

IAR Embedded Workbench® IDE

User Guide

for Advanced RISC Machines Ltd's
ARM® Cores

COPYRIGHT NOTICE

Copyright © 1999–2008 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. μ C/OS-II is a trademark of Micrium, Inc. CMX-RTX is a trademark of CMX Systems, Inc. ThreadX is a trademark of Express Logic. RTX-C 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

Fourteenth edition: June 2008

Part number: UARM-14

This guide describes version 5.2x of the IAR Embedded Workbench® IDE for Advanced RISC Machines Ltd's ARM core family.

Internal reference: 5.3.0. ISUD.

Brief contents

Tables	xxv
Figures	xxix
Preface	xxxvii
Part 1. Product overview	1
Product introduction	3
Installed files	19
Part 2. Tutorials	27
Creating an application project	29
Debugging using the IAR C-SPY® Debugger	41
Mixing C and assembler modules	51
Using C++	55
Simulating an interrupt	59
Creating and using libraries	69
Part 3. Project management and building	73
The development environment	75
Managing projects	81
Building	91
Editing	99
Part 4. Debugging	109
The IAR C-SPY® Debugger	111

Executing your application	121
Working with variables and expressions	127
Using breakpoints	135
Monitoring memory and registers	143
Using the C-SPY® macro system	149
Analyzing your application	157
Part 5. The C-SPY® Simulator	163
Simulator-specific debugging	165
Simulating interrupts	185
Part 6. C-SPY hardware debugger systems	197
Introduction to C-SPY® hardware debugger systems	199
Hardware-specific debugging	213
Using flash loaders	255
Part 7. Reference information	261
IAR Embedded Workbench® IDE reference	263
C-SPY® reference	343
General options	379
Compiler options	387
Assembler options	401
Converter options	409
Custom build options	411
Build actions options	413

Linker options	415
Library builder options	427
Debugger options	429
The C-SPY Command Line Utility—cspybat	433
C-SPY® macros reference	459
Index	493

Contents

Tables	xxv
Figures	xxix
Preface	xxxvii
Who should read this guide	xxxvii
How to use this guide	xxxvii
What this guide contains	xxxviii
Other documentation	xli
Document conventions	xlii
Typographic conventions	xlii
Naming conventions	xliii
 Part I. 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
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
IAR C-SPY ROM-monitor driver	13

IAR C-SPY ANGEL debug monitor driver	14
IAR C/C++ Compiler	14
Features	14
Runtime environment	15
Documentation	15
IAR Assembler	16
Features	16
Documentation	16
IAR ILINK Linker and accompanying tools	16
Features	17
Documentation	17
Installed files	19
Directory structure	19
Root directory	19
The ARM directory	20
The common directory	21
File types	22
files with non-default filename extensions	24
Documentation	24
The user and reference guides	24
Online help	25
IAR on the web	26
 Part 2. Tutorials	 27
Creating an application project	29
Setting up a new project	29
Creating a Workspace window	29
Creating the new project	30
Adding files to the project	31
Setting project options	32
Compiling and linking the application	35
Compiling the source files	35

Viewing the list file	36
Linking the application	38
Viewing the map file	39
Debugging using the IAR C-SPY® Debugger	41
Debugging the application	41
Starting the debugger	41
Organizing the windows	41
Inspecting source statements	42
Inspecting variables	44
Setting and monitoring breakpoints	45
Debugging in disassembly mode	46
Monitoring memory	47
Viewing terminal I/O	48
Reaching program exit	49
Mixing C and assembler modules	51
Examining the calling convention	51
Adding an assembler module to the project	52
Setting up the project	53
Using C++	55
Creating a C++ application	55
Compiling and linking the C++ application	55
Setting a breakpoint and executing to it	56
Printing the Fibonacci numbers	58
Simulating an interrupt	59
Adding an interrupt handler	59
The application—a brief description	59
Writing an interrupt handler	59
Setting up the project	60
Setting up the simulation environment	60
Defining a C-SPY setup macro file	61
Setting C-SPY options	62

