
</tr>
<tr>
<td>Show cycles</td>
<td>Displays the Cycles column in the Data Log window and in the Interrupt Log window, respectively.</td>
</tr>
<tr>
<td>Clear</td>
<td>Deletes the log information. Note that this will happen also when you reset the debugger, or if you change the execution frequency in the SWO Settings dialog box.</td>
</tr>
</tbody>
</table>

Table 52: Data Log window columns (Continued)

<table>
<thead>
<tr>
<th>Column</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Address</td>
<td>The actual memory address that is accessed. For example, if only a byte of a word is accessed, only the address of the byte is displayed. The address is calculated as base address + offset, where the base address is retrieved from the Data Log breakpoint dialog box and the offset is retrieved from the logs. If the log from the target system does not provide the debugger with an offset, the offset contains + ?. If you want the offset to be displayed, select the Value + exact addr option in the SWO Settings dialog box.</td>
</tr>
</tbody>
</table>

Table 52: Data Log window columns (Continued)
Analyzing your program using driver-specific tools

**DATA LOG SUMMARY WINDOW**

The Data Log Summary window—available from the J-Link menu—displays a summary of data accesses to specific memory location or areas.

**Using this window**

To use this window you must:

- Enable data logging on the context menu
- Set Data Log breakpoints for the memory locations or areas you want to log accesses to, see Data Log breakpoints dialog box, page 278
- Make sure to select the relevant Data Log Events option, see SWO Trace Window Settings dialog box, page 178.

**The display area**

Each row in the display area displays the type and the number of accesses to each memory location or area. More specifically, this information is provided:

<table>
<thead>
<tr>
<th>Column</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Data*</td>
<td>The name of the data object you have selected to log accesses to. To specify what data object you want to log accesses to, use the Data Log breakpoint dialog box.</td>
</tr>
<tr>
<td>Total accesses**</td>
<td>The number of total accesses.</td>
</tr>
<tr>
<td>Read accesses</td>
<td>The number of total read accesses.</td>
</tr>
</tbody>
</table>

Table 54: Data Log Summary window columns
Using the data and interrupt logging systems

<table>
<thead>
<tr>
<th>Column</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Write accesses</td>
<td>The number of total write accesses.</td>
</tr>
</tbody>
</table>

Table 54: Data Log Summary window columns (Continued)

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

Data Log Summary window context menu
See Data Log window context menu, page 298.

INTERRUPT LOG WINDOW
The Interrupt Log window—available from the J-Link menu—logs entrances to and exits from interrupts.

Figure 127: Interrupt Log window

Using this window
To use this window you must:
- Enable interrupt logging on the context menu
- Make sure the options Interrupt Log Events and Timestamps are enabled. See the SWO Trace Window Settings dialog box, page 178.
This window is also available in the C-SPY simulator.

**The display area**

Each row in the display area shows the time, type of interrupt, address, and action for each logged event. More specifically, this information is provided:

<table>
<thead>
<tr>
<th>Column</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Time</td>
<td>The time for the interrupt entrance, based on the clock frequency specified in the SWO Configuration dialog box. If a time is displayed in italics, the target system has not been able to collect a correct time, but instead had to approximate it. This column is available when you have selected Show time from the context menu.</td>
</tr>
<tr>
<td>Cycles</td>
<td>The number of cycles from the start of the execution until the event. A cycle displayed in italics indicates an approximative value. Italics is used when the target system has not been able to collect a correct time, but instead had to approximate it. This column is available when you have selected Show cycles from the context menu.</td>
</tr>
<tr>
<td>Interrupt</td>
<td>The type of interrupt that occurred. If the column displays Overflow in red, the communication channel failed to transmit all interrupt logs from the target system.</td>
</tr>
<tr>
<td>Address*</td>
<td>The address of the entry in the interrupt vector table.</td>
</tr>
<tr>
<td>Action</td>
<td>The type of event, either Enter for entering the interrupt or Leave for leaving the interrupt.</td>
</tr>
<tr>
<td>Execution time/cycles</td>
<td>The time spent in the interrupt, calculated using the enter and leave time stamps. This includes time spent in any subroutines or other interrupts that occurred in the specific interrupt.</td>
</tr>
</tbody>
</table>

* You can double-click an address. If it is available in the source code, the editor window displays the corresponding source code, for example for the interrupt handler (this does not include library source code).

**Note:** If you change the clock frequency in the SWO Configuration dialog box, all current logs are deleted.

**Interrupt Log window context menu**

See Data Log window context menu, page 298.
Using the data and interrupt logging systems

**INTERRUPT LOG SUMMARY WINDOW**

The Interrupt Log Summary window—available from the J-Link menu—displays a summary of logs of entrances and exits to and from interrupts.

![Interrupt Log Summary window](image)

**Using this window**

To use this window you must:

- Enable interrupt logging on the context menu
- Make sure the options **Interrupt Log Events** and **Timestamps** are enabled. See the *SWO Trace Window Settings dialog box*, page 178.

This window is also available in the C-SPY simulator.

**The display area**

Each row in the display area shows some statistics about the specific interrupt based on the log information. More specifically, this information is provided:

<table>
<thead>
<tr>
<th>Column</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Interrupt*</td>
<td>The type of interrupt that occurred.</td>
</tr>
<tr>
<td>Count</td>
<td>The number of times the interrupt occurred.</td>
</tr>
<tr>
<td>First Time</td>
<td>The first time the interrupt was executed.</td>
</tr>
<tr>
<td>Total Time*</td>
<td>The accumulated time spent in the interrupt.</td>
</tr>
<tr>
<td>Fastest**</td>
<td>The fastest execution of a single interrupt of this type.</td>
</tr>
<tr>
<td>Slowest**</td>
<td>The slowest execution of a single interrupt of this type.</td>
</tr>
<tr>
<td>Min Interval†</td>
<td>The shortest time between two interrupts of this type.</td>
</tr>
<tr>
<td>Max Interval†</td>
<td>The longest time between two interrupts of this type.</td>
</tr>
</tbody>
</table>

* At the bottom of the column, approximative time count displays the number of logs that contain an approximative time. Overflow count displays the number of overflows.
Analyzing your program using driver-specific tools

** Calculated in the same way as for the Execution time/cycles in the Interrupt Log window.
† The interval is specified as the time interval between the entry time for two consecutive interrupts.

Note: If you change the clock frequency in the SWO Settings dialog box, all current logs are deleted.

Interrupt Log Summary window context menu
See Data Log window context menu, page 298.

Using the profiler

This section gives you information about using the profiler. More specifically, these topics are covered:

- Requirements for using the profiler, page 303
- Reasons for using the profiler, page 303
- How to use the profiler on function level, page 304
- How to use the profiler on instruction level, page 305
- Related reference information, page 306.

Note: This profiler is part of the C-SPY driver and should not be confused with the profiling system provided as a plug-in module, see Function-level profiling, page 163.

REQUIREMENTS FOR USING THE PROFILER

To use the profiler, you need one of these setups:

- A J-Link or a J-Trace debug probe with an SWD/SWO interface between the probe and the target system, which must be based on a Cortex-M device
- A J-Trace debug probe and an ARM7/9 device with ETM trace.

REASONS FOR USING THE PROFILER

- Function profiling information is displayed in the Function Profiler window, that is, timing information for the functions in an application. The profiler measures the time between the entry and return of a function. This means that time consumed in a function is not added until the function returns or another function is called. You will only notice this if you are stepping into a function. Profiling must be turned on explicitly using a button on the window’s toolbar, and will remain 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.
Function profiling can help you find the functions where most time is spent during execution. 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 function into the memory which uses the most efficient addressing mode. For detailed information about efficient memory usage, see the IAR C/C++ Development Guide for ARM®.

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.

HOW TO USE 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;Include debug information in output</td>
</tr>
</tbody>
</table>

Table 57: Project options for enabling the profiler

2. To set up the profiler for function profiling:

- If you use the SWD/SWO interface, make sure that the PC Sampling option is selected in the SWO Settings dialog box.
- If you use ETM trace, make sure that the Cycle accurate tracing option is selected in the Trace Settings dialog box.

3. When you have built your application and started C-SPY, choose J-Link>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.

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

5. Profiling information is displayed in the Function Profiler window. To sort, click on the relevant column header.

6. When you start a new sampling, you can click the Clear button—alternatively, use the context menu—to clear the data.
HOW TO USE THE PROFILER ON INSTRUCTION LEVEL

To display instruction profiling information in the Disassembly window, follow these steps:

1. To set up the profiler for instruction profiling:
   - For the SWD/SWO interface, make sure that the **PC Sampling** option is selected in the **SWO Settings** dialog box
   - For ETM trace, no specific settings are required.

2. When you have built your application and started C-SPY, 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 program to collect the profiling information.

5. 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.

For each instruction, the number of times it has been executed is displayed.

If more than one source for the profiling data is supported by the C-SPY driver that you are using, the driver will try to use trace information as the source, unless you have chosen to use a different source. You can change the source to be used from the context menu that is available in the Profiler window.
RELATED REFERENCE INFORMATION

To use the profiler system, you might need reference information about these windows:

- Function Profiler window, page 306
- Disassembly window, page 406
- SWO Configuration dialog box, page 180.

FUNCTION PROFILER WINDOW

The Function Profiler window—available from the J-Link menu or the Simulator menu—displays function profiling information.

<table>
<thead>
<tr>
<th>Function</th>
<th>Calls</th>
<th>Flat Time</th>
<th>Flat Time (%)</th>
<th>Acc Time</th>
<th>Acc Time (%)</th>
</tr>
</thead>
<tbody>
<tr>
<td>DmllFunc(void*)</td>
<td>92</td>
<td>78.67</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Tim2Handler()</td>
<td>8</td>
<td>6.67</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>GPIO_WireR(GPIO_TypeD...)</td>
<td>6</td>
<td>5.06</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>HD44780InitIO()</td>
<td>5</td>
<td>4.17</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>HD44780_BusyCheck(mB*)</td>
<td>2</td>
<td>1.67</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>ExtIntSection()</td>
<td>2</td>
<td>1.67</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>GPIO_Init(GPIO_TypeDef...)</td>
<td>1</td>
<td>0.83</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>UsbAudioClassInit()</td>
<td>1</td>
<td>0.83</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Figure 130: Function Profiler window

Toolbar

The toolbar at the top of the window provides these buttons:

<table>
<thead>
<tr>
<th>Toolbar button</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><img src="icon" alt="Enable/Disable" /></td>
<td>Enables or disables the profiler.</td>
</tr>
<tr>
<td><img src="icon" alt="Clear" /></td>
<td>Clears all profiling data.</td>
</tr>
<tr>
<td><img src="icon" alt="Graphical view" /></td>
<td>Overlays the values in the percentage columns with a graphical bar.</td>
</tr>
<tr>
<td><img src="icon" alt="Save" /></td>
<td>Opens a standard Save As dialog box where you can save the contents of the window, with tab-separated columns. Only non-expanded rows are included in the list file.</td>
</tr>
</tbody>
</table>

Table 58: Function Profiler window toolbar
Analyzing your program using driver-specific tools

The display area

The content in the display area depends on which source is used for the profiling information:

- For the sources Breakpoints, Trace, and Builtin, 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 Sampling 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. This is because it is only known for each sample where the application was executing at that specific point in time. If the PC is sampled inside a runtime library function, it is not possible to know from which C function it was called.

More specifically, the display area provides this information:

<table>
<thead>
<tr>
<th>Column</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Function</td>
<td>The name of the profiled C function. For Sampling source, also sections of code from the runtime library or from other code without debug information, denoted only by the corresponding assembler labels, is displayed.</td>
</tr>
<tr>
<td>Calls</td>
<td>The number of times the function has been called.</td>
</tr>
<tr>
<td>Flat time</td>
<td>The time in cycles spent inside the function. For Sampling source, flat time shows the number of samples for a given function.</td>
</tr>
</tbody>
</table>

Table 58: Function Profiler window columns (Continued)
Using the profiler

**Function Profiler window context menu**

This context menu is available in the Function Profiler window:

<table>
<thead>
<tr>
<th>Menu command</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Enable</td>
<td>Enables the profiler. The system will collect information also when the window is closed.</td>
</tr>
<tr>
<td>Clear</td>
<td>Clears all profiling data.</td>
</tr>
<tr>
<td>Source*</td>
<td>Selects which source to be used for the profiling information. Choose between: Sampling, supported by the J-Link debug probe and the J-Trace debug probe. The instruction count for instruction profiling represents the number of samples for each instruction. Trace, supported by the J-Trace debug probe. The instruction count for instruction profiling is only as complete as the collected trace data.</td>
</tr>
</tbody>
</table>

*Available sources depend on the C-SPY driver you are using.*

---

**Table 60: Commands on the Function Profiler window context menu**

*For Sampling source, calls are not possible to deduce which means the column will be empty.*

---

**Table 59: Function Profiler window columns (Continued)**

<table>
<thead>
<tr>
<th>Column</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Flat time (%)</td>
<td>Flat time in cycles expressed as a percentage of the total time.</td>
</tr>
<tr>
<td>Acc. time</td>
<td>The time in cycles spent in this function and everything called by this function.</td>
</tr>
<tr>
<td>Acc. time (%)</td>
<td>Accumulated time in cycles expressed as a percentage of the total time.</td>
</tr>
</tbody>
</table>

---

**Figure 131: Function Profiler window context menu**

These commands are available on the menu:
Using flash loaders

This chapter describes the flash loader, what it is and how to use it.

The flash loader

A flash loader is an agent that is downloaded to the target. It fetches your application from the debugger and programs it into flash memory. The flash loader uses the file I/O mechanism to read the application program from the host. You can select one or several flash loaders, where each flash loader loads a selected part of your application. This means that you can use different flash loaders for loading different parts of your application.

A set of flash loaders for various microcontrollers is provided with IAR Embedded Workbench for ARM. In addition to these, more flash loaders are provided by chip manufacturers and third-party vendors. The flash loader API, documentation, and several implementation examples are available to make it possible for you to implement your own flash loader.

SETTING UP THE FLASH LOADER(S)

To use a flash loader for downloading your application:

1. Choose Project>Options.
2. Choose the Debugger category and click the Download tab.
3. Select the Use Flash loader(s) option. A default flash loader configured for the device you have specified will be used. The configuration is specified in a preconfigured board file.
4. To override the default flash loader or to modify the behavior of the default flash loader to suit your board, select the Override default .board file option, and Edit to open the Flash Loader Configuration dialog box. A copy of the *.board file will be created in your project directory and the path to the *.board file will be updated accordingly.
5. The Flash Loader Overview dialog box lists all currently configured flash loaders; see Flash Loader Overview dialog box, page 310. You can either select a flash loader or open the Flash Loader Configuration dialog box.

In the Flash Loader Configuration dialog box, you can configure the download. For reference information about the various flash loader options, see Flash Loader Configuration dialog box, page 312.
THE FLASH LOADING MECHANISM

When the Use flash loader(s) option is selected and one or several flash loaders have been configured, these steps are performed when the debug session starts:

1. C-SPY downloads the flash loader into target RAM.
2. C-SPY starts execution of the flash loader.
3. The flash loader programs the application code into flash memory.
4. The flash loader terminates.
5. C-SPY switches context to the user application.

Steps 2 to 4 are performed for each memory range of the application.
The steps 1 to 4 are performed for each selected flash loader.

BUILD CONSIDERATIONS

When you build an application that will be downloaded to flash, special consideration is needed. Two output files must be generated. The first is the usual ELF/DWARF file (out) that provides the debugger with debug and symbol information. The second file is a simple-code file (filename extension sim) that will be opened and read by the flash loader when it downloads the application to flash memory.

The simple-code file must have the same path and name as the ELF/DWARF file except for the filename extension. This file is automatically generated by the linker.

FLASH LOADER OVERVIEW DIALOG BOX

The Flash Loader Overview dialog box—available from the Debugger>Download page—lists all defined flash loaders. If you have selected a device on the General Options>Target page for which there is a flash loader, this flash loader is by default listed in the Flash Loader Overview dialog box.

Figure 132: Flash Loader Overview dialog box
The display area

Each row in the display area shows how you have set up one flash loader for flashing a specific part of memory:

<table>
<thead>
<tr>
<th>Column</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Range</td>
<td>The part of your application to be programmed by the selected flash loader.</td>
</tr>
<tr>
<td>Offset/Address</td>
<td>The start of the memory where your application will be flashed. If the address is preceded with a, the address is absolute. Otherwise, it is a relative offset to the start of the memory.</td>
</tr>
<tr>
<td>Loader Path</td>
<td>The path to the flash loader <em>.flash file to be used (</em> .out for old-style flash loaders).</td>
</tr>
<tr>
<td>Extra Parameters</td>
<td>List of extra parameters that will be passed to the flash loader.</td>
</tr>
</tbody>
</table>

Table 61: Flash Loader Overview dialog box columns

Click on the column headers to sort the list by range, offset/address, etc.

Function buttons

These function buttons are available:

<table>
<thead>
<tr>
<th>Button</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>OK</td>
<td>The selected flash loader(s) will be used for downloading your application to memory.</td>
</tr>
<tr>
<td>Cancel</td>
<td>Standard cancel.</td>
</tr>
<tr>
<td>New</td>
<td>Opens the Flash Loader Configuration dialog box where you can specify what flash loader to use; see Flash Loader Configuration dialog box, page 312.</td>
</tr>
<tr>
<td>Edit</td>
<td>Opens the Flash Loader Configuration dialog box where you can modify the settings for the selected flash loader; see Flash Loader Configuration dialog box, page 312.</td>
</tr>
<tr>
<td>Delete</td>
<td>Deletes the selected flash loader configuration.</td>
</tr>
</tbody>
</table>

Table 62: Function buttons in the Flash Loader Overview dialog box
FLASH LOADER CONFIGURATION DIALOG BOX

In the Flash Loader Configuration dialog box — available from the Flash Loader Overview dialog box — you can configure the download to suit your board. A copy of the default board file will be created in your project directory.

![Flash Loader Configuration dialog box](image)

**Figure 133: Flash Loader Configuration dialog box**

**Memory range**

Use the Memory range options to specify the part of your application to be downloaded to flash memory. Choose between:

- **All**: The whole application is downloaded using this flash loader.
- **Start/End**: The part of the application available in the memory range will be downloaded. Use the **Start** and **End** text fields to specify the memory range.

**Relocate**

Use the Relocate option to override the default flash base address, that is relocate the location of the application in memory. This means that you can flash your application to a different location from where it was linked. Choose between:

- **Offset**: A numeric value for a relative offset. This offset will be added to the addresses in the application file.
You can use these numeric formats:

- 123456: Decimal numbers
- 0x123456: Hexadecimal numbers
- 0123456: Octal numbers

The default base address used for writing the first byte—the lowest address—to flash is specified in the linker configuration file used for your application. However, it can sometimes be necessary to override the flash base address and start at a different location in the address space. This can, for example, be necessary for devices that remap the location of the flash memory.

**Flash loader path**

Use the text box to specify the path to the flash loader file (*.flash) to be used by your board configuration.

**Extra parameters**

Some flash loaders define their own set of specific options. Use this text box to specify options to control the flash loader. For information about available flash loader options, see the Parameter descriptions field.

**Parameter descriptions**

The Parameter descriptions field displays a description of the extra parameters specified in the Extra parameters text box.
The flash loader
Part 7. Reference information

This part of the IAR Embedded Workbench® IDE User Guide contains the following chapters:

- IAR Embedded Workbench® IDE reference
- C-SPY® reference
- General options
- Compiler options
- Assembler options
- Converter options
- Custom build options
- Build actions options
- Linker options
- Library builder options
- Debugger options
- The C-SPY Command Line Utility—cspybat
- C-SPY® macros reference.
IAR Embedded Workbench® IDE reference

This chapter contains reference information about the windows, menus, menu commands, and the corresponding components that are found in the IDE. For information about how to best use the IDE for your purposes, see parts 3 to 7 in this guide. This chapter contains the following sections:

- Windows, page 317
- Menus, page 350.

The IDE is a modular application. Which menus are available depends on which components are installed.

Windows

The available windows are:

- IAR Embedded Workbench IDE window
- Workspace window
- Editor window
- Source Browser window
- Breakpoints window
- Message windows.

In addition, a set of C-SPY®-specific windows becomes available when you start the debugger. For reference information about these windows, see the chapter C-SPY® reference in this guide.
The figure shows the main window of the IDE and its various components. The window might look different depending on which plugin modules you are using.

![Figure 134: IAR Embedded Workbench IDE window](image)

Each window item is explained in greater detail in the following sections.

**Menu bar**

Gives access to the IDE menus.

<table>
<thead>
<tr>
<th>Menu</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>File</td>
<td>The File menu provides commands for opening source and project files, saving and printing, and exiting from the IDE.</td>
</tr>
<tr>
<td>Edit</td>
<td>The Edit menu provides commands for editing and searching in editor windows and for enabling and disabling breakpoints in C-SPY.</td>
</tr>
</tbody>
</table>

*Table 63: IDE menu bar*
Part 7. Reference information

IAR Embedded Workbench® IDE reference

For reference information for each menu, see Menus, page 350.

**Toolbar**

The IDE toolbar—available from the View menu—provides buttons for the most useful commands on the IDE menus, and a text box for typing a string to do a quick search.

For a description of any button, point to it with the mouse button. When a command is not available, the corresponding toolbar button is dimmed, and you will not be able to click it.

This figure shows the menu commands corresponding to each of the toolbar buttons:

![IDE toolbar](image)

**Note**: When you start C-SPY, the Download and Debug button will change to a Make and Debug button and the Debug without Downloading will change to a Restart Debugger button.

---

**Table 63: IDE menu bar (Continued)**

<table>
<thead>
<tr>
<th>Menu</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>View</td>
<td>Use the commands on the View menu to open windows and decide which toolbars to display.</td>
</tr>
<tr>
<td>Project</td>
<td>The Project menu provides commands for adding files to a project, creating groups, and running the IAR Systems tools on the current project.</td>
</tr>
<tr>
<td>Tools</td>
<td>The Tools menu is a user-configurable menu to which you can add tools for use with the IDE.</td>
</tr>
<tr>
<td>Window</td>
<td>With the commands on the Window menu you can manipulate the IDE windows and change their arrangement on the screen.</td>
</tr>
<tr>
<td>Help</td>
<td>The commands on the Help menu provide help about the IDE.</td>
</tr>
</tbody>
</table>

For reference information for each menu, see Menus, page 350.
Status bar

The status bar at the bottom of the window displays the number of errors and warnings generated during a build, the position of the insertion point in the editor window, and the state of the modifier keys. The status bar is available from the View menu.

As you are editing, the status bar shows the current line and column number containing the insertion point, and the Caps Lock, Num Lock, and Overwrite status.

![Figure 136: IAR Embedded Workbench IDE window status bar](image)

WORKSPACE WINDOW

The Workspace window, available from the View menu, is where you can access your projects and files during the application development.

![Figure 137: Workspace window](image)

Toolbar

At the top of the window there is a drop-down list where you can choose a build configuration to display in the window for a specific project.
The display area

The display area is divided into different columns.

The **Files** column displays the name of the current workspace and a tree representation of the projects, groups and files included in the workspace. One or more of these icons are displayed:

- Workspace
- Project
- Project with multi-file compilation
- Group of files
- Group excluded from the build
- Group of files, part of multi-file compilation
- Group of files, part of multi-file compilation, but excluded from the build
- Object file or library
- Assembler source file
- C source file
- C++ source file
- Source file excluded from the build
- Header file
- Text file
- HTML text file
- Control file, for example the linker configuration file
- IDE internal file
- Other file

The column that contains status information about option overrides can have one of three icons for each level in the project:

- Blank There are no settings/overrides for this file/group
- Black check mark There are local settings/overrides for this file/group
The column that contains build status information can have one of three icons for each file in the project:

- **Red check mark**: There are local settings/overrides for this file/group, but they are either identical to the inherited settings or they are ignored because you use of multi-file compilation, which means that the overrides are not needed.

- **Blank**: The file will not be rebuilt next time the project is built

- **Red star**: The file will be rebuilt next time the project is built

- **Gearwheel**: The file is being rebuilt.

For details about the various source code control icons, see *Source code control states*, page 326.

At the bottom of the window you can choose which project to display. Alternatively, you can choose to display an overview of the entire workspace.

For more information about project management and using the Workspace window, see the chapter *Managing projects in Part 3. Project management and building* in this guide.

**Workspace window context menu**

Clicking the right mouse button in the Workspace window displays a context menu which gives you convenient access to several commands.

![Workspace window context menu](image)

*Figure 138: Workspace window context menu*
These commands are available on the context menu:

<table>
<thead>
<tr>
<th>Menu command</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Options</td>
<td>Displays a dialog box where you can set options for each build tool on the selected item in the Workspace window. You can set options on the entire project, on a group of files, or on an individual file.</td>
</tr>
<tr>
<td>Make</td>
<td>Brings the current target up to date by compiling, assembling, and linking only the files that have changed since the last build.</td>
</tr>
<tr>
<td>Compile</td>
<td>Compiles or assembles the currently active file as appropriate. You can choose the file either by selecting it in the Workspace window, or by selecting the editor window containing the file you want to compile.</td>
</tr>
<tr>
<td>Rebuild All</td>
<td>Recompiles and relinks all files in the selected build configuration.</td>
</tr>
<tr>
<td>Clean</td>
<td>Deletes intermediate files.</td>
</tr>
<tr>
<td>Stop Build</td>
<td>Stops the current build operation.</td>
</tr>
<tr>
<td>Add&gt;Add Files</td>
<td>Opens a dialog box where you can add files to the project.</td>
</tr>
<tr>
<td>Add&gt;Add &quot;filename&quot;</td>
<td>Adds the indicated file to the project. This command is only available if there is an open file in the editor.</td>
</tr>
<tr>
<td>Add&gt;Add Group</td>
<td>Opens a dialog box where you can add new groups to the project.</td>
</tr>
<tr>
<td>Remove</td>
<td>Removes selected items from the Workspace window.</td>
</tr>
<tr>
<td>Rename</td>
<td>Opens the Rename Group dialog box where you can rename a group. For more information about groups, see Groups, page 89.</td>
</tr>
<tr>
<td>Source Code Control</td>
<td>Opens a submenu with commands for source code control, see Source Code Control menu, page 324.</td>
</tr>
<tr>
<td>File Properties</td>
<td>Opens a standard File Properties dialog box for the selected file.</td>
</tr>
<tr>
<td>Set as Active</td>
<td>Sets the selected project in the overview display to be the active project. It is the active project that will be built when the Make command is executed.</td>
</tr>
</tbody>
</table>

Table 64: Workspace window context menu commands
Rename Group dialog box

The **Select Source Code Control Provider** dialog box is displayed if several SCC systems from different vendors are available. Use this dialog box to choose the SCC system you want to use.

Source Code Control menu

The **Source Code Control** menu is available from the **Project** menu and from the context menu in the Workspace window. This menu contains some of the most commonly used commands of external, third-party source code control systems.

<table>
<thead>
<tr>
<th>Command</th>
</tr>
</thead>
<tbody>
<tr>
<td>Check In...</td>
</tr>
<tr>
<td>Check Out...</td>
</tr>
<tr>
<td>Lock Check Out</td>
</tr>
<tr>
<td>Get Latest Version</td>
</tr>
<tr>
<td>Compare...</td>
</tr>
<tr>
<td>History...</td>
</tr>
<tr>
<td>Properties...</td>
</tr>
<tr>
<td>Refresh</td>
</tr>
<tr>
<td>Connect Project to SCC Project...</td>
</tr>
<tr>
<td>Disconnect Project from SCC Project...</td>
</tr>
</tbody>
</table>

Figure 139: Select Source Code Control Provider dialog box

Figure 140: Source Code Control menu
For more information about interacting with an external source code control system, see *Source code control*, page 94.

These commands are available on the submenu:

<table>
<thead>
<tr>
<th>Menu command</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Check In</td>
<td>Opens the <strong>Check In Files</strong> dialog box where you can check in the selected files; see <em>Check In Files dialog box</em>, page 327. Any changes you have made in the files will be stored in the archive. This command is enabled when currently checked-out files are selected in the Workspace window.</td>
</tr>
<tr>
<td>Check Out</td>
<td>Checks out the selected file or files. Depending on the SCC system you are using, a dialog box might appear; see <em>Check Out Files dialog box</em>, page 328. This means you get a local copy of the file(s), which you can edit. This command is enabled when currently checked-in files are selected in the Workspace window.</td>
</tr>
<tr>
<td>Undo Check out</td>
<td>The selected files revert to the latest archived version; the files are no longer checked-out. Any changes you have made to the files will be lost. This command is enabled when currently checked-out files are selected in the Workspace window.</td>
</tr>
<tr>
<td>Get Latest Version</td>
<td>Replaces the selected files with the latest archived version.</td>
</tr>
<tr>
<td>Compare</td>
<td>Displays—in a SCC-specific window—the differences between the local version and the most recent archived version.</td>
</tr>
<tr>
<td>History</td>
<td>Displays SCC-specific information about the revision history of the selected file.</td>
</tr>
<tr>
<td>Properties</td>
<td>Displays information available in the SCC system for the selected file.</td>
</tr>
<tr>
<td>Refresh</td>
<td>Updates the SCC display status for all the files that are part of the project. This command is always enabled for all projects under SCC.</td>
</tr>
<tr>
<td>Connect Project to SCC Project</td>
<td>Opens a dialog box, which originates from the SCC client application, to let you create a connection between the selected IAR Embedded Workbench project and an SCC project; the IAR Embedded Workbench project will then be an SCC-controlled project. After creating this connection, a special column that contains status information will appear in the Workspace window.</td>
</tr>
<tr>
<td>Disconnect Project From SCC Project</td>
<td>Removes the connection between the selected IAR Embedded Workbench project and an SCC project; your project will no longer be a SCC-controlled project. The column in the Workspace window that contains SCC status information will no longer be visible for that project.</td>
</tr>
</tbody>
</table>

*Table 65: Description of source code control commands*
Source code control states

Each source code-controlled file can be in one of several states.

<table>
<thead>
<tr>
<th>SCC state</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>![Checked out to you. The file is editable.]</td>
<td>Checked out to you. The file is editable.</td>
</tr>
<tr>
<td>![Checked out to you. The file is editable and you have modified the file.]</td>
<td>Checked out to you. The file is editable and you have modified the file.</td>
</tr>
<tr>
<td>![Grey padlock]</td>
<td>Checked in. In many SCC systems this means that the file is write-protected.</td>
</tr>
<tr>
<td>![Grey padlock]</td>
<td>Checked in. A new version is available in the archive.</td>
</tr>
<tr>
<td>![Red padlock]</td>
<td>Checked out exclusively to another user. In many SCC systems this means that you cannot check out the file.</td>
</tr>
<tr>
<td>![Red padlock]</td>
<td>Checked out exclusively to another user. A new version is available in the archive. In many SCC systems this means that you cannot check out the file.</td>
</tr>
</tbody>
</table>

Table 66: Description of source code control states

Note: The source code control in IAR Embedded Workbench depends on the information provided by the SCC system. If the SCC system provides incorrect or incomplete information about the states, IAR Embedded Workbench might display incorrect symbols.
Select Source Code Control Provider dialog box

The Select Source Code Control Provider dialog box is displayed if several SCC systems from different vendors are available. Use this dialog box to choose the SCC system you want to use.

Check In Files dialog box

The Check In Files dialog box is available by choosing the Project>Source Code Control>Check In command, alternatively available from the Workspace window context menu.
Comment

A text box in which you can write a comment—typically a description of your changes—that will be stored in the archive together with the file revision. This text box is only enabled if the SCC system supports the adding of comments at check in.

Keep checked out

The file(s) will continue to be checked out after they have been checked in. Typically, this is useful if you want to make your modifications available to other members in your project team, without stopping your own work with the file.

Advanced

Opens a dialog box, originating from the SCC client application, that contains advanced options. This button is only available if the SCC system supports setting advanced options at check in.

Files

A list of the files that will be checked in. The list will contain all files that were selected in the Workspace window when this dialog box was opened.

Check Out Files dialog box

The Check Out Files dialog box is available by choosing the Project>Source Code Control>Check Out command, alternatively available from the Workspace window.
context menu. However, this dialog box is only available if the SCC system supports adding comments at check-out or advanced options.

![Check Out Files dialog box](image)

**Comment**

A text field in which you can write a comment—typically the reason why the file is checked out—that will be placed in the archive together with the file revision. This text box is only enabled if the SCC system supports the adding of comments at check-out.

**Advanced**

Opens a dialog box, originating from the SCC client application, that contains advanced options. This button is only available if the SCC system supports setting advanced options at check out.

**Files**

A list of files that will be checked out. The list will contain all files that were selected in the Workspace window when this dialog box was opened.
EDITOR WINDOW

Source code files and HTML files are displayed in editor windows. You can have one or several editor windows open at the same time. The editor window is always docked, and its size and position depend on other currently open windows.

The name of the open file is displayed on the tab. If a file is read-only, a padlock icon is visible at the bottom left corner of the editor window. If a file has been modified after it was last saved, an asterisk appears after the filename on the tab, for example Utilities.c *. All open files are available from the drop-down menu in the upper right corner of the editor window.

For information about using the editor, see the chapter Editing, page 105.

HTML files

Use the File > Open command to open HTML documents in the editor window. From an open HTML document you can navigate to other documents using hyperlinks:

- A link to an html or htm file works like in normal web browsing
- A link to an eww workspace file opens the workspace in the IDE, and closes any currently open workspace and the open HTML document.
Split commands

Use the Window>Split command—or the Splitter controls—to split the editor window horizontally or vertically into multiple panes.

On the Window menu you also find commands for opening multiple editor windows, and commands for moving files between the editor windows.

Go to function

Click the Go to function button in the bottom left-hand corner of the editor window to list all functions of the C or C++ editor window.

Double-click the function that you want to show in the editor window.

Editor window tab context menu

This is the context menu that appears if you right-click on a tab in the editor window:

<table>
<thead>
<tr>
<th>Menu command</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Save file</td>
<td>Saves the file.</td>
</tr>
<tr>
<td>Close</td>
<td>Closes the file.</td>
</tr>
<tr>
<td>File Properties</td>
<td>Displays a standard file properties dialog box.</td>
</tr>
</tbody>
</table>

Table 67: Description of commands on the editor window tab context menu
**Editor window context menu**

The context menu available in the editor window provides convenient access to several commands.

<table>
<thead>
<tr>
<th>Menu command</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Cut, Copy, Paste</td>
<td>Standard window commands.</td>
</tr>
<tr>
<td>Complete</td>
<td>Attempts to complete the word you have begun to type, basing the guess on the contents of the rest of the editor document.</td>
</tr>
<tr>
<td>Match Brackets</td>
<td>Selects all text between the brackets immediately surrounding the insertion point, increases the selection to the next hierarchic pair of brackets, or beeps if there is no higher bracket hierarchy.</td>
</tr>
</tbody>
</table>

**Note:** The contents of this menu are dynamic, which means it might contain other commands than in this figure. All available commands are described in Table 68, *Description of commands on the editor window context menu*.

These commands are available on the editor window context menu:
<table>
<thead>
<tr>
<th>Menu command</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Insert Template</td>
<td>Displays a list in the editor window from which you can choose a code template to be inserted at the location of the insertion point. If the code template you choose requires any field input, the Template dialog box appears; for information about this dialog box, see Template dialog box, page 361. For information about using code templates, see Using and adding code templates, page 109.</td>
</tr>
<tr>
<td>Open &quot;header.h&quot;</td>
<td>Opens the header file &quot;header.h&quot; in an editor window. This menu command is only available if the insertion point is located on an #include line when you open the context menu.</td>
</tr>
<tr>
<td>Open Header/Source File</td>
<td>Jumps from the current file to the corresponding header or source file. If the destination file is not open when performing the command, the file will first be opened. This menu command is only available if the insertion point is located on any line except an #include line when you open the context menu. This command is also available from the File&gt;Open menu.</td>
</tr>
<tr>
<td>Go to definition of symbol</td>
<td>Shows the declaration of the symbol where the insertion point is placed.</td>
</tr>
<tr>
<td>Check In</td>
<td>Commands for source code control; for more details, see Source Code Control menu, page 324. These menu commands are only available if the current source file in the editor window is SCC-controlled. The file must also be a member of the current project.</td>
</tr>
<tr>
<td>Check Out</td>
<td></td>
</tr>
<tr>
<td>Undo Checkout</td>
<td></td>
</tr>
<tr>
<td>Toggle Breakpoint (Code)</td>
<td>Toggles a code breakpoint at the statement or instruction containing or close to the cursor in the source window. For information about code breakpoints, see Code breakpoints dialog box, page 341.</td>
</tr>
<tr>
<td>Toggle Breakpoint (Log)</td>
<td>Toggles a log breakpoint at the statement or instruction containing or close to the cursor in the source window. For information about log breakpoints, see Log breakpoints dialog box, page 343.</td>
</tr>
<tr>
<td>Toggle Breakpoint (Trace Start)</td>
<td>Toggles a Trace Start breakpoint. When the breakpoint is triggered, the trace system is started. For information about Trace Start breakpoints, see Trace Start breakpoints dialog box, page 193. Note that this menu command is only available if the C-SPY driver you are using supports the trace system.</td>
</tr>
<tr>
<td>Toggle Breakpoint (Trace Stop)</td>
<td>Toggles a Trace Stop breakpoint. When the breakpoint is triggered, the trace system is stopped. For information about Trace Stop breakpoints, see Trace Stop breakpoints dialog box, page 194. Note that this menu command is only available if the C-SPY driver you are using supports the trace system.</td>
</tr>
</tbody>
</table>

Table 68: Description of commands on the editor window context menu (Continued)
### Menu command Description

<table>
<thead>
<tr>
<th>Command</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Toggle Breakpoint (Trace Filter)</td>
<td>Toggles a Trace Filter breakpoint. When the breakpoint is triggered, the trace system runs only when the trace filter condition is true. For information about Trace Filter breakpoints, see Trace Filter breakpoints dialog box, page 292. Note that this menu command is only available if the C-SPY driver you are using supports the trace system.</td>
</tr>
<tr>
<td>Enable/disable Breakpoint</td>
<td>Toggles a breakpoint between being disabled, but not actually removed—making it available for future use—and being enabled again.</td>
</tr>
<tr>
<td>Set Data Breakpoint for variable</td>
<td>Toggles a data breakpoint on variables with static storage duration.</td>
</tr>
<tr>
<td></td>
<td>Requires support in the C-SPY driver you are using.</td>
</tr>
<tr>
<td>Set Data Log Breakpoint for variable</td>
<td>Toggles a data log breakpoint on variables with static storage duration.</td>
</tr>
<tr>
<td></td>
<td>Requires support in the C-SPY driver you are using.</td>
</tr>
<tr>
<td>Set Trace Start Breakpoint for variable</td>
<td>Toggles a trace start breakpoint on variables with static storage duration. Requires support in the C-SPY driver you are using.</td>
</tr>
<tr>
<td>Set Trace Stop Breakpoint for variable</td>
<td>Toggles a trace stop breakpoint on variables with static storage duration. Requires support in the C-SPY driver you are using.</td>
</tr>
<tr>
<td>Set Trace Filter Breakpoint for variable</td>
<td>Toggles a trace filter breakpoint on variables with static storage duration. Requires support in the C-SPY driver you are using.</td>
</tr>
<tr>
<td>Edit Breakpoint</td>
<td>Displays the Edit 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>Set Next Statement</td>
<td>Sets the PC directly to the selected statement or instruction without executing any code. Use this menu command with care. This menu command is only available when you are using the debugger.</td>
</tr>
<tr>
<td>Quick Watch</td>
<td>Opens the Quick Watch window, see Quick Watch window, page 422. This menu command is only available when you are using the debugger.</td>
</tr>
<tr>
<td>Add to Watch</td>
<td>Adds the selected symbol to the Watch window. This menu command is only available when you are using the debugger.</td>
</tr>
<tr>
<td>Move to PC</td>
<td>Moves the insertion point to the current PC position in the editor window. This menu command is only available when you are using the debugger.</td>
</tr>
<tr>
<td>Run to Cursor</td>
<td>Executes from the current statement or instruction up to a selected statement or instruction. This menu command is only available when you are using the debugger.</td>
</tr>
<tr>
<td>Options</td>
<td>Displays the IDE Options dialog box, see Tools menu, page 373.</td>
</tr>
</tbody>
</table>

Table 68: Description of commands on the editor window context menu (Continued)
Source file paths

The IDE supports relative source file paths to a certain degree.

If a source file is located in the project file directory or in any subdirectory of the project file directory, the IDE will use a path relative to the project file when accessing the source file.

Editor key summary

The following tables summarize the editor’s keyboard commands.

Use these keys and key combinations for moving the insertion point:

<table>
<thead>
<tr>
<th>To move the insertion point</th>
<th>Press</th>
</tr>
</thead>
<tbody>
<tr>
<td>One character left</td>
<td>Arrow left</td>
</tr>
<tr>
<td>One character right</td>
<td>Arrow right</td>
</tr>
<tr>
<td>One word left</td>
<td>Ctrl+Arrow left</td>
</tr>
<tr>
<td>One word right</td>
<td>Ctrl+Arrow right</td>
</tr>
<tr>
<td>One line up</td>
<td>Arrow up</td>
</tr>
<tr>
<td>One line down</td>
<td>Arrow down</td>
</tr>
<tr>
<td>To the start of the line</td>
<td>Home</td>
</tr>
<tr>
<td>To the end of the line</td>
<td>End</td>
</tr>
<tr>
<td>To the first line in the file</td>
<td>Ctrl+Home</td>
</tr>
<tr>
<td>To the last line in the file</td>
<td>Ctrl+End</td>
</tr>
</tbody>
</table>

Table 69: Editor keyboard commands for insertion point navigation

Use these keys and key combinations for scrolling text:

<table>
<thead>
<tr>
<th>To scroll</th>
<th>Press</th>
</tr>
</thead>
<tbody>
<tr>
<td>Up one line</td>
<td>Ctrl+Arrow up</td>
</tr>
<tr>
<td>Down one line</td>
<td>Ctrl+Arrow down</td>
</tr>
<tr>
<td>Up one page</td>
<td>Page Up</td>
</tr>
<tr>
<td>Down one page</td>
<td>Page Down</td>
</tr>
</tbody>
</table>

Table 70: Editor keyboard commands for scrolling

Use these key combinations for selecting text:

<table>
<thead>
<tr>
<th>To select</th>
<th>Press</th>
</tr>
</thead>
<tbody>
<tr>
<td>The character to the left</td>
<td>Shift+Arrow left</td>
</tr>
<tr>
<td>The character to the right</td>
<td>Shift+Arrow right</td>
</tr>
</tbody>
</table>

Table 71: Editor keyboard commands for selecting text
SOURCE BROWSER WINDOW

The Source Browser window—available from the View menu—displays an hierarchical view in alphabetical order of all symbols defined in the active build configuration.

The window consists of two separate display areas.
The upper display area

The upper display area contains two columns:

<table>
<thead>
<tr>
<th>Column</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Icons</td>
<td>An icon that corresponds to the Symbol type classification, see Icons used for the symbol types, page 338.</td>
</tr>
<tr>
<td>Name</td>
<td>The names of global symbols and functions defined in the project. Note that unnamed types, for example a struct or a union without a name, will get a name based on the filename and line number where it is defined. These pseudonames are enclosed in angle brackets.</td>
</tr>
</tbody>
</table>

If you click in the window header, you can sort the symbols either by name or by symbol type.

In the upper display area you can also access a context menu; see Source Browser window context menu, page 338.

The lower display area

For a symbol selected in the upper display area, the lower area displays its properties:

<table>
<thead>
<tr>
<th>Property</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Full name</td>
<td>Displays the unique name of each element, for instance classname::membername.</td>
</tr>
<tr>
<td>Symbol type</td>
<td>Displays the symbol type for each element, see Icons used for the symbol types, page 338.</td>
</tr>
<tr>
<td>Filename</td>
<td>Specifies the path to the file in which the element is defined.</td>
</tr>
</tbody>
</table>

Table 72: Columns in Source Browser window

Table 73: Information in Source Browser window
Icons used for the symbol types

These are the icons used:

- Base class
- Class
- Configuration
- Enumeration
- Enumeration constant
- (Yellow rhomb) Field of a struct
- (Purple rhomb) Function
- Macro
- Namespace
- Template class
- Template function
- Type definition
- Union
- (Yellow rhomb) Variable

Usage

For further details about how to use the Source Browser window, see Displaying browse information, page 93.

Source Browser window context menu

This is the context menu available in the upper display area:

- Go to Definition
- Move to Source
- All Symbols
  - All Functions & Variables
  - All Member Functions & Variables
  - Types
  - Constants & Macros
- All Files
  - Include System Includes
  - Only Project Members

Figure 149: Source Browser window context menu
These commands are available on the context menu:

<table>
<thead>
<tr>
<th>Menu command</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Go to Definition</td>
<td>The editor window will display the definition of the selected item.</td>
</tr>
<tr>
<td>Move to Parent</td>
<td>If the selected element is a member of a class, struct, union, enumeration, or namespace, this menu command can be used for moving to its enclosing element.</td>
</tr>
<tr>
<td>All Symbols</td>
<td>Type filter; all global symbols and functions defined in the project will be displayed.</td>
</tr>
<tr>
<td>All Functions &amp; Variables</td>
<td>Type filter; all functions and variables defined in the project will be displayed.</td>
</tr>
<tr>
<td>Non-Member Functions &amp; Variables</td>
<td>Type filter; all functions and variables that are not members of a class will be displayed</td>
</tr>
<tr>
<td>Types</td>
<td>Type filter; all types such as structures and classes defined in the project will be displayed.</td>
</tr>
<tr>
<td>Constants &amp; Macros</td>
<td>Type filter; all constants and macros defined in the project will be displayed.</td>
</tr>
<tr>
<td>All Files</td>
<td>File filter; symbols from all files that you have explicitly added to your project and all files included by them will be displayed.</td>
</tr>
<tr>
<td>Exclude System Includes</td>
<td>File filter; symbols from all files that you have explicitly added to your project and all files included by them will be displayed, except the include files in the IAR Embedded Workbench installation directory.</td>
</tr>
<tr>
<td>Only Project Members</td>
<td>File filter; symbols from all files that you have explicitly added to your project will be displayed, but no include files.</td>
</tr>
</tbody>
</table>

*Table 74: Source Browser window context menu commands*
BREAKPOINTS WINDOW

The Breakpoints window—available from the View menu—lists all breakpoints. From the window you can conveniently monitor, enable, and disable breakpoints; you can also define new breakpoints and modify existing breakpoints.

All breakpoints you define are displayed in the Breakpoints window.

For more information about the breakpoint system and how to set breakpoints, see the chapter Using breakpoints in Part 4. Debugging.

Breakpoints window context menu

Right-clicking in the Breakpoints window displays a context menu with several commands.

These commands are available on the context menu:

<table>
<thead>
<tr>
<th>Menu command</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Go to Source</td>
<td>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.</td>
</tr>
<tr>
<td>Edit</td>
<td>Opens the Edit Breakpoint dialog box for the selected breakpoint.</td>
</tr>
</tbody>
</table>

Table 75: Breakpoints window context menu commands
Code breakpoints dialog box

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.

To set a code breakpoint, right-click in the Breakpoints window and choose New Breakpoint > Code on the context menu. To modify an existing breakpoint, select it in the Breakpoints window and choose Edit on the context menu.
The **Code** breakpoints dialog box appears.

![Figure 152: Code breakpoints page](image)

**Break At**

Specify the location of the breakpoint in the **Break At** text box. Alternatively, click the **Edit** browse button to open the **Enter Location** dialog box; see **Enter Location dialog box**, page 345.

**Size**

Optionally, you can specify a size—in practice, a range—of locations. Each fetch access to the specified memory range will trigger the breakpoint. There are two different ways to specify the size:

- **Auto**, the size will be set automatically, typically to 1
- **Manual**, you specify the size of the breakpoint range manually in the **Size** text box.

**Action**

You can optionally connect an action to a 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

You can specify simple and complex conditions.

<table>
<thead>
<tr>
<th>Conditions</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Expression</td>
<td>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 must be fulfilled before a break occurs (integer).</td>
</tr>
</tbody>
</table>

Table 76: Breakpoint conditions

Log breakpoints dialog box

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 halt and print the specified message in the C-SPY Debug Log window. This is a convenient way to add trace printouts during the execution of your application, without having to add any code to the application source code.

To set a log breakpoint, right-click in the Breakpoints window and choose New Breakpoint→Log on the context menu. To modify an existing breakpoint, select it in the Breakpoints window and choose Edit on the context menu.

The Log breakpoints dialog box appears.
The quickest—and typical—way to set a log breakpoint is by choosing **Toggle Breakpoint (Log)** from the context menu available when you right-click in either the editor or the Disassembly window. For more information about how to set breakpoints, see *Defining breakpoints*, page 141.

**Break At**

Specify the location of 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 345.

**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 530.

**Conditions**

You can specify simple and complex conditions.

<table>
<thead>
<tr>
<th>Conditions</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Expression</td>
<td>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>
</tbody>
</table>

*Table 77: Log breakpoint conditions*
Enter Location dialog box

Use the Enter Location dialog box—available from a breakpoints dialog box—to specify the location of the breakpoint.

![Enter Location dialog box](image)

Figure 154: Enter Location dialog box

You can choose between these locations and their possible settings:

<table>
<thead>
<tr>
<th>Location type</th>
<th>Description/Examples</th>
</tr>
</thead>
<tbody>
<tr>
<td>Expression</td>
<td>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, <code>my_var</code> refers to the location of the variable <code>my_var</code>, and <code>arr[3]</code> refers to the third element of the array <code>arr</code>.</td>
</tr>
<tr>
<td>Absolute address</td>
<td>An absolute location on the form <code>zone:hexaddress</code> or simply <code>hexaddress</code>. Zone specifies in which memory the address belongs. For example <code>Memory:0x42</code>. If you enter a combination of a zone and address that is not valid, C-SPY will indicate the mismatch.</td>
</tr>
<tr>
<td>Source location</td>
<td>A location in the C source code using the syntax: <code>{file path}.row.column</code>. <code>file</code> specifies the filename and full path. <code>row</code> specifies the row in which you want the breakpoint. <code>column</code> specifies the column in which you want the breakpoint. Note that the Source Location type is usually meaningful only for code breakpoints. For example, <code>{C:\my_projects\Utilities.c}.22.3</code> sets a breakpoint on the third character position on line 22 in the source file <code>Utilities.c</code>.</td>
</tr>
</tbody>
</table>

Table 78: Location types
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.

![Resolve Source Ambiguity dialog box](image)

**All**

All listed locations will be used.

**Selected**

Select one or more of the locations in the list. Only the selected locations will be used.

**Cancel**

No location will be used.

**Automatically choose all**

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, page 389.
BUILD WINDOW

The Build window—available by choosing View>Messages—displays the messages generated when building a build configuration. When opened, this window is, by default, grouped together with the other message windows, see Windows, page 317.

Double-clicking a message in the Build window opens the appropriate file for editing, with the insertion point at the correct position.

Right-clicking in the Build window displays a context menu which allows you to copy, select, and clear the contents of the window.

Find in Files window—available by choosing View>Messages—displays the output from the Edit>Find and Replace>Find in Files command. When opened, this
window is, by default, grouped together with the other message windows, see *Windows*, page 317.

![Find in Files window (message window)](image1)

Double-clicking an entry in the window opens the appropriate file with the insertion point positioned at the correct location.

Right-clicking in the Find in Files window displays a context menu which allows you to copy, select, and clear the contents of the window.

![Find in Files window context menu](image2)

**TOOL OUTPUT WINDOW**

The Tool Output window—available by choosing View>Messages—displays any messages output by user-defined tools in the Tools menu, provided that you have selected the option Redirect to Output Window in the Configure Tools dialog box;
see Configure Tools dialog box, page 395. When opened, this window is, by default, grouped together with the other message windows, see Windows, page 317.

Right-clicking in the Tool Output window displays a context menu which allows you to copy, select, and clear the contents of the window.

**DEBUG LOG WINDOW**

The Debug Log window—available by choosing View>Messages—displays debugger output, such as diagnostic messages and trace information. 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 Windows, page 317.
Menus

Right-clicking in the Tool Output window displays a context menu which allows you to copy, select, and clear the contents of the window.

Figure 163: Debug Log window context menu

Menus

These menus are available in the IDE:

- File menu
- Edit menu
- View menu
- Project menu
- Tools menu
- Window menu
- Help menu.

In addition, a set of C-SPY-specific menus become available when you start the debugger. For reference information about these menus, see the chapter C-SPY® reference, page 403.

**FILE MENU**

The File menu provides commands for opening workspaces and source files, saving and printing, and exiting from the IDE.
The menu also includes a numbered list of the most recently opened files and workspaces. To open one of them, choose it from the menu.

<table>
<thead>
<tr>
<th>Menu command</th>
<th>Shortcut</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>New</td>
<td>CTRL+N</td>
<td>Displays a submenu with commands for creating a new workspace, or a new text file.</td>
</tr>
<tr>
<td>Open&gt;File</td>
<td>CTRL+O</td>
<td>Displays a submenu from which you can select a text file or an HTML document to open.</td>
</tr>
<tr>
<td>Open&gt;Workspace</td>
<td></td>
<td>Displays a submenu from which you can select a workspace file to open. Before a new workspace is opened you will be prompted to save and close any currently open workspaces.</td>
</tr>
<tr>
<td>Open&gt;Header/Source File</td>
<td>CTRL+H</td>
<td>Opens the header file or source file that corresponds to the current file, and jumps from the current file to the newly opened file. This command is also available from the context menu available from the editor window.</td>
</tr>
<tr>
<td>Close</td>
<td></td>
<td>Closes the active window. You will be given the opportunity to save any files that have been modified before closing.</td>
</tr>
<tr>
<td>Open Workspace</td>
<td></td>
<td>Displays a dialog box where you can open a workspace file. You will be given the opportunity to save and close any currently open workspace file that has been modified before opening a new workspace.</td>
</tr>
<tr>
<td>Save Workspace</td>
<td></td>
<td>Saves the current workspace file.</td>
</tr>
<tr>
<td>Close Workspace</td>
<td></td>
<td>Closes the current workspace file.</td>
</tr>
</tbody>
</table>

Table 79: File menu commands
### File menu commands (Continued)

<table>
<thead>
<tr>
<th>Menu command</th>
<th>Shortcut</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Save</td>
<td>CTRL+S</td>
<td>Saves the current text file or workspace file.</td>
</tr>
<tr>
<td>Save As</td>
<td></td>
<td>Displays a dialog box where you can save the current file with a new name.</td>
</tr>
<tr>
<td>Save All</td>
<td></td>
<td>Saves all open text documents and workspace files.</td>
</tr>
<tr>
<td>Page Setup</td>
<td></td>
<td>Displays a dialog box where you can set printer options.</td>
</tr>
<tr>
<td>Print</td>
<td>CTRL+P</td>
<td>Displays a dialog box where you can print a text document.</td>
</tr>
<tr>
<td>Recent Files</td>
<td></td>
<td>Displays a submenu where you can quickly open the most recently opened text documents.</td>
</tr>
<tr>
<td>Recent Workspaces</td>
<td></td>
<td>Displays a submenu where you can quickly open the most recently opened workspace files.</td>
</tr>
<tr>
<td>Exit</td>
<td></td>
<td>Exits from the IDE. You will be asked whether to save any changes to text windows before closing them. Changes to the project are saved automatically.</td>
</tr>
</tbody>
</table>

Table 79: File menu commands (Continued)
EDIT MENU

The Edit menu provides several commands for editing and searching.

<table>
<thead>
<tr>
<th>Menu command</th>
<th>Shortcut</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Undo</td>
<td>CTRL+Z</td>
<td>Undoes the last edit made to the current editor window.</td>
</tr>
<tr>
<td>Redo</td>
<td>CTRL+Y</td>
<td>Redoes the last Undo in the current editor window. You can undo and redo an unlimited number of edits independently in each editor window.</td>
</tr>
<tr>
<td>Cut</td>
<td>CTRL+X</td>
<td>The standard Windows command for cutting text in editor windows and text boxes.</td>
</tr>
<tr>
<td>Copy</td>
<td>CTRL+C</td>
<td>The standard Windows command for copying text in editor windows and text boxes.</td>
</tr>
<tr>
<td>Paste</td>
<td>CTRL+V</td>
<td>The standard Windows command for pasting text in editor windows and text boxes.</td>
</tr>
<tr>
<td>Paste Special</td>
<td></td>
<td>Provides you with a choice of the most recent contents of the clipboard to choose from when pasting in editor documents.</td>
</tr>
<tr>
<td>Select All</td>
<td>CTRL+A</td>
<td>Selects all text in the active editor window.</td>
</tr>
</tbody>
</table>

Table 80: Edit menu commands
### Find and Replace

<table>
<thead>
<tr>
<th>Menu command</th>
<th>Shortcut</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Find and Replace&gt;Find</td>
<td>CTRL+F</td>
<td>Displays the Find dialog box where you can search for text within the current editor window. Note that if the insertion point is located in the Memory window when you choose the Find command, the dialog box will contain a different set of options than it would otherwise do. If the insertion point is located in the Trace window when you choose the Find command, the Find in Trace dialog box is opened; the contents of this dialog box depend on the C-SPY driver you are using, see the driver documentation for more information.</td>
</tr>
<tr>
<td>Find Next</td>
<td>F3</td>
<td>Finds the next occurrence of the specified string.</td>
</tr>
<tr>
<td>Find Previous</td>
<td>SHIFT+F3</td>
<td>Finds the previous occurrence of the specified string.</td>
</tr>
<tr>
<td>Find Next (Selected)</td>
<td>CTRL+F3</td>
<td>Searches for the next occurrence of the currently selected text or the word currently surrounding the insertion point.</td>
</tr>
<tr>
<td>Find Previous (Selected)</td>
<td>CTRL+SHIFT+F3</td>
<td>Searches for the previous occurrence of the currently selected text or the word currently surrounding the insertion point.</td>
</tr>
<tr>
<td>Replace</td>
<td>CTRL+H</td>
<td>Displays a dialog box where you can search for a specified string and replace each occurrence with another string. Note that if the insertion point is located in the Memory window when you choose the Replace command, the dialog box will contain a different set of options than it would otherwise do.</td>
</tr>
<tr>
<td>Find in Files</td>
<td></td>
<td>Displays a dialog box where you can search for a specified string in multiple text files; see Find in Files dialog box, page 358.</td>
</tr>
<tr>
<td>Incremental Search</td>
<td>CTRL+I</td>
<td>Displays a dialog box where you can gradually fine-tune or expand the search by continuously changing the search string.</td>
</tr>
<tr>
<td>Go To</td>
<td>CTRL+G</td>
<td>Displays the Go to Line dialog box where you can move the insertion point to a specified line and column in the current editor window.</td>
</tr>
<tr>
<td>Toggle Bookmark</td>
<td>CTRL+F2</td>
<td>Toggles a bookmark at the line where the insertion point is located in the active editor window.</td>
</tr>
<tr>
<td>Go to Bookmark</td>
<td>F2</td>
<td>Moves the insertion point to the next bookmark that has been defined with the Toggle Bookmark command.</td>
</tr>
</tbody>
</table>

*Table 80: Edit menu commands (Continued)*
### Table 80: Edit menu commands (Continued)

<table>
<thead>
<tr>
<th>Menu command</th>
<th>Shortcut</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Navigate Backward</td>
<td>ALT+Left arrow</td>
<td>Navigates backward in the insertion point history. The current position of the insertion point is added to the history by actions like Go to definition and clicking on a result from the Find in Files command.</td>
</tr>
<tr>
<td>Navigate Forward</td>
<td>ALT+Right arrow</td>
<td>Navigates forward in the insertion point history. The current position of the insertion point is added to the history by actions like Go to definition and clicking on a result from the Find in Files command.</td>
</tr>
<tr>
<td>Go to Definition</td>
<td>F12</td>
<td>Shows the declaration of the selected symbol or the symbol where the insertion point is placed. This menu command is available when browse information has been enabled, see Project options, page 386.</td>
</tr>
<tr>
<td>Insert Template</td>
<td>CTRL+SHIFT+SPACE</td>
<td>Displays a list in the editor window from which you can choose a code template to be inserted at the location of the insertion point. If the code template you choose requires any field input, the Template dialog box appears; for information about this dialog box, see Template dialog box, page 361. For information about using code templates, see Using and adding code templates, page 109.</td>
</tr>
<tr>
<td>Edit Templates</td>
<td></td>
<td>Opens the current code template file, where you can modify existing code templates and add your own code templates. For information about using code templates, see Using and adding code templates, page 109.</td>
</tr>
<tr>
<td>Next Error/Tag</td>
<td>F4</td>
<td>If the Messages window contains a list of error messages or the results from a Find in Files search, this command will display the next item from that list in the editor window.</td>
</tr>
<tr>
<td>Previous Error/Tag</td>
<td>SHIFT+F4</td>
<td>If the Messages window contains a list of error messages or the results from a Find in Files search, this command will display the previous item from that list in the editor window.</td>
</tr>
<tr>
<td>Complete</td>
<td>CTRL+SPACE</td>
<td>Attempts to complete the word you have begun to type, basing the guess on the contents of the rest of the editor document.</td>
</tr>
<tr>
<td>Match Brackets</td>
<td></td>
<td>Selects all text between the brackets immediately surrounding the insertion point, increases the selection to the next hierarchic pair of brackets, or beeps if there is no higher bracket hierarchy.</td>
</tr>
</tbody>
</table>
Find dialog box

The Find dialog box is available from the Edit menu. Note that the contents of this dialog box look different if you search in an editor window compared to if you search in the Memory window.

<table>
<thead>
<tr>
<th>Option</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Find what</td>
<td>Selects the text to search for.</td>
</tr>
<tr>
<td>Match case</td>
<td>Searches only occurrences that exactly match the case of the specified text. Otherwise specifying int will also find INT and Int. This option is only available when you search in an editor window.</td>
</tr>
<tr>
<td>Match whole word</td>
<td>Searches the specified text only if it occurs as a separate word. Otherwise specifying int will also find printf, sprintf etc. This option is only available when you search in an editor window.</td>
</tr>
<tr>
<td>Search as hex</td>
<td>Searches for the specified hexadecimal value. This option is only available when you search in the Memory window.</td>
</tr>
<tr>
<td>Find next</td>
<td>Finds the next occurrence of the selected text.</td>
</tr>
<tr>
<td>Find previous</td>
<td>Finds the previous occurrence of the selected text.</td>
</tr>
<tr>
<td>Stop</td>
<td>Stops an ongoing search. This button is only available during a search in the Memory window.</td>
</tr>
</tbody>
</table>

Table 81: Find dialog box options
Replace dialog box

The Replace dialog box is available from the Edit menu.

<table>
<thead>
<tr>
<th>Option</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Find what</td>
<td>Selects the text to search for.</td>
</tr>
<tr>
<td>Replace with</td>
<td>Selects the text to replace each found occurrence in the Replace With box.</td>
</tr>
<tr>
<td>Match whole word</td>
<td>Searches the specified text only if it occurs as a separate word.</td>
</tr>
<tr>
<td></td>
<td>Otherwise int will also find print, sprintf etc. This checkbox is not</td>
</tr>
<tr>
<td></td>
<td>available when you perform the search in the Memory window.</td>
</tr>
<tr>
<td>Search as hex</td>
<td>Searches for the specified hexadecimal value. This checkbox is only</td>
</tr>
<tr>
<td></td>
<td>available when you perform the search in the Memory window.</td>
</tr>
<tr>
<td>Find next</td>
<td>Searches the next occurrence of the text you have specified.</td>
</tr>
<tr>
<td>Replace</td>
<td>Replaces the searched text with the specified text.</td>
</tr>
<tr>
<td>Replace all</td>
<td>Replaces all occurrences of the searched text in the current editor window.</td>
</tr>
</tbody>
</table>

Table 82: Replace dialog box options
Find in Files dialog box

Use the Find in Files dialog box—available from the Edit menu—to search for a string in files.

The result of the search appears in the Find in Files messages window—available from the View menu. You can then go to each occurrence by choosing the Edit>Next Error/Tag command, alternatively by double-clicking the messages in the Find in Files messages window. This opens the corresponding file in an editor window with the insertion point positioned at the start of the specified text. A blue flag in the left-most margin indicates the line.

In the Find in Files dialog box, you specify the search criteria with the following settings.

Find what:

A text field in which you type the string you want to search for or a regular expression. Choose between these commands:

- **Match case**  Searches only for occurrences that exactly match the case of the specified text. Otherwise specifying int will also find INT and Int.
- **Match whole word**  Searches only for the string when it occurs as a separate word (short cut &w). Otherwise int will also find print, sprintf and so on.
Look in

The options in the Look in area let you specify which files you want to search in for a specified string. Choose between:

- **For all projects in workspace**
  The search traverses all projects in the workspace, not just the active project.

- **Project files**
  The search is performed in all files that you have explicitly added to your project.

- **Project files and user include files**
  The search is performed in all files that you have explicitly added to your project and all files that they include, except the include files in the IAR Embedded Workbench installation directory.

- **Project files and all include files**
  The search is performed in all project files that you have explicitly added to your project and all files that they include.

- **Directory**
  The search is performed in the directory that you specify. Recent search locations are saved in the drop-down list. Locate the directory using the browse button.

- **Look in subdirectories**
  The search is performed in the directory that you have specified and all its subdirectories.

File types

This is a filter for choosing which type of files to search; the filter applies to all options in the Look in area. Choose the appropriate filter from the drop-down list. Note that the File types text field is editable, which means that you can add your own filters. Use the * character to indicate zero or more unknown characters of the filters, and the ? character to indicate one unknown character.

Stop

Stops an ongoing search. This function button is only available during an ongoing search.
Incremental Search dialog box

The Incremental Search dialog box—available from the Edit menu—lets you gradually fine-tune or expand the search string.

Find What

Type the string to search for. The search is performed from the location of the insertion point—the start point. Gradually incrementing the search string will gradually expand the search criteria. Backspace will remove a character from the search string; the search is performed on the remaining string and will start from the start point.

If a word in the editor window is selected when you open the Incremental Search dialog box, this word will be displayed in the Find What text box.

Match Case

Use this option to find only occurrences that exactly match the case of the specified text. Otherwise searching for int will also find INT and Int.

Function buttons

<table>
<thead>
<tr>
<th>Function button</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Find Next</td>
<td>Searches for the next occurrence of the current search string. If the Find What text box is empty when you click the Find Next button, a string to search for will automatically be selected from the drop-down list. To search for this string, click Find Next.</td>
</tr>
<tr>
<td>Close</td>
<td>Closes this dialog box.</td>
</tr>
</tbody>
</table>

Table 83: Incremental Search function buttons
Template dialog box

Use the Template dialog box to specify any field input that is required by the source code template you insert. This dialog box appears when you insert a code template that requires any field input.

![Figure 168: Template dialog box](image)

**Note:** This figure reflects the default code template that can be used for automatically inserting code for a `for` loop.

The contents of this dialog box match the code template. In other words, which fields that appear depends on how the code template is defined.

At the bottom of the dialog box, the code that would result from the code template is displayed.

For more information about using code templates, see *Using and adding code templates*, page 109.
VIEW MENU

With the commands on the View menu you can choose what to display in the IAR Embedded Workbench IDE. During a debug session you can also open debugger-specific windows from the View menu.

<table>
<thead>
<tr>
<th>Menu command</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Messages</td>
<td>Opens a submenu which gives access to the message windows—Build, Find in Files, Tool Output, Debug Log—that display messages and text output from the IAR Embedded Workbench commands. If the window you choose from the menu is already open, it becomes the active window.</td>
</tr>
<tr>
<td>Workspace</td>
<td>Opens the current Workspace window.</td>
</tr>
<tr>
<td>Source Browser</td>
<td>Opens the Source Browser window.</td>
</tr>
<tr>
<td>Breakpoints</td>
<td>Opens the Breakpoints window.</td>
</tr>
<tr>
<td>Toolbars</td>
<td>The options <strong>Main</strong> and <strong>Debug</strong> toggle the two toolbars on and off.</td>
</tr>
<tr>
<td>Status bar</td>
<td>Toggles the status bar on and off.</td>
</tr>
<tr>
<td>Debugger windows</td>
<td>During a debugging session, the various debugging windows are also available from the View menu: Disassembly window, Memory window, Symbolic Memory window, Register window, Watch window, Locals window, Statics window, Auto window, Live Watch window, Quick Watch window, Call Stack window, Terminal I/O window, Code Coverage window</td>
</tr>
</tbody>
</table>

Table 84: View menu commands
**PROJECT MENU**

The **Project** menu provides commands for working with workspaces, projects, groups, and files, and for specifying options for the build tools, and running the tools on the current project.

<table>
<thead>
<tr>
<th>Menu command</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Add Files...</td>
<td>Displays a dialog box that where you can select which files to include to the current project.</td>
</tr>
<tr>
<td>Add Group...</td>
<td>Displays a dialog box where you can create a new group. In the <strong>Group Name</strong> text box, specify the name of the new group. For more information about groups, see Groups, page 89.</td>
</tr>
</tbody>
</table>

**Figure 170: Project menu**

These commands are available on the menu:
<table>
<thead>
<tr>
<th>Menu Command</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Import File List</td>
<td>Displays a standard <strong>Open</strong> dialog box where you can import information about files and groups from projects created using another IAR Systems tool chain. To import information from project files which have one of the older filename extensions <em>pew</em> or <em>prj</em> you must first have exported the information using the context menu command <strong>Export File List</strong> available in your own IAR Embedded Workbench.</td>
</tr>
<tr>
<td>Edit Configurations</td>
<td>Displays the <strong>Configurations for project</strong> dialog box, where you can define new or remove existing build configurations.</td>
</tr>
<tr>
<td>Remove</td>
<td>In the Workspace window, removes the selected item from the workspace.</td>
</tr>
<tr>
<td>Create New Project</td>
<td>Displays a dialog box where you can create a new project and add it to the workspace.</td>
</tr>
<tr>
<td>Add Existing Project</td>
<td>Displays a dialog box where you can add an existing project to the workspace.</td>
</tr>
<tr>
<td>Options (Alt+F7)</td>
<td>Displays the <strong>Options for node</strong> dialog box, where you can set options for the build tools on the selected item in the Workspace window. You can set options on the entire project, on a group of files, or on an individual file.</td>
</tr>
<tr>
<td>Source Code Control</td>
<td>Opens a submenu with commands for source code control, see <strong>Source Code Control menu</strong>, page 324.</td>
</tr>
<tr>
<td>Make (F7)</td>
<td>Brings the current build configuration up to date by compiling, assembling, and linking only the files that have changed since the last build.</td>
</tr>
<tr>
<td>Compile (Ctrl+F7)</td>
<td>Compiles or assembles the currently selected file, files, or group. One or more files can be selected in the Workspace window—all files in the same project, but not necessarily in the same group. You can also select the editor window containing the file you want to compile. The <strong>Compile</strong> command is only enabled if every file in the selection is individually suitable for the command. You can also select a group, in which case the command is applied to each file in the group (including inside nested groups) that can be compiled, even if the group contains files that cannot be compiled, such as header files. If the selected file is part of a multi-file compilation group, the command will still only affect the selected file.</td>
</tr>
<tr>
<td>Rebuild All</td>
<td>Rebuilds and relinks all files in the current target.</td>
</tr>
<tr>
<td>Clean</td>
<td>Removes any intermediate files.</td>
</tr>
</tbody>
</table>

*Table 85: Project menu commands (Continued)*
In the **Erase Memory** dialog box—displayed when you have chosen the **Project>Erase Memory** command and your flash memory system configuration file (filename...
extension .board) specifies two or more flash memories—you can to erase one or more of the flash memories.

![Erase Memory dialog box](image)

Each line lists the path to the flash memory device configuration file (filename extension .flash) and the associated memory range. Select the memory you want to erase.

These buttons are available:

- **Erase all**: All memories listed in the dialog box are erased, regardless of individually selected lines.
- **Erase**: Erases the selected memories.
- **Cancel**: Cancels the dialog box.

**Argument variables summary**

You can use these argument variables for paths and arguments:

<table>
<thead>
<tr>
<th>Variable</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>$CONFIG_NAME$</td>
<td>The name of the current build configuration, for example Debug or Release.</td>
</tr>
<tr>
<td>$CUR_DIR$</td>
<td>Current directory</td>
</tr>
<tr>
<td>$CUR_LINE$</td>
<td>Current line</td>
</tr>
<tr>
<td>$DATE$</td>
<td>Today's date</td>
</tr>
<tr>
<td>$EW_DIR$</td>
<td>Top directory of IAR Embedded Workbench, for example <code>c:\program files\iar systems\embedded workbench 5.5</code></td>
</tr>
<tr>
<td>$EXE_DIR$</td>
<td>Directory for executable output</td>
</tr>
<tr>
<td>$FILE_BNAME$</td>
<td>Filename without extension</td>
</tr>
</tbody>
</table>

Table 86: Argument variables
Configurations for project dialog box

In the Configuration for project dialog box—available by choosing Project>Editor>Configurations—you can define new build configurations for the selected project; either entirely new, or based on a previous project.

Table 86: Argument variables (Continued)

<table>
<thead>
<tr>
<th>Variable</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>$FILE_BPATH$</td>
<td>Full path without extension</td>
</tr>
<tr>
<td>$FILE_DIR$</td>
<td>Directory of active file, no filename</td>
</tr>
<tr>
<td>$FILE_FNAME$</td>
<td>Filename of active file without path</td>
</tr>
<tr>
<td>$FILE_PATH$</td>
<td>Full path of active file (in Editor, Project, or Message window)</td>
</tr>
<tr>
<td>$LIST_DIR$</td>
<td>Directory for list output</td>
</tr>
<tr>
<td>$OBJ_DIR$</td>
<td>Directory for object output</td>
</tr>
<tr>
<td>$PROJ_DIR$</td>
<td>Project directory</td>
</tr>
<tr>
<td>$PROJ_FNAME$</td>
<td>Project file name without path</td>
</tr>
<tr>
<td>$PROJ_PATH$</td>
<td>Full path of project file</td>
</tr>
<tr>
<td>$TARGET_DIR$</td>
<td>Directory of primary output file</td>
</tr>
<tr>
<td>$TARGET_BNAME$</td>
<td>Filename without path of primary output file and without extension</td>
</tr>
<tr>
<td>$TARGET_BPATH$</td>
<td>Full path of primary output file without extension</td>
</tr>
<tr>
<td>$TARGET_FNAME$</td>
<td>Filename without path of primary output file</td>
</tr>
<tr>
<td>$TARGET_PATH$</td>
<td>Full path of primary output file</td>
</tr>
<tr>
<td>$TOOLKIT_DIR$</td>
<td>Directory of the active product, for example c:\program files\iar systems\embedded workbench 5.n.arm</td>
</tr>
<tr>
<td>$USER_NAME$</td>
<td>Your host login name</td>
</tr>
<tr>
<td>$<em>ENVVAR</em>$_</td>
<td>The environment variable ENVVAR. Any name within $ and $ will be expanded to that system environment variable.</td>
</tr>
</tbody>
</table>

Figure 172: Configurations for project dialog box
The dialog box contains the following:

<table>
<thead>
<tr>
<th>Operation</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Configurations</td>
<td>Lists existing configurations, which can be used as templates for new configurations.</td>
</tr>
<tr>
<td>New</td>
<td>Opens a dialog box where you can define new build configurations.</td>
</tr>
<tr>
<td>Remove</td>
<td>Removes the configuration that is selected in the Configurations list.</td>
</tr>
</tbody>
</table>

Table 87: Configurations for project dialog box options

New Configuration dialog box

In the New Configuration dialog box—available by clicking New in the Configurations for project dialog box—you can define new build configurations; either entirely new, or based on any currently defined configuration.

The dialog box contains the following:

<table>
<thead>
<tr>
<th>Item</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Name</td>
<td>The name of the build configuration.</td>
</tr>
<tr>
<td>Tool chain</td>
<td>The target to build for. If you have several versions of IAR Embedded Workbench for different targets installed on your host computer, the drop-down list can contain these targets.</td>
</tr>
<tr>
<td>Based on configuration</td>
<td>A currently defined build configuration that you want the new configuration to be based on. The new configuration will inherit the project settings and information about the factory settings from the old configuration. If you select None, the new configuration will have default factory settings and not be based on an already defined configuration.</td>
</tr>
</tbody>
</table>

Table 88: New Configuration dialog box options
The Create New Project dialog box is available from the Project menu, and lets you create a new project based on a template project. Template projects are available for C/C++ applications, assembler applications, and library projects. You can also create your own template projects.

Figure 174: Create New Project dialog box

The dialog box contains the following:

<table>
<thead>
<tr>
<th>Item</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Tool chain</td>
<td>Specifies the default factory settings—either Debug or Release—that you want to apply to your new build configuration. These factory settings will be used by your project if you press the Factory Settings button in the Options dialog box.</td>
</tr>
<tr>
<td>Project templates</td>
<td>Lists all available template projects that you can base a new project on.</td>
</tr>
<tr>
<td>Description</td>
<td></td>
</tr>
</tbody>
</table>

Table 89: Description of Create New Project dialog box
The **Options** dialog box is available from the **Project** menu.

In the **Category** list you can select the build tool for which you want to set options. The options available in the **Category** list will depend on the tools installed in your IAR Embedded Workbench IDE, and will typically include these options:

<table>
<thead>
<tr>
<th>Category</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>General Options</td>
<td>General options</td>
</tr>
<tr>
<td>C/C++ Compiler</td>
<td>IAR C/C++ Compiler options</td>
</tr>
<tr>
<td>Assembler</td>
<td>IAR Assembler options</td>
</tr>
<tr>
<td>Converter</td>
<td>Options for converting ELF output to Motorola or Intel-standard.</td>
</tr>
<tr>
<td>Custom Build</td>
<td>Options for extending the tool chain</td>
</tr>
<tr>
<td>Build Actions</td>
<td>Options for pre-build and post-build actions</td>
</tr>
<tr>
<td>Linker</td>
<td>IAR ILINK Linker options. This category is available for application projects.</td>
</tr>
<tr>
<td>Library Builder</td>
<td>Library builder options. This category is available for library projects.</td>
</tr>
<tr>
<td>Debugger</td>
<td>IAR C-SPY Debugger options</td>
</tr>
<tr>
<td>Simulator</td>
<td>Simulator-specific options</td>
</tr>
</tbody>
</table>

Table 90: Project option categories

**Note:** Additional debugger categories might be available depending on the debugger drivers installed.

Selecting a category displays one or more pages of options for that component of the IDE.

For detailed information about each option, see the option reference chapters:

- **General options**
- **Compiler options**
- **Assembler options**
- **Converter options**
- **Custom build options**
- **Build actions options**
- **Linker options**
- **Library builder options**
- **Debugger options**
For information about the options related to available hardware debugger systems, see Part 6. C-SPY hardware debugger systems.

**Batch Build dialog box**

The **Batch Build** dialog box—available by choosing **Project>Batch build**—lists all defined batches of build configurations.

![Batch Build dialog box](image)

The dialog box contains the following:

<table>
<thead>
<tr>
<th>Item</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Batches</td>
<td>Lists all currently defined batches of build configurations.</td>
</tr>
<tr>
<td>New</td>
<td>Displays the <strong>Edit Batch Build</strong> dialog box, where you can define new batches of build configurations.</td>
</tr>
<tr>
<td>Remove</td>
<td>Removes the selected batch.</td>
</tr>
<tr>
<td>Edit</td>
<td>Displays the <strong>Edit Batch Build</strong> dialog box, where you can modify already defined batches.</td>
</tr>
<tr>
<td>Build</td>
<td>Consists of the three build commands <strong>Make</strong>, <strong>Clean</strong>, and <strong>Rebuild All</strong>.</td>
</tr>
</tbody>
</table>

*Table 91: Description of the Batch Build dialog box*
**Edit Batch Build dialog box**

In the **Edit Batch Build** dialog box—available from the **Batch Build** dialog box—you can create new batches of build configurations, and edit already existing batches.

![Edit Batch Build dialog box](image)

The dialog box contains the following:

<table>
<thead>
<tr>
<th>Item</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Name</td>
<td>The name of the batch.</td>
</tr>
<tr>
<td>Available configurations</td>
<td>Lists all build configurations that are part of the workspace.</td>
</tr>
<tr>
<td>Configurations to build</td>
<td>Lists all the build configurations you select to be part of a named batch.</td>
</tr>
</tbody>
</table>

*Table 92: Description of the Edit Batch Build dialog box*

To move appropriate build configurations from the **Available configurations** list to the **Configurations to build** list, use the arrow buttons. Note also that you can drag the build configurations in the **Configurations to build** field to specify the order between the build configurations.
TOOLS MENU

The Tools menu provides commands for customizing the environment, such as changing common fonts and shortcut keys.

It is a user-configurable menu to which you can add tools for use with IAR Embedded Workbench. Thus, it might look different depending on which tools have been preconfigured to appear as menu items. See Configure Tools dialog box, page 395.

<table>
<thead>
<tr>
<th>Tools menu commands</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Menu command</strong></td>
</tr>
<tr>
<td>Options</td>
</tr>
<tr>
<td>Configure Tools</td>
</tr>
<tr>
<td>Filename Extensions</td>
</tr>
<tr>
<td>Configure Viewers</td>
</tr>
<tr>
<td>Notepad</td>
</tr>
</tbody>
</table>

Table 93: Tools menu commands
COMMON FONTS OPTIONS

Use the Common Fonts options—available by choosing Tools>Options—for configuring the fonts used for all project windows except the editor windows.

With the Font buttons you can change the fixed and proportional width fonts, respectively.

Any changes to the Fixed Width Font options will apply to the Disassembly, Register, and Memory windows. Any changes to the Proportional Width Font options will apply to all other windows.

None of the settings made on this page apply to the editor windows. For information about how to change the font in the editor windows, see Editor Colors and Fonts options, page 383.
KEY BINDINGS OPTIONS

Use the Key Bindings options—available by choosing Tools>Options—to customize the shortcut keys used for the IDE menu commands.

Menu

Use the drop-down list to choose the menu you want to edit. Any currently defined shortcut keys are shown in the scroll list under the drop-down list.

Command

All commands available on the selected menu are listed in the Commands column. Select the menu command for which you want to configure your own shortcut keys.

Press shortcut key

Use the text field to type the key combination you want to use as shortcut key. You cannot set or add a shortcut if it is already used by another command.

Primary

The shortcut key will be displayed next to the command on the menu. Click the Set button to set the combination for the selected command, or the Clear button to delete the shortcut.
**Menu**

**Alias**

The shortcut key will work but not be displayed on the menu. Click either the **Add** button to make the key take effect for the selected command, or the **Clear** button to delete the shortcut.

**Reset All**

Reverts all command shortcut keys to the factory settings.

**LANGUAGE OPTIONS**

Use the **Language** options—available by choosing **Tools** > **Options**—to specify the language to be used in windows, menus, dialog boxes, etc.

**Language**

Use the drop-down list to choose the language to be used. In the IAR Embedded Workbench IDE, only **English (United States)** is available.

**Note:** If you have IAR Embedded Workbench IDE installed for several different tool chains in the same directory, the IDE might be in mixed languages if the tool chains are available in different languages.
EDITOR OPTIONS

Use the Editor options—available by choosing Tools>Options—to configure the editor.

For more information about the IAR Embedded Workbench IDE Editor and how to use it, see Editing, page 105.