Building the project	63
Starting the simulator	63
Specifying a simulated interrupt	63
Setting an immediate breakpoint	64
Simulating the interrupt	65
Executing the application	65
Using macros for interrupts and breakpoints	67
Creating and using libraries	69
Using libraries	69
Creating a new project	70
Creating a library project	70
Using the library in your application project	70
 Part 3. Project management and building	73
The development environment	75
The IAR Embedded Workbench IDE	75
The tool chain	75
Running the IDE	76
Exiting	77
Customizing the environment	77
Organizing the windows on the screen	77
Customizing the IDE	78
Invoking external tools	79
Managing projects	81
The project model	81
How projects are organized	81
Creating and managing workspaces	83
Navigating project files	85
Viewing the workspace	86
Displaying browse information	87
Source code control	88
Interacting with source code control systems	88

Building	91
Building your application	91
Setting options	91
Building a project	93
Building multiple configurations in a batch	93
Using pre- and post-build actions	94
Correcting errors found during build	94
Building from the command line	95
Extending the tool chain	95
Tools that can be added to the tool chain	96
Adding an external tool	96
Editing	99
Using the IAR Embedded Workbench editor	99
Editing a file	99
Using and adding code templates	103
Navigating in and between files	105
Searching	105
Customizing the editor environment	105
Using an external editor	106
 Part 4. Debugging	 109
The IAR C-SPY® Debugger	111
Debugger concepts	111
C-SPY and target systems	111
Debugger	112
Target system	112
User application	112
C-SPY Debugger systems	112
ROM-monitor program	113
Third-party debuggers	113
The C-SPY environment	113
An integrated environment	113

Setting up C-SPY	114
Choosing a debug driver	114
Executing from reset	114
Using a setup macro file	115
Selecting a device description file	115
Loading plugin modules	116
Starting C-SPY	116
Executable files built outside of the IDE	117
Redirecting debugger output to a file	117
Adapting C-SPY to target hardware	117
Device description file	118
Remapping memory	119
Executing your application	121
Source and disassembly mode debugging	121
Executing	121
Step	122
Go	123
Run to Cursor	124
Highlighting	124
Using breakpoints to stop	124
Using the Break button to stop	125
Stop at program exit	125
Call stack information	125
Terminal input and output	126
Working with variables and expressions	127
C-SPY expressions	127
C symbols	127
Assembler symbols	128
Macro functions	128
Macro variables	129
Limitations on variable information	129
Effects of optimizations	129

Viewing variables and expressions	130
Working with the windows	130
Using the trace system	131
Viewing assembler variables	132
Using breakpoints	135
The breakpoint system	135
Defining breakpoints	135
Toggling a simple code breakpoint	136
Setting a breakpoint in the Memory window	137
Defining breakpoints using the dialog box	137
Defining breakpoints using system macros	138
Viewing all breakpoints	139
Using the Breakpoint Usage dialog box	140
Monitoring memory and registers	143
Memory addressing	143
Windows for monitoring memory and registers	144
Using the Memory window	145
Using the Stack window	145
Working with registers	147
Using the C-SPY® macro system	149
The macro system	149
The macro language	150
The macro file	150
Setup macro functions	151
Using C-SPY macros	152
Using the Macro Configuration dialog box	152
Registering and executing using setup macros and setup files	153
Executing macros using Quick Watch	154
Executing a macro by connecting it to a breakpoint	155

Analyzing your application	157
Function-level profiling	157
Using the profiler	158
Code coverage	160
Using Code Coverage	160
 Part 5. The C-SPY® Simulator	163
Simulator-specific debugging	165
The C-SPY Simulator introduction	165
Features	165
Selecting the simulator driver	165
Simulator-specific menus	166
Simulator menu	166
Pipeline Trace window	167
Memory Access Configuration	168
Memory access costs dialog box	169
Using the trace system in the simulator	170
Trace window	170
Trace toolbar	171
Function Trace window	172
Trace Expressions window	173
Find In Trace window	174
Find in Trace dialog box	174
Memory access checking	176
Memory Access setup dialog box	176
Edit Memory Access dialog box	179
Using breakpoints in the simulator	179
Data breakpoints	180
Immediate breakpoints	182
Breakpoint Usage dialog box	184