**Tab size**

Use this option to specify the number of character spaces corresponding to each tab.

**Indent size**

Use this option to specify the number of character spaces to be used for indentation.

**Tab Key Function**

Use this option to specify how the Tab key is used. Choose between:

- Insert tab
- Indent with spaces
EOL character
Use this option to select the line break character to be used when editor documents are saved. Choose between:

- **PC** (default)  Windows and DOS end of line characters. The PC format is used by default.
- **Unix**  UNIX end of line characters.
- **Preserve**  The same end of line character as the file had when it was opened, either PC or UNIX. If both types or neither type are present in the opened file, PC end of line characters are used.

Show right margin
The area of the editor window outside the right-side margin is displayed as a light gray field. You can choose to set the size of the text field between the left-side margin and the right-side margin. Choose to set the size based on:

- **Printing edge**  Size based on the printable area which is based on general printer settings.
- **Columns**  Size based on number of columns.

Syntax highlighting
Use this option to make the editor display the syntax of C or C++ applications in different text styles.

To read more about syntax highlighting, see *Editor Colors and Fonts options*, page 383, and *Syntax coloring*, page 107.

Auto indent
Use this option to ensure that when you press Return, the new line is indented automatically. For C/C++ source files, indentation is performed as configured in the *Configure Auto Indent* dialog box. Click the *Configure* button to open the dialog box where you can configure the automatic indentation; see *Configure Auto Indent dialog box*, page 379. For all other text files, the new line will have the same indentation as the previous line.

Show line numbers
Use this option to display line numbers in the editor window.
Scan for changed files
Use this option to reload files that have been modified by another tool.
If a file is open in the IDE, and the same file has concurrently been modified by another tool, the file will be automatically reloaded in the IDE. However, if you already started to edit the file, you will be prompted before the file is reloaded.

Show bookmarks
Use this option to display a column on the left side in the editor window, with icons for compiler errors and warnings, Find in Files results, user bookmarks and breakpoints.

Enable virtual space
Use this option to allow the insertion point to move outside the text area.

Remove trailing blanks
Use this option to remove trailing blanks from files when they are saved to disk. Trailing blanks are blank spaces between the last non-blank character and the end of line character.

CONFIGURE AUTO INDENT DIALOG BOX
Use the Configure Auto Indent dialog box to configure the automatic indentation performed by the editor for C/C++ source code. To open the dialog box:

1. Choose Tools>Options.
2. Click the Editor tab.
3. Select the Auto indent option.
4 Click the **Configure** button.

![Configure Auto Indent dialog box](image)

To read more about indentation, see *Automatic text indentation*, page 108.

**Opening Brace (a)**

Use the text box to type the number of spaces used for indenting an opening brace.

**Body (b)**

Use the text box to type the number of additional spaces used for indenting code after an opening brace, or a statement that continues onto a second line.

**Label (c)**

Use the text box to type the number of additional spaces used for indenting a label, including case labels.

**Sample code**

This area reflects the settings made in the text boxes for indentation. All indentations are relative to the preceding line, statement, or other syntactic structures.
EXTERNAL EDITOR OPTIONS

Use the External Editor options—available by choosing Tools>Options—to specify an external editor of your choice.

![External Editor options dialog box](image)

Figure 183: External Editor options

Note: The appearance of this dialog box depends on the setting of the Type option.

See also Using an external editor, page 112.

**Use External Editor**

Use this option to enable the use of an external editor.

**Type**

Use the drop-down list to select the type of interface. Choose between:

- Command Line
- DDE (Windows Dynamic Data Exchange).

**Editor**

Use the text field to specify the filename and path of your external editor. A browse button is available for your convenience.

**Arguments**

Use the text field to specify any arguments to pass to the editor. Only applicable if you have selected Command Line as the interface type, see Type, page 381.
**Service**

Use the text field to specify the DDE service name used by the editor. Only applicable if you have selected DDE as the interface type, see *Type*, page 381.

The service name depends on the external editor that you are using. Refer to the user documentation of your external editor to find the appropriate settings.

**Command**

Use the text field to specify a sequence of command strings to send to the editor. The command strings should be typed as:

```
DDE-Topic CommandString
DDE-Topic CommandString
```

Only applicable if you have selected DDE as the interface type, see *Type*, page 381.

The command strings depend on the external editor that you are using. Refer to the user documentation of your external editor to find the appropriate settings.

*Note:* You can use variables in arguments. See *Argument variables summary*, page 366, for information about available argument variables.

**EDITOR SETUP FILES OPTIONS**

Use the Editor Setup Files options—available by choosing *Tools>Options*—to specify setup files for the editor.

![Figure 184: Editor Setup Files options](image-url)
Use Custom Keyword File

Use this option to specify a text file containing keywords that you want the editor to highlight. For information about syntax coloring, see Syntax coloring, page 107.

Use Code Templates

Use this option to specify a text file with code templates that you can use for inserting frequently used code in your source file. For information about using code templates, see Using and adding code templates, page 109.

EDITOR COLORS AND FONTS OPTIONS

Use the Editor Colors and Fonts options—available by choosing Tools>Options—to specify the colors and fonts used for text in the editor windows.

Editor Font

Press the Font button to open the standard Font dialog box where you can choose the font and its size to be used in editor windows.
Syntax Coloring

Use the Syntax Coloring options to choose the color and type style for selected item.

Use these options:

Scroll-bar list

Lists the possible items for which you can specify color and type style. Select an item in the list and choose the color and type style for it. Note that the User keyword list entry refers to the keywords that you have listed in the custom keyword file, see Use Custom Keyword File, page 383.

Color

Provides a list of colors to choose from for the selected element. Choose Custom from the list to define your own color. The standard Windows Color dialog box appears.

Type Style

Provides a list of type styles to choose from.

Sample

Displays the current setting for the selected item.

Background Color

Provides a list of background colors to choose from for the editor window.

The keywords controlling syntax highlighting for assembler and C or C++ source code are specified in the files syntax_icc.cfg and syntax_asm.cfg, respectively. These files are located in the config directory.
MESSAGES OPTIONS

Use the Messages options—available by choosing Tools>Options—to choose the amount of output in the Build messages window.

Show build messages

Use this drop-down menu to specify the amount of output in the Build messages window. Choose between:

- **All**: Shows all messages, including compiler and linker information.
- **Messages**: Shows messages, warnings, and errors.
- **Warnings**: Shows warnings and errors.
- **Errors**: Shows errors only.

Log File

Use these options to write build messages to a log file. To enable the options, select the Enable build log file option. Choose between:

- **Append to end of file**: Appends the messages at the end of the specified file.
**Menus**

**Overwrite old file**  
Replaces the contents in the file you specify.

Type the filename you want to use in the text box. A browse button is available for your convenience.

**Enable All Dialogs**  
The Enable All Dialogs button enables all suppressed dialog boxes.

You can suppress some dialog boxes by selecting a Don’t show again check box, for example:

![Message dialog box containing a Don’t show again option](image)

**PROJECT OPTIONS**

Use the Project options—available by choosing Tools>Options—to set options for the Make and Build commands.

![Project options](image)
These options are available:

<table>
<thead>
<tr>
<th>Option</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Stop build operation on</td>
<td>Specifies when the build operation should stop.</td>
</tr>
<tr>
<td></td>
<td><strong>Never</strong>: Do not stop.</td>
</tr>
<tr>
<td></td>
<td><strong>Warnings</strong>: Stop on warnings and errors.</td>
</tr>
<tr>
<td></td>
<td><strong>Errors</strong>: Stop on errors.</td>
</tr>
<tr>
<td>Save editor windows before</td>
<td><strong>Always</strong>: Always save before Make or Build.</td>
</tr>
<tr>
<td>building</td>
<td><strong>Ask</strong>: Prompt before saving.</td>
</tr>
<tr>
<td></td>
<td><strong>Never</strong>: Do not save.</td>
</tr>
<tr>
<td>Save workspace and projects</td>
<td><strong>Always</strong>: Always save before Make or Build.</td>
</tr>
<tr>
<td>before building</td>
<td><strong>Ask</strong>: Prompt before saving.</td>
</tr>
<tr>
<td></td>
<td><strong>Never</strong>: Do not save.</td>
</tr>
<tr>
<td>Make before debugging</td>
<td><strong>Always</strong>: Always perform the Make command before debugging.</td>
</tr>
<tr>
<td></td>
<td><strong>Ask</strong>: Always prompt before performing the Make command.</td>
</tr>
<tr>
<td></td>
<td><strong>Never</strong>: Do not perform the Make command before debugging.</td>
</tr>
<tr>
<td>Reload last workspace at startup</td>
<td>Select this option if you want the last active workspace to load automatically the next time you start IAR Embedded Workbench.</td>
</tr>
<tr>
<td>Play a sound after build operations</td>
<td>Plays a sound when the build operations are finished.</td>
</tr>
<tr>
<td>Generate browse information</td>
<td>Enables the use of the Source Browser window.</td>
</tr>
</tbody>
</table>

Table 94: Project IDE options
SOURCE CODE CONTROL OPTIONS

Use the Source Code Control options—available by choosing Tools>Options—to configure the interaction between an IAR Embedded Workbench project and an SCC project.

**Keep items checked out when checking in**

Determines the default setting for the option Keep Checked Out in the Check In Files dialog box; see Check In Files dialog box, page 327.

**Save editor windows before performing source code control commands**

Specifies whether editor windows should be saved before you perform any source code control commands. Choose between:

- **Ask**
  When you perform any source code control commands, you will be asked about saving editor windows first.

- **Never**
  Editor windows will never be saved first when you perform any source code control commands.

- **Always**
  Editor windows will always be saved first when you perform any source code control commands.
DEBUGGER OPTIONS

Use the Debugger options—available by choosing Tools>Options—for configuring the debugger environment.

When source resolves to multiple function instances

Some source code corresponds to multiple code instances, for example template code. When specifying a source location in such code, for example when setting a source breakpoint, you can make C-SPY act on all instances or a subset of instances. Use the Automatically choose all instances option to let C-SPY act on all instances without asking first.

Source code color in Disassembly window

Use the Color button to select the color of the source code in the Disassembly window. To define your own color, choose Custom from the list of colors. The standard Windows Color dialog box appears.

Step into functions

Use this option to control the behavior of the Step Into command. Choose between:

All functions  The debugger will step into all functions.
Functions with source only

The debugger will only step into functions for which the source code is known. This helps you avoid stepping into library functions or entering disassembly mode debugging.

STL container expansion

The Depth value decides how many elements that are shown initially when a container value is expanded, for example, the Watch window. To show additional elements, click the expansion arrow.

Update intervals

The Update intervals options specify how often the contents of the Live Watch window and the Memory window are updated.

These options are available if the C-SPY driver you are using has access to the target system memory while executing your application.

Default integer format

Use the drop-down list to set the default integer format in the Watch, Locals, and related windows.

STACK OPTIONS

Use the Stack options—available by choosing Tools>Options or from the context menu in the Memory window—to set options specific to the Stack window.

![Figure 191: Stack options](image-url)

IAR Embedded Workbench® IDE

User Guide for ARM
Enable graphical stack display and stack usage tracking

Use this option to enable the graphical stack bar available at the top of the Stack window. At the same time, it enables the functionality needed to detect stack overflows. To read more about the stack bar and the information it provides, see The graphical stack bar, page 434.

% stack usage threshold

Use this text field to specify the percentage of stack usage above which C-SPY should issue a warning for stack overflow.

Warn when exceeding stack threshold

Use this option to make C-SPY issue a warning when the stack usage exceeds the threshold specified in the % stack usage threshold option.

Warn when stack pointer is out of bounds

Use this option to make C-SPY issue a warning when the stack pointer is outside the stack memory range.

Stack pointer(s) not valid until reaching

Use this option to specify a location in your application code from where you want the stack display and verification to occur. The Stack window will not display any information about stack usage until execution has reached this location. By default, C-SPY will not track the stack usage before the main function. If your application does not have a main function, for example, if it is an assembler-only project, you should specify your own start label. If this option is used, after each reset C-SPY keeps a breakpoint on the given location until it is reached.

Typically, the stack pointer is set up in the system initialization code cstartup, but not necessarily from the very first instruction. If you use this option, you can avoid incorrect warnings or misleading stack display for this part of the application.

Warnings

You can choose to issue warnings using one of these options:

Log

Warnings are issued in the Debug Log window

Log and alert

Warnings are issued in the Debug Log window and as alert dialog boxes.
Limit stack display to

Use this option to limit the amount of memory displayed in the Stack window by specifying a number, counting from the stack pointer. This can be useful if you have a big stack or if you are only interested in the topmost part of the stack. Using this option can improve the Stack window performance, especially if reading memory from the target system is slow. By default, the Stack window shows the whole stack, or in other words, from the stack pointer to the bottom of the stack. If the debugger cannot determine the memory range for the stack, the byte limit is used even if the option is not selected.

Note: The Stack window does not affect the execution performance of your application, but it might read a large amount of data to update the displayed information when the execution stops.

REGISTER FILTER OPTIONS

Use the Register Filter options—available by choosing Tools>Options when the debugger is running—to display registers in the Register window in groups you have created yourself. For more information about register groups, see Register groups, page 152.

These options are available:

<table>
<thead>
<tr>
<th>Option</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Use register filter</td>
<td>Enables the usage of register filters.</td>
</tr>
</tbody>
</table>

Figure 192: Register Filter options
TERMINAL I/O OPTIONS

Use the Terminal I/O options—available by choosing Tools>Options when the debugger is running—to configure the C-SPY terminal I/O functionality.

<table>
<thead>
<tr>
<th>Option</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Filter Files</td>
<td>Displays a dialog box where you can select or create a new filter file.</td>
</tr>
<tr>
<td>Groups</td>
<td>Lists available groups in the register filter file, alternatively displays the new register group.</td>
</tr>
<tr>
<td>New Group</td>
<td>The name for the new register group.</td>
</tr>
<tr>
<td>Group members</td>
<td>Lists the registers selected from the register scroll bar window.</td>
</tr>
<tr>
<td>Base</td>
<td>Changes the default integer base.</td>
</tr>
</tbody>
</table>

Table 95: Register Filter options (Continued)

**TERMINAL I/O OPTIONS**

**Keyboard**

Use the Keyboard option to make the input characters be read from the keyboard. Choose between:

- **Buffered**
  - Input characters are buffered.
- **Direct**
  - Input characters are not buffered.
Menus

File
Use the File option to make the input characters be read from a file. A browse button is available for locating the file. Choose between:

**Text**
Input characters are read from a text file.

**Binary**
Input characters are read from a binary file.

Input Echoing
Input characters can be echoed either in a log file, or in the C-SPY Terminal I/O window. To echo input in a file requires that you have enabled the option Debug>Logging>Enable log file.

Show target reset in Terminal I/O window
When the target resets, a message is displayed in the C-SPY Terminal I/O window.
CONFIGURE TOOLS DIALOG BOX

In the Configure Tools dialog box—available from the Tools menu—you can specify a user-defined tool to add to the Tools menu.

Figure 194: Configure Tools dialog box

Note: If you intend to add an external tool to the standard build tool chain, see Extending the tool chain, page 101.

These options are available:

<table>
<thead>
<tr>
<th>Option</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Menu Content</td>
<td>Lists all available user defined menu commands.</td>
</tr>
<tr>
<td>Menu Text</td>
<td>Specifies the text for the menu command. If you add the sign ‘&amp;’, the following letter, N in this example, will appear as the mnemonic key for this command. The text you type in this field will be reflected in the Menu Content field.</td>
</tr>
<tr>
<td>Command</td>
<td>Specifies the command, and its path, to be run when you choose the command from the menu. A browse button is available for your convenience.</td>
</tr>
</tbody>
</table>

Table 96: Configure Tools dialog box options
Menus

<table>
<thead>
<tr>
<th>Option</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Argument</td>
<td>Optionally type an argument for the command.</td>
</tr>
<tr>
<td>Initial Directory</td>
<td>Specifies an initial working directory for the tool.</td>
</tr>
<tr>
<td>Redirect to Output window</td>
<td>Specifies any console output from the tool to the Tool Output page in the Messages window. Tools that are launched with this option cannot receive any user input, for instance input from the keyboard. Tools that require user input or make special assumptions regarding the console that they execute in, will not work at all if launched with this option.</td>
</tr>
<tr>
<td>Prompt for Command Line</td>
<td>Displays a prompt for the command line argument when the command is chosen from the Tools menu.</td>
</tr>
<tr>
<td>Tool Available</td>
<td>Specifies in which context the tool should be available, only when debugging or only when not debugging.</td>
</tr>
</tbody>
</table>

**Note:** You can use variables in the arguments, which allows you to set up useful tools such as interfacing to a command line revision control system, or running an external tool on the selected file.

To remove a command from the Tools menu, select it in this list and click Remove.

Click OK to confirm the changes you have made to the Tools menu.

The menu items you have specified will then be displayed on the Tools menu.

---

**Specifying command line commands or batch files**

Command line commands or batch files must be run from a command shell, so to add these to the Tools menu you can specify an appropriate command shell in the Command text box. These are the command shells that you can enter as commands:

<table>
<thead>
<tr>
<th>Command shell</th>
<th>System</th>
</tr>
</thead>
<tbody>
<tr>
<td>cmd.exe (recommended) or command.com</td>
<td>Windows XP/Vista/7</td>
</tr>
</tbody>
</table>

**Table 97: Command shells**

For an example, see Adding command line commands, page 86.
FILENAME EXTENSIONS DIALOG BOX

In the Filename Extensions dialog box—available from the Tools menu—you can customize the filename extensions recognized by the build tools. This is useful if you have many source files that have a different filename extension.

If you have an IAR Embedded Workbench for a different microprocessor installed on your host computer, it can appear in the Tool Chain box. In that case you should select the tool chain you want to customize.

![Filename Extensions dialog box](image)

*Figure 196: Filename Extensions dialog box*

Note the * sign which indicates user-defined overrides. If there is no * sign, factory settings are used.

Click Edit to open the Filename Extension Overrides dialog box.
FILENAME EXTENSION OVERRIDES DIALOG BOX

The Filename Extension Overrides dialog box—available by clicking Edit in the Filename Extensions dialog box—lists the available tools in the build chain, their factory settings for filename extensions, and any defined overrides.

<table>
<thead>
<tr>
<th>Tool</th>
<th>Factory Setting</th>
<th>Override</th>
</tr>
</thead>
<tbody>
<tr>
<td>GCC Compiler</td>
<td>c.o, c.c, cpp</td>
<td>c.o, c.c</td>
</tr>
<tr>
<td>Assembler</td>
<td>.asm, s, sjmps</td>
<td></td>
</tr>
<tr>
<td>Browser Into Compiler</td>
<td>c.o, c.c, cpp</td>
<td></td>
</tr>
<tr>
<td>Linker</td>
<td>.o, .lib</td>
<td></td>
</tr>
<tr>
<td>Library Builder</td>
<td>.dll</td>
<td></td>
</tr>
<tr>
<td>Browser Into Builder</td>
<td>.bin</td>
<td></td>
</tr>
</tbody>
</table>

Select the tool for which you want to define more recognized filename extensions, and click Edit to open the Edit Filename Extensions dialog box.

EDIT FILENAME EXTENSIONS DIALOG BOX

The Edit File Extensions dialog box—available by clicking Edit in the Filename Extension Overrides dialog box—lists the filename extensions accepted by default, and you can also define new filename extensions.

Click Override and type the new filename extension you want to be recognized. Extensions can be separated by commas or semicolons, and should include the leading period.
CONFIGURE VIEWERS DIALOG BOX

The Configure Viewers dialog box—available from the Tools menu—lists the filename extensions of document formats that IAR Embedded Workbench can handle, and which viewer application that are used for opening the document type. Explorer Default in the Action column means that the default application associated with the specified type in Windows Explorer is used for opening the document type.

![Configure Viewers dialog box](image)

To specify how to open a new document type or editing the setting for an existing document type, click New or Edit to open the Edit Viewer Extensions dialog box.

EDIT VIEWER EXTENSIONS DIALOG BOX

Type the filename extension for the document type—including the separating period (.)—in the Filename extensions box.

![Edit Viewer Extensions dialog box](image)

Then choose one of the Action options:

- **Built-in text editor**—select this option to open all documents of the specified type with the IAR Embedded Workbench text editor.
- **Use file explorer associations**—select this option to open all documents with the default application associated with the specified type in Windows Explorer.
• **Command line**—select this option and type or browse your way to the viewer application, and give any command line options you would like to the tool.

**WINDOW MENU**

Use the commands on the Window menu to manipulate the IDE windows and change their arrangement on the screen.

The last section of the Window menu lists the windows currently open on the screen. Choose the window you want to switch to.

![Figure 201: Window menu](image)

These commands are available on the Window menu:

<table>
<thead>
<tr>
<th>Menu command</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Close Tab</td>
<td>Closes the active tab.</td>
</tr>
<tr>
<td>Close Window</td>
<td>CTRL+F4 Closes the active editor window.</td>
</tr>
<tr>
<td>Split</td>
<td>Splits an editor window horizontally or vertically into two, or four panes, to allow you to see more parts of a file simultaneously.</td>
</tr>
<tr>
<td>New Vertical Editor Window</td>
<td>Opens a new empty window next to current editor window.</td>
</tr>
<tr>
<td>New Horizontal Editor Window</td>
<td>Opens a new empty window under current editor window.</td>
</tr>
<tr>
<td>Move Tabs To Next Window</td>
<td>Moves all tabs in current window to next window.</td>
</tr>
<tr>
<td>Move Tabs To Previous Window</td>
<td>Moves all tabs in current window to previous window.</td>
</tr>
<tr>
<td>Close All Tabs Except Active</td>
<td>Closes all the tabs except the active tab.</td>
</tr>
<tr>
<td>Close All Editor Tabs</td>
<td>Closes all tabs currently available in editor windows.</td>
</tr>
</tbody>
</table>

*Table 98: Window menu commands*
HELP MENU

The Help menu provides help about IAR Embedded Workbench and displays the version numbers of the user interface and of the IDE.

You can also access the Information Center from the Help menu. The Information Center is an integrated navigation system that gives easy access to the information resources you need to get started and during your project development: tutorials, example projects, user guides, support information, and release notes. It also provides shortcuts to useful sections on the IAR Systems web site.
C-SPY® reference

This chapter contains reference information about the windows, menus, menu commands, and the corresponding components that are specific for the IAR C-SPY Debugger. This chapter contains the following sections:

- C-SPY windows, page 403
- C-SPY menus, page 437.

C-SPY windows

The following windows specific to C-SPY are available:

- 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
- Terminal I/O window
- Code Coverage window
- Profiling window
- Profiling window
- Images window
- Stack window
- Symbols window.

Additional windows are available depending on which C-SPY driver you are using. For information about driver-specific windows, see the driver-specific documentation.
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:

<table>
<thead>
<tr>
<th>Key</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Enter</td>
<td>Makes an item editable and saves the new value.</td>
</tr>
<tr>
<td>Esc</td>
<td>Cancels a new value.</td>
</tr>
</tbody>
</table>

Table 99: Editing in C-SPY windows

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
```

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. Typically, this menu contains menu commands for opening driver-specific windows and dialog boxes. See the driver-specific documentation for more information
- A special debug toolbar
- Several windows and dialog boxes specific to C-SPY.

The window might look different depending on which components you are using.

Each window item is explained in greater detail in the following sections.

Menu bar

In addition to the menus available in the development environment, 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. These menus are available when C-SPY is running:

<table>
<thead>
<tr>
<th>Menu</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Debug</td>
<td>The Debug menu provides commands for executing and debugging the source application. Most of the commands are also available as icon buttons in the debug toolbar.</td>
</tr>
<tr>
<td>Disassembly</td>
<td>The Disassembly menu provides commands for controlling the disassembly processor mode.</td>
</tr>
<tr>
<td>Simulator</td>
<td>The Simulator menu provides access to the dialog boxes for setting up interrupt simulation and memory maps. Only available when the C-SPY Simulator is used.</td>
</tr>
</tbody>
</table>

Table 100: C-SPY menu

Additional menus might be available, depending on which debugger drivers have been installed; for information, see the driver-specific documentation.

**Debug toolbar**

The debug toolbar provides buttons for the most frequently-used commands on the Debug menu.

For a description of any button, point to it with the mouse pointer. When a command is not available the corresponding button is dimmed and you will not be able to select it.

This diagram shows the command corresponding to each button:

![C-SPY debug toolbar diagram](image)

Figure 202: C-SPY debug toolbar
DISASSEMBLY WINDOW

The C-SPY Disassembly window—available from the View menu—shows the application being debugged as disassembled application code.

**Figure 203: C-SPY Disassembly window**

**Toolbar**

At toolbar at the top of the window provides these toolbar buttons:

<table>
<thead>
<tr>
<th>Toolbar button</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Go to</td>
<td>The location you want to view. This can be a memory address, or the name of a variable, function, or label.</td>
</tr>
<tr>
<td>Zone display</td>
<td>Lists the available memory zones to display. Read more about Zones in the section Memory addressing, page 149.</td>
</tr>
<tr>
<td>Toggle Mixed-Mode</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>

Table 101: Disassembly window toolbar
The display area

The current position—highlighted in green—indicates the next assembler instruction to be executed. To move the cursor to any line in the Disassembly window, click on the line. Alternatively, move the cursor using the navigation keys. Double-click in the gray left-side margin of the window to set a breakpoint, which is indicated in red. Code that has been executed—code coverage—is indicated with a green diamond.

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.

To change the default color of the source code in the Disassembly window, choose Tools>Options>Debugger. Set the default color using the Set 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.

Disassembly context menu

This is the context menu available in the Disassembly window:

<table>
<thead>
<tr>
<th>Menu command</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Move to PC</td>
<td>Displays code at the current program counter location.</td>
</tr>
<tr>
<td>Run to Cursor</td>
<td></td>
</tr>
<tr>
<td>Code Coverage</td>
<td></td>
</tr>
<tr>
<td>Instruction Profiling</td>
<td></td>
</tr>
<tr>
<td>Toggle Breakpoint (Code)</td>
<td></td>
</tr>
<tr>
<td>Toggle Breakpoint (Loc)</td>
<td></td>
</tr>
<tr>
<td>Toggle Breakpoint (Trace Start)</td>
<td></td>
</tr>
<tr>
<td>Toggle Breakpoint (Trace Stop)</td>
<td></td>
</tr>
<tr>
<td>Toggle Breakpoint (Trace Filter)</td>
<td></td>
</tr>
<tr>
<td>Enable/Disable Breakpoint</td>
<td></td>
</tr>
<tr>
<td>Set Next Statement</td>
<td></td>
</tr>
<tr>
<td>Copy Window Contents</td>
<td></td>
</tr>
<tr>
<td>Mixed-Mode</td>
<td></td>
</tr>
</tbody>
</table>

*Figure 204: Disassembly window context menu*

**Note:** The contents of this menu are dynamic, which means it might contain other commands than in this figure. All available commands are described in Table 102, Disassembly context menu commands.

These commands are available on the menu:

<table>
<thead>
<tr>
<th>Menu command</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Move to PC</td>
<td>Displays code at the current program counter location.</td>
</tr>
</tbody>
</table>

*Table 102: Disassembly context menu commands*
C-SPY windows

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

Code Coverage
Opens 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 and off.
Show, toggles the display of code coverage. Executed code is indicated by a green diamond.
Clear, clears all code coverage information.

Instruction Profiling
Opens 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 and off.
Show, toggles the display of instruction profiling. For each instruction, the left-side margin displays how many times the instruction has been executed.
Clear, clears all instruction profiling information.

Toggle Breakpoint (Code)
Toggles a code breakpoint. Assembler instructions at which code breakpoints have been set are highlighted in red. For information about code breakpoints, see Code breakpoints dialog box, page 341.

Toggle Breakpoint (Log)
Toggles a log breakpoint for trace printouts. Assembler instructions at which log breakpoints have been set are highlighted in red. For information about log breakpoints, see Log breakpoints dialog box, page 343.

Toggle Breakpoint (Trace Start)
Toggles a Trace Start breakpoint. When the breakpoint is triggered, the trace system is started. For information about Trace Start breakpoints, see Trace Start breakpoints dialog box, page 193. Note that this menu command is only available if the C-SPY driver you are using supports the trace system.

Toggle Breakpoint (Trace Stop)
Toggles a Trace Stop breakpoint. When the breakpoint is triggered, the trace system is stopped. For information about Trace Stop breakpoints, see Trace Stop breakpoints dialog box, page 194. Note that this menu command is only available if the C-SPY driver you are using supports the trace system.

Table 102: Disassembly context menu commands (Continued)
### 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 205: 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 be used.

MEMORY WINDOW

The Memory window—available from the View menu—gives an up-to-date display of a specified area of memory 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.

Figure 206: Memory window

Toolbar

The toolbar at the top of the window provides these commands:

<table>
<thead>
<tr>
<th>Operation</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Go to</td>
<td>The location you want to view. This can be a memory address, or the name of a variable, function, or label.</td>
</tr>
</tbody>
</table>

Table 103: Memory window operations
The display area shows the addresses currently being viewed, the memory contents in the format you have chosen, and the memory contents in ASCII format. You can edit the contents of the Memory window, both in the hexadecimal part and the ASCII part of the window.

Data coverage is displayed with these colors:
- Yellow indicates data that has been read
- Blue indicates data that has been written
- Green indicates data that has been both read and written.

Note: Data coverage is not supported by all C-SPY drivers. Data coverage is supported by the C-SPY Simulator.

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

---

<table>
<thead>
<tr>
<th>Operation</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Zone display</td>
<td>Lists the available memory zones to display. Read more about Zones in Memory addressing, page 149.</td>
</tr>
<tr>
<td>Context menu button</td>
<td>Displays the context menu, see Memory window context menu, page 412.</td>
</tr>
<tr>
<td>Update Now</td>
<td>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.</td>
</tr>
<tr>
<td>Live Update</td>
<td>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&gt;Debugger dialog box.</td>
</tr>
</tbody>
</table>

Table 103: Memory window operations (Continued)
C-SPY windows

Memory window context menu

This context menu is available in the Memory window:

![Memory window context menu](image)

These commands are available on the menu:

<table>
<thead>
<tr>
<th>Menu command</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Copy, Paste</td>
<td>Standard editing commands.</td>
</tr>
<tr>
<td>Zone</td>
<td>Lists the available memory zones to display. Read more about Zones in Memory addressing, page 149.</td>
</tr>
<tr>
<td>x1, x2, x4 Units</td>
<td>Switches between displaying the memory contents in units of 8, 16, or 32 bits</td>
</tr>
<tr>
<td>Little Endian</td>
<td>Switches between displaying the contents in big-endian or little-endian order.</td>
</tr>
<tr>
<td>Big Endian</td>
<td>Switches between displaying the contents in big-endian or little-endian order.</td>
</tr>
<tr>
<td>Data Coverage</td>
<td>Enable toggles data coverage on and off.</td>
</tr>
<tr>
<td></td>
<td>Show toggles between showing and hiding data coverage.</td>
</tr>
<tr>
<td></td>
<td>Clear clears all data coverage information.</td>
</tr>
<tr>
<td>Memory Fill</td>
<td>Displays the Fill dialog box, where you can fill a specified area with a value, see Fill dialog box, page 413.</td>
</tr>
<tr>
<td>Memory Save</td>
<td>Displays the Memory Save dialog box, where you can save the contents of a specified memory area to a file, see Memory Save dialog box, page 414.</td>
</tr>
</tbody>
</table>

Table 104: Commands on the memory window context menu
### Table 104: Commands on the memory window context menu (Continued)

<table>
<thead>
<tr>
<th>Menu command</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Memory Restore</td>
<td>Displays the Memory Restore 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 415.</td>
</tr>
<tr>
<td>Set Data Breakpoint</td>
<td>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.</td>
</tr>
<tr>
<td>Set Trace Start Breakpoint</td>
<td>Sets a Trace Start breakpoint 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 Trace Start breakpoints dialog box, page 193.</td>
</tr>
<tr>
<td>Set Trace Stop Breakpoint</td>
<td>Sets a Trace Stop breakpoint 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 Trace Stop breakpoints dialog box, page 194.</td>
</tr>
<tr>
<td>Set Trace Filter Breakpoint</td>
<td>Sets a Trace Filter breakpoint 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 Trace Filter breakpoints dialog box, page 292.</td>
</tr>
</tbody>
</table>

### FILL DIALOG BOX

In the Fill dialog box—available from the context menu in the Memory window—you can fill a specified area of memory with a value.

![Fill dialog box](image)

Figure 208: Fill dialog box
Options

<table>
<thead>
<tr>
<th>Option</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Start Address</td>
<td>Type the start address—in binary, octal, decimal, or hexadecimal notation.</td>
</tr>
<tr>
<td>Length</td>
<td>Type the length—in binary, octal, decimal, or hexadecimal notation.</td>
</tr>
<tr>
<td>Zone</td>
<td>Select memory zone.</td>
</tr>
<tr>
<td>Value</td>
<td>Type the 8-bit value to be used for filling each memory location.</td>
</tr>
</tbody>
</table>

Table 105: Fill dialog box options

These are the available memory fill operations:

<table>
<thead>
<tr>
<th>Operation</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Copy</td>
<td>The Value will be copied to the specified memory area.</td>
</tr>
<tr>
<td>AND</td>
<td>An AND operation will be performed between the Value and the existing contents of memory before writing the result to memory.</td>
</tr>
<tr>
<td>XOR</td>
<td>An XOR operation will be performed between the Value and the existing contents of memory before writing the result to memory.</td>
</tr>
<tr>
<td>OR</td>
<td>An OR operation will be performed between the Value and the existing contents of memory before writing the result to memory.</td>
</tr>
</tbody>
</table>

Table 106: Memory fill operations

MEMORY SAVE DIALOG BOX

Use the Memory Save dialog box—available by choosing Debug>Memory>Save or from the context menu in the Memory window—to save the contents of a specified memory area to a file.

![Memory Save dialog box](image-url)
Zone
The available memory zones.

Start address
The start address of the memory range to be saved.

Stop address
The stop address of the memory range to be saved.

File format
The file format to be used, which is Intel-extended by default.

Filename
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
Use the Memory Restore dialog box—available by choosing Debug>Memory>Save or from the context menu in the Memory window—to load the contents of a file in Intel-extended or Motorola S-record format to a specified memory zone.

Figure 210: Memory Restore dialog box

Zone
The available memory zones.

Filename
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.

**SYMBOLIC MEMORY WINDOW**

The Symbolic Memory window—available from the View menu when the debugger is running—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 spotting alignment holes or for understanding problems caused by buffers being overwritten.

![Symbolic Memory window](image)

**Figure 211: Symbolic Memory window**

**Toolbar**

The toolbar at the top of the window provides these toolbar buttons:

<table>
<thead>
<tr>
<th>Operation</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Go to</td>
<td>The memory location or symbol you want to view.</td>
</tr>
<tr>
<td>Zone display</td>
<td>Lists the available memory zones to display. To read more about zones, see Memory addressing, page 149.</td>
</tr>
<tr>
<td>Previous</td>
<td>Jumps to the previous symbol.</td>
</tr>
<tr>
<td>Next</td>
<td>Jumps to the next symbol.</td>
</tr>
</tbody>
</table>

*Table 107: Symbolic Memory window toolbar*
The display area

The display area displays the memory space, where information is provided in these columns:

<table>
<thead>
<tr>
<th>Column</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Location</td>
<td>The memory address.</td>
</tr>
<tr>
<td>Data</td>
<td>The memory contents in hexadecimal format. The data is grouped according to the size of the symbol. This column is editable.</td>
</tr>
<tr>
<td>Variable</td>
<td>The variable name; requires that the variable has a fixed memory location. Local variables are not displayed.</td>
</tr>
<tr>
<td>Value</td>
<td>The value of the variable. This column is editable.</td>
</tr>
<tr>
<td>Type</td>
<td>The type of the variable.</td>
</tr>
</tbody>
</table>

Table 108: Symbolic Memory window columns

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.

Symbolic Memory window context menu

This context menu is available in the Symbolic Memory window:

<table>
<thead>
<tr>
<th>Menu command</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Next Symbol</td>
<td>Jumps to the next symbol.</td>
</tr>
<tr>
<td>Previous Symbol</td>
<td>Jumps to the previous symbol.</td>
</tr>
</tbody>
</table>

Table 109: Commands on the Symbolic Memory window context menu
The Register window—available from the View menu—gives an up-to-date display of the contents of the processor registers, and allows you to edit them. When a value changes it becomes highlighted. Some registers are expandable, which means that the register contains interesting bits or subgroups of bits.

You can open several instances of this window, which is very convenient if you want to keep track of different register groups.

Use the drop-down list to select which register group to display in the Register window. To define application-specific register groups, see Defining application-specific groups, page 153.

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.

The Watch window—available from the View menu—allows you to monitor the values of C-SPY expressions or variables. You can view, add, modify, and remove expressions...
in the Watch window. Tree structures of arrays, structs, and unions are expandable, which means that you can study each item of these.

![Watch window](image)

Every time execution in C-SPY stops, a value that has changed since the last stop is highlighted. In fact, every time memory changes, the values in the Watch window are recomputed, including updating the red highlights.

**Watch window context menu**

This context menu is available in the Watch window:

![Watch window context menu](image)

The menu contains these commands:

<table>
<thead>
<tr>
<th>Menu command</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Add, Remove</td>
<td>Adds or removes the selected expression.</td>
</tr>
</tbody>
</table>

*Table 110: Watch window context menu commands*
The display format setting affects different types of expressions in different ways:

<table>
<thead>
<tr>
<th>Type of expressions</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, 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 111: Effects of display format setting on different types of expressions

**LOCALS WINDOW**

The Locals window—available from the View menu—automatically displays the local variables and function parameters.

Figure 216: Locals window
Locals window context menu

The context menu available in the Locals window provides commands for changing the display format of expressions; for information about these commands, see Watch window context menu, page 419.

AUTO WINDOW

The Auto window—available from the View menu—automatically displays a useful selection of variables and expressions in, or near, the current statement.

<table>
<thead>
<tr>
<th>Expression</th>
<th>Value</th>
<th>Location</th>
<th>Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>i</td>
<td>5</td>
<td>R4</td>
<td>short</td>
</tr>
<tr>
<td>Rprof()</td>
<td>0</td>
<td>0x6f03214</td>
<td>unsigned int</td>
</tr>
<tr>
<td>Fib</td>
<td>&lt;empty&gt;</td>
<td>0x6f0229</td>
<td>unsigned int[16]</td>
</tr>
<tr>
<td>GetFib</td>
<td>GetFib(i) (i=2)</td>
<td></td>
<td>unsigned int...</td>
</tr>
</tbody>
</table>

Figure 217: Auto window

Auto window context menu

The context menu available in the Auto window provides commands for changing the display format of expressions; for information about these commands, see Watch window context menu, page 419.

LIVE WATCH WINDOW

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

<table>
<thead>
<tr>
<th>Expression</th>
<th>Value</th>
<th>Location</th>
<th>Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>getsfib</td>
<td>getsfib (0x1139)</td>
<td></td>
<td>unsigned int(*)</td>
</tr>
<tr>
<td></td>
<td>getsfib (0x1139)</td>
<td>Memory/0x1139</td>
<td>unsigned int[16]</td>
</tr>
</tbody>
</table>

Figure 218: Live Watch window

Typically, this window is useful for hardware target systems supporting this feature.

Live Watch window context menu

The context menu available in the Live Watch window provides commands for adding and removing expressions, changing the display format of expressions, and commands...
for changing the default type interpretation of variables. For information about these commands, see Watch window context menu, page 419.

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.

### QUICK WATCH WINDOW

In the Quick Watch window—available from the **View** menu—you can watch the value of a variable or expression and evaluate expressions.

![Quick Watch window](image)

Type the expression you want to examine in the **Expressions** text box. Click the **Recalculate** button to calculate the value of the expression. For examples about how to use the Quick Watch window, see Using the Quick Watch window, page 139 and Executing macros using Quick Watch, page 160.

**Quick Watch window context menu**

The context menu available in the Quick Watch window provides commands for changing the display format of expressions, and commands for changing the default type interpretation of variables. For information about these commands, see Watch window context menu, page 419.

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

### STATICS WINDOW

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

<table>
<thead>
<tr>
<th>Expression</th>
<th>Value</th>
<th>Location</th>
<th>Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>cell_count + 10</td>
<td>DATA:0x000080</td>
<td>0x00000000</td>
<td>int</td>
</tr>
<tr>
<td>cell_count</td>
<td>DATA:0x000080</td>
<td>0x00000000</td>
<td>int</td>
</tr>
<tr>
<td>cell_count</td>
<td>DATA:0x000080</td>
<td>0x00000000</td>
<td>int</td>
</tr>
<tr>
<td>cell_count</td>
<td>DATA:0x000080</td>
<td>0x00000000</td>
<td>int</td>
</tr>
<tr>
<td>cell_count</td>
<td>DATA:0x000080</td>
<td>0x00000000</td>
<td>int</td>
</tr>
<tr>
<td>cell_count</td>
<td>DATA:0x000080</td>
<td>0x00000000</td>
<td>int</td>
</tr>
<tr>
<td>cell_count</td>
<td>DATA:0x000080</td>
<td>0x00000000</td>
<td>int</td>
</tr>
<tr>
<td>cell_count</td>
<td>DATA:0x000080</td>
<td>0x00000000</td>
<td>int</td>
</tr>
</tbody>
</table>

Figure 220: Statics window

The display area

The display area shows the values of variables with static storage duration, where information is provided in these columns:

<table>
<thead>
<tr>
<th>Column</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Expression</td>
<td>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.</td>
</tr>
<tr>
<td>Value</td>
<td>The value of the variable. Values that have changed are highlighted in red. This column is editable.</td>
</tr>
<tr>
<td>Location</td>
<td>The location in memory where this variable is stored.</td>
</tr>
<tr>
<td>Type</td>
<td>The data type of the variable.</td>
</tr>
</tbody>
</table>

Table 112: Statics window columns

Statics window context menu

This context menu is available in the Statics window:

Figure 221: Statics window context menu
The menu contains these commands:

<table>
<thead>
<tr>
<th>Menu command</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Default Format,</td>
<td>Changes the display format of expressions. The display format setting affects different types of expressions in different ways, see Table 111, Effects of display format setting on different types of expressions. Your selection of display format is saved between debug sessions.</td>
</tr>
<tr>
<td>Binary Format,</td>
<td></td>
</tr>
<tr>
<td>Octal Format,</td>
<td></td>
</tr>
<tr>
<td>Decimal Format,</td>
<td></td>
</tr>
<tr>
<td>Hexadecimal Format,</td>
<td></td>
</tr>
<tr>
<td>Char Format</td>
<td></td>
</tr>
<tr>
<td>Select Statics</td>
<td>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 424.</td>
</tr>
</tbody>
</table>

Table 113: Statics window context menu commands

**SELECT STATICS DIALOG BOX**

Use the Select Statics dialog box—available from the context menu in the Statics window—to select which variables should be displayed in the Statics window.

**Show all variables with static storage duration**

Use this option to make all variables be displayed in the Statics window, including new variables that are added to your application between debug sessions.

**Show selected variables only**

Use this option to select which variables you want to be displayed in the Statics window. Note that in this case if you add a new variable to your application between two debug sessions, it will not be displayed until you select it again.
sessions, this variable will not automatically be displayed in the Statics window. If the checkbox next to a variable is selected, the variable will be displayed.

**CALL STACK WINDOW**

The Call stack window—available from the View menu—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.

Each entry 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.

If the Step Into command steps into 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.

**Call Stack window context menu**

The context menu available when you right-click in the Call Stack window provides these commands:

- Go to Source
- Show Arguments
- Run to Cursor
- Toggle breakpoint (Code)
- Toggle breakpoint (Loc)
- Enable/disable breakpoint

Figure 223: Call Stack window

Figure 224: Call Stack window context menu
Commands

Go to Source Displays the selected functions in the Disassembly or editor windows.

Show Arguments Shows function arguments.

Run to Cursor Executes 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

In the Terminal I/O window—available from the View menu—you can enter input to the application, and display output from it. To use this window, you must build the application with the Semihosted or the IAR breakpoint option. 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.

Figure 225: Terminal I/O window

Clicking the Ctrl codes button opens a menu with submenus for input of special characters, such as EOF (end of file) and NUL.

Figure 226: Ctrl codes menu
Clicking the Input Mode button 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)

For reference information about the options available in the dialog box, see Terminal I/O options, page 393.

**CODE COVERAGE WINDOW**

The Code Coverage window—available from the View menu—reports the status of the current code coverage analysis, that is, what parts of the code that have been executed at least once since the start of the analysis. The compiler generates detailed stepping information in the form of step points at each statement, and at each function call. The report includes information about all modules and functions. It reports the amount of all step points, in percentage, that have been executed and lists all step points that have not been executed up to the point where the application has been stopped. The coverage will continue until turned off.

![Code Coverage window](image)
Note: You can enable the Code Coverage plugin module on the Debugger>Plugins page available in the Options dialog box.

Code coverage is not supported by all C-SPY drivers. For information about whether the C-SPY driver you are using supports code coverage, see the driver-specific documentation in Part 6: C-SPY hardware debugger systems. Code coverage is supported by the C-SPY Simulator.

Toolbar

The toolbar at the top of the window provides these buttons:

<table>
<thead>
<tr>
<th>Toolbar button</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Activate</td>
<td>Switches code coverage on and off during execution.</td>
</tr>
<tr>
<td>Clear</td>
<td>Clears the code coverage information. All step points are marked as not executed.</td>
</tr>
<tr>
<td>Refresh</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>Auto-refresh</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>Save As</td>
<td>Saves the current code coverage result in a text file.</td>
</tr>
<tr>
<td>Save session</td>
<td>Saves your code coverage session data to a *.dat file.</td>
</tr>
<tr>
<td>Restore session</td>
<td>Restores previously saved code coverage session data.</td>
</tr>
</tbody>
</table>

Table 114: Code Coverage window toolbar

The display area

These icons are used to give you an overview of the current status on all levels:

- A red diamond signifies that 0% of the code has been executed
- A green diamond signifies that 100% of the code has been executed
- A red and green diamond signifies that some of the code has been executed
- A yellow diamond signifies a step point that has not been executed.

For step point lines, the information displayed is the column number range and the row number of the step point in the source window, followed by the address of the step point.

<column start>-<column end>:<row>.
**Code Coverage window context menu**

This context menu is available in the Code Coverage window:

![Code coverage window context menu](image)

These commands are available on the menu:

<table>
<thead>
<tr>
<th>Menu command</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Activate</td>
<td>Switches code coverage on and off during execution.</td>
</tr>
<tr>
<td>Clear</td>
<td>Clears the code coverage information. All step points are marked as not executed.</td>
</tr>
<tr>
<td>Refresh</td>
<td>Updates the code coverage information and refreshes the window. All step points that has been executed since the last refresh are removed from the tree.</td>
</tr>
<tr>
<td>Auto-refresh</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>Save As</td>
<td>Saves the current code coverage result in a text file.</td>
</tr>
</tbody>
</table>

Table 115: Code Coverage window context menu commands

**PROFILING WINDOW**

The Profiling window—available from the **View** menu—displays profiling information, that is, timing information for the functions in an application. Profiling must be turned on explicitly using a button in the window’s toolbar, and will stay active until it is turned off.
The profiler measures time at the entry and return of a function. This means that time consumed in a function is not added until the function returns or another function is called. You will only notice this if you are stepping into a function.

<table>
<thead>
<tr>
<th>Function</th>
<th>Calls</th>
<th>Net Time (cycles)</th>
<th>Net Time (%)</th>
<th>Accomulated Time</th>
<th>Accomulated Time</th>
</tr>
</thead>
<tbody>
<tr>
<td>Outside main</td>
<td>0</td>
<td>247</td>
<td>4.20</td>
<td>207</td>
<td>4.20</td>
</tr>
<tr>
<td>_patcher</td>
<td>24</td>
<td>72</td>
<td>1.45</td>
<td>72</td>
<td>1.45</td>
</tr>
<tr>
<td>_main</td>
<td>0</td>
<td>88</td>
<td>0.80</td>
<td>88</td>
<td>0.80</td>
</tr>
<tr>
<td>do_bmpRound_p</td>
<td>10</td>
<td>240</td>
<td>0.75</td>
<td>390</td>
<td>0.23</td>
</tr>
<tr>
<td>exit</td>
<td>3</td>
<td>0.46</td>
<td>3</td>
<td>0.06</td>
<td></td>
</tr>
<tr>
<td>get_vlog</td>
<td>26</td>
<td>126</td>
<td>8.66</td>
<td>390</td>
<td>8.06</td>
</tr>
<tr>
<td>int8</td>
<td>1</td>
<td>248</td>
<td>6.12</td>
<td>400</td>
<td>10.09</td>
</tr>
<tr>
<td>main</td>
<td>1</td>
<td>154</td>
<td>3.29</td>
<td>4027</td>
<td>15.60</td>
</tr>
<tr>
<td>next_counter</td>
<td>10</td>
<td>72</td>
<td>1.45</td>
<td>70</td>
<td>1.45</td>
</tr>
<tr>
<td>print</td>
<td>10</td>
<td>3316</td>
<td>68.50</td>
<td>3460</td>
<td>71.90</td>
</tr>
<tr>
<td>patcher</td>
<td>24</td>
<td>72</td>
<td>1.45</td>
<td>144</td>
<td>2.38</td>
</tr>
</tbody>
</table>

Figure 230: Profiling window

**Note:** You can enable the Profiling plugin module on the Debugger>Plugins page available in the Options dialog box.

When profiling on hardware, there will be no cycle counter statistics available.

Profiling is not supported by all C-SPY drivers. For information about whether the C-SPY driver you are using supports profiling, see the driver-specific documentation in Part 6. C-SPY hardware debugger systems. Profiling is supported by the C-SPY Simulator.

**Profiling commands**

In addition to the toolbar buttons, the context menu available in the Profiling window gives you access to these and some extra commands:

![Profiling window context menu](image)

You can find these commands on the menu:

- **Activate**
  - Toggles profiling on and off during execution.

- **New measurement**
  - Starts a new measurement. To reset the displayed values to zero, click the button.
The Profiling window contains these columns:

<table>
<thead>
<tr>
<th>Column</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Function</td>
<td>The name of each function.</td>
</tr>
<tr>
<td>Calls</td>
<td>The number of times each function has been called.</td>
</tr>
<tr>
<td>Flat Time</td>
<td>The total time spent in each function in cycles or as a percentage of the</td>
</tr>
<tr>
<td></td>
<td>total number of cycles, excluding all function calls made from that</td>
</tr>
<tr>
<td></td>
<td>function.</td>
</tr>
<tr>
<td>Accumulated Time</td>
<td>Time spent in each function in cycles or as a percentage of the total</td>
</tr>
<tr>
<td></td>
<td>number of cycles, including all function calls made from that function.</td>
</tr>
</tbody>
</table>

There is always an item in the list called Outside main. This is time that cannot be placed in any of the functions in the list. That is, code compiled without debug information, for instance, all startup and exit code, and C/C++ library code.
 IMAGES WINDOW

The Images window—available from the View menu—lists all currently loaded images (debug files).

<table>
<thead>
<tr>
<th>Name</th>
<th>Path</th>
</tr>
</thead>
<tbody>
<tr>
<td>All images</td>
<td>[Combines debug information from all images]</td>
</tr>
<tr>
<td>project</td>
<td>C:\Documents and Settings\My Documents\IAR Embedded Workbench\DebugEval\project1.out</td>
</tr>
<tr>
<td>externImage</td>
<td>C:\Documents and Settings\My Documents\IAR Embedded Workbench\DebugEval\externImage.out</td>
</tr>
</tbody>
</table>

Figure 232: Images window

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

The display area

The display area lists the loaded images:

<table>
<thead>
<tr>
<th>Column</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Name</td>
<td>The name of the loaded image.</td>
</tr>
<tr>
<td>Path</td>
<td>The path to the loaded image.</td>
</tr>
</tbody>
</table>

Table 117: Images window columns

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.

Images window context menu

This context menu is available in the Images window:

- Show all images
- Show only 'project'

Figure 233: Images window context menu
These commands are available on the menu:

<table>
<thead>
<tr>
<th>Menu command</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Show all images</td>
<td>Shows debug information for all loaded debug images.</td>
</tr>
<tr>
<td>Show only image</td>
<td>Shows debug information for the selected debug image.</td>
</tr>
</tbody>
</table>

Table 118: Images window context menu commands

**Related information**

For related information, see:

- Loading multiple images, page 124
- __loadImage__, page 549
- Images, page 495.

**STACK WINDOW**

The Stack window is a memory window that displays the contents of the stack. In addition, some integrity checks of the stack can be performed to detect and warn about problems with stack overflow. For example, the Stack window is useful for determining the optimal size of the stack.

Before you can open the Stack window you must make sure it is enabled: choose **Project>Options>Debugger>Plugins** and select **Stack** from the list of plugins. In C-SPY, you can then open a Stack window by choosing **View>Stack**. You can open several Stack windows, each showing a different stack—if several stacks are available—or the same stack with different display settings.

![Figure 234: Stack window](image)

**The stack drop-down menu**

If the core you are using has multiple stacks, you can use the stack drop-down menu at the top of the window to select which stack to view.
The graphical stack bar

At the top of the window, a stack bar displays the state of the stack graphically. To view the stack bar you must make sure it is enabled: choose Tools>Options>Stack and select the option Enable graphical stack display and stack usage tracking.

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. A green line represents the current value of the stack pointer. The part of the stack memory that has been used during execution is displayed in a dark gray color, and the unused part in a light gray color. The graphical stack bar turns red when the stack usage exceeds a threshold that you can specify.

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 range, without actually modifying any of the bytes near the stack range. Likewise, your application might modify memory within the stack range by mistake. Furthermore, the Stack window cannot detect a stack overflow when it happens, but can only detect the signs it leaves behind.

Note: The size and location of the stack is retrieved from the definition of the section holding the stack, typically CSTACK, 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 section 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++ Development Guide for ARM®.

When the stack bar is enabled, the functionality needed to detect and warn about stack overflows is also enabled, see Stack options, page 390.

The Stack window columns

The main part of the window displays the contents of stack memory in these columns:

<table>
<thead>
<tr>
<th>Column</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Location</td>
<td>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.</td>
</tr>
</tbody>
</table>

Table 119: Stack window columns
The Stack window context menu

This context menu is available if you right-click in the Stack window:

<table>
<thead>
<tr>
<th>Column</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Data</td>
<td>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.</td>
</tr>
<tr>
<td>Variable</td>
<td>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.</td>
</tr>
<tr>
<td>Value</td>
<td>Displays the value of the variable that is displayed in the Variable column.</td>
</tr>
<tr>
<td>Frame</td>
<td>Displays the name of the function the call frame corresponds to.</td>
</tr>
</tbody>
</table>

Table 119: Stack window columns (Continued)

The commands are available on the context menu:

- **Show variables**: Separate columns named Variables, Value, and Frame are displayed in the Stack window. Variables located at memory addresses listed in the Stack window are displayed in these columns.
- **Show offsets**: When this option is selected, locations in the Location column are displayed as offsets from the stack pointer. When deselected, locations are displayed as absolute addresses.
- **1x Units**: The data in the Data column is displayed as single bytes.
- **2x Units**: The data in the Data column is displayed as 2-byte groups.
- **4x Units**: The data in the Data column is displayed as 4-byte groups.
- **Options**: Opens the IDE Options dialog box where you can set options specific to the Stack window, see Stack options, page 390.

Figure 235: Stack window context menu
Overriding the default stack setup

The Stack window retrieves information about the stack size and placement from the definition of the sections holding the stacks made in the linker configuration file. The sections are described in the IAR C/C++ Development Guide for ARM®.

For applications that set up the stacks using other mechanisms, it is possible to override the default mechanism. Use any of the C-SPY command-line options, see --proc_stack_stack, page 523.

Syntax

```
--proc_stack_stack stackstart, stackend
```

The parameters stackstart and stackend follow the standard C-SPY expression syntax. Whitespace characters are not allowed in the expression.

SYMBOLS WINDOW

The Symbols window—available from the View menu—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.

![Symbols window](image)

**The display area**

The display area lists the symbols, where information is provided in these columns:

<table>
<thead>
<tr>
<th>Column</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Symbol</td>
<td>The symbol name.</td>
</tr>
<tr>
<td>Location</td>
<td>The memory address.</td>
</tr>
<tr>
<td>Full Name</td>
<td>The symbol name; often the same as the contents of the Symbol column but differs for example for C++ member functions.</td>
</tr>
</tbody>
</table>

Table 120: Symbols window columns
Click on the column headers to sort the list by name, location, or full name.

**Symbols window context menu**

This context menu is available in the Symbols window:

![Symbols window context menu](image)

These commands are available on the menu:

<table>
<thead>
<tr>
<th>Menu command</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Function</td>
<td>Toggles the display of function symbols in the list.</td>
</tr>
<tr>
<td>Variables</td>
<td>Toggles the display of variables in the list.</td>
</tr>
<tr>
<td>Labels</td>
<td>Toggles the display of labels in the list.</td>
</tr>
</tbody>
</table>

Table 121: Commands on the Symbols window context menu

**C-SPY menus**

In addition to the menus available in the development environment, the Debug and Disassembly menus are available when C-SPY is running.

Additional menus are available depending on which C-SPY driver you are using. For information about driver-specific menus, see the driver-specific documentation in *Part 6. C-SPY hardware debugger systems*. 
## DEBUG MENU

The **Debug** menu provides commands for executing and debugging your application. Most of the commands are also available as toolbar buttons.

### Menu Command Table

<table>
<thead>
<tr>
<th>Menu Command</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Go F5</td>
<td>Executes from the current statement or instruction until a breakpoint or program exit is reached.</td>
</tr>
<tr>
<td>Break</td>
<td>Stops the application execution.</td>
</tr>
<tr>
<td>Reset</td>
<td>Resets the target processor.</td>
</tr>
</tbody>
</table>
| Stop Debugging   | **Ctrl+Shift+D**
|                  | Stops the debugging session and returns you to the project manager.        |
| Step Over F10    | Executes the next statement, function call, or instruction, without entering C or C++ functions or assembler subroutines. |
| Step Into F11    | Executes the next statement or instruction, 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. |

*Figure 238: Debug menu*
In the Autostep settings dialog box, you can customize autostepping by setting the delay between steps or selecting different autostep settings.

**Autostep settings dialog box**

In the Autostep settings dialog box—available from the Debug menu—you can customize autostepping.
The drop-down menu lists the available step commands. The Delay text box lets you specify the delay between each step.

**Macro Configuration dialog box**

In the Macro Configuration dialog box—available by choosing Debug > Macros—you can list, register, and edit your macro files and functions.

Macro functions that have been registered using the dialog box will be deactivated when you exit the debug session, and will not automatically be registered at the next debug session.

![Macro Configuration dialog box](image)

**Registering macro files**

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.
Once you have selected the macro files you want to use click **Register** to register them, replacing any previously defined macro functions or variables. Registered macro functions are displayed in the scroll window under **Registered Macros**. Note that system macros cannot be removed from the list, they are always registered.

**Listing macro functions**

Selecting **All** displays all macro functions, selecting **User** displays all user-defined macros, and selecting **System** displays all system macros.

Clicking on either **Name** or **File** under **Registered Macros** displays the column contents sorted by macro names or by file. Clicking a second time sorts the contents in the reverse order.

**Modifying macro files**

Double-clicking a user-defined macro function in the **Name** column automatically opens the file in which the function is defined, allowing you to modify it, if needed.

**Log File dialog box**

The Log File dialog box—available by choosing **Debug>Logging>Set Log File**—allows you to log output from C-SPY to a file.

![Figure 241: Log File dialog box](image)

Enable or disable logging to the file with the **Enable Log file** check box.

The information printed in the file is, by default, the same as the information listed in the Log window. To change the information logged, use the **Include** options:

<table>
<thead>
<tr>
<th>Option</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Errors</td>
<td>C-SPY has failed to perform an operation.</td>
</tr>
<tr>
<td>Warnings</td>
<td>A suspected error.</td>
</tr>
</tbody>
</table>

*Table 123: Log file options*
**Terminal I/O Log File dialog box**

The Terminal I/O Log Files dialog box—available by choosing **Debug>Logging**—allows you to select a destination log file, and to log terminal I/O input and output from C-SPY to this file.

- **Enable Terminal I/O log file**: Select this option to enable terminal I/O log file.
- **Terminal I/O Log File**: Enter the name of the log file.

Click the browse button to open a standard **Save As** dialog box. Click **Save** to select the specified file—the default filename extension is `.log`.

**DISASSEMBLY MENU**

The commands on the **Disassembly** menu allow you to select which disassembly mode to use.

- **Disassemble in Thumb mode**
- **Disassemble in ARM mode**
- **Disassemble in Current processor mode**
- **Disassemble in Auto mode**

**Note**: After changing disassembly mode, you must scroll the window contents up and down a couple of times to refresh the view.
The Disassembly menu contains the following menu commands:

<table>
<thead>
<tr>
<th>Menu command</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Disassemble in Thumb mode</td>
<td>Select this option to disassemble your application in Thumb mode.</td>
</tr>
<tr>
<td>Disassemble in ARM Mode</td>
<td>Select this option to disassemble your application in ARM mode.</td>
</tr>
<tr>
<td>Disassemble in Current processor mode</td>
<td>Select this option to disassemble your application in the current processor mode.</td>
</tr>
<tr>
<td>Disassemble in Auto mode</td>
<td>Select this option to disassemble your application in automatic mode. This is the default option.</td>
</tr>
</tbody>
</table>

Table 124: Description of Disassembly menu commands

See also *Disassembly window*, page 406.
C-SPY menus
General options

This chapter describes the general options in the IAR Embedded Workbench® IDE.

For information about how to set options, see Setting options, page 97.

Target

The Target options specify the processor variant, FPU, and byte order for the IAR C/C++ Compiler and Assembler.

Figure 244: Target options

PROCESSOR VARIANT

Choose between these two options to specify the processor variant:

Core

The processor core you are using. For a description of the available variants, see the IAR C/C++ Development Guide for ARM®.

Device

The device you are using. The choice of device will automatically determine the default C-SPY® device description file. For information about how to override the default files, see Device description file, page 494.
**ENDIAN MODE**

Choose between the following two options to select the byte order for your project:

**Little**
- The lowest byte is stored at the lowest address in memory. The highest byte is the most significant; it is stored at the highest address.

**Big**
- The lowest address holds the most significant byte, while the highest address holds the least significant byte.
- There are two variants of the big-endian mode. Choose:
  - BE8 to make data big-endian and code little-endian
  - BE32 to make both data and code big-endian.

**FPU**

Choose between the following options to generate code that carries out floating-point operations using a Vector Floating Point (VFP) coprocessor:

- **None** (default) - The software floating-point library is used.
- **VFPv1** - A VFP unit conforming to architecture VFPv1.
- **VFPv2** - A VFP unit conforming to architecture VFPv2.
- **VFPv3** - A VFP unit conforming to architecture VFPv3.
- **VFPv4** - A VFP unit conforming to architecture VFPv4.
- **VFP9-S** - A VFPv2 architecture that can be used with the ARM9E family of CPU cores. Selecting this coprocessor is therefore identical to selecting the VFPv2 architecture.

By selecting a VFP coprocessor, you will override the use of the software floating-point library for all supported floating-point operations.
Output

With the **Output** options you can specify the type of output file—**Executable** or **Library**. You can also specify the destination directories for executable files, object files, and list files.

**OUTPUT FILE**

Use these options to choose the type of output file. Choose between:

**Executable** *(default)*

As a result of the build process, the linker will create an application (an executable output file). When this option is selected, linker options will be available in the **Options** dialog box. Before you create the output you should set the appropriate linker options.

**Library**

As a result of the build process, the library builder will create a **library file**. When this option is selected, library builder options will be available in the **Options** dialog box, and **Linker** will disappear from the list of categories. Before you create the library you can set the options.

**OUTPUT DIRECTORIES**

Use these options to specify paths to destination directories. Note that incomplete paths are relative to your project directory. You can specify the paths to these destination directories:

**Executables/libraries**

Use this option to override the default directory for executable or library files. Type the name of the directory where you want to save executable files for the project.
Library Configuration

With the **Library Configuration** options you can specify which library to use.

![Library Configuration options](image)

**Object files**

Use this option to override the default directory for object files. Type the name of the directory where you want to save object files for the project.

**List files**

Use this option to override the default directory for list files. Type the name of the directory where you want to save list files for the project.

**LIBRARY**

In the **Library** drop-down list you choose which runtime library to use. For information about available libraries, see the IAR C/C++ Development Guide for ARM®.

The names of the library object file and library configuration file that actually will be used are displayed in the **Library file** and **Configuration file** text boxes, respectively.

**CONFIGURATION FILE**

The **Configuration file** text box displays the library configuration file that will be used. A library configuration file is chosen automatically depending on the project settings. If you have chosen **Custom** in the **Library** drop-down list, you must specify your own library configuration file.
LIBRARY LOW-LEVEL INTERFACE IMPLEMENTATION

Use these options to choose what type of low-level interface for I/O to be included in the library.

For Cortex-M, choose between:

**None**
No low-level support for I/O available in the libraries. You must provide your own `__write` function to use the I/O functions part of the library.

**Semihosted and with stdout/stderr via semihosting**
Semihosted I/O which uses the BKPT instruction.

**Semihosted and with stdout/stderr via SWO**
Semihosted I/O which uses the BKPT instruction for all functions except for the `stdout` and `stderr` output where the SWO interface—which is available on some J-Link debug probes—is used. This means a much faster mechanism where the application does not need to halt execution to transfer data.

**IAR breakpoint**
Not available.

For other cores, choose between:

**None**
No low-level support for I/O available in the libraries. You must provide your own `__write` function to use the I/O functions part of the library.

**Semihosted**
Semihosted I/O which uses the SVC instruction (earlier SWI).

**IAR breakpoint**
The IAR proprietary variant of semihosting, which does not use the SVC instruction and thus does not need to set a breakpoint on the SVC vector. This is an advantage for applications which require the SVC vector for their own use, for example an RTOS. This method can also lead to performance improvements. However, note that this method does not work with applications, libraries, and object files that are built using tools from other vendors.
Library Options

With the options on the Library Options page you can choose printf and scanf formatters.

See the IAR C/C++ Development Guide for ARM® for more information about the formatting capabilities.

PRINTF FORMATTER

The full formatter version is memory-consuming, and provides facilities that are not required in many embedded applications. To reduce the memory consumption, alternative versions are also provided.

Printf formatters in the library are: Full, Large, Small, and Tiny.

SCANF FORMATTER

The full formatter version is memory-consuming, and provides facilities that are not required in many embedded applications. To reduce the memory consumption, alternative versions are also provided.

Scanf formatters in the library are: Full, Large, and Small.
Use the options on the MISRA-C:1998 and MISRA-C:2004 pages to control how the IDE checks the source code for deviations from the MISRA C rules. The settings are used for both the compiler and the linker.

For details about specific option, see the IAR Embedded Workbench® MISRA C:2004 Reference Guide or the IAR Embedded Workbench® MISRA C:1998 Reference Guide available from the Help menu.
Compiler options

This chapter describes the compiler options available in the IAR Embedded Workbench® IDE.

For information about how to set options, see Setting options, page 99.

Multi-file compilation

Before you set specific compiler options, you can decide if you want to use multi-file compilation, which is an optimization technique. If the compiler is allowed to compile multiple source files in one invocation, it can in many cases optimize more efficiently.

You can use this option for the entire project or for individual groups of files. All C/C++ source files in such a group are compiled together using one invocation of the compiler.

In the Options dialog box, select Multi-file Compilation to enable multi-file compilation for the group of project files that you have selected in the workspace window. Use Discard Unused Publics to discard any unused public functions and variables from the compilation unit.

If you use this option, all files included in the selected group are compiled using the compiler options which have been set on the group or nearest higher enclosing node which has any options set. Any overriding compiler options on one or more files are ignored when building, because a group compilation must use exactly one set of options.

For information about how multi-file compilation is displayed in the workspace window, see Workspace window, page 280.

For more information about multi-file compilation and discarding unused public functions, see the IAR C/C++ Development Guide for ARM®.

Language

The Language options enable the use of target-dependent extensions to the C or C++ language.
With the **Language** options you can specify the language support you need.

For information about Embedded C++ and Extended Embedded C++, see the *IAR C/C++ Development Guide for ARM*.

**C**

By default, the IAR C/C++ Compiler runs in ISO/ANSI C mode, in which features specific to Embedded C++ and Extended Embedded C++ cannot be used.

**Embedded C++**

In Embedded C++ mode, the compiler treats the source code as Embedded C++. This means that features specific to Embedded C++, such as classes and overloading, can be used.

**Extended Embedded C++**

In Extended Embedded C++ mode, you can take advantage of features like namespaces or the standard template library in your source code.

**Automatic**

If you select **Automatic**, language support is decided automatically depending on the filename extension of the file being compiled:

- Files with the filename extension `.c` will be compiled as C source files
- Files with the filename extension `.cpp` will be compiled as Extended Embedded C++ source files.
REQUIRE PROTOTYPES

This option forces the compiler to verify that all functions have proper prototypes. Using this option means that code containing any of the following will generate an error:

- A function call of a function with no declaration, or with a Kernighan & Ritchie C declaration
- A function definition of a public function with no previous prototype declaration
- An indirect function call through a function pointer with a type that does not include a prototype.

LANGUAGE CONFORMANCE

Language extensions must be enabled for the compiler to be able to accept ARM-specific keywords as extensions to the standard C or C++ language. In the IDE, the option Allow IAR extensions is enabled by default.

The option Relaxed ISO/ANSI disables IAR Systems extensions, but does not adhere to strict ISO/ANSI.

Select the option Strict ISO/ANSI to adhere to the strict ISO/ANSI C standard.

For details about language extensions, see the IAR C/C++ Development Guide for ARM®.

PLAIN 'CHAR' IS

Normally, the compiler interprets the char type as unsigned char. Use this option to make the compiler interpret the char type as signed char instead, for example for compatibility with another compiler.

Note: The runtime library is compiled with unsigned plain characters. If you select the Signed option, you might get type mismatch warnings from the linker as the library uses unsigned char.

ENABLE MULTIBYTE SUPPORT

By default, multibyte characters cannot be used in C or Embedded C++ source code. If you use this option, multibyte characters in the source code are interpreted according to the host computer’s default setting for multibyte support.

Multibyte characters are allowed in C and C++ style comments, in string literals, and in character constants. They are transferred untouched to the generated code.
The **Code** options determine several target-specific settings for code generation.

![Code Options Diagram](image)

**GENERATE INTERWORK CODE**

Use this option, which is selected by default, to be able to mix ARM and Thumb code.

**PROCESSOR MODE**

Choose between the following two options to select the processor mode for your project:

- **Arm**
  Generates code that uses the full 32-bit instruction set.

- **Thumb**
  Generates code that uses the reduced 16-bit instruction set. Thumb code minimizes memory usage and provides higher performance in 8/16-bit bus environments.

**POSITION-INDEPENDENCE**

Choose between the following options for position-independent code and data:

- **Code and read-only data (ropi)**
  Generates code that uses PC-relative references to address code and read-only data.

- **Read/write data (rwpi)**
  Generates code that uses an offset from the static base register to address writable data.

- **No dynamic read/write initialization**
  Disables runtime initialization of static C variables.
For detailed reference information about these compiler options, see the *IAR C/C++ Development Guide for ARM*.

**Optimizations**

The **Optimizations** options determine the type and level of optimization for generation of object code.

![Optimizations Options](image)

**OPTIMIZATIONS**

The compiler supports various levels of optimizations, and for the highest level you can fine-tune the optimizations explicitly for an optimization goal—size or speed. Choose between:

- **None** (best debug support)
- **Low**
- **Medium**
- **High, balanced** (balancing between speed and size)
- **High, speed** (favors speed)
- **High, size** (favors size).

By default, a debug project will have a size optimization that is fully debuggable, while a release project will have a high balanced optimization that generates small code without sacrificing speed.

For a list of optimizations performed at each optimization level, see the *IAR C/C++ Development Guide for ARM*. 
Enabled transformations
These transformations are available on different level of optimizations:
● Common subexpression elimination
● Loop unrolling
● Function inlining
● Code motion
● Type-based alias analysis
● Static variable clustering
● Instruction scheduling.

When a transformation is available, you can enable or disable it by selecting its check box.

In a debug project the transformations are, by default, disabled. In a release project the transformations are, by default, enabled.

For a brief description of the transformations that can be individually disabled, see the IAR C/C++ Development Guide for ARM®.

Output

The **Output** options determine settings for the generated output.

![Output options](image)

*Figure 252: Compiler output options*
**GENERATE DEBUG INFORMATION**

This option causes the compiler to include additional information in the object modules that is required by C-SPY® and other symbolic debuggers.

The **Generate debug information** option is selected by default. Deselect this option if you do not want the compiler to generate debug information.

**Note:** The included debug information increases the size of the object files.

**CODE SECTION NAME**

The compiler places functions into named sections which are referred to by the IAR ILINK Linker. Use the text field to specify a different name than the default name to place any part of your application source code into separate non-default sections. This is useful if you want to control placement of your code to different address ranges and you find the @ notation, alternatively the #pragma location directive, insufficient.

**Note:** Take care when explicitly placing a function in a predefined section other than the one used by default. This is useful in some situations, but incorrect placement can result in anything from error messages during compilation and linking to a malfunctioning application. Carefully consider the circumstances; there might be strict requirements on the declaration and use of the function or variable.

Note that any changes to the section names require a corresponding modification in the linker configuration file.

For detailed information about sections and the various methods for controlling placement of code, see the *IAR C/C++ Development Guide for ARM®*. 
The List options determine whether a list file is produced, and the information is included in the list file.

Normally, the compiler does not generate a list file. Select any of the following options to generate a list file or an assembler file. The list file will be saved in the List directory, and its filename will consist of the source filename, plus the filename extension lst. You can open the output files directly from the Output folder which is available in the Workspace window.

**OUTPUT LIST FILE**

Select the Output list file option and choose the type of information to include in the list file:

- **Assembler mnemonics**
  - Includes assembler mnemonics in the list file.
- **Diagnostics**
  - Includes diagnostic information in the list file.

**OUTPUT ASSEMBLER FILE**

Select the Output assembler file option and choose the type of information to include in the list file:

- **Include source**
  - Includes source code in the assembler file.
- **Include call frame information**
  - Includes compiler-generated information for runtime model attributes, call frame information, and frame size information.
Preprocessor

The Preprocessor options allow you to define symbols and include paths for use by the compiler.

![Preprocessor options](image)

**IGNORE STANDARD INCLUDE DIRECTORIES**

If you select this option, the standard include files will not be used when the project is built.

**ADDITIONAL INCLUDE DIRECTORIES**

The Additional include directories option adds a path to the list of #include file paths. The paths required by the product are specified automatically based on your choice of runtime library.

Type the full file path of your #include files.

**Note**: Any additional directories specified using this option are searched before the standard include directories.

To make your project more portable, use the argument variable $TOOLKIT_DIR$ for the subdirectories of the active product and $PROJ_DIR$ for the directory of the current project. For an overview of the argument variables, see [*Argument variables summary*, page 328].
**PREINCLUDE FILE**

Use this option to make the compiler include the specified include file before it starts to read the source file. This is useful if you want to change something in the source code for the entire application, for instance if you want to define a new symbol.

**DEFINED SYMBOLS**

The Defined symbols option is useful for conveniently specifying a value or choice that would otherwise be specified in the source file.

Type the symbols that you want to define for the project, for example:

```plaintext
TESTVER=1
```

Note that there should be no space around the equal sign.

The Defined symbols option has the same effect as a `#define` statement at the top of the source file.

For example, you might arrange your source to produce either the test or production version of your application depending on whether the symbol `TESTVER` was defined. To do this you would use include sections such as:

```plaintext
#ifdef TESTVER
  ...
#endif
```

You would then define the symbol `TESTVER` in the Debug target but not in the Release target.

**PREPROCESSOR OUTPUT TO FILE**

By default, the compiler does not generate preprocessor output.

Select the Preprocessor output to file option if you want to generate preprocessor output. You can also choose to preserve comments and/or to generate `#line` directives.

---

**Diagnostics**

The Diagnostics options determine how diagnostics are classified and displayed. Use the diagnostics options to override the default classification of the specified diagnostics.
Note: The diagnostics cannot be suppressed for fatal errors, and fatal errors cannot be reclassified.

Figure 255: Compiler diagnostics options

**ENABLE REMARKS**

The least severe diagnostic messages are called *remarks*. A remark indicates a source code construct that might cause strange behavior in the generated code.

By default, remarks are not issued. Select the **Enable remarks** option if you want the compiler to generate remarks.

**SUPPRESS THESE DIAGNOSTICS**

This option suppresses the output of diagnostics for the tags that you specify.

For example, to suppress the warnings `Pe117` and `Pe177`, type:

```
Pe117,Pe177
```

**TREAT THESE AS REMARKS**

A remark is the least severe type of diagnostic message. It indicates a source code construct that might cause strange behavior in the generated code. Use this option to classify diagnostics as remarks.

For example, to classify the warning `Pe177` as a remark, type:

```
Pe177
```
TREAT THESE AS WARNINGS

A warning indicates an error or omission that is of concern, but which will not cause the compiler to stop before compilation is completed. Use this option to classify diagnostic messages as warnings.

For example, to classify the remark Pe826 as a warning, type:

Pe826

TREAT THESE AS ERRORS

An error indicates a violation of the C or C++ language rules, of such severity that object code will not be generated, and the exit code will be non-zero. Use this option to classify diagnostic messages as errors.

For example, to classify the warning Pe117 as an error, type:

Pe117

TREAT ALL WARNINGS AS ERRORS

Use this option to make the compiler treat all warnings as errors. If the compiler encounters an error, object code is not generated.

MISRA C

Use these options to override the options set on the MISRA-C:1998 and MISRA-C:2004 pages of the General Options category.

For details about specific option, see the IAR Embedded Workbench® MISRA C:2004 Reference Guide or the IAR Embedded Workbench® MISRA C:1998 Reference Guide available from the Help menu.
Extra Options

The Extra Options page provides you with a command line interface to the compiler.

Figure 256: Extra Options page for the compiler

USE COMMAND LINE OPTIONS

Additional command line arguments for the compiler (not supported by the GUI) can be specified here.
Assembler options

This chapter describes the assembler options available in the IAR Embedded Workbench® IDE.

For information about how to set options, see Setting options, page 97.

Language

The Language options control the code generation of the assembler.

- **User symbols are case sensitive**
- **Enable multibyte support**

Multibyte characters are allowed in comments, in string literals, and in character constants. They are transferred untouched to the generated code.
MACRO QUOTE CHARACTERS

The Macro quote characters option sets the characters used for the left and right quotes of each macro argument.

By default, the characters are `<` and `>`. This option allows you to change the quote characters to suit an alternative convention or simply to allow a macro argument to contain `<` or `>`. From the drop-down list, choose one of four types of brackets to be used as macro quote characters.

ALLOW ALTERNATIVE REGISTER NAMES, MNEMONICS AND OPERANDS

To enable migration from an existing application to the IAR Assembler for ARM, alternative register names, mnemonics, and operands can be allowed. This is controlled by the assembler command line option `-j`. Use this option for assembler source code written for the ARM ADS/RVCT assembler. For more information, see the ARM® IAR Assembler Reference Guide.
The **Output** options allow you to generate information to be used by a debugger such as the IAR C-SPY® Debugger.

**GENERATE DEBUG INFORMATION**

The **Generate debug information** option must be selected if you want to use a debugger with your application. By default, this option is selected in a Debug project, but not in a Release project.
The **List** options are used for making the assembler generate a list file, for selecting the list file contents, and generating other listing-type output.

![Figure 260: Assembler list file options](image)

By default, the assembler does not generate a list file. Selecting **Output list file** causes the assembler to generate a listing and send it to the file `sourcename.lst`.

**Note:** If you want to save the list file in another directory than the default directory for list files, use the **Output Directories** option in the **General Options** category; see *Output*, page 447, for additional information.

**INCLUDE HEADER**

The header of the assembler list file contains information about the product version, date and time of assembly, and the command line equivalents of the assembler options that were used. Use this option to include the list file header in the list file.

**INCLUDE LISTING**

Use the suboptions under **Include listing** to specify which type of information to include in the list file:

<table>
<thead>
<tr>
<th>Option</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>#included text</td>
<td>Includes <code>#include</code> files in the list file.</td>
</tr>
<tr>
<td>Macro definitions</td>
<td>Includes macro definitions in the list file.</td>
</tr>
<tr>
<td>Macro expansions</td>
<td>Includes macro expansions in the list file.</td>
</tr>
<tr>
<td>Macro execution info</td>
<td>Prints macro execution information on every call of a macro.</td>
</tr>
</tbody>
</table>

*Table 125: Assembler list file options*
**INCLUDE CROSS-REFERENCE**

The **Include cross reference** option causes the assembler to generate a cross-reference table at the end of the list file. See the *ARM® IAR Assembler Reference Guide* for details.

**LINES/PAGE**

The default number of lines per page is 80 for the assembler list file. Use the **Lines/page** option to set the number of lines per page, within the range 10 to 150.

**TAB SPACING**

By default, the assembler sets eight character positions per tab stop. Use the **Tab spacing** option to change the number of character positions per tab stop, within the range 2 to 9.

### Preprocessor

The **Preprocessor** options allow you to define include paths and symbols in the assembler.

![Preprocessor options](image)

*Figure 261: Assembler preprocessor options*
PREPROCESSOR

IGNORE STANDARD INCLUDE DIRECTORIES
If you select this option, the standard include files will not be used when the project is built.

ADDITIONAL INCLUDE DIRECTORIES
The Additional include directories option adds paths to the list of #include file paths. The path required by the product is specified automatically.
Type the full path of the directories that you want the assembler to search for #include files.
To make your project more portable, use the argument variable $TOOLKIT_DIR$ for the subdirectories of the active product and $PROJ_DIR$ for the directory of the current project. For an overview of the argument variables, see Table 86, Argument variables, page 366.
See the ARM® IAR Assembler Reference Guide for information about the #include directive.

Note: By default, the assembler also searches for #include files in the paths specified in the IASMARM_INC environment variable. We do not, however, recommend that you use environment variables in the IDE.

DEFINED SYMBOLS
This option provides a convenient way of specifying a value or choice that you would otherwise have to specify in the source file.
Type the symbols you want to define, one per line.

- For example, you might arrange your source to produce either the test or production version of your application depending on whether the symbol TESTVER was defined. To do this you would use include sections such as:

  #ifdef TESTVER
  ... ; additional code lines for test version only
  #endif

  You would then define the symbol TESTVER in the Debug target but not in the Release target.

- Alternatively, your source might use a variable that you need to change often, for example FRAMERATE. You would leave the variable undefined in the source and use this option to specify a value for the project, for example FRAMERATE=3.

To delete a user-defined symbol, select in the Defined symbols list and press the Delete key.
Diagnostics

Use the **Diagnostics** options to disable or enable individual warnings or ranges of warnings.

![Diagnostics options](image)

*Figure 262: Assembler diagnostics options*

The assembler displays a warning message when it finds an element of the source code that is legal, but probably the result of a programming error.

By default, all warnings are enabled. The **Diagnostics** options allow you to enable only some warnings, or to disable all or some warnings.

Use the radio buttons and entry fields to specify which warnings you want to enable or disable.

For additional information about assembler warnings, see the *ARM® IAR Assembler Reference Guide*.

**MAX NUMBER OF ERRORS**

By default, the maximum number of errors reported by the assembler is 100. This option allows you to decrease or increase this number, for example, to see more errors in a single assembly.
Extra Options

The Extra Options page provides you with a command line interface to the assembler.

Figure 263: Extra Options page for the assembler

USE COMMAND LINE OPTIONS

Additional command line arguments for the assembler (not supported by the GUI) can be specified here.
Converter options

This chapter describes the options available in the IAR Embedded Workbench® IDE for converting output files from the ELF format.

For information about how to set options, see Setting options, page 97.

Output

The Output options are used for specifying details about the producable output format and the level of debugging information included in the output file.

PROMABLE OUTPUT FORMAT

The ILINK linker generates ELF as output, optionally including DWARF for debug information. Use the Promable output format drop-down list to convert the ELF output to a different format, for example Motorola or Intel-extended. The ielftool converter is used for converting the file. For more information about the converter, see the IAR C/C++ Development Guide for ARM®.

OUTPUT FILE

Use Output file to specify the name of the ielftool converted output file. If a name is not specified, the linker will use the project name with a filename extension. The filename extension depends on which output format you choose; for example, either srec or hex.
Override default

Use this option to specify a filename or filename extension other than the default.
Custom build options

This chapter describes the Custom Build options available in the IAR Embedded Workbench® IDE.

For information about how to set options, see Setting options, page 97.

Custom Tool Configuration

To set custom build options in the IDE, choose Project>Options to display the Options dialog box. Then select Custom Build in the Category list to display the Custom Tool Configuration page:

![Custom Tool Configuration](image)

In the Filename extensions text box, specify the filename extensions for the types of files that are to be processed by this custom tool. You can enter several filename extensions. Use commas, semicolons, or blank spaces as separators. For example:

```
.htm; .html
```

In the Command line text box, type the command line for executing the external tool.

In the Output files text box, enter the output files from the external tool.

If any additional files are used by the external tool during the building process, these files should be added in the Additional input files text box. If these additional input files, dependency files, are modified, the need for a rebuild is detected.

For an example, see Extending the tool chain, page 101.
Custom Tool Configuration
Build actions options

This chapter describes the options for pre-build and post-build actions available in the IAR Embedded Workbench® IDE.

For information about how to set options, see Setting options, page 97.

Build Actions Configuration

To set options for pre-build and post-build actions in the IDE, choose Project > Options to display the Options dialog box. Then select Build Actions in the Category list to display the Build Actions Configuration page.

These options apply to the whole build configuration, and cannot be set on groups or files.

PRE-BUILD COMMAND LINE

Type a command line to be executed directly before a build; a browse button for locating an extended command line file is available for your convenience. The commands will not be executed if the configuration is already up-to-date.

POST-BUILD COMMAND LINE

Type a command line to be executed directly after each successful build; a browse button is available for your convenience. The commands will not be executed if the
configuration was up-to-date. This is useful for copying or post-processing the output file.
Linker options

This chapter describes the ILINK options available in the IAR Embedded Workbench® IDE.

For information about how to set options, see Setting options, page 97.

Config

With the Config options you can specify the path and name of the linker command file and define symbols for the configuration file.

![Figure 267: Linker configuration options]

**LINKER CONFIGURATION FILE**

A default linker configuration file is selected automatically based on your project settings. To override this, select the Override default option and specify an alternative file.

The argument variables $TOOLKIT_DIR$ or $PROJ_DIR$ can be used here too, to specify a project-specific or predefined configuration file.

**CONFIGURATION FILE SYMBOL DEFINITIONS**

Define constant configuration symbols to be used in the configuration file. Such a symbol has the same effect as a symbol defined using the define symbol directive in the linker command file.
With the options on the Library page you can make settings for library usage.

![Library page screenshot]

**Figure 268: Library Usage page**

See the *IAR C/C++ Development Guide for ARM®* for more information about available libraries.

**AUTOMATIC RUNTIME LIBRARY SELECTION**

Use this option to make ILINK automatically choose the appropriate library based on your project settings.

**ADDITIONAL LIBRARIES**

Use the text box to specify additional libraries that you want the linker to include during the link process. You can only specify one library per line and you must specify the full path to the library. The argument variables `$PROJ_DIR$` and `$TOOLKIT_DIR$` can be used.

Alternatively, you can add an additional library directly to your project in the workspace window. You can find an example of this in the tutorial for creating and using libraries.

**OVERRIDE DEFAULT PROGRAM ENTRY**

By default, the program entry is the label `__iar_program_start`. The linker will make sure that a module containing the program entry label is included, and that the section containing that label is not discarded.
Use the option **Override default program entry** to override the default entry label. Choose between:

- **Entry symbol**
  Specifies a different entry symbol than used by default. Use the text field to specify a symbol other than `__iar_program_start` to use for the program entry.

- **Defined by application**
  Disables the use of an entry symbol. The linker will, as always, include all program modules, and enough library modules to satisfy all symbol references, keeping all sections that are marked with the root attribute or that are referenced, directly or indirectly, from such a section.

## Input

The **Input** options are used for specifying how to handle input to the linker.

![Image: ILINK input file options](image)

**KEEP SYMBOLS**

Normally, the linker keeps a symbol only if your application needs it.

Use the text box to specify a symbol, or several symbols separated by commas, that you want to always be included in the final application.
RAW BINARY IMAGE

Use the Raw binary image options to link pure binary files in addition to the ordinary input files. Use the text boxes to specify these parameters:

- **File**: The pure binary file you want to link.
- **Symbol**: The symbol defined by the section where the binary data is placed.
- **Section**: The section where the binary data is placed.
- **Align**: The alignment of the section where the binary data is placed.

The entire contents of the file are placed in the section you specify, which means it can only contain pure binary data, for example, the raw-binary output format. The section where the contents of the specified file is placed, is only included if the specified symbol is required by your application. Use the `--keep` linker option if you want to force a reference to the symbol. Read more about single output files and the `--keep` option in the IAR C/C++ Development Guide for ARM®.

Output

The Output options are used for specifying details about the output.

![Output file options](image)

**OUTPUT FILE**

Use Output file to specify the name of the ILINK output file. If a name is not specified, the linker will use the project name with the filename extension `.out`.
OUTPUT DEBUG INFORMATION

Use **Output debug information** to make the linker generate an ELF output file including DWARF for debug information.

**List**

The **List** options determine the generation of a linker listing.

![List Options](image)

**GENERATE LINKER MAP FILE**

Use the **Generate linker map file** option to produce a linker memory map file. The map file has the filename extension `map`. For detailed information about the map file and its contents, see the *IAR C/C++ Development Guide for ARM®*.

**GENERATE LOG**

Use the **Generate log** options to save log information to a file. The log file will be placed in the `list` directory and have the filename extension `log`. The log information can be useful for understanding why an executable image became the way it is. You can optionally choose to log:

- Automatic library selection
- Initialization decisions
- Module selections
- Redirected symbols
- Section selections
- Unused section fragments
- Unused statics
You can define symbols with the `#define` option.

![Defined symbols](image)

**DEFINE SYMBOL**

Use `Define symbol` to define absolute symbols at link time. This is especially useful for configuration purposes.

Type the symbols that you want to define for the project, for example:

```c
TESTVER=1
```

Note that there should be no space around the equal sign.

Any number of symbols can be defined in a linker configuration file. The symbol(s) defined in this manner will be located in a special module called `?ABS_ENTRY_MOD`, which is generated by the linker.

The linker will display an error message if you attempt to redefine an existing symbol.

## Diagnostics

The **Diagnostics** options determine how diagnostics are classified and displayed. Use the diagnostics options to override the default classification of the specified diagnostics.
Note: The diagnostics cannot be suppressed for fatal errors, and fatal errors cannot be reclassified.

Figure 273: Linker diagnostics options

ENABLE REMARKS
The least severe diagnostic messages are called *remarks*. A remark indicates a construction that might cause strange behavior in the generated code.

By default, remarks are not issued. Select the **Enable remarks** option if you want the linker to generate remarks.

SUPPRESS THESE DIAGNOSTICS
This option suppresses the output of diagnostics for the tags that you specify.

For example, to suppress the warnings Pe117 and Pe177, type:

Pe117, Pe177

TREAT THESE AS REMARKS
A remark is the least severe type of diagnostic message. It indicates a construction that might cause strange behavior in the generated code or the executable image. Use this option to classify diagnostics as remarks.

For example, to classify the warning Pe177 as a remark, type:

Pe177
**TREAT THESE AS WARNINGS**

A **warning** indicates an error or omission that is of concern, but which will not cause the linker to stop before linking is completed. Use this option to classify diagnostic messages as warnings.

For example, to classify the remark **Pe826** as a warning, type:

```
Pe826
```

**TREAT THESE AS ERRORS**

An **error** indicates a violation of the linking rules, of such severity that an executable image will not be generated, and the exit code will be non-zero. Use this option to classify diagnostic messages as errors.

For example, to classify the warning **Pe117** as an error, type:

```
Pe117
```

**TREAT ALL WARNINGS AS ERRORS**

Use this option to make the linker treat all warnings as errors. If the linker encounters an error, an executable image is not generated.

---

**Checksum**

With the **Checksum** options you can specify details about how the code is generated.

![Checksum Options](image)

For more information about filling and checksumming, see the *IAR C/C++ Development Guide for ARM*.

---

*IAR Embedded Workbench® IDE 488 User Guide for ARM*
FILL UNUSED CODE MEMORY
Use Fill unused code memory to fill unused memory in the supplied range.

Fill pattern
Use this option to specify size, in hexadecimal notation, of the filler to be used in gaps between segment parts.

Start address
Use this option to specify the start address of the range to be filled.

End address
Use this option to specify the end address of the range to be filled.

Generate checksum
Use Generate checksum to checksum the supplied range.

Size
Size specifies the number of bytes in the checksum, which can be 1, 2, or 4.

Algorithms
One of the following algorithms can be used:

<table>
<thead>
<tr>
<th>Algorithms</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Arithmetic sum</td>
<td>Simple arithmetic sum. The result is truncated to one byte. Use the Result in full size option to get the result in the specified size.</td>
</tr>
<tr>
<td>CRC16</td>
<td>CRC16, generating polynomial 0x11021 (default)</td>
</tr>
<tr>
<td>CRC32</td>
<td>CRC32, generating polynomial 0x104C11DB7</td>
</tr>
<tr>
<td>Crc polynomial</td>
<td>CRC with a generating polynomial of the value you enter</td>
</tr>
</tbody>
</table>

Table 126: Linker checksum algorithms

Complement
Use the Complement drop-down list to specify the one’s complement or two’s complement.
**Bit order**

By default it is the most significant 1, 2, or 4 bytes (MSB) of the result that will be output, in the natural byte order for the processor. Choose LSB from the **Bit order** drop-down list if you want the least significant bytes to be output.

**Alignment**

Use this option to specify an optional alignment for the checksum. If you do not specify an alignment explicitly, an alignment of 2 is used.

**Initial value**

Use this option to specify the initial value of the checksum.

---

### Extra Options

The **Extra Options** page provides you with a command line interface to the linker.

![Extra Options](image)

*Figure 275: Extra Options page for the linker*

**USE COMMAND LINE OPTIONS**

Additional command line arguments for the linker (not supported by the GUI) can be specified here.
Library builder options

This chapter describes the library builder options available in the IAR Embedded Workbench® IDE.

For information about how to set options, see Setting options, page 97.

Output

Options for the library builder are not available by default. Before you can set these options in the IDE, you must add the library builder tool to the list of categories. Choose Project > Options to display the Options dialog box, and select the General Options category. On the Output page, select the Library option.

If you select the Library option, Library Builder appears as a category in the Options dialog box. As a result of the build process, the library builder will create a library output file. Before you create the library you can set output options.

To set options, select Library Builder from the category list to display the options.

Figure 276: Library builder output options
To restore all settings to the default factory settings, click the **Factory Settings** button.
The **Output file** option overrides the default name of the output file. Enter a new name in the **Override default** text box.
Debugger options

This chapter describes the C-SPY® options available in the IAR Embedded Workbench® IDE.

For information about how to set options, see Setting options, page 97.

In addition, for information about options specific to the C-SPY hardware debugger systems, see the chapter Hardware-specific debugging.

Setup

To set C-SPY options in the IDE, choose Project>Options to display the Options dialog box. Then select Debugger in the Category list. The Setup page contains the generic C-SPY options.

![Figure 277: Generic C-SPY options](image)

To restore all settings to the default factory settings, click the Factory Settings button.

The Setup options specify the C-SPY driver, the setup macro file, and device description file to be used, and which default source code location to run to.

**DRIVER**

Selects the appropriate driver for use with C-SPY, for example a simulator or an emulator.
These drivers are currently available:

<table>
<thead>
<tr>
<th>C-SPY driver</th>
<th>Filename</th>
</tr>
</thead>
<tbody>
<tr>
<td>Simulator</td>
<td>armssim2.dll</td>
</tr>
<tr>
<td>Angel</td>
<td>armangel.dll</td>
</tr>
<tr>
<td>GDB Server</td>
<td>armgdbserv.dll</td>
</tr>
<tr>
<td>J-Link/J-Trace</td>
<td>armjlink.dll</td>
</tr>
<tr>
<td>LMI FTDI</td>
<td>armlmiftdi.dll</td>
</tr>
<tr>
<td>Macraigor</td>
<td>armjtag.dll</td>
</tr>
<tr>
<td>RDI</td>
<td>armdii.dll</td>
</tr>
<tr>
<td>ROM-monitor for serial port</td>
<td>armrom.dll</td>
</tr>
<tr>
<td>ROM-monitor for USB</td>
<td>armromUSB.dll</td>
</tr>
<tr>
<td>ST-Link</td>
<td>armstlink.dll</td>
</tr>
</tbody>
</table>

Table 127: C-SPY driver options

Contact your distributor or IAR Systems representative, or visit the IAR Systems web site at www.iar.com for the most recent information about the available C-SPY drivers.

**RUN TO**

Use this option to specify a location you want C-SPY to run to when you start the debugger and after a reset.

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 example function names.

If the option is deselected, the program counter will contain the regular hardware reset address at each reset.

**SETUP MACROS**

To register the contents of a setup macro file in the C-SPY startup sequence, select Use macro file and enter 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.

It is possible to specify up to two different macro files.

**DEVICE DESCRIPTION FILE**

Use this option to load a device description file that contains device-specific information.
For details about the device description file, see *Device description file*, page 125.

Device description files for each ARM device are provided in the directory `arm\config` and have the filename extension `ddf`.

## Download

Options specific to the C-SPY drivers are described in the chapter *Hardware-specific debugging*, page 243 in Part 6, *C-SPY hardware debugger systems*.

## Images

On the **Images** page you can specify three additional debug files to download.

![Images page for C-SPY](image)

*Figure 278: Images page for C-SPY*

If you want to download more images, use the related C-SPY macro, see **__loadImage**, page 549.

**DOWNLOAD EXTRA IMAGE**

Select **Download extra image** to enable the options for the additional debug file to download.

Use these options to specify the additional image that you want to download:

- **Path**: Specifies the debug file to download. A browse button is available for your convenience.

- **Offset**: An integer that specifies the destination address for the downloaded image.
Extra Options

The Extra Options page provides you with a command line interface to C-SPY.

![Extra Options page for C-SPY](image)

**USE COMMAND LINE OPTIONS**

Additional command line arguments for C-SPY (not supported by the GUI) can be specified here.

Plugins

On the Plugins page you can specify C-SPY plugin modules to be loaded and made 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.

**Figure 280: C-SPY plugin options**

By default, **Select plugins to load** lists the plugin modules delivered with the product installation.

If you have any C-SPY plugin modules delivered by any third-party vendor, these will also appear in the list.

Any plugin modules for real-time operating systems will also appear in the list of plugin modules. Some information about the CMX-RTX plugin module can be found in the document **cmx_quickstart.pdf**, delivered with this product. The μC/OS-II plugin module is documented in the **μC/OS-II Kernel Awareness for C-SPY User Guide**, available from Micrium, Inc.

The `common\plugins` directory is intended for generic plugin modules. The `arm\plugins` directory is intended for target-specific plugin modules.
The C-SPY Command Line Utility—cspybat

You can execute the IAR C-SPY Debugger in batch mode, using the C-SPY Command Line Utility—cspybat.exe—which is described in this chapter.

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 arm\bin.</td>
</tr>
<tr>
<td>driver_DLL</td>
<td>The C-SPY driver DLL file; available in arm\bin.</td>
</tr>
<tr>
<td>debug_file</td>
<td>The object file that you want to debug (filename extension out).</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 505.</td>
</tr>
<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 505.</td>
</tr>
</tbody>
</table>

Table 128: cspybat parameters
Example

This example starts **cspybat** using the simulator driver:

```
EW_DIR\common\bin\cspybat  EW_DIR\\arm\bin\armproc.dll
EW_DIR\arm\bin\armsim.dll  PROJ_DIR\myproject.out --plugin
EW_DIR\arm\bin\armbat.dll  --backend sim -B --cpu arm -p
EW_DIR\arm\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.

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**.
- 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.

---

**C-SPY command line options**

**GENERAL CSPYBAT OPTIONS**

```
--backend
```

Marks the beginning of the parameters to be sent to the C-SPY driver (mandatory).
OPTIONS AVAILABLE FOR ALL C-SPY DRIVERS

-B Enables batch mode.

--BE8 Uses the big-endian format BE8. For reference information, see the IAR C/C++ Development Guide for ARM®.

--BE32 Uses the big-endian format BE32. For reference information, see the IAR C/C++ Development Guide for ARM®.

--cpu Specifies a processor variant. For reference information, see the IAR C/C++ Development Guide for ARM®.

--device Specifies the name of the device.

--drv_attach_to_program Attaches the debugger to a running application at its current location. For reference information, see Attach to program, page 245.

--drv_catch_exceptions Makes the application stop for certain exceptions.

--drv_communication Specifies the communication link to be used.

--drv_communication_log Creates a log file.

--drv_default_breakpoint Sets the type of breakpoint resource to be used when setting breakpoints.

--drv_reset_to_cpu_start Omits setting the PC when starting or resetting the debugger.
C-SPY command line options

--drv_restore_breakpoints Restores automatically any breakpoints that were destroyed during system startup.

--drv_suppress_download Suppresses download of the executable image. For reference information, see Suppress download, page 245.

--drv_vector_table_base Specifies the location of the Cortex-M reset vector and the initial stack pointer value.

--drv_verify_download Verifies the target program. For reference information, see Verify download, page 245. Available for Angel, GDB Server, IAR ROM-monitor, J-Link/J-Trace, LMI FTDI, Macraigor, RDI, and ST-Link.

--endian Specifies the byte order of the generated code and data. For reference information, see the IAR C/C++ Development Guide for ARM®.

--fpu Selects the type of floating-point unit. For reference information, see the IAR C/C++ Development Guide for ARM®.

-p Specifies the device description file to be used.

--proc_stack_stack Provides information to the C-SPY plugin module about reserved stacks.

--semihosting Enables semihosted I/O.

OPTIONS AVAILABLE FOR THE SIMULATOR DRIVER

--disable_interrupts Disables the interrupt simulation.

--mapu Activates memory access checking.
OPTIONS AVAILABLE FOR THE C-SPY ANGEL DEBUG MONITOR DRIVER

--rdi_heartbeat               Makes C-SPY poll your target system periodically. For reference information, see Send heartbeat, page 247.

--rdi_step_max_one            Executes one instruction.

OPTIONS AVAILABLE FOR THE C-SPY GDB SERVER DRIVER

--gdbserv_exec_command        Sends a command string to the GDB Server.

OPTIONS AVAILABLE FOR THE C-SPY IAR ROM-MONITOR DRIVER

There are no additional options specific to the C-SPY IAR ROM-monitor driver.

OPTIONS AVAILABLE FOR THE C-SPY J-LINK/J-TRACE DRIVER

--jlink_device_select         Selects a specific device in the JTAG scan chain.

--jlink_exec_command          Calls the __jlinkExecCommand macro after target connection has been established.

--jlink_initial_speed         Sets the initial JTAG communication speed in kHz.

--jlink_interface             Specifies the communication between the J-Link debug probe and the target system.

--jlink_ir_length             Sets the number of IR bits before the ARM device to be debugged.

--jlink_reset_strategy        Selects the reset strategy to be used at debugger startup.

--jlink_speed                 Sets the JTAG communication speed in kHz.

OPTIONS AVAILABLE FOR THE C-SPY LMI FTDI DRIVER

--lmiftdi_speed               Sets the JTAG communication speed in kHz.

OPTIONS AVAILABLE FOR THE C-SPY MACRAIGOR DRIVER

--mac_handler_address         Specifies the location of the debug handler used by Intel XScale devices.

--mac_interface               Specifies the communication between the Macraigor debug probe and the target system.
C-SPY command line options

OPTIONS AVAILABLE FOR THE C-SPY RDI DRIVER

--mac_jtag_device   Selects the device corresponding to the hardware interface.
--mac_multiple_targets  Specifies the device to connect to, if there are more than one device on the JTAG scan chain.
--mac_reset_pulls_reset  Makes C-SPY generate an initial hardware reset.
--mac_set_temp_reg_buffer  Provides the driver with a physical RAM address for accessing the coprocessor.
--mac_speed   Sets the JTAG speed between the JTAG interface and the ARM JTAG ICE port.
--mac_xscale_ir7  Specifies that the XScale ir7 architecture is used.

--rdi_allow_hardware_reset  Performs a hardware reset.
--rdi_driver_dll  Specifies the path to the RDI driver DLL file.
--rdi_use_etm  Enables C-SPY to use and display ETM trace.
--rdi_step_max_one  Executes one instruction.

OPTIONS AVAILABLE FOR THE C-SPY ST-LINK DRIVER

--stlink_interface  Specifies the communication between the ST-Link debug probe and the target system.

OPTIONS AVAILABLE FOR THE THIRD-PARTY DRIVERS

For information about any options specific to the third-party driver you are using, see its documentation.
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.

-B

Syntax
-B

Applicability
All C-SPY drivers.

Description
Use this option to enable batch mode.

--backend

Syntax
--backend {driver options}

Parameters

driver options Any option available to the C-SPY driver you are using.

Applicability
Sent to cspybat (mandatory).

Description
Use this option to send options to the C-SPY driver. All options that follow --backend will be passed to the C-SPY driver, and will not be processed by cspybat itself.

--code_coverage_file

Syntax
--code_coverage_file file

Parameters

file The name of the destination file for the code coverage information.

Applicability
Sent to cspybat.

Description
Use this option to enable the generation of code coverage information. The code coverage information will be generated after the execution has completed and you can find it in the specified file.

Note that this option requires that the C-SPY driver you are using supports code coverage. If you try to use this option with a C-SPY driver that does not support code coverage, an error message will be directed to stderr.
Descriptions of C-SPY command line options

See also  
*Code Coverage window*, page 427.

---

**--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.

---

**--device**

*Syntax*  
`--device=device_name`

*Parameters*  
`device_name`  
The name of the device, for example, ADuC7030, AT91SAM7S256, LPC2378, STR912FM44, or TMS470R1B1M.

*Applicability*  
All C-SPY drivers.

*Description*  
Use this option to specify the name of the device.

To set related option, choose:

*Project>*Options>*General Options>*Target>*Device

---

**--disable_interrupts**

*Syntax*  
`--disable_interrupts`

*Applicability*  
The C-SPY Simulator driver.
Description
Use this option to disable the interrupt simulation.

To set this option, choose Simulator>Interrupt Setup and deselect the Enable interrupt simulation option.

--download_only

Syntax
--download_only

Applicability
Sent to cspybat.

Description
Use this option to execute the flash loader without starting a debug session afterwards.

See also
The IAR Embedded Workbench flash loader User Guide.

To set related options, choose:
Project>Download

--drv_catch_exceptions

Syntax
--drv_catch_exceptions=value

Parameters
value
(for ARM9 and Cortex-R4) A value in the range of 0-0x1FF. Each bit specifies which exception to catch:
Bit 0 = Reset
Bit 1 = Undefined instruction
Bit 2 = SWI
Bit 3 = Not used
Bit 4 = Data abort
Bit 5 = Prefetch abort
Bit 6 = IRQ
Bit 7 = FIQ
Bit 8 = Other errors
Descriptions of C-SPY command line options

_value_

A value in the range of 0–0x7FF. Each bit specifies which exception to catch:
- Bit 0 = CORERESET - Reset Vector
- Bit 4 = MMERR - Memory Management Fault
- Bit 5 = NOCPERR - Coprocessor Access Error
- Bit 6 = CHKERR - Checking Error
- Bit 7 = STATERR - State Error
- Bit 8 = BUSERR - Bus Error
- Bit 9 = INTERR - Interrupt Service Errors
- Bit 10 = HARDERR - Hard Fault

Applicability
The C-SPY Angel debug monitor driver.
The C-SPY J-Link/J-Trace driver
The C-SPY RDI driver.

Description
Use this option to make the application stop when a certain exception occurs.

See also
Breakpoints on vectors, page 280.

For the C-SPY Angel debug monitor driver, use:
Project>Options>Debugger>Extra Options

For the C-SPY J-Link/J-Trace driver, use:
Project>Options>Debugger>J-Link/J-Trace>Catch exceptions

For the C-SPY RDI driver, use:
Project>Options>Debugger>RDI>Catch exceptions

_value_

A value in the range of 0–0x7FF. Each bit specifies which exception to catch:

Syntax
--drv_communication=connection

Parameters
Where _connection_ is one of these for the C-SPY Angel debug monitor driver:

Via Ethernet
- UDP:ip_address
- UDP:ip_address,port
- UDP:hostname
- UDP:hostname,port
Where `connection` is one of these for the C-SPY GDB Server driver:

### Via serial port

```
port:baud,parity,stop_bit,handshake
port = COM1–COM256 (default COM1)
baud = 9600, 19200, 38400, 57600, or 115200 (default 9600 baud)
parity = N (no parity)
stop_bit = 1 (one stop bit)
handshake = NONE or RTSCTS (default NONE for no handshaking)
For example, COM1:9600,N,8,1,NONE.
```

### Via Ethernet

```
TCPIP:ip_address
TCPIP:ip_address,port
TCPIP:hostname
TCPIP:hostname,port
```

Note that if no port is specified, port 3333 is used by default.

Where `connection` is one of these for the C-SPY IAR ROM-monitor driver:

### Via serial port

```
port:baud,parity,stop_bit,handshake
port = COM1–COM256 (default COM1)
baud = 9600, 19200, 38400, 57600, or 115200 (default 9600 baud)
parity = N (no parity)
stop_bit = 1 (one stop bit)
handshake = NONE or RTSCTS (default NONE for no handshaking)
For example, COM1:9600,N,8,1,NONE.
```

Where `connection` is one of these for the C-SPY J-Link/J-Trace driver:

### Via USB directly to J-Link

```
USB0–USB3
```

### Via J-Link server

```
TCPIP:ip_address
TCPIP:ip_address,port
TCPIP:hostname
TCPIP:hostname,port
```

Note that if no port is specified, port 19020 is used by default.
Descriptions of C-SPY command line options

Where connection is one of these for the C-SPY Macraigor driver:

For mpDemon

port: baud
port = COM1–COM4
baud = 9600, 19200, 38400, 57600, or 115200 (default 9600 baud)

For mpDemon

TCP: ip_address
TCP: ip_address, port
TCP: hostname
TCP: hostname, port
Note that if no port is specified, port 19020 is used by default.

Via USB to usbDemon and USB ports = USB0–USB3

Applicability
The C-SPY Angel debug monitor driver
The C-SPY GDB Server driver
The C-SPY IAR ROM-monitor driver.
The C-SPY J-Link/J-Trace driver
The C-SPY Macraigor driver.

Description
Use this option to choose communication link.

Project>Options>Debugger>Angel>Communication
Project>Options>Debugger>GDB Server>TCP/IP address or hostname [,port]
Project>Options>Debugger>IAR ROM-monitor>Communication
Project>Options>Debugger>J-Link/J-Trace>Connection>Communication
To set related options for the C-SPY Macraigor driver, choose:

Project>Options>Debugger>Macraigor

--drv_communication_log

Syntax

--drv_communication_log=filename

Parameters

filename
The name of the log file.

Applicability
All C-SPY drivers.
**Description**

Use this option to log the communication between C-SPY and the target system to a file. To interpret the result, detailed knowledge of the communication protocol is required.

*Project>Options>Debugger>Driver>Log communication*

---

**--drv_default_breakpoint**

**Syntax**

```
--drv_default_breakpoint={0|1|2}
```

**Parameters**

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>Auto (default)</td>
</tr>
<tr>
<td>1</td>
<td>Hardware</td>
</tr>
<tr>
<td>2</td>
<td>Software</td>
</tr>
</tbody>
</table>

**Applicability**

The C-SPY GDB Server driver  
The C-SPY J-Link/J-Trace driver.  
The C-SPY Macraigor driver.

**Description**

Use this option to select the type of breakpoint resource to be used when setting a breakpoint.

*See also*  
*Default breakpoint type, page 273.*

*Project>Options>Debugger>Driver>Breakpoints>Default breakpoint type*

---

**--drv_reset_to_cpu_start**

**Syntax**

```
--drv_reset_to_cpu_start
```

**Applicability**

The C-SPY Angel debug monitor driver  
The C-SPY GDB Server driver  
The C-SPY J-Link/J-Trace driver  
The C-SPY LMI FTDI driver  
The C-SPY Macraigor driver  
The C-SPY RDI driver.
## Description

Use this option to omit setting the PC when starting or resetting the debugger. Instead PC will have the original value set by the CPU, which is the address of the application entry point.

To set this option, use **Project>Options>Debugger>Extra Options.**

### --drv_restore_breakpoints

**Syntax**

```
--drv_restore_breakpoints=location
```

**Parameters**

- **location**
  - Address or function name label

**Applicability**

- The C-SPY GDB Server driver
- The C-SPY J-Link/J-Trace driver
- The C-SPY Macraigor driver.

**Description**

Use this option to restore automatically any breakpoints that were destroyed during system startup.

**See also**

*Restore software breakpoints at*, page 273.

### --drv_vector_table_base

**Syntax**

```
--drv_vector_table_base=expression
```

**Parameters**

- **expression**
  - A label or an address

**Applicability**

- The C-SPY GDB Server driver
- The C-SPY J-Link/J-Trace driver
- The C-SPY LMI FTDI driver
- The C-SPY Macraigor driver
- The C-SPY RDI driver
- The C-SPY Simulator driver.
Use this option for Cortex-M to specify the location of the reset vector and the initial stack pointer value. This is useful if you want to override the default __vector_table label—defined in the system startup code—in the application or if the application lacks this label, which can be the case if you debug code that is built by tools from another vendor.

To set this option, use Project>Options>Debugger>Extra Options.

---

**--flash_loader**

**Syntax**

```
--flash_loader filename
```

**Parameters**

- `filename` The flash loader specification XML file.

**Applicability**

Sent to cspybat.

**Description**

Use this option to specify a flash loader specification xml file which contains all relevant information about the flash loading. There can be more than one such argument, in which case each argument will be processed in the specified order, resulting in several flash programming passes.

**See also**

The *IAR Embedded Workbench flash loader User Guide*.

---

**--gdbserv_exec_command**

**Syntax**

```
--gdbserv_exec_command="string"
```

**Parameters**

- `string` String or command sent to the GDB Server; see its documentation for more information.

**Applicability**

The C-SPY GDB Server driver.

**Description**

Use this option to send strings or commands to the GDB Server.

To set this option, use Project>Options>Debugger>Extra Options.
--jlink_device_select

Syntax
---jlink_device_select=tap_number

Parameters

tap_number
The TAP position of the device you want to connect to.

Applicability
The C-SPY J-Link/J-Trace driver.

Description
If there is more than one device on the JTAG scan chain, use this option to select a specific device.

See also
JTAG scan chain, page 256.

Project>Options>Debugger>J-Link/J-Trace>Connection>JTAG scan chain>TAP number

--jlink_exec_command

Syntax
---jlink_exec_command=cmdstr1; cmdstr2; cmdstr3 ...

Parameters

cmdstrn
J-Link/J-Trace command string.

Applicability
The C-SPY J-Link/J-Trace driver.

Description
Use this option to make the debugger call the __jlinkExecCommand macro with one or several command strings, after target connection has been established.

See also
__jlinkExecCommand, page 544.

Project>Options>Debugger>Extra Options

--jlink_initial_speed

Syntax
---jlink_initial_speed=speed

Parameters

speed
The initial communication speed in kHz. If no speed is specified, 32 kHz will be used as the initial speed.
Applicability
The C-SPY J-Link/J-Trace driver.

Description
Use this option to set the initial JTAG communication speed in kHz.

See also
JTAG/SWD speed, page 255.

--jlink_interface

Syntax
--jlink_interface={JTAG|SWD}

Parameters
- JTAG
  Uses JTAG communication with the target system (default).
- SWD
  Uses SWD communication with the target system (Cortex-M only); uses fewer pins than JTAG communication.

Applicability
The C-SPY J-Link/J-Trace driver.

Description
Use this option to specify the communication channel between the J-Link debug probe and the target system.

See also
Interface, page 256.

--jlink_ir_length

Syntax
--jlink_ir_length=\texttt{length}

Parameters
- length
  The number of IR bits before the ARM device to be debugged, for JTAG scan chains that mix ARM devices with other devices.

Applicability
The C-SPY J-Link/J-Trace driver.

Description
Use this option to set the number of IR bits before the ARM device to debugged.
Description of C-SPY command line options

See also

JTAG scan chain, page 256.

Project>Options>Debugger>J-Link/J-Trace>Connection>JTAG scan chain>Preceding bits

--jlink_reset_strategy

Syntax

--jlink_reset_strategy={delay|strategy}

Parameters

delay

For Cortex-M and ARM 7/9/11 with strategies 1–9, delay should be 0 (ignored). For ARM 7/9/11 with strategy 0, the delay should be one of 0–10000.

strategy

Strategies for ARM 7/9/11:

0 = Hardware, halt after reset (delay is used for this strategy)
1 = Hardware, halt with BP@0
2 = Software, for Analog Devices ADuC7xxx MCUs
4 = Hardware, halt with WP
5 = Hardware, halt with DBGQR
8 = Software, for Atmel AT91SAM7 MCUs
9 = Hardware, for NXP LPCxxxx MCUs

Strategies for Cortex-M:

0 = Normal reset via reset pin, halt after reset
1 = Reset only core
2 = Reset via reset pin
3 = Connect during reset, for STM32 devices
4 = Halt after bootloader, for LPC11xx and LPC13xx devices
5 = Halt before bootloader, for LPC11xx and LPC13xx devices

Applicability

The C-SPY J-Link/J-Trace driver.

Description

Use this option to select the reset strategy to be used at debugger startup.

See also

Reset, page 253.

Project>Options>Debugger>J-Link/J-Trace>Setup>Reset
--jlink_speed

Syntax
```
--jlink_speed={fixed|auto|adaptive}
```

Parameters

<table>
<thead>
<tr>
<th>Option</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>fixed</td>
<td>1–12000</td>
</tr>
<tr>
<td>auto</td>
<td>The highest possible frequency for reliable operation (default)</td>
</tr>
<tr>
<td>adaptive</td>
<td>For ARM devices that have the RTCK JTAG signal available</td>
</tr>
</tbody>
</table>

Applicability
The C-SPY J-Link/J-Trace driver.

Description
Use this option to set the JTAG communication speed in kHz.

See also
`JTAG/SWD speed`, page 255.

Project>Options>Debugger>J-Link/J-Trace>Setup>JTAG speed

--lmiftdi_speed

Syntax
```
--lmiftdi_speed=frequency
```

Parameters

<table>
<thead>
<tr>
<th>Option</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>frequency</td>
<td>The frequency in kHz.</td>
</tr>
</tbody>
</table>

Applicability
The C-SPY LMI FTDI driver.

Description
Use this option to set the JTAG communication speed in kHz.

See also
`JTAG/SWD speed`, page 261.

Project>Options>Debugger>LMI FTDI>Setup>JTAG speed

--mac_handler_address

Syntax
```
--mac_handler_address=address
```

Parameters

<table>
<thead>
<tr>
<th>Option</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>address</td>
<td>The start address of the memory area for the debug handler.</td>
</tr>
</tbody>
</table>

Applicability
The C-SPY Macraigor driver
Descriptions of C-SPY command line options

Description
Use this option to specify the location—the memory address—of the debug handler used by Intel XScale devices.

See also
Debug handler address, page 264.

Project>Options>Debugger>Macraigor>Debug handler address

--mac_interface

Syntax
--mac_interface={JTAG|SWD}

Parameters

JTAG
Uses JTAG communication with the target system (default).

SWD
Uses SWD communication with the target system (Cortex-M only); uses fewer pins than JTAG communication.

Applicability
The C-SPY Macraigor driver.

Description
Use this option to specify the communication channel between the Macraigor debug probe and the target system.

Project>Options>Debugger>Macraigor>Interface

--mac_jtag_device

Syntax
--mac_jtag_device=device

Parameters

device
The device corresponding to the hardware interface that is used. Choose between Macraigor mpDemon, usbdemon, and usb2demon.

Applicability
The C-SPY Macraigor driver.

Description
Use this option to select the device corresponding to the hardware interface that is used.

See also
OCD interface device, page 262.

Project>Options>Debugger>Macraigor>OCD interface device
--mac_multiple_targets

Syntax
--mac_multiple_targets=<tap-no>@dev0, dev1, dev2, dev3,...

Parameters

*tap-no* The TAP number of the device to connect to, where 0 connects to the first device, 1 to the second, and so on.

*dev0-devn* The nearest TDO pin on the Macraigor JTAG interface.

Applicability
The C-SPY Macraigor driver.

Description
If there is more than one device on the JTAG scan chain, each device must be defined. Use this option to specify which device you want to connect to.

Example
--mac_multiple_targets=0@ARM7TDMI, ARM7TDMI

See also
JTAG scan chain with multiple targets, page 263.

--mac_reset_pulls_reset

Syntax
--mac_reset_pulls_reset=time

Parameters

*time* 0–2000 which is the delay in milliseconds after reset.

Applicability
The C-SPY Macraigor driver.

Description
Use this option to make C-SPY perform an initial hardware reset when the debugger is started, and to specify the delay for the reset.

See also
Hardware reset, page 263.
Descriptions of C-SPY command line options

--macro

Syntax

    --macro filename

Parameters

    filename          The C-SPY macro file to be used (filename extension .mac).

Applicability

Sent to cspybat.

Description

Use this option to specify a C-SPY macro file to be loaded before executing the target application. This option can be used more than once on the command line.

See also

The macro file, page 156.

--mac_set_temp_reg_buffer

Syntax

    --mac_set_temp_reg_buffer=address

Parameters

    address          The start address of the RAM area.

Applicability

Sent to the C-SPY Macraigor driver.

Description

Use this option to specify the start address of the RAM area that is used for controlling the MMU and caching via the CP15 coprocessor.

To set this option, use Project->Options->Debugger->Extra Options.

--mac_speed

Syntax

    --mac_speed={factor}

Parameters

    factor          The factor by which the JTAG interface clock is divided when generating the scan clock. The number must be in the range 1–8 where 1 is the fastest.

Applicability

The C-SPY Macraigor driver.

Description

Use this option to set the JTAG speed between the JTAG interface and the ARM JTAG ICE port.
See also JTAG speed, page 262.

Project>Options>Debugger>Macraigor>JTAG speed

--mac_xscale_ir7

Syntax
--mac_xscale_ir7

Applicability
The C-SPY Macraigor driver.

Description
Use this option to specify that the X Scale ir7 core is used, instead of X Scale ir5. Note that this option is mandatory when using the X Scale ir7 core.

These X Scale cores are supported by the C-SPY Macraigor driver:

- Intel X Scale Core 1 (5-bit instruction register—ir5)
- Intel X Scale Core 2 (7-bit instruction register—ir7)

To set this option, use Project>Options>Debugger>Extra Options.

--mapu

Syntax
--mapu

Applicability
Sent to C-SPY simulator driver.

Description
Specify this option to use the section 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 203.

To set related options, choose:
Simulator>Memory Access Setup
Descriptions of C-SPY command line options

-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
Device description file, page 125

--plugin
Syntax
--plugin filename
Parameters
filename The plugin file to be used (filename extension dll).
Applicability
Sent to csybat.
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 csybat, a dedicated plugin module called armbat.dll located in the arm\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 csybat specifically. This means that the C-SPY plugin modules located in the common\plugins directory cannot normally be used with csybat.
**--proc_stack_stack**

**Syntax**

```
--proc_stack_stack=startaddress, endaddress
```

where `stack` is one of `usr`, `svc`, `irq`, `fiq`, `und`, or `abt` for ARM7/9/11 and XScale and where `stack` is one of `main`, or `proc` for Cortex-M

**Parameters**

- `startaddress`: The start address of the stack, specified either as a value or as an expression.
- `endaddress`: The end address of the stack, specified either as a value or as an expression.

**Applicability**

All C-SPY drivers. Note that this command line option is only available when using C-SPY from the IDE; not in batch mode using `cspybat`.

**Description**

Use this option to provide information to the C-SPY stack plugin module about reserved stacks. By default, C-SPY receives this information from the system startup code, but if you for some reason want to override the default values, this option can be useful.

*Statics window*, page 422.

To set this option, use `Project>Options>Debugger>Extra Options`.

**--rdi_allow_hardware_reset**

**Syntax**

```
--rdi_allow_hardware_reset
```

**Applicability**

The C-SPY RDI driver.

**Description**

Use this option to allow the emulator to perform a hardware reset of the target. Requires support by the emulator.

**See also**

*Allow hardware reset*, page 265.

To set this option, use `Project>Options>Debugger>RDI>Allow hardware reset`.
Descriptions of C-SPY command line options

--rdi_driver_dll

Syntax  
--rdi_driver_dll filename

Parameters  
filename  The file or path to the RDI driver DLL file.

Applicability  
The C-SPY RDI driver.

Description  
Use this option to specify the path to the RDI driver DLL file provided with the JTAG pod.

See also  
Manufacturer RDI driver, page 265.

--rdi_use_etm

Syntax  
--rdi_use_etm

Applicability  
The C-SPY RDI driver.

Description  
Use this option to enable C-SPY to use and display ETM trace.

See also  
ETM trace, page 266.

--rdi_step_max_one

Syntax  
--rdi_step_max_one

Applicability  
The C-SPY Angel debug monitor driver

The C-SPY RDI driver.

Description  
Use this option to execute only one instruction. The debugger will turn off interrupts while stepping and, if necessary, simulate the instruction instead of executing it.

To set this option, use Project>Options>Debugger>Extra Options.
--semihosting

**Syntax**

```
--semihosting={none|iar_breakpoint}
```

**Parameters**

- **No parameter**
  - Use standard semihosting.
- **none**
  - Does not use semihosted I/O.
- **iar_breakpoint**
  - Uses the IAR proprietary semihosting variant.

**Applicability**

All C-SPY drivers.

**Description**

Use this option to enable semihosted I/O and to choose the kind of semihosting interface to use. Note that if this option is not used, semihosting will by default be enabled and C-SPY will try to choose the correct semihosting mode automatically. This means that normally you do not have to use this option if your application is linked with semihosting.

To make semihosting work, your application must be linked with a semihosting library.

**See also**

The *IAR C/C++ Development Guide for ARM®* for more information about linking with semihosting.

---

--silent

**Syntax**

```
--silent
```

**Applicability**

Sent to cspybat.

**Description**

Use this option to omit the sign-on message.

---

--stlink_interface

**Syntax**

```
--stlink_interface={JTAG|SWD}
```

**Parameters**

- **JTAG**
  - Uses JTAG communication with the target system (default).
- **SWD**
  - Uses SWD communication with the target system.
Descriptions of C-SPY command line options

Applicability
The C-SPY ST-Link driver.

Description
Use this option to specify the communication channel between the ST-Link debug probe and the target system.

See also
*Interface*, page 268.

---

**--timeout**

Syntax

```bash
--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.
C-SPY® macros reference

This chapter gives reference information about the C-SPY macros. First a syntax description of the macro language is provided. Then, the available setup macro functions and the pre-defined system macros are summarized. Finally, each system macro is described in detail.

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. You can define macro functions, with or without parameters and return value. You can use built-in system macros, similar to C library functions. Finally, you can define global and local macro variables. You can collect your macro functions in a macro file (filename extension mac).

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.

PREDEFINED SYSTEM MACRO FUNCTIONS

The macro language also includes a wide set of predefined system macro functions (built-in functions), similar to C library functions. For detailed information about each system macro, see Description of C-SPY system macros, page 535.
MACRO VARIABLES

A macro variable is a variable defined and allocated outside your application space. It can then be used in a C-SPY expression. For detailed information about C-SPY expressions, see the chapter C-SPY expressions, page 135.

The syntax for defining one or more macro variables is:

```c
__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:

```c
myvar = 3.5;  // myvar is now type float, value 3.5.
myvar = (int*)i;  // myvar is now type pointer to int, and the value is the same as i.
```

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.

Table 129: Examples of C-SPY macro variables

<table>
<thead>
<tr>
<th>Expression</th>
<th>What it means</th>
</tr>
</thead>
<tbody>
<tr>
<td>myvar = 3.5;</td>
<td><code>myvar</code> is now type <code>float</code>, value 3.5.</td>
</tr>
<tr>
<td>myvar = (int*)i;</td>
<td><code>myvar</code> is now type pointer to <code>int</code>, and the value is the same as <code>i</code>.</td>
</tr>
</tbody>
</table>
The macro function __toString is used for converting from a NULL-terminated C string in your application (char* or char[]) to a macro string. For example, assume this definition of a C string in your application:

```c
char const *cstr = "Hello";
```

Then examine these examples:

```c
__var str;         /* A macro variable */
str = cstr         /* str is now just a pointer to char */
sizeof str         /* same as sizeof (char*), typically 2 or 4 */
str = __toString(cstr,512)  /* str is now a macro string */
sizeof str         /* 5, the length of the string */
str[1]             /* 101, the ASCII code for 'e' */
str += " World!"    /* str is now "Hello World!" */
```

See also Formatted output, page 530.

**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**

```c
expression;
```

For detailed information about C-SPY expressions, see C-SPY expressions, page 135.

**Conditional statements**

```c
if (expression)
  statement

if (expression)
  statement
else
  statement
```

**Loop statements**

```c
for (init_expression; cond_expression; update_expression)
  statement

while (expression)
  statement

do
  statement
```
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:

```c
__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 551.

Examples

Use the __message statement, as in this example:

```c
var1 = 42;
var2 = 37;
__message "This line prints the values ", var1, " and ", var2, " in the Log window.";
```

This should produce this message in the Log window:

This line prints the values 42 and 37 in the Log window.
Use __fmessage to write the output to the designated file, for example:

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

Finally, use __smessage to produce strings, for example:

```
myMacroVar = __smessage 42, " is the answer."
```

myMacroVar now contains the string "42 is the answer".

### Specifying display format of arguments

To override the default display format of a scalar argument (number or pointer) in `argList`, suffix it with a : followed by a format specifier. Available specifiers are `%b` for binary, `%o` for octal, `%d` for decimal, `%x` for hexadecimal and `%c` for character. These match the formats available in the Watch and Locals windows, but number prefixes and quotes around strings and characters are not printed. Another example:

```
__message "The character '\', cvar:%c, ' has the decimal value ", cvar;
```

This might produce:

The character 'A' has the decimal value 65

**Note:** A character enclosed in single quotes (a character literal) is an integer constant and is not automatically formatted as a character. For example:

```
__message 'A', " is the numeric value of the character ", 'A':%c;
```

would produce:

65 is the numeric value of the character A

**Note:** The default format for certain types is primarily designed to be useful in the Watch window and other related windows. For example, a value of type `char` is formatted as 'A' (0x41), while a pointer to a character (potentially a C string) is formatted as 0x8102 "Hello", where the string part shows the beginning of the string (currently up to 60 characters).

When printing a value of type `char*`, use the `%x` format specifier to print just the pointer value in hexadecimal notation, or use the system macro `__toString` to get the full string value.
**Setup macro functions summary**

This table summarizes the available setup macro functions:

<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>execUserFlashInit</td>
<td>Called once before the flash loader is downloaded to RAM. Implement this macro typically for setting up the memory map required by the flash loader. This macro is only called when you are programming flash, and it should only be used for flash loader functionality.</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>execUserFlashReset</td>
<td>Called once after the flash loader is downloaded to RAM, but before execution of the flash loader. This macro is only called when you are programming flash, and it should only be used for flash loader functionality.</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>
<tr>
<td>execUserFlashExit</td>
<td>Called once when the debug session ends. Implement this macro to save status data etc. This macro is useful for flash loader functionality.</td>
</tr>
</tbody>
</table>

Table 130: C-SPY setup macros

**Note:** 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*, page 63.

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.
### C-SPY system macros summary

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>__emulatorSpeed</td>
<td>Sets the emulator clock frequency</td>
</tr>
<tr>
<td>__emulatorStatusCheckOnRead</td>
<td>Enables or disables the verification of the CPSR register after each read operation</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>__gdbserver_exec_command</td>
<td>Send strings or commands to the GDB Server.</td>
</tr>
<tr>
<td>__hwReset</td>
<td>Performs a hardware reset and halt of the target CPU</td>
</tr>
<tr>
<td>__hwResetRunToBp</td>
<td>Performs a hardware reset and then executes to the specified address</td>
</tr>
<tr>
<td>__hwResetWithStrategy</td>
<td>Performs a hardware reset and halt with delay of the target CPU</td>
</tr>
<tr>
<td>__isBatchMode</td>
<td>Checks if C-SPY is running in batch mode or not.</td>
</tr>
<tr>
<td>__jlinkExecCommand</td>
<td>Sends a low-level command to the J-Link/J-Trace driver.</td>
</tr>
<tr>
<td>__jtagCommand</td>
<td>Sends a low-level command to the JTAG instruction register</td>
</tr>
<tr>
<td>__jtagCP15IsPresent</td>
<td>Checks if coprocessor CP15 is available</td>
</tr>
<tr>
<td>__jtagCP15ReadReg</td>
<td>Returns the coprocessor CP15 register value</td>
</tr>
<tr>
<td>__jtagCP15WriteReg</td>
<td>Writes to the coprocessor CP15 register</td>
</tr>
<tr>
<td>__jtagData</td>
<td>Sends a low-level data value to the JTAG data register</td>
</tr>
<tr>
<td>__jtagRawRead</td>
<td>Returns the read data from the JTAG interface</td>
</tr>
<tr>
<td>__jtagRawSync</td>
<td>Writes accumulated data to the JTAG interface</td>
</tr>
<tr>
<td>__jtagRawWrite</td>
<td>Accumulates data to be transferred to the JTAG</td>
</tr>
</tbody>
</table>

*Table 131: Summary of system macros*
<table>
<thead>
<tr>
<th>Macro</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>__jtagResetTRST</td>
<td>Resets the ARM TAP controller via the TRST JTAG signal</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>
<tr>
<td>__restoreSoftwareBreakpoint</td>
<td>Restores any breakpoints that were destroyed during system startup.</td>
</tr>
<tr>
<td>__setCodeBreak</td>
<td>Sets a code breakpoint</td>
</tr>
<tr>
<td>__setDataBreak</td>
<td>Sets a data breakpoint</td>
</tr>
<tr>
<td>__setLogBreak</td>
<td>Sets a log breakpoint</td>
</tr>
<tr>
<td>__setSimBreak</td>
<td>Sets a simulation breakpoint</td>
</tr>
<tr>
<td>__setTraceStartBreak</td>
<td>Sets a trace start breakpoint</td>
</tr>
<tr>
<td>__setTraceStopBreak</td>
<td>Sets a trace stop breakpoint</td>
</tr>
<tr>
<td>__sourcePosition</td>
<td>Returns the file name and source location if the current execution location corresponds to a source location</td>
</tr>
<tr>
<td>__strFind</td>
<td>Searches a given string for the occurrence of another string</td>
</tr>
<tr>
<td>__subString</td>
<td>Extracts a substring from another string</td>
</tr>
<tr>
<td>__targetDebuggerVersion</td>
<td>Returns the version of the target debugger</td>
</tr>
</tbody>
</table>

Table 131: Summary of system macros (Continued)
This section gives reference information about each of the C-SPY system macros.

__cancelAllInterrupts

Syntax
__cancelAllInterrupts()

Return value
int 0

Description
Cancels all ordered interrupts.

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

__toLowerCase
Returns a copy of the parameter string where all the characters have been converted to lower case

__toString
Prints strings

__toUpperCase
Returns a copy of the parameter string where all the characters have been converted to upper case

__unloadImage
Unloads a debug image.

__writeFile
Writes to the specified file

__writeFileByte
Writes one byte to the specified file

__writeMemory8,
__writeMemoryByte
Writes one byte to the specified memory location

__writeMemory16
Writes a two-byte word to the specified memory location

__writeMemory32
Writes a four-byte word to the specified memory location

Table 131: Summary of system macros (Continued)
Description of C-SPY system macros

__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 number</td>
</tr>
</tbody>
</table>

Table 132: __cancelInterrupt return values

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
Defining breakpoints, page 141.

__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

<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 133: __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 between one of these:

- "angel" corresponds to the Angel driver
- "gdbserv" corresponds to the Angel driver
- "generic" corresponds to third-party drivers
- "jlink" corresponds to the J-Link/J-Trace driver
- "jtag" corresponds to the Macraigor driver
- "lmiftdi" corresponds to the LMI FTDI driver
- "rdi" corresponds to the RDI driver
- "rom" corresponds to the ROM-monitor driver
- "sim" corresponds to the simulator driver
- "stlink" corresponds to the ST-Link 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 134: __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.

__emulatorSpeed

Syntax

__emulatorSpeed(speed)

Parameter

speed

The emulator speed in Hz. Use 0 (zero) to make the speed automatically detected. Use -1 for adaptive speed (only for emulators supporting adaptive speed).
__emulatorSpeed

**Syntax**

```c
__emulatorSpeed(0)
```

**Parameter**

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>status</td>
<td>Use 0 to enable checks (default). Use 1 to disable checks.</td>
</tr>
</tbody>
</table>

**Return value**

<table>
<thead>
<tr>
<th>Return value</th>
<th>Result</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>Successful</td>
<td>The previous speed, or 0 (zero) if unknown</td>
<td></td>
</tr>
<tr>
<td>Unsuccessful</td>
<td>the speed is not supported by the emulator</td>
<td></td>
</tr>
</tbody>
</table>

**Description**

Sets the emulator clock frequency. For JTAG interfaces, this is the JTAG clock frequency as seen on the TCK signal.

**Example**

```c
__emulatorSpeed(0)
```

Sets the emulator speed to be automatically detected.

**Applicability**

This system macro is available for the J-Link/J-Trace JTAG interface.

__emulatorStatusCheckOnRead

**Syntax**

```c
__emulatorStatusCheckOnRead(status)
```

**Parameter**

- `status`  
  Use 0 to enable checks (default). Use 1 to disable checks.

**Return value**

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

**Description**

Enables or disables the driver verification of CPSR (current processor status register) after each read operation. Typically, this macro can be used for initiating JTAG connections on some CPUs, like Texas Instruments’ TMS470R1B1M.

**Note:** Enabling this verification can cause problems with some CPUs, for example if invalid CPSR values are returned. However, if this verification is disabled (SetCheckModeAfterRead = 0), the success of read operations cannot be verified and possible data aborts are not detected.

**Example**

```c
__emulatorStatusCheckOnRead(1)
```

Disables the checks for data aborts on memory reads.

**Applicability**

This system macro is available for the J-Link/J-Trace JTAG interface.
Description of C-SPY system macros

__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 136: __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 137: __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.
__gdbserver_exec_command

Syntax

`__gdbserver_exec_command("string")`

Parameter

`string`

String or command sent to the GDB Server; see its documentation for more information.

Description

Use this option to send strings or commands to the GDB Server.

Applicability

This system macro is available for the GDB Server interfaces.

__hwReset

Syntax

`__hwReset(halt_delay)`

Parameter

`halt_delay`

The delay, in microseconds, between the end of the reset pulse and the halt of the CPU. Use 0 (zero) to make the CPU halt immediately after reset.

Return value

<table>
<thead>
<tr>
<th>Result</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>Successful. The actual delay value implemented by the emulator</td>
<td>&gt;=0</td>
</tr>
<tr>
<td>Successful. The delay feature is not supported by the emulator</td>
<td>-1</td>
</tr>
<tr>
<td>Unsuccessful. Hardware reset is not supported by the emulator</td>
<td>-2</td>
</tr>
</tbody>
</table>

Table 138: __hwReset return values

Description

Performs a hardware reset and halt of the target CPU.

Example

`__hwReset(0)`

Resets the CPU and immediately halts it.

Applicability

This system macro is available for all JTAG interfaces.
**__hwResetRunToBp**

**Syntax**

```
__hwResetRunToBp(strategy, breakpoint_address, timeout)
```

**Parameter**

- **strategy**
  
  The reset strategy for halting the core. Choose between:
  
  0 (zero) to halt after the reset.
  
  1 to halt with a breakpoint at address 0x0.
  
  2 for software reset (for Analog devices only).

- **breakpoint_address**
  
  The address of the breakpoint to execute to, specified as an integer value (symbols cannot be used).

- **timeout**
  
  A time out for the breakpoint, specified in milliseconds. If the breakpoint is not reached within the specified time, the core will be halted.

**Return value**

<table>
<thead>
<tr>
<th>Value</th>
<th>Result</th>
</tr>
</thead>
<tbody>
<tr>
<td>&gt;=0</td>
<td>Successful. The approximate execution time in ms until the breakpoint is hit.</td>
</tr>
<tr>
<td>-2</td>
<td>Unsuccessful. Hardware reset is not supported by the emulator.</td>
</tr>
<tr>
<td>-3</td>
<td>Unsuccessful. The reset strategy is not supported by the emulator.</td>
</tr>
</tbody>
</table>

**Description**

Performs a hardware reset, sets a breakpoint at the specified address, executes to the breakpoint, and then removes it. The breakpoint address should be the start address of the downloaded image after it has been copied to RAM.

This macro is intended for running a boot loader that copies the application image from flash to RAM. The macro should be executed after the image has been downloaded to flash, but before the image is verified. The macro can be run in `execUserFlashExit` or `execUserPreload`.

**Example**

```
__hwResetRunToBp(0, 0x400000, 10000)
```

 Resets the CPU with the reset strategy 0 and executes to the address 0x400000. If the breakpoint is not reached within 10 seconds, execution stops in accordance with the specified time out.

**Applicability**

This system macro is available for the J-Link/J-Trace JTAG interfaces.
__hwResetWithStrategy

Syntax

```c
__hwResetWithStrategy(halt_delay, strategy)
```

Parameter

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>halt_delay</code></td>
<td>The delay, in milliseconds, between the end of the reset pulse and the halt of the CPU. Use 0 (zero) to make the CPU halt immediately after reset; only when <code>strategy</code> is set to 0.</td>
<td><code>&gt;=0</code></td>
</tr>
<tr>
<td><code>strategy</code></td>
<td>The reset strategy for halting the core. Choose between: 0 (zero) to halt after the reset, 1 to halt with a breakpoint at address 0x0, 2 for software reset (for Analog devices only).</td>
<td><code>-1</code></td>
</tr>
</tbody>
</table>

Return value

<table>
<thead>
<tr>
<th>Result</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>Successful. The actual delay in milliseconds, as implemented by the emulator</td>
<td><code>&gt;=0</code></td>
</tr>
<tr>
<td>Successful. The delay feature is not supported by the emulator</td>
<td><code>-1</code></td>
</tr>
<tr>
<td>Unsuccessful. Hardware reset is not supported by the emulator</td>
<td><code>-2</code></td>
</tr>
<tr>
<td>Unsuccessful. The reset strategy is not supported by the emulator</td>
<td><code>-3</code></td>
</tr>
</tbody>
</table>

Table 140: __hwResetWithStrategy return values

Description

Performs a hardware reset and a halt with delay of the target CPU.

Applicability

This system macro is available for the J-Link/J-Trace JTAG interface.

Example

```c
__hwResetWithStrategy(0, 1)
```

Resets the CPU and halts it using a breakpoint at memory address zero.

__isBatchMode

Syntax

```c
__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 141: __isBatchMode return values
Description

This macro returns True if the debugger is running in batch mode, otherwise it returns False.

__jlinkExecCommand

Syntax

__jlinkExecCommand(cmdstr)

Parameter

cmdstr J-Link/J-Trace command string

Return value

int 0

Description

Sends a low-level command to the J-Link/J-Trace driver, see the J-Link/J-Trace User's Guide.

Applicability

This system macro is available for the J-Link/J-Trace JTAG interface.

__jtagCommand

Syntax

__jtagCommand(ir)

Parameter

ir can be one of:

<table>
<thead>
<tr>
<th>Value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>2</td>
<td>SCAN_N</td>
</tr>
<tr>
<td>4</td>
<td>RESTART</td>
</tr>
<tr>
<td>12</td>
<td>INTEST</td>
</tr>
<tr>
<td>14</td>
<td>IDCODE</td>
</tr>
<tr>
<td>15</td>
<td>BYPASS</td>
</tr>
</tbody>
</table>

Return value

int 0

Description

Sends a low-level command to the JTAG instruction register IR.

Applicability

This system macro is available for the J-Link/J-Trace JTAG interface.

Example

__jtagCommand(14);
Id = __jtagData(0,32);

Returns the JTAG ID of the ARM target device.
**__jtagCP15IsPresent**

**Syntax**

```
__jtagCP15IsPresent()
```

**Return value**

1 if CP15 is available, otherwise 0.

**Description**

Checks if the coprocessor CP15 is available.

**Applicability**

This system macro is available for the J-Link/J-Trace JTAG interface.

**__jtagCP15ReadReg**

**Syntax**

```
__jtagCP15ReadReg(CRn, CRm, op1, op2)
```

**Parameter**

The parameters—registers and operands—of the **MRC** instruction. For details, see the *ARM Architecture Reference Manual*. Note that **op1** should always be 0.

**Return value**

The register value.

**Description**

Reads the value of the CP15 register and returns its value.

**Applicability**

This system macro is available for the J-Link/J-Trace JTAG interface.

**__jtagCP15WriteReg**

**Syntax**

```
__jtagCP15WriteReg(CRn, CRm, op1, op2, value)
```

**Parameter**

The parameters—registers and operands—of the **MCR** instruction. For details, see the *ARM Architecture Reference Manual*. Note that **op1** should always be 0. **value** is the value to be written.

**Description**

Writes a value to the CP15 register.

**Applicability**

This system macro is available for the J-Link/J-Trace JTAG interface.

**__jtagData**

**Syntax**

```
__jtagData(dr, bits)
```

**Parameter**

```
<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>dr</code></td>
<td>32-bit data register value</td>
</tr>
</tbody>
</table>
```

Part 7. Reference information 545
Return value

Returns the result of the operation; the number of bits in the result is given by the \textit{bits} parameter.

Description

Sends a low-level data value to the JTAG data register \textit{DR}. The bit shifted out of \textit{DR} is returned.

Example

\begin{verbatim}
__jtagCommand(14);
Id = __jtagData(0,32);
\end{verbatim}

Returns the JTAG ID of the ARM target device.

Applicability

This system macro is available for the J-Link/J-Trace JTAG interface.

\_jtagRawRead

Syntax

\begin{verbatim}
__jtagRawRead(bitpos, numbits)
\end{verbatim}

Parameter

\textit{bitpos} \hspace{1cm} The start bit position in the returned JTAG bits to return data from

\textit{numbits} \hspace{1cm} The number of bits to read. The maximum value is 32.

Description

Returns the data read from the JTAG \textit{TDO}. Only the least significant bits contain data; the last bit read is from the least significant bit. This function can be called an arbitrary number of times to get all bits returned by an operation. This function also makes an implicit synchronization of any accumulated write bits.

Example

The following piece of pseudocode illustrates how the data is written to the JTAG (on the \textit{TMS} and \textit{TDI} pins) and read (from \textit{TDO}):

\begin{verbatim}
__var Id;
__var BitPos;
спешesor sor sor sor sor sor sor sor sor sor sor sor sor sor sor sor sor sor sor sor sor sor sor sor sor sor sor sor sor sor sor sor sor sor sor sor sor sor sor sor sor sor sor sor sor sor sor sor sor sor sor sor sor sor sor sor sor sor sor sor sor sor sor sor sor sor sor sor sor sor sor sor sor sor sor sor sor sor sor sor sor sor sor sor sor sor sor sor sor sor sor sor sor sor sor sor sor sor sor sor sor sor sor sor sor sor sor sor sor sor sor sor sor sor sor sor sor sor sor sor sor sor sor sor sor sor sor sor sor sor sor sor sor sor sor sor sor sor sor sor sor sor sor sor sor sor sor sor sor sor sor sor sor sor sor sor sor sor sor sor sor sor sor sor sor sor sor sor sor sor sor sor sor sor sor sor sor sor sor sor sor sor sor sor sor sor sor sor sor sor sor sor sor sor sor sor sor sor sor sor sor sor sor sor sor sor sor sor sor sor sor sor sor sor sor sor sor sor sor sor sor sor sor sor sor sor sor sor sor sor sor sor sor sor sor sor sor sor sor sor sor sor sor sor sor sor sor sor sor sor sor sor sor sor sor sor sor sor sor sor sor sor sor sor sor sor sor sor sor sor sor sor sor sor sor sor sor sor sor sor sor sor sor sor sor sor sor sor sor sor sor sor sor sor sor sor sor sor sor sor sor sor sor sor sor sor sor sor sor sor sor sor sor sor sor sor sor sor sor sor sor sor sor sor sor sor sor sor sor sor sor sor sor sor sor sor sor sor sor sor sor sor sor sor sor sor sor sor sor sor sor sor sor sor sor sor sor sor sor sor sor sor sor sor sor sor sor sor sor sor sor sor sor sor sor sor sor sor sor sor sor sor sor sor sor sor sor sor sor sor sor sor sor sor sor sor sor sor sor sor sor sor sor sor sor sor sor sor sor sor sor sor sor sor sor sor sor sor sor sor sor sor sor sor sor sor sor sor sor sor sor sor sor sor sor sor sor sor sor sor sor sor sor sor sor sor sor sor sor sor sor sor sor sor sor sor sor sor sor sor sor sor sor sor sor sor sor sor sor sor sor sor sor sor sor sor sor sor sor sor sor sor sor sor sor sor sor sor sor sor sor sor sor sor sor sor sor sor sor sor sor sor sor sor sor sor sor sor sor sor sor sor sor sor sor sor sor sor sor sor sor sor sor sor sor sor sor sor sor sor sor sor sor so
BitPos = __jtagRawWrite(0, 0x80000000, 32); /* Shift 32 bits into DR. Remember BitPos for Read operation */
__jtagRawWrite(0, 0x1, 2); /* Goto IDLE */
Id = __jtagRawRead(BitPos, 32); /* Read the Id */
__message "JTAG Id: ", Id;%x, 
"n";
}

Applicability
This system macro is available for the J-Link/J-Trace JTAG interface.

__jtagRawSync

Syntax
__jtagRawSync()

Return value
int 0

Description
Sends arbitrary data to the JTAG interface. All accumulated bits using __jtagRawWrite will be written to the JTAG scan chain. The data is sent synchronously with TCK and typically sampled by the device on rising edge of TCK.

Example
The following piece of pseudocode illustrates how the data is written to the JTAG (on the TMS and TDI pins) and read (from TDO):

int i;
U32 tdo;
for (i = 0; i < numBits; i++) {
    TDI = tdi & 1; /* Set TDI pin */
    TMS = tms & 1; /* Set TMS pin */
    TCK = 0;
    TCK = 1;
    tdo <<= 1;
    if (TDO) {
        tdo |= 1;
    }
    tdi >>= 1;
    tms >>= 1;
}

Applicability
This system macro is available for the J-Link/J-Trace JTAG interface.
Description of C-SPY system macros

__jtagRawWrite

Syntax

```
__jtagRawWrite(tdi, tms, numbits)
```

Parameter

- `tdi` The data output to the TDI pin. This data is sent with the least significant bit first.
- `tms` The data output to the TMS pin. This data is sent with the least significant bit first.
- `numbits` The number of bits to transfer. Every bit results in a falling and rising edge of the JTAG TCK line. The maximum value is 64.

Return value

Returns the bit position of the data in the accumulated packet. Typically, this value is used when reading data from the JTAG.

Description

Accumulates bits to be transferred to the JTAG. If 32 bits are not enough, this function can be called multiple times. Both data output lines (TMS and TDI) can be controlled separately.

Example

```
/* Send five 1 bits on TMS to go to TAP-RESET state */
__jtagRawWrite(0x1F, 0, 5);  /* Store bits in buffer */
__jtagRawSync();  /* Transfer buffer, writing tms, tdi, reading tdo */
```

__jtagResetTRST

Syntax

```
__jtagResetTRST()
```

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 142: __jtagResetTRST return values

Description

Resets the ARM TAP controller via the TRST JTAG signal.

Applicability

This system macro is available for the J-Link/J-Trace JTAG interface.
__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. See Argument variables summary, page 366.

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

debugInfoOnly
A non-zero integer value if no code or data should be downloaded to the target system, which means that C-SPY will only read the debug information from the debug file. Or, 0 (zero) for download.

Return value

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

Table 143: __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
*Images*, page 495 and *Loading multiple images*, page 124.

__memoryRestore

**Syntax**

__memoryRestore(zone, filename)

**Parameters**

- **zone**  
  The memory zone name (string); for a list of available zones, see *Memory addressing*, page 149

- **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. See *Argument variables summary*, page 366.

**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 415.

__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

- **format**  
  A string that specifies the format to be used for the saved memory. Choose between:
  - intel-extended
  - motorola
  - motorola-s19
  - motorola-s28
  - motorola-s37.
__memorySave

Return value
int 0

Description
Saves the contents of a specified memory area to a file.

Example
__memorySave("Memory:0x00", "Memory:0xFF", "intel-extended", "c:\\temp\\saved_memory.hex");

See also
Memory Save dialog box, page 414.

__openFile

Syntax
__openFile(filename, access)

Parameters

filename A string that specifies the file to write to. The filename must include a path, which must either be absolute or use argument variables. See Argument variables summary, page 366.

access The access type (string).
These are mandatory but mutually exclusive:
"a" append, new data will be appended at the end of the open file
"r" read
"w" write
These are optional and mutually exclusive:
"b" binary, opens the file in binary mode
"t" ASCII text, opens the file in text mode
This access type is optional:
"+" together with r, w, or a; r+ or w+ is read and write, while a+ is read and append

Return value

<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 144: __openFile return values

Description
Opens a file for I/O operations. The default base directory of this macro is where the currently open project file (*.ewp) is located. The argument to __openFile can
specify a location relative to this directory. In addition, you can use argument variables such as $PROJ_DIR$ and $TOOLKIT_DIR$ in the path argument.

Example

```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

`Argument variables summary`, page 366.

**__orderInterrupt**

Syntax

```c
__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( "IRQ", 4000, 2000, 0, 1, 0, 100 );
```

__popSimulatorInterruptExecutingStack

**Syntax**

```c
__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.

__readFile

**Syntax**

```c
__readFile(fileHandle, valuePtr)
```

**Parameters**

- `fileHandle`: A macro variable used as filehandle by the __openFile macro
- `valuePtr`: A pointer to a variable

**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 145: __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)
{
```
**__readFileByte**

**Syntax**

```c
__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**

```c
__var byte;
while ( (byte = __readFileByte(myFileHandle)) != -1 )
{
    /* Do something with byte */
}
```

**__readMemory8, __readMemoryByte**

**Syntax**

```c
__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 Memory addressing, page 149

**Return value**

The macro returns the value from memory.

**Description**

Reads one byte from a given memory location.

**Example**

```c
__readMemory8(0x0108, "Memory");
```
___readMemory16

Syntax            ___readMemory16(address, zone)

Parameters  

address  The memory address (integer)

zone  The memory zone name (string); for a list of available zones, see Memory addressing, page 149

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 Memory addressing, page 149

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. See Argument variables summary, page 366.

Return value  int 0
Description of C-SPY system macros

__registerMacroFile

Description
Registers macros from a setup macro file. With this function you can register multiple macro files during C-SPY startup.

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

See also
Registering and executing using setup macros and setup files, page 159.

__resetFile

Syntax
__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.

__restoreSoftwareBreakpoint

Syntax
__restoreSoftwareBreakpoint()

Return value
int 0

Description
Restores automatically any breakpoints that were destroyed during system startup.

This can be useful if you have an application that is copied to RAM during startup and is then executing in RAM. This can, for example, be the case if you use the initialize by copy directive for code in the linker configuration file or if you have any __ramfunc declared functions in your application. In this case, any breakpoints will be destroyed during the RAM copying when the C-SPY debugger starts.

By using the this macro, C-SPY will restore the destroyed breakpoints.

Applicability
This system macro is available for the J-Link/J-Trace JTAG interface and the Macraigor interface.
__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 146: __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

Defining breakpoints, page 141.
Description of C-SPY system macros

__setDataBreak

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

Parameters

<table>
<thead>
<tr>
<th>Location</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>location</td>
<td>A string with a location description. This can be either:</td>
</tr>
<tr>
<td></td>
<td>A source location on the form <code>{filename}.line.col</code> (for example <code>{D:\src\prog.c}.12.9</code>), although this is not very useful for data breakpoints</td>
</tr>
<tr>
<td></td>
<td>An absolute location on the form <code>zone:hexaddress</code> or simply <code>hexaddress</code> (for example <code>Memory:0x42</code>)</td>
</tr>
<tr>
<td></td>
<td>An expression whose value designates a location (for example <code>my_global_variable</code>)</td>
</tr>
<tr>
<td>Count</td>
<td>The number of times that a breakpoint condition must be fulfilled before a break occurs (integer)</td>
</tr>
<tr>
<td>Condition</td>
<td>The breakpoint condition (string)</td>
</tr>
<tr>
<td>Cond_type</td>
<td>The condition type; either “CHANGED” or “TRUE” (string)</td>
</tr>
<tr>
<td>Access</td>
<td>The memory access type: “R” for read, “W” for write, or “RW” for read/write</td>
</tr>
<tr>
<td>Action</td>
<td>An expression, typically a call to a macro, which is evaluated when the breakpoint is detected</td>
</tr>
</tbody>
</table>

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 147: __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
__var brk;
brk = __setDataBreak("Memory:0x4710", 3, "d>6", "TRUE", "W", "ActionData()" );
__clearBreak(brk);

See also
Defining breakpoints, page 141.

__setLogBreak

Syntax
__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.

condition
The breakpoint condition (string)

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

Return value

Result Value
Successful An unsigned integer uniquely identifying the breakpoint. The same value must be used when you want to clear the breakpoint.
Unsuccessful 0

Table 148: __setLogBreak return values

Description
Sets a log breakpoint, that is, a breakpoint which is triggered when an instruction is fetched from the specified location. If you have set the breakpoint on a specific machine instruction, the breakpoint will be triggered and the execution will temporarily halt and print the specified message in the C-SPY Debug Log window.

Example
__var logBp1;
__var logBp2;
logOn()
Description of C-SPY system macros

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

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

See also  
Formatted output, page 530, Log breakpoints dialog box, page 343, and  
Defining breakpoints, page 141.

__setSimBreak

Syntax  
__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 149: __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**

```
__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 150: __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**

```
__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 Defining breakpoints, page 141.

__setTraceStopBreak

Syntax

__setTraceStopBreak(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</td>
</tr>
<tr>
<td></td>
<td>value must be used when you want to clear the breakpoint.</td>
</tr>
<tr>
<td>Unsuccessful</td>
<td>int 0</td>
</tr>
</tbody>
</table>

Table 151: __setTraceStopBreak return values

Description

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

Applicability

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

Example

__var brk;
brk = __setDataBreak("Memory:0x4710", 3, "d>6", "TRUE",
    "W", "ActionData()");
...
__clearBreak(brk);
See also  Defining breakpoints, page 141.

__sourcePosition

Syntax

__sourcePosition(linePtr, colPtr)

Parameters

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>linePtr</td>
<td>Pointer to the variable storing the line number</td>
</tr>
<tr>
<td>colPtr</td>
<td>Pointer to the variable storing the column number</td>
</tr>
</tbody>
</table>

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 152: __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

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>macroString</td>
<td>The macro string to search in</td>
</tr>
<tr>
<td>pattern</td>
<td>The string pattern to search for</td>
</tr>
<tr>
<td>position</td>
<td>The position where to start the search. The first position is 0</td>
</tr>
</tbody>
</table>

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 528.
**__subString**

**Syntax**

```macro__subString(macroString, position, length)```

**Parameters**

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>macroString</td>
<td>The macro string from which to extract a substring.</td>
</tr>
<tr>
<td>position</td>
<td>The start position of the substring. The first position is 0.</td>
</tr>
<tr>
<td>length</td>
<td>The length of the substring.</td>
</tr>
</tbody>
</table>

**Return value**

A substring extracted from the given macro string.

**Description**

This macro extracts a substring from another string.

**Example**

```macro__subString("Compiler", 0, 2)
  The resulting macro string contains Co.
__subString("Compiler", 3, 4)
  The resulting macro string contains pille.
```

**See also**

Macro strings, page 528.

**__targetDebuggerVersion**

**Syntax**

```macro__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**

```macro__var toolVer;
  toolVer = __targetDebuggerVersion();
  __message "The target debugger version is, ", toolVer;
```

**__toLower**

**Syntax**

```macro__toLower(macroString)```

**Parameter**

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>macroString</td>
<td>is any macro string.</td>
</tr>
</tbody>
</table>

**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 528.

__toString

Syntax
__toString(C_string, maxlength)

Parameter

<table>
<thead>
<tr>
<th>string</th>
<th>Any null-terminated C string</th>
</tr>
</thead>
<tbody>
<tr>
<td>maxLength</td>
<td>The maximum length of the returned macro string</td>
</tr>
</tbody>
</table>

Return value
Macro string.

Description
This macro is used for converting C strings (char* or char[]) into macro strings.

Example
Assuming your application contains this definition:
char const * hptr = "Hello World!";
this macro call:
__toString(hptr, 5)
would return the macro string containing Hello.

See also
Macro strings, page 528.

__toUpper

Syntax
__toUpper(macroString)

Parameter
macroString is any macro string.

Return value
The converted string.
Description of C-SPY system macros

Description
This macro returns a copy of the parameter macroString where all the characters have been converted to upper case.

Example
__toUpper('string')
The resulting macro string contains STRING.

See also
Macro strings, page 528.

__unloadImage
Syntax
__unloadImage(module_id)

Parameter
module_id
An integer which represents a unique module identification, which is retrieved as a return value from the corresponding __loadImage C-SPY macro.

Return value
Value             Result
module_id         A unique module identification (the same as the input parameter).
int 0             The unloading failed.

Table 153: __unloadImage return values

Description
Unloads debug information from an already downloaded image.

See also

__writeFile
Syntax
__writeFile(file, value)

Parameters
fileHandle       A macro variable used as filehandle by the __openFile macro
value            An integer

Return value
int 0

Description
Prints the integer value in hexadecimal format (with a trailing space) to the file file.
Note: The __fmessage statement can do the same thing. The __writeFile macro is provided for symmetry with __readFile.

__writeFileByte

Syntax
__writeFileByte(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 Memory addressing, page 149

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)
Description of C-SPY system macros

__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 Memory addressing, page 149

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 Memory addressing, page 149

Return value int 0

Description Writes four bytes to a given memory location.

Example

__writeMemory32(0x5555FFFF, 0x8020, "Memory");
Glossary

This is a general glossary for terms relevant to embedded systems programming. Some of the terms do not apply to the IAR Embedded Workbench® version that you are using.

A

Absolute location
A specific memory address for an object specified in the source code, as opposed to the object being assigned a location by the linker.

Address expression
An expression which has an address as its value.

AEABI
Embedded Application Binary Interface for ARM, defined by ARM Limited.

Application
The program developed by the user of the IAR Systems toolkit and which will be run as an embedded application on a target processor.

Ar
The GNU binary utility for creating, modifying, and extracting from archives, that is, libraries. See also larchive.

Architecture
A term used by computer designers to designate the structure of complex information-processing systems. It includes the kinds of instructions and data used, the memory organization and addressing, and the methods by which the system is implemented. The two main architecture types used in processor design are Harvard architecture and von Neumann architecture.

Archive
See Library.

Assembler directives
The set of commands that control how the assembler operates.

Assembler language
A machine-specific set of mnemonics used to specify operations to the target processor and input or output registers or data areas. Assembler language might sometimes be preferred over C/C++ to save memory or to enhance the execution speed of the application.

Assembler options
Parameters you can specify to change the default behavior of the assembler.

Attributes
See Section attributes.

Auto variables
The term refers to the fact that each time the function in which the variable is declared is called, a new instance of the variable is created automatically. This can be compared with the behavior of local variables in systems using static overlay, where a local variable only exists in one instance, even if the function is called recursively. Also called local variables. Compare Register variables.

B

Backtrace
Information for keeping call frame information up to date so that the IAR C-SPY® Debugger can return from a function correctly. See also Call frame information.

Bank
See Memory bank.

Bank switching
Switching between different sets of memory banks. This software technique increases a computer's usable memory by allowing different pieces of memory to occupy the same address space.

Banked code
Code that is distributed over several banks of memory. Each function must reside in only one bank.
Banked data
Data that is distributed over several banks of memory. Each data object must fit inside one memory bank.

Banked memory
Has multiple storage locations for the same address. See also Memory bank.

Bank-switching routines
Code that selects a memory bank.

Batch files
A text file containing operating system commands which are executed by the command line interpreter. In Unix, this is called a “shell script” because it is the Unix shell which includes the command line interpreter. Batch files can be used as a simple way to combine existing commands into new commands.

Bitfield
A group of bits considered as a unit.

Block, in linker configuration file
A continuous piece of code or data. It is either built up of blocks, overlays, and sections or it is empty. A block has a name, and the start and end address of the block can be referred to from the application. It can have attributes such as a maximum size, a specific size, or a minimum alignment. The contents can have a specific order or not.

Breakpoint
1. Code breakpoint. A point in a program that, when reached, triggers some special behavior useful to the process of debugging. Generally, breakpoints are used for stopping program execution or dumping the values of some or all of the program variables. Breakpoints can be part of the program itself, or they can be set by the programmer as part of an interactive session with a debugging tool for scrutinizing the program's execution.

2. Data breakpoint. A point in memory that, when accessed, triggers some special behavior useful to the process of debugging. Generally, data breakpoints are used to stop program execution when an address location is accessed either by a read operation or a write operation.

3. Immediate breakpoint. A point in memory that, when accessed, trigger some special behavior useful in the process of debugging. Immediate breakpoints are generally used for halting the program execution in the middle of a memory access instruction (before or after the actual memory access depending on the access type) while performing some user-specified action. The execution is then resumed. This feature is only available in the simulator version of C-SPY.

C

Call frame information
Information that allows the IAR C-SPY® Debugger to show, without any runtime penalty, the complete stack of function calls—call stack—wherever the program counter is, provided that the code comes from compiled C functions. See also Backtrace.

Calling convention
A calling convention describes the way one function in a program calls another function. This includes how register parameters are handled, how the return value is returned, and which registers that will be preserved by the called function. The compiler handles this automatically for all C and C++ functions. All code written in assembler language must conform to the rules in the calling convention to be callable from C or C++, or to be able to call C and C++ functions. The C calling convention and the C++ calling conventions are not necessarily the same.

Cheap
As in cheap memory access. A cheap memory access either requires few cycles to perform, or few bytes of code to implement. A cheap memory access is said to have a low cost. See Memory access cost.

Checksum
A computed value which depends on the ROM content of the whole or parts of the application, and which is stored along with the application to detect corruption of the data. The checksum is produced by the linker to be verified with the application. Several algorithms are supported. Compare CRC (cyclic redundancy checking).
**Code banking**  
See Banked code.

**Code model**  
The code model controls how code is generated for an application. Typically, the code model controls behavior such as how functions are called and in which code sections functions will be located. All object files of an application must be compiled using the same code model.

**Code pointers**  
A code pointer is a function pointer. As many cores allow several different methods of calling a function, compilers for embedded systems usually provide the users with the ability to use all these methods.

Do not confuse code pointers with data pointers.

**Code sections**  
Read-only sections that contain code. See also Section.

**Compilation unit**  
See Translation unit.

**Compiler options**  
Parameters you can specify to change the default behavior of the compiler.

**Cost**  
See Memory access cost.

**CRC (cyclic redundancy checking)**  
A number derived from, and stored with, a block of data to detect corruption. A CRC is based on polynomials and is a more advanced way of detecting errors than a simple arithmetic checksum. Compare Checksum.

**C-SPY options**  
Parameters you can specify to change the default behavior of the IAR C-SPY Debugger.

**Cstartup**  
Code that sets up the system before the application starts executing.

**C-style preprocessor**  
A preprocessor is either a stand-alone application or an integrated part of a compiler, that performs preprocessing of the input stream before the actual compilation occurs. A C-style preprocessor follows the rules set up in Standard C and implements commands like `#define`, `#if`, and `#include`, which are used to handle textual macro substitution, conditional compilation, and inclusion of other files.

**Data banking**  
See Banked data.

**Data model**  
The data model specifies the default memory type. This means that the data model typically controls one or more of the following: The method used and the code generated to access static and global variables, dynamically allocated data, and the runtime stack. It also controls the default pointer type and in which data sections static and global variables will be located. A project can only use one data model at a time, and the same model must be used by all user modules and all library modules in the project.

**Data pointers**  
Many cores have different addressing modes to access different memory types or address spaces. Compilers for embedded systems usually have a set of different data pointer types so they can access the available memory efficiently.

**Data representation**  
How different data types are laid out in memory and what value ranges they represent.

**Declaration**  
A specification to the compiler that an object, a variable or function, exists. The object itself must be defined in exactly one translation unit (source file). An object must either be declared or defined before it is used. Normally an object that is used in many files is defined in one source file. A declaration is normally placed in a header file that is included by the files that use the object.
For example:

```c
/* Variable "a" exists somewhere. Function 
"b" takes two int parameters and returns an 
int. */

extern int a;
int b(int, int);
```

**Definition**
The variable or function itself. Only one definition can exist 
for each variable or function in an application. See also 
Tentative definition.

For example:

```c
int a;
int b(int x, int y)
{
    return x + y;
}
```

**Demangling**
To restore a mangled name to the more common C/C++ name. See also **Mangling**.

**Device description file**
A file used by C-SPY that contains various device-specific 
information such as I/O registers (SFR) definitions, interrupt 
vectors, and control register definitions.

**Device driver**
Software that provides a high-level programming interface to 
a particular peripheral device.

**Digital signal processor (DSP)**
A device that is similar to a microprocessor, except that the 
internal CPU is optimized for use in applications involving 
discrete-time signal processing. In addition to standard 
microprocessor instructions, digital signal processors usually 
support a set of complex instructions to perform common 
signal-processing computations quickly.

**Disassembly window**
A C-SPY window that shows the memory contents 
disassembled as machine instructions, interspersed with the 
corresponding C source code (if available).

**DWARF**
An industry-standard debugging format which supports source 
level debugging. This is the format used by the IAR ILINK 
Linker for representing debug information in an object.

**Dynamic initialization**
Variables in a program written in C are initialized during the 
initial phase of execution, before the main function is called. 
These variables are always initialized with a static value, 
which is determined either at compile time or at link time. This 
is called static initialization. In C++, variables might require 
initialization to be performed by executing code, for example, 
runtime the constructor of global objects, or performing 
dynamic memory allocation.

**Dynamic memory allocation**
There are two main strategies for storing variables: statically at 
link time, or dynamically at runtime. Dynamic memory 
allocation is often performed from the heap and it is the size of 
the heap that determines how much memory that can be used 
for dynamic objects and variables. The advantage of dynamic 
memory allocation is that several variables or objects that are 
not active at the same time can be stored in the same memory, 
thus reducing the memory requirements of an application. See 
also **Heap memory**.

**Dynamic object**
An object that is allocated, created, destroyed, and released at 
runtime. Dynamic objects are almost always stored in memory 
that is dynamically allocated. Compare **Static object**.

**EEPROM**
Electrically Erasable, Programmable Read-Only Memory. A 
type of ROM that can be erased electronically, and then be 
re-programmed.
**Glossary**

**ELF**
Executable and Linking Format, an industry-standard object file format. This is the format used by the IAR ILINK Linker. The debug information is formatted using DWARF.

**Embedded C++**
A subset of the C++ programming language, which is intended for embedded systems programming. The fact that performance and portability are particularly important in embedded systems development was considered when defining the language.

**Embedded system**
A combination of hardware and software, designed for a specific purpose. Embedded systems are often part of a larger system or product.

**Emulator**
An emulator is a hardware device that performs emulation of one or more derivatives of a processor family. An emulator can often be used instead of the actual core and connects directly to the printed circuit board—where the core would have been connected—via a connecting device. An emulator always behaves exactly as the processor it emulates, and is used when debugging requires all systems actuators, or when debugging device drivers.

**Enea OSE Load module format**
A specific ELF format that is loadable by the OSE operating system. See also **ELF**.

**Enumeration**
A type which includes in its definition an exhaustive list of possible values for variables of that type. Common examples include Boolean, which takes values from the list [true, false], and day-of-week which takes values [Sunday, Monday, Tuesday, Wednesday, Thursday, Friday, Saturday]. Enumerated types are a feature of typed languages, including C and Ada.

Characters, (fixed-size) integers, and even floating-point types might be (but are not usually) considered to be (large) enumerated types.

**EPROM**
Erasable, Programmable Read-Only Memory. A type of ROM that can be erased by exposing it to ultraviolet light, and then be re-programmed.

**Executable image**
Contains the executable; the result of linking several relocatable object files and libraries. The file format used for an object file is ELF with embedded DWARF for debug information.

**Exceptions**
An exception is an interrupt initiated by the processor hardware, or hardware that is tightly coupled with the processor, for instance, a memory management unit (MMU). The exception signals a violation of the rules of the architecture (access to protected memory), or an extreme error condition (division by zero).

Do not confuse this use of the word exception with the term **exception** used in the C++ language (but not in Embedded C++).

**Expensive**
As in expense memory access. An expensive memory access either requires many cycles to perform, or many bytes of code to implement. An expensive memory access is said to have a high cost. See **Memory access cost**.

**Extended keywords**
Non-standard keywords in C and C++. These usually control the definition and declaration of objects (that is, data and functions). See also **Keywords**.

**F**

**Filling**
How to fill up bytes—with a specific fill pattern—that exists between the sections in an executable image. These bytes exist because of the alignment demands on the sections.
**Format specifiers**

Used to specify the format of strings sent by library functions such as `printf`. In the following example, the function call contains one format string with one format specifier, `%c`, that prints the value of `a` as a single ASCII character:

```c
printf("a = %c", a);
```

**General options**

Parameters you can specify to change the default behavior of all tools that are included in the IDE.

**Generic pointers**

Pointers that have the ability to point to all different memory types in, for example, a core based on the Harvard architecture.

**Harvard architecture**

A core based on the Harvard architecture has separate data and instruction buses. This allows execution to occur in parallel. As an instruction is being fetched, the current instruction is executing on the data bus. Once the current instruction is complete, the next instruction is ready to go. This theoretically allows for much faster execution than a von Neumann architecture, but adds some silicon complexity. Compare von Neumann architecture.

**Heap memory**

The heap is a pool of memory in a system that is reserved for dynamic memory allocation. An application can request parts of the heap for its own use; once memory is allocated from the heap it remains valid until it is explicitly released back to the heap by the application. This type of memory is useful when the number of objects is not known until the application executes. Note that this type of memory is risky to use in systems with a limited amount of memory or systems that are expected to run for a very long time.

**Heap size**

Total size of memory that can be dynamically allocated.

**Host**

The computer that communicates with the target processor. The term is used to distinguish the computer on which the debugger is running from the core the embedded application you develop runs on.

**Iarchive**

The IAR Systems utility for creating archives, that is, libraries. Iarchive is delivered with IAR Embedded Workbench.

**IDE (integrated development environment)**

A programming environment with all necessary tools integrated into one single application.

**Ielfdumparm**

The IAR Systems utility for creating a text representation of the contents of ELF relocatable or executable image.

**Ielftool**

The IAR Systems utility for performing various transformations on an ELF executable image, such as fill, checksum, and format conversion.

**ILINK**

The IAR ILINK Linker which produces absolute output in the ELF/DWARF format.

**ILINK configuration**

The definition of available physical memories and the placement of sections—pieces of code and data—into those memories. ILINK requires a configuration to build an executable image.

**Image**

See Executable image.

**Include file**

A text file which is included into a source file. This is often done by the preprocessor.
Initialization setup in linker configuration file
Defines how to initialize RAM sections with their initializers. Normally, only non-constant non-noinit variables are initialized but, for example, pieces of code can be initialized as well.

Initialized sections
Read-write sections that should be initialized with specific values at startup. See also Section.

Inline assembler
Assembler language code that is inserted directly between C statements.

Inlining
An optimization that replaces function calls with the body of the called function. This optimization increases the execution speed and can even reduce the size of the generated code.

Instruction mnemonics
A word or acronym used in assembler language to represent a machine instruction. Different processors have different instruction sets and therefore use a different set of mnemonics to represent them, such as, ADD, BR (branch), BLT (branch if less than), MOVE, LDR (load register).

Interrupt vector
A small piece of code that will be executed, or a pointer that points to code that will be executed when an interrupt occurs.

Interrupt vector table
A table containing interrupt vectors, indexed by interrupt type. This table contains the processor's mapping between interrupts and interrupt service routines and must be initialized by the programmer.

Interruptions
In embedded systems, the use of interrupts is a method of detecting external events immediately, for example a timer overflow or the pressing of a button.

Interrupts are asynchronous events that suspend normal processing and temporarily divert the flow of control through an “interrupt handler” routine. Interrupts can be caused by both hardware (I/O, timer, machine check) and software (supervisor, system call or trap instruction). Compare Trap.

Intrinsic
An adjective describing native compiler objects, properties, events, and methods.

Intrinsic functions
1. Function calls that are directly expanded into specific sequences of machine code. 2. Functions called by the compiler for internal purposes (that is, floating-point arithmetic etc.).

Iobjmanip
The IAR Systems utility for performing low-level manipulation of ELF object files.

K

Key bindings
Key shortcuts for menu commands used in the IDE.

Keywords
A fixed set of symbols built into the syntax of a programming language. All keywords used in a language are reserved—they cannot be used as identifiers (in other words, user-defined objects such as variables or procedures). See also Extended keywords.

L

L-value
A value that can be found on the left side of an assignment and thus be changed. This includes plain variables and de-referenced pointers. Expressions like \((x + 10)\) cannot be assigned a new value and are therefore not L-values.

Language extensions
Target-specific extensions to the C language.
Library
See Runtime library.

Library configuration file
A file that contains a configuration of the runtime library. The file contains information about what functionality is part of the runtime environment. The file is used for tailoring a build of a runtime library. See also Runtime library.

Linker configuration file
A file that contains a configuration used by the IAR ILINK Linker when building an executable image. See also ILINK configuration.

Local variable
See Auto variables.

Location counter
See Program location counter (PLC).

Logical address
See Virtual address (logical address).

MAC (Multiply and accumulate)
A special instruction, or on-chip device, that performs a multiplication together with an addition. This is very useful when performing signal processing where many filters and transforms have the form:

\[ y_j = \sum_{i=0}^{N} c_i \cdot x_{i+j} \]

The accumulator of the MAC usually has a higher precision (more bits) than normal registers. See also Digital signal processor (DSP).

Macro
1. Assembler macros are user-defined sets of assembler lines that can be expanded later in the source file by referring to the given macro name. Parameters will be substituted if referred to.

2. C macro. A text substitution mechanism used during preprocessing of source files. Macros are defined using the #define preprocessing directive. The replacement text of each macro is then substituted for any occurrences of the macro name in the rest of the translation unit.

3. C-SPY macros are programs that you can write to enhance the functionality of C-SPY. A typical application of C-SPY macros is to associate them with breakpoints; when such a breakpoint is hit, the macro is run and can for example be used to simulate peripheral devices, to evaluate complex conditions, or to output a trace.

The C-SPY macro language is like a simple dialect of C, but is less strict with types.

Mailbox
A mailbox in an RTOS is a point of communication between two or more tasks. One task can send messages to another task by placing the message in the mailbox of the other task. Mailboxes are also known as message queues or message ports.

Mangling
Mangling is a technique used for mapping a complex C/C++ name into a simple name. Both mangled and unmangled names can be produced for C/C++ symbols in ILINK messages.

Memory, in linker configuration file
A physical memory. The number of units it contains and how many bits a unit consists of, are defined in the linker configuration file. The memory is always addressable from 0x0 to size -1.

Memory access cost
The cost of a memory access can be in clock cycles, or in the number of bytes of code needed to perform the access. A memory which requires large instructions or many instructions is said to have a higher access cost than a memory which can be accessed with few, or small instructions.

Memory area
A region of the memory.
Memory bank
The smallest unit of continuous memory in banked memory. One memory bank at a time is visible in a core’s physical address space.

Memory map
A map of the different memory areas available to the core.

Memory model
Specifies the memory hierarchy and how much memory the system can handle. Your application must use only one memory model at a time, and the same model must be used by all user modules and all library modules.

Microcontroller
A microprocessor on a single integrated circuit intended to operate as an embedded system. In addition to a CPU, a microcontroller typically includes small amounts of RAM, PROM, timers, and I/O ports.

Microprocessor
A CPU contained on one (or a few) integrated circuits. A single-chip microprocessor can include other components such as memory, memory management, caches, floating-point unit, I/O ports and timers. Such devices are also known as microcontrollers.

Module
An object. An object file contains a module and library contains one or more objects. The basic unit of linking. A module contains definitions for symbols (exports) and references to external symbols (imports). When you compile C/C++, each translation unit produces one module.

Multi-file compilation
A technique which means that the compiler compiles several source files as one compilation unit, which enables for interprocedural optimizations such as inlining, cross call, and cross jump on multiple source files in a compilation unit.

Nested interrupts
A system where an interrupt can be interrupted by another interrupt is said to have nested interrupts.

Non-banked memory
Has a single storage location for each memory address in a core’s physical address space.

Non-initialized memory
Memory that can contain any value at reset, or in the case of a soft reset, can remember the value it had before the reset.

No-init sections
Read-write sections that should not be initialized at startup. See also Section.

Non-volatile storage
Memory devices such as battery-backed RAM, ROM, magnetic tape and magnetic disks that can retain data when electric power is shut off. Compare Volatile storage.

NOP
No operation. This is an instruction that does not do anything, but is used to create a delay. In pipelined architectures, the NOP instruction can be used for synchronizing the pipeline. See also Pipeline.

Objcopy
A GNU binary utility for converting an absolute object file in ELF format into an absolute object file, for example the format Motorola-std or Intel-std. See also Ielftool.

Object
An object file or a library member.

Object file, absolute
See Executable image.
**Object file, relocatable**
The result of compiling or assembling a source file. The file format used for an object file is ELF with embedded DWARF for debug information.

**Operator**
A symbol used as a function, with infix syntax if it has two arguments (+, for example) or prefix syntax if it has only one (for instance, bitwise negation, –). Many languages use operators for built-in functions such as arithmetic and logic.

**Operator precedence**
Each operator has a precedence number assigned to it that determines the order in which the operator and its operands are evaluated. The highest precedence operators are evaluated first. Use parentheses to group operators and operands to control the order in which the expressions are evaluated.

**Options**
A set of commands that control the behavior of a tool, for example the compiler or linker. The options can be specified on the command line or via the IDE.

**Output image**
The resulting application after linking. This term is equivalent to executable image, which is the term used in the IAR Systems user documentation.

**Overlay, in linker configuration file**
Like a block, but it contains several overlaid entities, each built up of blocks, overlays, and sections. The size of an overlay is determined by its largest constituent.

**Parameter passing**
See Calling convention.

**Peripheral unit**
A hardware component other than the processor, for example memory or an I/O device.

**Pipeline**
A structure that consists of a sequence of stages through which a computation flows. New operations can be initiated at the start of the pipeline even though other operations are already in progress through the pipeline.

**Placement, in linker configuration file**
How to place blocks, overlays, and sections into a region. It determines how pieces of code and data are actually placed in the available physical memory.

**Pointer**
An object that contains an address to another object of a specified type.

**#pragma**
During compilation of a C/C++ program, the #pragma preprocessing directive causes the compiler to behave in an implementation-defined manner. This can include, for example, producing output on the console, changing the declaration of a subsequent object, changing the optimization level, or enabling/disabling language extensions.

**Pre-emptive multitasking**
An RTOS task is allowed to run until a higher priority process is activated. The higher priority task might become active as the result of an interrupt. The term preemptive indicates that although a task is allotted to run a given length of time (a timeslice), it might lose the processor at any time. Each time an interrupt occurs, the task scheduler looks for the highest priority task that is active and switches to that task. If the located task is different from the task that was executing before the interrupt, the previous task is suspended at the point of interruption.

Compare Round Robin.

**Preprocessing directives**
A set of directives that are executed before the parsing of the actual code is started.

**Preprocessor**
See C-style preprocessor.
Processor variant
The different chip setups that the compiler supports.

Program counter (PC)
A special processor register that is used to address instructions. Compare Program location counter (PLC).

Program location counter (PLC)
Used in the IAR Assembler to denote the code address of the current instruction. The PLC is represented by a special symbol (typically $) that can be used in arithmetic expressions. Also called simply location counter (LC).

Project
The user application development project.

Project options
General options that apply to an entire project, for example the target processor that the application will run on.

PROM
Programmable Read-Only Memory. A type of ROM that can be programmed only once.

Q
Qualifiers
See Type qualifiers.

R
Range, in linker configuration file
A range of consecutive addresses in a memory. A region is built up of ranges.

Read-only sections
Sections that contain code or constants. See also Section.

Real-time operating system (RTOS)
An operating system which guarantees the latency between an interrupt being triggered and the interrupt handler starting, and how tasks are scheduled. An RTOS is typically much smaller than a normal desktop operating system. Compare Real-time system.

Real-time system
A computer system whose processes are time-sensitive. Compare Real-time operating system (RTOS).

Region, in linker configuration file
A set of non-overlapping ranges. The ranges can lie in one or more memories. For ILINK, blocks, overlays, and sections are placed into regions in the linker configuration file.

Region expression, in linker configuration file
A region built up from region literals, regions, and the common set operations possible in the linker configuration file.

Region literal, in linker configuration file
A literal that defines a set of one or more non-overlapping ranges in a memory.

Register
A small on-chip memory unit, usually just one or a few bytes in size, which is particularly efficient to access and therefore often reserved as a temporary storage area during program execution.

Register constant
A register constant is a value that is loaded into a dedicated processor register when the system is initialized. The compiler can then generate code that assumes that the constants are present in the dedicated registers.

Register locking
Register locking means that the compiler can be instructed that some processor registers shall not be used during normal code generation. This is useful in many situations. For example, some parts of a system might be written in assembler language to gain speed. These parts might be given dedicated processor registers. Or the register might be used by an operating system, or by other third-party software.

Register variables
Typically, register variables are local variables that are placed in registers instead of on the (stack) frame of the function. Register variables are much more efficient than other variables because they do not require memory accesses, so the compiler can use shorter/faster instructions when working with them. See also Auto variables.
Relay
A synonym to veneer, see Veneer.

Relocatable sections
Sections that have no fixed location in memory before linking.

Reset
A reset is a restart from the initial state of a system. A reset can originate from hardware (hard reset), or from software (soft reset). A hard reset can usually not be distinguished from the power-on condition, which a soft reset can be.

ROM-monitor
A piece of embedded software designed specifically for use as a debugging tool. It resides in the ROM of the evaluation board chip and communicates with a debugger via a serial port or network connection. The ROM-monitor provides a set of primitive commands to view and modify memory locations and registers, create and remove breakpoints, and execute your application. The debugger combines these primitives to fulfill higher-level requests like program download and single-step.

Round Robin
Task scheduling in an operating system, where all tasks have the same priority level and are executed in turn, one after the other. Compare Pre-emptive multitasking.

RTOS
See Real-time operating system (RTOS).

Runtime library
A collection of relocatable object files that will be included in the executable image only if referred to from an object file, in other words conditionally linked.

Runtime model attributes
A mechanism that is designed to prevent modules that are not compatible to be linked into an application. A runtime attribute is a pair constituted of a named key and its corresponding value.

ILINK uses the runtime model attributes when automatically choosing a library, to verify that the correct one is used.

R-value
A value that can be found on the right side of an assignment. This is just a plain value. See also L-value.

Saturation arithmetics
Most, if not all, C and C++ implementations use %2N 2-complement-based arithmetics where an overflow wraps the value in the value domain, that is, \((127 + 1) = -128\). Saturation arithmetics, on the other hand, does not allow wrapping in the value domain, for instance, \((127 + 1) = 127\), if 127 is the upper limit. Saturation arithmetics is often used in signal processing, where an overflow condition would have been fatal if value wrapping had been allowed.

Scheduler
The part of an RTOS that performs task-switching. It is also responsible for selecting which task that should be allowed to run. Many scheduling algorithms exist, but most of them are either based on static scheduling (performed at compile-time), or on dynamic scheduling (where the actual choice of which task to run next is taken at runtime, depending on the state of the system at the time of the task-switch). Most real-time systems use static scheduling, because it makes it possible to prove that the system will not violate the real-time requirements.

Scope
The section of an application where a function or a variable can be referenced by name. The scope of an item can be limited to file, function, or block.

Section
An entity that either contains data or text. Typically, one or more variables, or functions. A section is the smallest linkable unit.

Section attributes
Each section has a name and an attribute. The attribute defines what a section contains, that is, if the section content is read-only, read/write, code, data, etc.
Section fragment
A part of a section, typically a variable or a function.

Section selection
In the linker configuration file, defining a set of sections by using section selectors. A section belongs to the most restrictive section selector if it can be part of more than one selection. Three different selectors can be used individually or in conjunction to select the set of sections: section attribute (selecting by the section content), section name (selecting by the section name), and object name (selecting from a specific object).

Semaphore
A semaphore is a type of flag that is used for guaranteeing exclusive access to resources. The resource can be a hardware port, a configuration memory, or a set of variables. If several tasks must access the same resource, the parts of the code (the critical sections) that access the resource must be made exclusive for every task. This is done by obtaining the semaphore that protects that resource, thus blocking all other tasks from it. If another task wishes to use the resource, it also must obtain the semaphore. If the semaphore is already in use, the second task must wait until the semaphore is released. After the semaphore is released, the second task is allowed to execute and can obtain the semaphore for its own exclusive access.

Severity level
The level of seriousness of the diagnostic response from the assembler, compiler, or debugger, when it notices that something is wrong. Typical severity levels are remarks, warnings, errors, and fatal errors. A remark just points to a possible problem, while a fatal error means that the programming tool exits without finishing.

Sharing
A physical memory that can be addressed in several ways. For ILINK, defined in the linker configuration file.

Short addressing
Many cores have special addressing modes for efficient access to internal RAM and memory mapped I/O. Short addressing is therefore provided as an extended feature by many compilers for embedded systems. See also Data pointers.

Side effect
An expression in C or C++ is said to have a side-effect if it changes the state of the system. Examples are assignments to a variable, or using a variable with the post-increment operator. The C and C++ standards state that a variable that is subject to a side-effect should not be used more than once in an expression. As an example, this statement violates that rule:

*d++ = *d;

Signal
Signals provide event-based communication between tasks. A task can wait for one or more signals from other tasks. Once a task receives a signal it waits for, execution continues. A task in an RTOS that waits for a signal does not use any processing time, which allows other tasks to execute.

Simulator
A debugging tool that runs on the host and behaves as similar to the target processor as possible. A simulator is used to debug the application when the hardware is unavailable, or not needed for proper debugging. A simulator is usually not connected to any physical peripheral devices. A simulated processor is often slower, or even much slower, than the real hardware.

Single stepping
Executing one instruction or one C statement at a time in the debugger.

Skeleton code
An incomplete code framework that allows the user to specialize the code.

Special function register (SFR)
A register that is used to read and write to the hardware components of the core.
Stack frames
Data structures containing data objects like preserved registers, local variables, and other data objects that must be stored temporary for a particular scope (usually a function).

Earlier compilers usually had a fixed size and layout on a stack frame throughout a complete function, while modern compilers might have a very dynamic layout and size that can change anywhere and anytime in a function.

Stack sections
The section or sections that reserve space for the stack(s). Most processors use the same stack for calls and parameters, but some have separate stacks.

Standard libraries
The C and C++ library functions as specified by the C and C++ standard, and support routines for the compiler, like floating-point routines.

Static object
An object whose memory is allocated at link-time and is created during system startup (or at first use). Compare Dynamic object.

Static overlay
Instead of using a dynamic allocation scheme for parameters and auto variables, the linker allocates space for parameters and auto variables at link time. This generates a worst-case scenario of stack usage, but might be preferable for small chips with expensive stack access or no stack access at all.

Statically allocated memory
This kind of memory is allocated once and for all at link-time, and remains valid all through the execution of the application. Variables that are either global or declared static are allocated this way.

Structure value
A collecting names for structs and unions. A struct is a collection of data object placed sequentially in memory (possibly with pad bytes between them). A union is a collection of data sharing the same memory location.

Symbolic location
A location that uses a symbolic name because the exact address is unknown.

Target
1. An architecture. 2. A piece of hardware. The particular embedded system you are developing the application for. The term is usually used to distinguish the system from the host system.

Task (thread)
A task is an execution thread in a system. Systems that contain many tasks that execute in parallel are called multitasking systems. Because a processor only executes one instruction stream at the time, most systems implement some sort of task-switch mechanism (often called context switch) so that all tasks get their share of processing time. The process of determining which task that should be allowed to run next is called scheduling. Two common scheduling methods are Pre-emptive multitasking and Round Robin.

Tentative definition
A variable that can be defined in multiple files, provided that the definition is identical and that it is an absolute variable.

Terminal I/O
A simulated terminal window in C-SPY.

Timer
A peripheral that counts independent of the program execution.

Timeslice
The (longest) time an RTOS allows a task to run without running the task-scheduling algorithm. A task might be allowed to execute during several consecutive timeslices before being switched out. A task might also not be allowed to use its entire time slice, for example if, in a preemptive system, a higher priority task is activated by an interrupt.
**Translation unit**
A source file together with all the header files and source files included via the preprocessor directive `#include`, except for the lines skipped by conditional preprocessor directives such as `#if` and `#ifdef`.

**Trap**
A trap is an interrupt initiated by inserting a special instruction into the instruction stream. Many systems use traps to call operating system functions. Another name for trap is software interrupt.

**Type qualifiers**
In Standard C/C++, `const` or `volatile`. IAR Systems compilers usually add target-specific type qualifiers for memory and other type attributes.

**UBROF (Universal Binary Relocatable Object Format)**
File format produced by some of the IAR Systems programming tools, if your product package includes the XLINK linker.

**Value expressions, in linker configuration file**
A constant number that can be built up out of expressions that has a syntax similar to C expressions.

**Veneer**
A small piece of code that is inserted as a springboard between caller and callee when:

- There is a mismatch in mode, for example ARM and Thumb
- The call instruction does not reach its destination.

**Virtual address (logical address)**
An address that must be translated by the compiler, linker or the runtime system into a physical memory address before it is used. The virtual address is the address seen by the application, which can be different from the address seen by other parts of the system.

**Virtual space**
An IAR Embedded Workbench Editor feature which allows you to place the insertion point outside of the area where there are actual characters.

**Volatile storage**
Data stored in a volatile storage device is not retained when the power to the device is turned off. To preserve data during a power-down cycle, you should store it in non-volatile storage. This should not be confused with the C keyword `volatile`. Compare Non-volatile storage.

**von Neumann architecture**
A computer architecture where both instructions and data are transferred over a common data channel. Compare Harvard architecture.

**Watchpoints**
Watchpoints keep track of the values of C variables or expressions in the C-SPY Watch window as the application is being executed.

**XLINK**
The IAR XLINK Linker which uses the UBROF output format.

**Zero-initialized sections**
Sections that should be initialized to zero at startup. See also Section.
Zero-overhead loop
A loop in which the loop condition, including branching back
to the beginning of the loop, does not take any time at all. This
is usually implemented as a special hardware feature of the
processor and is not available in all architectures.

Zone
Different processors have widely differing memory
architectures. Zone is the term C-SPY uses for a named
memory area. For example, on processors with separately
addressable code and data memory there would be at least two
zones. A processor with an intricate banked memory scheme
might have several zones.
A

absolute location
  definition of ............................... 569
  specifying for a breakpoint ............. 345
Access Type (Breakpoints dialog box) .......... 276
data breakpoint ........................... 208
  immediate breakpoint ..................... 210
Access type (Data Log dialog box) .......... 279
access type (in JTAG Watchpoints dialog box) .... 284
Access type (Trace Filter dialog box) ........... 293
Access type (Trace Start dialog box) ............ 288
Access type (Trace Stop dialog box) .......... 291
Action (Breakpoints dialog box) .............. 275
code breakpoint ............................ 342
data breakpoint ........................... 209
  immediate breakpoint ..................... 211
Additional include directories (assembler option) ... 472
Additional include directories (compiler option) ...... 461
Additional libraries (linker option) ............ 482
address expression, definition of ............. 569
address, in JTAG Watchpoints dialog box. ....... 283
AEABI, definition of .......................... 569
Alias (Key bindings option) ................... 376
Allow alternative register names, mnemonics and operands
  (assembler option) ......................... 468
Allow hardware reset (C-SPY RDI option) ........ 265
Angel driver, features ....................... 14
Angel interface, specifying ................... 247
Angel (C-SPY options) ....................... 247
application
  built outside the IDE ..................... 123
definition of ................................ 569
testing .................................... 100, 163
architecture, definition of ................... 569
archive, definition of ....................... 569
argument variables .......................... 396
  environment variables ..................... 367
in #include file paths
  assembler .................................. 472
  compiler .................................. 461
  summary ................................... 366
Arguments (External editor option) .............. 381
ARM code, mixing with Thumb code ............... 456
Arm (compiler option) ........................ 456
arm (directory) ................................ 19
ar, definition of ................................ 569
asm (filename extension) ...................... 21
assembler
  command line version ...................... 81
documentation ................................ 24
features ..................................... 16
assembler comments, text style in editor ........ 107
assembler directives
  definition of ................................ 569
text style in editor ......................... 107
assembler labels, viewing ...................... 139
assembler language, definition of .............. 569
assembler list files
  compiler call frame information
  conditional information, specifying .......... 470
cross-references, generating .................. 471
format ...................................... 58
generating ................................... 470
header, including ............................ 470
lines per page, specifying ................... 471
tab spacing, specifying ...................... 471
Assembler mnemonics (compiler option) .......... 460
assembler options ........................... 467
  Allow alternative register names, mnemonics
  and operands ............................... 468
  Diagnostics ................................ 473
  Language .................................. 467
  List ........................................ 470
  Output ..................................... 469
  Preprocessor ................................. 471
  assembler options, definition of ............ 569
  assembler output, including debug information .... 469
assembler preprocessor ........................................ 471
assembler symbols
   defining .................................................. 472
   using in C-SPY expressions ........................... 136
assembler variables, viewing .............................. 139
assert, in built applications ............................. 89
assumptions, programming experience .................. xli
Attach to program (C-SPY options) ...................... 245
attributes on sections, definition of .................. 580
Auto indent (editor option) .............................. 378
Auto window .................................................. 421
context menu ............................................... 421
Automatic runtime library selection (linker option) .. 482
Automatic (compiler option) ............................. 454
Autostep settings dialog box (Debug menu) ............ 439

B

-B (C-SPY command line option) ........................ 505
--backend (C-SPY command line option) .............. 505
Background color (IDE Tools option) .................. 384
backtrace information
   definition of .............................................. 569
   generated by compiler ................................. 133
   viewing in Call Stack window ......................... 425
bank switching, definition of ............................ 569
banked code, definition of ................................ 569
banked data, definition of ................................ 570
banked memory, definition of ............................. 570
bank-switching routines, definition of .................. 570
Base (Register filter option) ............................. 393
bat (filename extension) .................................. 21
Batch Build ................................................... 99
Batch Build Configuration dialog box (Project menu) .. 372
Batch Build dialog box (Project menu) ................. 371
batch files
   definition of .............................................. 570
   specifying from the Tools menu ....................... 86
   batch mode, using C-SPY in ........................... 499
Baud rate (C-SPY Angel option) ......................... 248
Baud rate (C-SPY IAR ROM-monitor option) .......... 251
Baud rate (C-SPY Macraigor option) ................... 263
--BE32 (C-SPY command line option) .................. 501
--BE8 (C-SPY command line option) ................... 501
Big endian (C-SPY target option) ....................... 446
bin, arm (subdirectory) ................................... 19
bin, common (subdirectory) .............................. 21
bitfield, definition of ..................................... 570
blocks, in C-SPY macros ................................ 530
Block, definition of ...................................... 570
Body (b) (Configure auto indent option) ............... 380
bold style, in this guide .................................. xlvi
bookmarks
   adding ...................................................... 112
   showing in editor ....................................... 379
break condition (in JTAG Watchpoints dialog box) .. 285
Break (button) .............................................. 133, 405
breakpoint condition, example ........................ 145–146
breakpoint icons ............................................ 142
Breakpoint type (Breakpoints dialog box) .............. 274
Breakpoint Usage dialog box (Simulator menu) ........ 211, 280
   using ...................................................... 147
breakpoints .................................................. 132
code, example ............................................ 557
   conditional, example ................................... 69
   connecting a C-SPY macro ............................. 161
   consumers ................................................. 148
data .......................................................... 207, 275, 278
   example ................................................... 558–559, 561–562
   immediate .................................................. 209
   example ................................................... 69
   in Memory window ....................................... 144
   in ramfunc functions .................................. 281
   in the simulator ......................................... 206
   listing all .................................................. 147
   on vectors, using Macraigor ........................... 280
   setting
      in memory window .................................... 144
<table>
<thead>
<tr>
<th>Topic</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>using system macros</td>
<td>145</td>
</tr>
<tr>
<td>using the dialog box</td>
<td>143</td>
</tr>
<tr>
<td>settings</td>
<td>369</td>
</tr>
<tr>
<td>single-stepping if not available</td>
<td>121</td>
</tr>
<tr>
<td>system, description of</td>
<td>141</td>
</tr>
<tr>
<td>toggling</td>
<td>143</td>
</tr>
<tr>
<td>useful tips</td>
<td>145</td>
</tr>
<tr>
<td>viewing</td>
<td>146</td>
</tr>
<tr>
<td>Breakpoints dialog box</td>
<td></td>
</tr>
<tr>
<td>Code</td>
<td>274, 341</td>
</tr>
<tr>
<td>Data</td>
<td>207, 275</td>
</tr>
<tr>
<td>Data Log</td>
<td>278</td>
</tr>
<tr>
<td>Immediate</td>
<td>210</td>
</tr>
<tr>
<td>Log</td>
<td>343</td>
</tr>
<tr>
<td>Trace Filter (J-Link)</td>
<td>292</td>
</tr>
<tr>
<td>Trace Start</td>
<td>193</td>
</tr>
<tr>
<td>Trace Start (J-Link)</td>
<td>287</td>
</tr>
<tr>
<td>Trace Stop</td>
<td>194</td>
</tr>
<tr>
<td>Trace Stop (J-Link)</td>
<td>289</td>
</tr>
<tr>
<td>Breakpoints window (View menu)</td>
<td>340</td>
</tr>
<tr>
<td>Breakpoints (J-Link/J-Trace option)</td>
<td>273</td>
</tr>
<tr>
<td>breakpoints, definition of</td>
<td>570</td>
</tr>
<tr>
<td>Broadcast all branch addresses (Trace Settings option)</td>
<td>177</td>
</tr>
<tr>
<td>-build (iarbuild command line option)</td>
<td>101</td>
</tr>
<tr>
<td>Build Actions</td>
<td>100</td>
</tr>
<tr>
<td>Build Actions Configuration (Build Actions options)</td>
<td>479</td>
</tr>
<tr>
<td>build configuration</td>
<td></td>
</tr>
<tr>
<td>creating</td>
<td>90</td>
</tr>
<tr>
<td>definition of</td>
<td>88</td>
</tr>
<tr>
<td>Build window context menu</td>
<td>347</td>
</tr>
<tr>
<td>Build window (View menu)</td>
<td>347</td>
</tr>
<tr>
<td>building</td>
<td></td>
</tr>
<tr>
<td>commands for</td>
<td>99</td>
</tr>
<tr>
<td>from the command line</td>
<td>101</td>
</tr>
<tr>
<td>options</td>
<td>386</td>
</tr>
<tr>
<td>pre- and post-actions</td>
<td>100</td>
</tr>
<tr>
<td>the process</td>
<td>97</td>
</tr>
<tr>
<td>byte order, specifying</td>
<td>446</td>
</tr>
</tbody>
</table>

**C**

C comments, text style in editor ........................................ 107
C compiler. **See** compiler
C function information, in C-SPY ........................................ 133
C keywords, text style in editor ....................................... 107
C symbols, using in C-SPY expressions ................................ 135
C variables, using in C-SPY expressions ................................ 135
c (filename extension) .................................................. 21
call chain, displaying in C-SPY ....................................... 133
call frame information
  definition of .................................................................. 570
  including in assembler list file .................................... 460
call frame information **See also** backtrace information
Call stack information ..................................................... 133
Call Stack window .......................................................... 425
context menu ...................................................................... 425
example ............................................................................. 68
for backtrace information ................................................ 133
calling convention
  definition of .................................................................. 570
  examining ....................................................................... 55
__cancelAllInterrupts (C-SPY system macro) ............................. 535
__cancelCurrentInterrupt (C-SPY system macro) ......................... 536
Catch exceptions (C-SPY J-Link option) ................................ 273
Catch exceptions (C-SPY RDI option) .................................... 266
category, in Options dialog box .......................................... 98, 370
cfg (filename extension) .................................................... 21
different characters, in assembler macro quotes ..................... 468
direct memory access, definition of ..................................... 570
Check In Files dialog box .................................................. 327
Check Out Files dialog box ................................................ 328
checksum
  definition of .................................................................. 570
  generating ...................................................................... 489
Checksum (linker options) .................................................. 488
chm (filename extension) .................................................... 21
-clean (iarbuild command line option) .................................. 101
__clearBreak (C-SPY system macro) ....................................... 536
this guide .................................................. xli
doc, arm (subdirectory) ............................... 20
doc, common (subdirectory) ....................... 21
Download (C-SPY options) ........................ 245
--download_only (C-SPY command line option) . 507
drag-and-drop
  of files in Workspace window .................. 90
text in editor window ............................... 107
Driver (C-SPY option) ............................... 493
drivers, arm (subdirectory) ......................... 20
__driverType (C-SPY system macro) .............. 538
--drv_attach_to_program (C-SPY command line option) . 501
--drv_catch_exceptions (C-SPY command line option) . 507
--drv_communication (C-SPY command line option) . 508
--drv_communication_log (C-SPY command line option) . 510
--drv_default_breakpoint (C-SPY command line option) . 511
--drv_reset_to_cpu_start (C-SPY command line option) . 511
--drv_restore_breakpoints (C-SPY command line option) . 512
--drv_suppress_download (C-SPY command line option) . 502
--drv_vector_table_base (C-SPY command line option) . 512
--drv_verify_download (C-SPY command line option) . 502
dSP. See digital signal processor
DWARF, definition of ................................. 572
Dynamic Data Exchange (DDE) ...................... 113
calling external editor ............................. 381
dynamic initialization, definition of ............ 572
dynamic memory allocation, definition of ......... 572
dynamic object, definition of ................... 572
commands ............................................. 107
customizing the environment ...................... 112
external ............................................. 112
features ............................................. 5
HTML files ........................................... 330
indentation ......................................... 108
keyboard commands .................................. 335
matching parentheses and brackets ............... 109
options .............................................. 377
shortcut to functions ............................... 112, 331
splitter controls ................................... 331
status bar, using in ................................ 109
using .................................................. 105
Editor Colors and Fonts (IDE Options dialog box) . 383
Editor Font (Editor colors and fonts option) .... 383
Editor Setup Files (IDE Options dialog box) .... 382
teditor setup files, options ......................... 382
Editor window ....................................... 330
tcontext menu ........................................ 332
tab, context menu ................................... 331
tEditor (External editor option) ................... 381
tEditor (IDE Options dialog box) ................. 377
tEEC++ syntax (compiler option) .................. 454
tEEPROM, definition of ............................ 572
tELF, converting from ............................... 475
tELF, definition of .................................. 573
Embedded C++
tdefinition of ....................................... 573
tsyntax, enabling in compiler ...................... 454
Embedded C++ Technical Committee ................ xlvi
Embedded C++ (compiler option) .................. 454
tenabled system, definition of .................... 573
Embedded Workbench
teditor .................................................. 105
texting from ......................................... 83
tlayout .................................................. 83
tmain window ......................................... 82, 318
treference information ................................ 317
running ............................................... 82

E

Edit Filename Extensions dialog box (Tools menu) .... 398
Edit Interrupt dialog box (Simulator menu) ........ 218
Edit Memory Access dialog box ..................... 206
Edit menu ............................................ 353
eting source files .................................... 105
teditor, user guide .................................. ii, ii
teditor
tcode templates ....................................... 109

----------

IAR Embedded Workbench® IDE

592  User Guide for ARM

UARM-17b
tracing incorrect function arguments .......... 145
using libraries ..................................... 75
variables
setting a watch point ............................... 49
watching in C-SPY ................................. 48
viewing assembler list file ......................... 58
viewing compiler list files ......................... 41
workspace, creating a new ......................... 33
examples, arm (subdirectory) ....................... 20
exceptions, definition of ......................... 573
execUserExit (C-SPY setup macro) ................. 532
execUserFlashExit (C-SPY setup macro) ............ 532
execUserFlashInit (C-SPY setup macro) ............. 532
execUserPreload (C-SPY setup macro) .............. 532
execUserReset (C-SPY setup macro) ............... 532
elements (C-SPY setup macro) ..................... 532
executable image, definition of .................... 573
Executable (output directory) ....................... 447
executing a program up to a breakpoint .......... 50
execution history, tracing ........................ 174
execution time, reducing ......................... 163
$EXE_DIR$ (argument variable) ..................... 366
Exit (File menu) ..................................... 83
exit, of user application ........................... 133
expense memory access, definition of .......... 573
expressions. See C-SPY expressions
Extend to cover requested range (Breakpoints dialog) 277
Extend to cover requested range (Trace Filter dialog box) 293
Extend to cover requested range (Trace Start dialog box) 288
Extend to cover requested range (Trace Stop dialog box) 291
Extend to cover requested range (Data Log dialog box) ......................... 279
extended command line file ......................... 23
    specifying for pre- and post-build actions ......... 479
Extended Embedded C++ syntax, enabling in compiler ... 454
extended keywords, definition of ................... 573
extensions. See filename extensions or language extensions
External Editor (IDE Options dialog box) ............ 381
external editor, using ................................ 112
external input (in JTAG Watchpoints dialog box) .... 284
Extra Options
for assembler ........................................ 474
for compiler ........................................ 465
for C-SPY ........................................... 496
for C-SPY hardware drivers ................. 246
for linker ........................................... 490
F
factory settings
    library builder .................................... 492
    restoring default settings ..................... 99
features
    assembler .......................................... 16
    compiler .......................................... 15
    editor ............................................. 5
    source code control ................................ 4
file extensions. See filename extensions
File menu ........................................... 350
file types
    device description ................................ 20
        specifying in Embedded Workbench ............ 121
    device selection ................................ 20
    documentation .................................... 20
    drivers .......................................... 20
    extended command line .......................... 23
    flash loader applications ....................... 20
    header .......................................... 20
    include .......................................... 20
    library .......................................... 20
    linker configuration files ...................... 20
    macro ............................................ 121, 494
    project templates ............................... 20
    readme .......................................... 20–21
    syntax coloring configuration .................. 20
    filename extensions ............................ 21
cfg, syntax highlighting ......................... 384
Flash Loader Overview dialog box .......................... 310
Flash loader path, specifying the path to a flash loader .......................... 313
flash memory
  load library module to ........................................ 549
  loading externally built applications to ....................... 123
flash (filename extension) ........................................ 22
flashdict (filename extension) .................................... 22
--flash_loader (C-SPY command line option) ..................... 513
floating windows .................................................. 83
fmt (filename extension) .......................................... 22
font
  Editor ......................................................... 383
  Fixed width .................................................. 374
  Proportional width ........................................... 374
for (macro statement) ........................................... 529
Force (C-SPY J-Link option) ...................................... 178-179
Forced Interrupt window (Simulator menu) ......................... 220
format specifiers, definition of .................................. 574
formats
  assembler list file ............................................ 58
  compiler list file ............................................ 41
  --fpu (C-SPY command line option) ......................... 502
FPU (General option) .............................................. 446
  function calls, displaying in C-SPY ......................... 68
  function level profiling ...................................... 163
  Function Profiler window ..................................... 306
  context menu ................................................. 308
  Function Trace (C-SPY window) ............................... 188
functions
  C-SPY running to when starting ............................... 121, 494
  intrinsic, definition of .................................... 575
  shortcut to in editor windows ............................... 112, 331

G

GDB Server Setup (C-SPY options) ............................ 249
__gdbserver_exec_command (C-SPY system macro) .................. 541
--gdbserv_exec_command (C-SPY command line option) 513

ddf, selecting device description file ...................... 121
eww, the workspace file ........................................ 83
icf, linker configuration file .................................. 44
mac
  the macro file .............................................. 156
  using macro file ........................................... 121
other than default ............................................. 23
sfr, register definitions for C-SPY ........................... 126
Filename Extensions dialog box (Tools menu) ................. 397
Filename Extensions Overrides dialog box (Tools menu) ....... 398
files
  adding to a project ........................................... 36
  checking in and out .......................................... 95
  compiling, example .......................................... 40
  editing ........................................................ 105
  navigating among ............................................. 91
readme.htm ...................................................... 23
$FILE_DIRS (argument variable) .................................. 367
$FILE_FAMES (argument variable) ............................. 367
$FILE_PATHS (argument variable) ............................. 367
Fill dialog box (Memory window) ............................... 413
Fill pattern (linker option) .................................... 489
Fill unused code memory (linker option) ....................... 489
filling, definition of ........................................... 573
Filter Files (Register filter option) ........................ 393
Find dialog box (Edit menu) .................................... 356
Find in Files dialog box (Edit menu) ........................ 358
Find in Files window (Edit menu) ............................... 347
context menu .................................................... 348
Find in Trace dialog box ....................................... 197
Find in Trace window ............................................ 198
Find Next (button) .............................................. 319
Find Previous (button) .......................................... 319
Find (button) ..................................................... 319
first activation time (interrupt property) ..................... 214
definition of ................................................... 214
Fixed width font (IDE option) .................................... 374
flash loader, using .............................................. 309
flash loader applications ........................................ 20
flash memory
  load library module to ........................................ 549
  loading externally built applications to ..................... 123
flash (filename extension) ....................................... 22
flashdict (filename extension) .................................. 22
--flash_loader (C-SPY command line option) .................... 513
floating windows ................................................ 83
fmt (filename extension) ....................................... 22
font
  Editor ......................................................... 383
  Fixed width .................................................. 374
  Proportional width ........................................... 374
for (macro statement) .......................................... 529
Force (C-SPY J-Link option) ..................................... 178-179
Forced Interrupt window (Simulator menu) ..................... 220
format specifiers, definition of ................................. 574
formats
  assembler list file ............................................ 58
  compiler list file ............................................ 41
  --fpu (C-SPY command line option) ........................ 502
FPU (General option) ............................................ 446
  function calls, displaying in C-SPY ......................... 68
  function level profiling ...................................... 163
  Function Profiler window ..................................... 306
  context menu ................................................. 308
  Function Trace (C-SPY window) ............................... 188
functions
  C-SPY running to when starting ............................... 121, 494
  intrinsic, definition of .................................... 575
  shortcut to in editor windows ............................... 112, 331

G

GDB Server Setup (C-SPY options) ............................ 249
__gdbserver_exec_command (C-SPY system macro) .................. 541
--gdbserv_exec_command (C-SPY command line option) 513

595
general options .............................................. 445
  specifying, example .................................... 37
Endian, mode ............................................. 446
FPU .......................................................... 446
Library Configuration ..................................... 448
Library Options ........................................... 450
MISRA C ..................................................... 451
Output ........................................................ 447
Target .......................................................... 445
general options, definition of ........................... 574
Generate browse information (IDE Project options) . 387
Generate checksum (linker option) ......................... 489
Generate debug info (assembler option) .................... 469
Generate debug information (compiler option) ............. 459
Generate interwork code (compiler option) ................. 456
Generate linker map file (linker option) .................... 485
Generate log (linker option) ................................ 485
generic pointers, definition of ............................. 574
glossary ........................................................ 569
Go to Bookmark (button) .................................... 319
Go to function (editor button) .............................. 331
Go to Line dialog box ...................................... 112, 331
Go To (button) .............................................. 319
Go (button) .................................................. 405
Go (Debug menu) ........................................... 132
Group members (Register filter option) .................... 393
Groups (Register filter option) ............................. 393
groups, definition of ....................................... 89
h (filename extension) ..................................... 22
Hardware reset (C-SPY Macraigor option) ................. 263
Harvard architecture, definition of ....................... 574
header files .................................................. 20
  quick access to .......................................... 111
heap memory, definition of ................................ 574
heap size, definition of .................................... 574
Help menu .................................................... 401
helpfiles (filename extension) .............................. 22
highlighting, in C-SPY ..................................... 132
hold time (interrupt property) ............................. 219
definition of ............................................... 214
host, definition of .......................................... 574
html (filename extension) ................................... 22
html (filename extension) ................................... 22
__hwReset (C-SPY system macro) .......................... 541
__hwResetRunToBp (C-SPY system macro) ................. 542
__hwResetWithStrategy (C-SPY system macro) .......... 543
i (filename extension) .................................... 22
IAR Assembler Reference Guide ......................... 24
IAR debugger driver plugin (Third-party Driver option) . 269
IAR Development Guide .................................... 24
IAR ROM-monitor interface, specifying ................... 251
IAR ROM-monitor (C-SPY options) ......................... 250
IAR Systems web site ..................................... 25
 iarbuild, building from the command line ............... 101
iarchive
  definition of ............................................. 574
  using for building libraries ............................. 75
larduino.exe ............................................... 82
icf (filename extension) ................................... 22
icons, in this guide ......................................... .xlvi
IDE .......................................................... 3–4
definition of ............................................... 574
if else (macro statement) ................................... 529
if (macro statement) ........................................ 529
Ignore standard include directories (assembler option) . 472
Ignore standard include directories (compiler option) . 461
ILINK
  See also linker
    definition of ............................................. 574
  illegal memory accesses, checking for .................. 203
Images window .............................................. 432
case menu ..................................................... 432

Images, loading multiple ........................................ 495
immediate breakpoints ........................................... 209
inc (filename extension) ......................................... 22
Include compiler call frame information (compiler option) ........... 460
include files ....................................................... 20
    assembler, specifying path .................................. 472
    compiler, specifying path ................................... 461
definition of .................................................... 574
Include header (assembler option) .................................. 470
Include listing (assembler option) .................................. 470
Include source (compiler option) .................................. 460
Incremental Search dialog box (Edit menu) ......................... 360
inc, arm (subdirectory) ........................................... 20
Indent size (editor option) ....................................... 377
indentation, in editor ............................................ 108
information, product ............................................. 23
inherited settings, overriding .................................... 98
ini (filename extension) .......................................... 22
initialization in ILINK config file, definition of .................... 575
initialized sections, definition of ................................ 575
    definition of ................................................. 575
    language facilities in compiler ................................ 15
    inlining, definition of ..................................... 575
input
    redirecting to Terminal I/O window .......................... 426
    special characters in Terminal I/O window ................... 426
Input Mode dialog box ............................................ 427
Input (linker options) ............................................ 483
Input (Linker option) ............................................. 483
insertion point, shortcut key for moving ........................... 107
installation directory ............................................ xlvi
installation path, default ........................................ 19
installed files ..................................................... 19
documentation ...................................................... 20–21
executable .......................................................... 21
include .............................................................. 20
library ............................................................... 20
instruction mnemonics, definition of ................................ 575
Integrated Development Environment (IDE) .......................... 3–4
    definition of .................................................. 574
    Intel-extended, C-SPY input format .............................. 119
    Interface (C-SPY J-Link option) ................................ 256, 268
    Interface (C-SPY LMI FTDI option) ............................. 260
    Interface (C-SPY Macraigor option) ............................. 262
    Internet, IAR Systems web site .................................. 25
    Interrupt Log Events (C-SPY J-Link option) .................... 179
    Interrupt Log Summary window ................................ 302
        context menu ............................................... 298
    Interrupt Log window .......................................... 300
        context menu ............................................... 298
    Interrupt Log ((SWO Hardware Settings) ......................... 182
    Interrupt Setup dialog box (Simulator menu) ................... 217
    interrupt system, using device description file ................ 217
    interrupt vector table, definition of .......................... 575
    interrupt vector, definition of ................................ 575
interrupts
    adapting C-SPY system for target hardware ...................... 216
    definition of .................................................. 575
    nested, definition of ......................................... 577
    options ........................................................ 219
    simulated, definition of ...................................... 213
timer, example ................................................. 222
    using system macros .......................................... 220
interwork code, generating ......................................... 456
intrinsic functions
    definition of .................................................. 575
    language facilities in compiler ................................ 15
    intrinsic, definition of ..................................... 575
    iobmanip, definition of ...................................... 575
    __isBatchMode (C-SPY system macro) .......................... 543
ISO/ANSI C
    library compliance with ....................................... 16
    making compiler adhering to ................................ 455
    sizeof operator in C-SPY ..................................... 136
    italic style, in this guide ..................................... xlvi
    ITM Stimulus Ports (SWO Hardware Settings) ................... 182
    i79 (filename extension) ...................................... 22
__jlinkExecCommand (C-SPY system macro) . . . . . . . . . 544
--jlink_device_select (C-SPY command line option) . . . 514
--jlink_exec_command (C-SPY command line option) . . . . 514
--jlink_initial_speed (C-SPY command line option) . . . . 514
--jlink_interface (C-SPY command line option) . . . . . . 515
--jlink_ir_length (C-SPY command line option) . . . . . . . 515
--jlink_reset_strategy (C-SPY command line option) . . . 516
--jlink_speed (C-SPY command line option) . . . . . . . . . . 517

JTAG interfaces
  EPI JEENI . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 247
  J-Link . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 252
  LMI FTDI . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 260

JTAG scan chain with multiple targets
(C-SPY Macraigor option) . . . . . . . . . . . . . . . . . . . . . . . . . 263
JTAG scan chain (C-SPY J-Link options) . . . . . . . . . . . . . . . 256
JTAG speed (C-SPY LMI FTDI option) . . . . . . . . . . . . . . . . . 261
JTAG speed (C-SPY Macraigor option) . . . . . . . . . . . . . . . . 262
JTAG watchpoints . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 282
JTAG Watchpoints (dialog box) . . . . . . . . . . . . . . . . . . . . . . . 283
__jtagCommand (C-SPY system macro) . . . . . . . . . . . . . . 544
__jtagCP15IsPresent (C-SPY system macro) . . . . . . . . . 545
__jtagCP15ReadReg (C-SPY system macro) . . . . . . . . . 545
__jtagCP15WriteReg (C-SPY system macro) . . . . . . . . . 545
__jtagData (C-SPY system macro) . . . . . . . . . . . . . . . . . . . . . . 545
__jtagRawRead (C-SPY system macro) . . . . . . . . . . . . . . 546
__jtagRawSync (C-SPY system macro) . . . . . . . . . . . . . . 547
__jtagRawWrite (C-SPY system macro) . . . . . . . . . . . . . . 548
__jtagResetTRST (C-SPY system macro) . . . . . . . . . . . . 548
JTAG/SWD speed (C-SPY J-Link option) . . . . . . . . . . . . . . 255
J-Link communication problem . . . . . . . . . . . . . . . . . . . . . . . 255
J-Link driver, features . . . . . . . . . . . . . . . . . . . . . . . . . . . . 10
J-Link JTAG interface . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 252
J-Link watchpoints . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 283
J-Link/J-Trace Connection (C-SPY options) . . . . . . . . . . . . . 256
J-Link/J-Trace Reset (C-SPY option) . . . . . . . . . . . . . . . . . . 253
J-Link/J-Trace Setup (C-SPY options) . . . . . . . . . . . . . . . . . 252

K

Keep symbol (Linker option) . . . . . . . . . . . . . . . . . . . . . . . . 483
Key bindings (IDE Options dialog box) . . . . . . . . . . . . . . . . . . 375
key bindings, definition of . . . . . . . . . . . . . . . . . . . . . . . . . . . 575
key summary, editor . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 335
keywords
  enable language extensions for . . . . . . . . . . . . . . . . . . . . . 455
  specify syntax color for in editor . . . . . . . . . . . . . . . . . . . 108
  keywords, definition of . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 575

L

Label (c) (Configure auto indent option) . . . . . . . . . . . . . . 380
labels (assembler), viewing . . . . . . . . . . . . . . . . . . . . . . . . . 139
Language conformance (compiler option) . . . . . . . . . . . . . 455
language extensions
  definition of . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 575
  disabling in compiler . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 455
language facilities, in compiler . . . . . . . . . . . . . . . . . . . . . . . 15
Language (assembler options). . . . . . . . . . . . . . . . . . . . . . . . 467
Language (compiler options) . . . . . . . . . . . . . . . . . . . . . . . . 453
Language (IDE Options dialog box) . . . . . . . . . . . . . . . . . . . . 376
Language (Language option) . . . . . . . . . . . . . . . . . . . . . . . . 376
layout, of Embedded Workbench . . . . . . . . . . . . . . . . . . . . . . . 83
library
  creating a project for . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 76
  runtime . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 16
library builder options, factory settings . . . . . . . . . . . . . . . . . 492
library builder, output options . . . . . . . . . . . . . . . . . . . . . . . . 491
library configuration file
  definition of . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 576
  specifying from IDE . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 448
Library Configuration (general options) . . . . . . . . . . . . . . . . . . 448
library files . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 20
library functions
  configurable . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 20
  reference information . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 106
Menu (Key bindings option) ........................................ 375
menus .............................................................. 350
specific to C-SPY ............................................. 437
Messages window, amount of output ....................... 385
Messages (IDE Options dialog box) ......................... 385
metadata (subdirectory) ......................................... 21
microcontroller, definition of .................................. 577
microprocessor, definition of .................................. 577
migration, from earlier IAR compilers ....................... xlv
Min.s (assembler tutorial file) .................................. 75
MISRA C
  compiler options ............................................. 464
documentation .................................................. 24
general options ................................................ 451
modules, definition of ......................................... 577
Motorola, C-SPY input format ................................. 119
Multiply and accumulate, definition of ................... 576
multitasking, definition of .................................. 578
multi-file compilation .......................................... 453
definition of .................................................... 577
Multi-ICE interface. See RealView Multi-ICE interface

N
naming conventions ............................................. xlvii
Navigate Backward (button) .................................. 319
NDEBUG, preprocessor symbol ............................... 89
nested interrupts, definition of .............................. 577
New Configuration dialog box (Project menu) ............ 368
New Document (button) ....................................... 319
New Group (Register filter option) ......................... 393
Next Statement (button) ....................................... 405
No dynamic read/write initialization (compiler option) .. 456
non-banked memory, definition of ......................... 577
non-initialized memory, definition of ..................... 577
non-volatile storage, definition of ......................... 577
NOP (assembler instruction)
  definition of .............................................. 577
no-init sections, definition of ............................. 577

O

o (filename extension) ........................................... 22
objcopy, definition of ....................................... 577
objdump, definition of ....................................... 574
object file (absolute), definition of ....................... 577
object file (relocatable), definition of ................... 578
object files, specifying output directory for ............. 448
object, definition of ......................................... 577
OBJ_DIR$ (argument variable) ............................... 367
OCD interface device (C-SPY Macraigor option) ........ 262
Offset (C-SPY image option) .................................. 495
online documentation
  available from Help menu .................................. 401
  common, in directory ...................................... 21
target-specific, in directory ................................. 20
online help ..................................................... 25
Open Workspace (File menu) .................................. 351
_openFile (C-SPY system macro) ............................. 551
Opening Brace (a) (Configure auto indent option) ....... 380
operator precedence, definition of ....................... 578
operators
definition of ................................................ 578
sizeof in C-SPY .............................................. 136
sizeof in C-SPY .............................................. 136
optimization levels .......................................... 457
Optimizations page (compiler options) .................... 457
Optimizations (compiler option) ........................... 457
optimizations, effects on variables ....................... 137
options
assembler ..................................................... 467
build actions .................................................. 479
compiler ....................................................... 453
converter ..................................................... 475
custom build .................................................. 477
C-SPY .......................................................... 493
C-SPY command line ......................................... 505
C-SPY Macraigor ............................................ 262
editor .......................................................... 377
general .................................................. 445
specifying ........................................... 37
hardware debugger systems ......................... 243
library builder ....................................... 491
linker .................................................. 481
setup files for editor ................................. 382
Options dialog box (Project menu) ................. 370
using .................................................... 98
options, definition of ................................ 578
__orderInterrupt (C-SPY system macro) .............. 552
out (filename extension) ............................... 22
output ................................................... 475
assembler, including debug information .............. 469
compiler
   including debug information ........................ 459
   preprocessor, generating ............................ 462
converting from ELF ................................... 475
from C-SPY, redirecting to a file .................... 125
linker
   specifying filename ................................ 484
   specifying filename for linker output .............. 485
Output assembler file (compiler option) .............. 460
Output debug information (linker option) ............. 485
Output file (compiler option) ........................ 460
Output file (converter option) ........................ 475
Output file (linker option) ............................ 484
Output list file (compiler option) ..................... 460
Output (assembler option) ............................ 469
Output (general options) .............................. 475
Output (library builder options) ...................... 491
Output (linker options) ................................ 484
overlay, definition of ................................ 578
Override default program entry (linker option) ...... 482

**P**

-p (C-SPY command line option) ..................... 522

parameters
   list of passed to the flash loader .......................... 311
   specify to control flash loader ................................ 313
   tracing incorrect values of .................................. 133
typographic convention .................................... xlvi
parentheses and brackets, matching (in editor) ........ 109
part number, of user guide ................................ ii, ii
Paste (button) .......................................... 319
Path (C-SPY image option) .............................. 495
paths
   assembler include files .................................. 472
   compiler include files .................................... 461
   relative, in Embedded Workbench ......................... 91, 335
   source files ............................................. 335
   pbd (filename extension) ................................ 22
   pfi (filename extension) ................................ 23
   PC Sampling (SWO Hardware Settings) .................... 181
peripheral units
   definition of .......................................... 578
   definitions in ddf file ................................... 121
device-specific ........................................... 125
   in Register window ...................................... 126
   pew (filename extension) ................................ 23
   pipeline, definition of ................................... 578
   placement, definition of ................................... 578
   Plain ‘char’ is (compiler option) ......................... 455
   Play a sound after build operations (IDE Project options) ........................... 387
--plugin (C-SPY command line option) ................ 522
   plugin modules (C-SPY) .................................. 8
   loading .................................................... 122
   plugins
      arm (subdirectory) ..................................... 20
      common (subdirectory) .................................. 21
   Plugins (C-SPY options) ................................ 496
   pointers
      definition of ............................................. 578
      for stack is outside of memory range .................. 151
      warn when stack pointer is out of range .............. 391
      __popSimulatorInterruptExecutingStack (C-SPY
      system macro) ........................................... 553
Index

Port (Angel C-SPY option) .............................................. 248
Port (C-SPY Macraigor option) ..................................... 263
Port (OKI ROM-monitor C-SPY option) .......................... 251
Position-independence (compiler option) ......................... 465
powerpac, arm (subdirectory) ........................................ 20
#pragma directive, definition of .................................. 578
precedence, definition of ............................................. 578
preemptive multitasking, definition of ............................ 578
Preinclude file (compiler option) ................................. 462
Preprocessor
   definition of. See C-style preprocessor
   specifying output to file ........................................ 462
preprocessor directives
   definition of ...................................................... 578
text style in editor ................................................... 107
Preprocessor output to file (compiler option) .................. 462
Preprocessor (assembler option) ................................... 471
Preprocessor (compiler options) ................................... 461
prerequisites, programming experience ........................... xli
Press shortcut key (Key bindings option) ........................ 375
Primary (Key bindings option) ...................................... 375
Print formatter (general option) ................................. 450
prj (filename extension) ............................................. 23
probability (interrupt property) ................................... 220
definition of ......................................................... 214
Processor mode (compiler option) ............................... 456
Processor variant (target option) ............................... 445
processor variant, definition of ................................... 579
--proc_stack_xxx (C-SPY command line option) .............. 523
product overview
   assembler .......................................................... 16
   compiler .......................................................... 15
   C-SPY Debugger .................................................. 5
directory structure .................................................... 19
documentation ........................................................ 23
file types ............................................................. 21
IAR Embedded Workbench IDE ..................................... 3
linker ................................................................. 17
profiling
   on function level ............................................... 304
   on instruction level ............................................ 305
   profiling information ......................................... 163
   on functions and instructions ............................... 303
   Profiling window ............................................... 429
   using ............................................................. 164
program counter, definition of ................................... 579
program execution, in C-SPY ....................................... 129
program location counter, definition of ........................ 579
programming experience ........................................... xli
Project Make, options .............................................. 386
Project menu .......................................................... 363
project model ........................................................ 87
project options, definition of .................................... 579
Project page (IDE Options dialog box) .......................... 386
projects
   adding files to ................................................. 90, 363
   example .......................................................... 36
   build configuration, creating .................................. 90
   building ......................................................... 99
   in batches ....................................................... 99
   compiling, example ............................................. 40
   creating ......................................................... 34, 90
   example ........................................................ 76
   definition of ..................................................... 87, 579
   excluding groups and files ................................... 90
   files
   checking in and out ............................................ 95
   moving ........................................................... 90
   for debugging externally built applications .................. 123
   groups, creating ............................................... 90
   managing ......................................................... 4, 87
   organization ..................................................... 87
   removing items ................................................ 90
   setting options ................................................ 97
   source code control .......................................... 94
testing .............................................................. 100
version control systems ........................................... 94

UARM-17b
User Guide for ARM

IAR Embedded Workbench® IDE

606 User Guide for ARM

UARM-17b

__setCodeBreak (C-SPY system macro) .................. 557
__setDataBreak (C-SPY system macro) ................. 558
__setLogBreak (C-SPY system macro) ................. 559
__setSimBreak (C-SPY system macro) .................. 560
settings (directory) ...................................... 23
__setTraceStartBreak (C-SPY system macro) .......... 561
__setTraceStopBreak (C-SPY system macro) .......... 562
Setup macros (C-SPY option) ............................ 494
setup macros, in C-SPY. See C-SPY macros
Setup (C-SPY options) ................................. 493
severity level
  changing default for compiler diagnostics .......... 462
  changing default for linker diagnostics .......... 486
  definition of ......................................... 581
SFR
  definition of ......................................... 581
  in header files ..................................... 20
  in Register window ................................. 152
sharing, definition of ................................ 581, 583
short cut keys ........................................... 581
short cut keys .......................................... 107
Show bookmarks (editor option) ......................... 379
Show line numbers (editor option) ..................... 378
Show right margin (editor option) ..................... 378
Show timestamp (Trace Settings option) .............. 177
side-effect, definition of ............................. 581
signals, definition of ................................ 581
--silent (C-SPY command line option) ............... 525
simulating interrupts, enabling/disabling ........... 217
simulator
  definition of ......................................... 581
  features ............................................... 10
  introduction ......................................... 201
simulator driver, selecting ............................ 201
Simulator menu .......................................... 202
size optimization ....................................... 457
Size (Breakpoints dialog) ............................... 208, 277, 342
Size (Data Log dialog) .................................. 279
Size (Trace Filter dialog) .............................. 293
Size (Trace Start dialog) ................................ 288
Size (Trace Stop dialog) ................................ 290
sizeof .................................................... 136
skeleton code
  definition of ......................................... 581
Source Browser window .................................. 336
  context menu ......................................... 338
  using ................................................... 93
source code
  including in compiler list file ...................... 460
  templates .............................................. 109
Source code color in Disassembly window (IDE option) 389
Source Code Control context menu ....................... 324
source code control systems ............................ 94
Source Code Control (IDE Options dialog box) ......... 388
source code control, features .......................... 4
source file paths ...................................... 91, 335
source files
  adding to a project .................................. 36
  editing ................................................. 105
  managing in projects ................................ 89
__sourcePosition (C-SPY system macro) ............... 563
special function registers (SFR)
  definition of ......................................... 581
  description files .................................... 126
  in header files ..................................... 20
  using as assembler symbols ........................... 136
  specifying options for ............................... 453
  speed optimization .................................. 457
  src, arm (subdirectory) .............................. 20
  stack frames, definition of .......................... 582
  stack segment, definition of ....................... 582
  Stack window ......................................... 433
  using .................................................. 150
  Stack (IDE Options dialog box) ..................... 390
  stack.mac .............................................. 155
Stall processor on FIFO full (Trace Settings option) 177
standard libraries, definition of ..................... 582
Start address (linker option) .......................... 489
static objects, definition of ............................. 582
static overlay, definition of ............................ 582
statically allocated memory, definition of .......... 582
Statics window ............................................ 422
status bar .................................................. 320
stdin and stdout
  redirecting to C-SPY window ......................... 134
  redirecting to file .................................. 134
Step Into .................................................. 405
description .............................................. 130
example of ................................................ 48
Step into functions (IDE option) ..................... 389
Step Out .................................................... 405
description .............................................. 131
Step Over .................................................. 405
description .............................................. 130
step points, definition of ............................... 130
stepping .................................................... 130
example ................................................... 47
stepping, definition of ................................ 581
STL container expansion (IDE option) ............... 390
--stlink_interface (C-SPY command line option) .... 525
Stop build operation on (IDE Project options) ..... 387
Stop Debugging (button) ................................ 405
__strFind (C-SPY system macro) ......................... 563
Strict ISO/ANSI (compiler option) .................... 455
strings, text style in editor ............................ 107
structure value, definition of ........................ 582
ST-Link driver, features ................................ 14
ST-Link menu ............................................. 268
ST-Link (C-SPY options) ................................ 268
__subString (C-SPY system macro) ...................... 564
support, technical ......................................... 25
Suppress download (C-SPY option) .................... 245
Suppress these diagnostics (compiler option) ....... 463
Suppress these diagnostics (linker option) ......... 487
SWD interface ............................................. 256, 261–262, 268
information in Trace window ........................... 172
SWO clock (SWO Hardware Settings) ................. 182
SWO communication channel
  enabling ..................................................... 256, 261–262
  for timestamps in trace ................................. 178
SWO Settings dialog box (Trace) ....................... 178, 180
symbolic location, definition of .................... 582
Symbolic Memory window ............................... 416
collection menu ........................................... 417
toolbar ..................................................... 416
symbols
  See also user symbols
    defining in assembler ................................ 472
    defining in compiler ................................ 462
    defining in linker .................................... 486
    definition of .......................................... 582
    using in C-SPY expressions ......................... 135
Symbols window .......................................... 436
collection menu .......................................... 437
syntax coloring
  configuration files ................................... 20
  in editor ............................................... 107
Syntax Coloring (Editor colors and fonts option) .. 384
Syntax highlighting (editor option) .................. 378
syntax highlighting, in editor window ............... 108
system macros ........................................... 527

T
Tab Key Function (editor option) ....................... 377
Tab size (editor option) ................................ 377
Tab spacing (assembler option) ....................... 471
target system, definition of ........................... 118
Target (general options) ................................ 445
__targetDebuggerVersion (C-SPY system macro) ........ 564
$TARGET_BNAME$ (argument variable) ................ 367
$TARGET_BPATH$ (argument variable) ................. 367
$TARGET_DIR$ (argument variable) .................... 367
Treat these as warnings (linker option) .................. 48
Trigger at (Trace Filter dialog box) ..................... 292
Trigger at (Trace Start dialog box) ..................... 287
Trigger at (Trace Stop dialog box) ...................... 290
Trigger range (Breakpoints dialog) .................... 277
Trigger range (Data Log dialog box) ................... 279
Trigger range (Trace Filter dialog box) ................. 293
Trigger range (Trace Start dialog box) .................. 288
Trigger range (Trace Stop dialog box) .................. 290
tutor, arm (subdirectory) .................................. 20
type qualifiers, definition of .............................. 583
Type (External editor option) ............................ 381
type-checking .............................................. 16
typographic conventions ................................ xlvi

UBROF, definition of ..................................... 583
Undo (button) .............................................. 319
__unloadImage(C-SPY system macro) .................... 566
Update intervals (IDE option) ........................... 390
Use Code Templates (editor option) .................... 383
Use Custom Keyword File (editor option) .............. 383
Use External Editor (External editor option) ........... 381
Use flash loader (C-SPY option) ........................ 246
Use register filter (Register filter option) .............. 392
user application, definition of ........................... 118
User symbols are case sensitive (assembler option) .... 467
$USER_NAME$ (argument variable) ..................... 367

value expressions, definition of ........................ 583
variables
   effects of optimizations ................................ 137
   information, limitation on ................................ 137
   using in arguments .................................... 396
   using in C-SPY expressions ............................ 135

watching in C-SPY ........................................ 138
example .................................................. 48
variance (interrupt property) .......................... 219
definition of ............................................ 214
Vector Catch dialog box (JTAG menu) .................... 280
Verify download (C-SPY option) .......................... 245
version control systems ................................ 94
version number, of Embedded Workbench ................. 401
VFPv1 (General option) .................................. 446
VFPv2 (General option) .................................. 446
VFPv3 (General option) .................................. 446
VFPv4 (General option) .................................. 446
VFP9-S (General option) ................................ 446
View menu ............................................... 362
virtual address, definition of ............................ 583
virtual space, definition of .............................. 583
visualSTATE
   C-SPY plugin module for ................................ 8
   part of the tool chain ................................... 81
   project file .............................................. 23
   volatile storage, definition of ........................ 583
   von Neumann architecture, definition of ............... 583
   vsp (filename extension) ............................... 23

Warn when exceeding stack threshold (Stack option) .... 391
Warn when stack pointer is out of bounds (Stack option) 391
warnings
   compiler ................................................ 464
   linker ................................................... 488
warnings icon, in this guide ................................ xlvi
Watch window ............................................. 418
context menu ............................................. 419
using ...................................................... 138
watchpoints
   JTAG ..................................................... 282
   J-Link watchpoints .................................... 283
   setting .................................................. 48
Symbols

cancelAllInterrupts (C-SPY system macro) ............... 535
cancelInterrupt (C-SPY system macro) .................. 536
clearBreak (C-SPY system macro) ...................... 536
closeFile (C-SPY system macro) ....................... 536
delay (C-SPY system macro) .............................. 537
disableInterrupts (C-SPY system macro) ................ 537
driverType (C-SPY system macro) ....................... 538
emulatorSpeed (C-SPY system macro) ................. 538
emulatorStatusCheckOnRead (C-SPY system macro) .... 539
enableInterrupts (C-SPY system macro) ............... 540
evaluate (C-SPY system macro) ......................... 540
message (C-SPY system macro) .......................... 530
gdbserver_exec_command (C-SPY system macro) ....... 541
hwReset (C-SPY system macro) ......................... 541
hwResetRunToBp (C-SPY system macro) ............... 542
hwResetWithStrategy (C-SPY system macro) .......... 543
isBatchMode (C-SPY system macro) .................... 543
jlinkExecCommand (C-SPY system macro) .............. 544
jtagCommand (C-SPY system macro) .................... 544
jtagCP15IsPresent (C-SPY system macro) ............ 545
jtagCP15ReadReg (C-SPY system macro) ............. 545
jtagCP15WriteReg (C-SPY system macro) ............ 545
jtagData (C-SPY system macro) ......................... 545
jtagRawRead (C-SPY system macro) ..................... 546
jtagRawSync (C-SPY system macro) ..................... 547
jtagRawWrite (C-SPY system macro) ................... 548
jtagResetTRST (C-SPY system macro) ................ 548
loadImage (C-SPY system macro) ....................... 549
memoryRestore (C-SPY system macro) ................. 550
memorySave (C-SPY system macro) ..................... 550
message (C-SPY system macro) .......................... 530
openFile (C-SPY system macro) ......................... 551
orderInterrupt (C-SPY system macro) ................. 551
popSimulatorInterruptExecutingStack (C-SPY system macro) ........... 552
readFile (C-SPY system macro) ......................... 553
readFileByte (C-SPY system macro) ................... 554

x

cxl (filename extension) ................................. 23
XLINK, definition of .................................. 583

z

zero-initialized sections, definition of ................ 583
zero-overhead loop, definition of ..................... 584
zone
definition of ............................................. 584
in C-SPY .................................................. 149
__readMemoryByte (C-SPY system macro) .................... 554
__readMemory8 (C-SPY system macro) .................. 554
__readMemory16 (C-SPY system macro) ............ 555
__readMemory32 (C-SPY system macro) ............ 555
__registerMacroFile (C-SPY system macro) .... 555
__resetFile (C-SPY system macro) ..................... 556
__restoreSoftwareBreakpoint (C-SPY system macro) .... 556
__setCodeBreak (C-SPY system macro) ............ 557
__setDataBreak (C-SPY system macro) ............ 558
__setLogBreak (C-SPY system macro) ............. 559
__setSimBreak (C-SPY system macro) ............. 560
__setTraceStartBreak (C-SPY system macro) .... 561
__setTraceStopBreak (C-SPY system macro) .... 562
__smessage (C-SPY macro statement) .............. 530
__sourcePosition (C-SPY system macro) ........ 563
__strFind (C-SPY system macro) ............... 563
__subString (C-SPY system macro) .............. 564
targetDebuggerVersion (C-SPY system macro) .... 564
toLower (C-SPY system macro) .......... 564
toString (C-SPY system macro) .......... 565
toUpper (C-SPY system macro) ............. 565
__unloadImage (C-SPY system macro) .......... 566
__writeFile (C-SPY system macro) .............. 566
__writeFileByte (C-SPY system macro) .......... 567
__writeMemoryByte (C-SPY system macro) ........ 567
__writeMemory8 (C-SPY system macro) ........ 567
__writeMemory16 (C-SPY system macro) ........ 567
__writeMemory32 (C-SPY system macro) ........ 568
-B (C-SPY command line option) .................. 505
-p (C-SPY command line option) .................. 522
--backend (C-SPY command line option) .......... 505
--BE32 (C-SPY command line option) .......... 501
--BE8 (C-SPY command line option) .......... 501
--code_coverage (C-SPY command line option) .... 505
--cpu (C-SPY command line option) .......... 501
--cycles (C-SPY command line option) ........ 506
--device (C-SPY command line option) .......... 506
--disable_interrupts (C-SPY command line option) .... 506
--download_only (C-SPY command line option) .... 507
--drv_attach_to_program (C-SPY command line option) .... 501
--drv_cache_exceptions (C-SPY command line option) .... 501
--drv_communication (C-SPY command line option) .... 508
--drv_communication_log (C-SPY command line option) .... 510
--drv_default_breakpoint (C-SPY command line option) .... 511
--drv_reset_to_cpu_start (C-SPY command line option) .... 511
--drv_restore_breakpoints (C-SPY command line option) .... 512
--drv_suppress_download (C-SPY command line option) .... 502
--drv_vector_table_base (C-SPY command line option) .... 512
--drv_verify_download (C-SPY command line option) .... 502
--endian (C-SPY command line option) .......... 502
--flash_loader (C-SPY command line option) .... 513
--fp (C-SPY command line option) ........... 502
--gdbserv_exec_command (C-SPY command line option) .... 513
--jlink_device_select (C-SPY command line option) .... 514
--jlink_exec_command (C-SPY command line option) .... 514
--jlink_initial_speed (C-SPY command line option) .... 514
--jlink_interface (C-SPY command line option) .... 515
--jlink_ir_length (C-SPY command line option) .... 515
--jlink_reset_strategy (C-SPY command line option) .... 516
--jlink_speed (C-SPY command line option) .... 517
--lmifdi_speed (C-SPY command line option) .... 517
--macro (C-SPY command line option) ........ 520
--mac_handler_address (C-SPY command line option) .... 517
--mac_interface (C-SPY command line option) .... 518
--mac_jtag_device (C-SPY command line option) .... 518
--mac_multiple_targets (C-SPY command line option) .... 519
--mac_reset_pulse (C-SPY command line option) .... 519
--mac_set_temp_reg_buffer (C-SPY command line option) .... 520
--mac_speed (C-SPY command line option) .... 520
--mac_xscale_ir7 (C-SPY command line option) .... 521
--mapu (C-SPY command line option) ........ 521
--plugin (C-SPY command line option) ........ 522
--proc_stack_xxx (C-SPY command line option) .... 523
--rdi_allow_hardware_reset (C-SPY command line option) .... 523
--rdi_driver_dll (C-SPY command line option) .... 524
--rdi_heartbeat (C-SPY command line option) .... 503
--rdi_step_max_one (C-SPY command line option) . . . . . 524
--rdi_use_etm (C-SPY command line option) ............ 524
--semihosting (C-SPY command line option) .......... 525
--stlink_interface (C-SPY command line option) .... 525
--timeout (C-SPY command line option) .............. 526
#define options (linker options) ..................... 486
#define statement, in compiler ....................... 462
#define directive, definition of .................... 578
% stack usage threshold (Stack option). .......... 391
$CONFIG_NAME$ (argument variable) ................. 366
$CUR_DIR$ (argument variable) ...................... 366
$CUR_LINE$ (argument variable) ..................... 366
$DATE$ (argument variable) ......................... 366
$EXE_DIR$ (argument variable) ...................... 366
$FILE_DIR$ (argument variable) ..................... 367
$FILE_FNAME$ (argument variable) .................. 367
$FILE_PATH$ (argument variable) ................... 367
$OBJ_DIR$ (argument variable) ...................... 367
$PROJ_DIR$ (argument variable) .................... 367
$PROJ_FNAME$ (argument variable) .................. 367
$PROJ_PATH$ (argument variable) ................... 367
$TARGET_BNAME$ (argument variable) ............... 367
$TARGET_BPATH$ (argument variable) ............... 367
$TARGET_DIR$ (argument variable) .................. 367
$TARGET_FNAME$ (argument variable) ............... 367
$TARGET_PATH$ (argument variable) ................. 367
$TOOLKIT_DIR$ (argument variable) ................ 367
$USER_NAME$ (argument variable) ................... 367

IAR Embedded Workbench® IDE
612 User Guide for ARM