Simulating interrupts	185
The C-SPY interrupt simulation system	185
Interrupt characteristics	186
Interrupt simulation states	187
Using the interrupt simulation system	188
Target-adapting the interrupt simulation system	189
Interrupt Setup dialog box	189
Edit Interrupt dialog box	190
Forced interrupt window	192
C-SPY system macros for interrupts	192
Interrupt Log window	194
Simulating a simple interrupt	195
 Part 6. C-SPY hardware debugger systems	197
Introduction to C-SPY® hardware debugger systems	199
The IAR C-SPY hardware debugger systems	199
Differences between the C-SPY drivers	200
Getting started	200
The IAR C-SPY Angel debug monitor driver	201
The IAR C-SPY GDB Server driver	203
Configuring the OpenOCD Server	204
The IAR C-SPY ROM-monitor driver	204
The IAR C-SPY J-Link/J-Trace drivers	205
Installing the J-Link USB driver	206
The IAR C-SPY LMI FTDI driver	207
Installing the FTDI USB driver	208
The IAR C-SPY Macraigor driver	208
The IAR C-SPY RDI driver	209
An overview of the debugger startup	211
Debugging code in flash	211
Debugging code in RAM	212

Hardware-specific debugging	213
C-SPY options for debugging using hardware systems	213
Download	215
Debugging using the Angel debug monitor driver	216
Angel	216
Debugging using the IAR C-SPY GDB Server driver	217
GDB Server	218
The GDB Server menu	218
Debugging using the IAR C-SPY ROM-monitor driver	219
IAR ROM-monitor	219
Debugging using the IAR C-SPY J-Link/J-Trace driver	220
Setup	221
Connection	224
The J-Link menu	225
SWO Setup dialog box	226
Live watch and use of DCC	228
Debugging using the IAR C-SPY LMI FTDI driver	229
Setup	229
The LMI FTDI menu	230
Debugging using the IAR C-SPY Macraigor driver	230
Macraigor	231
The Macraigor JTAG menu	233
Debugging using the RDI driver	233
RDI	233
RDI menu	235
Debugging using third-party drivers	236
Third-Party Driver	236
Using the trace system in hardware debugger systems	237
Trace Setup dialog box	238
Trace Save dialog box	240
Trace window	240
Trace toolbar	242

Using breakpoints in the hardware debugger systems	243
Available number of breakpoints	243
Breakpoints options	243
Code breakpoints dialog box	245
Data breakpoints dialog box	246
Data Log breakpoints dialog box	248
Breakpoint Usage dialog box	250
Breakpoints on vectors	250
Setting breakpoints in __ramfunc declared functions	251
Using JTAG watchpoints	251
The Watchpoint mechanism	252
JTAG watchpoints dialog box	252
Using flash loaders	255
The flash loader	255
Setting up the flash loader(s)	255
The flash loading mechanism	256
Build considerations	256
Flash Loader Overview dialog box	257
Flash Loader Configuration dialog box	258
 Part 7. Reference information	261
IAR Embedded Workbench® IDE reference	263
Windows	263
IAR Embedded Workbench IDE window	264
Workspace window	266
Editor window	274
Source Browser window	280
Breakpoints window	282
Build window	288
Find in Files window	288
Tool Output window	289
Debug Log window	290

Menus	291
File menu	291
Edit menu	294
View menu	302
Project menu	304
Tools menu	313
Common fonts options	314
Key Bindings options	315
Language options	316
Editor options	317
Configure Auto Indent dialog box	319
External Editor options	320
Editor Setup Files options	322
Editor Colors and Fonts options	323
Messages options	324
Project options	326
Source Code Control options	327
Debugger options	328
Stack options	330
Register Filter options	332
Terminal I/O options	333
Configure Tools dialog box	334
Filename Extensions dialog box	336
Filename Extension Overrides dialog box	337
Edit Filename Extensions dialog box	337
Configure Viewers dialog box	338
Edit Viewer Extensions dialog box	338
Window menu	339
Help menu	340
Embedded Workbench Startup dialog box	340
C-SPY® reference	343
C-SPY windows	343
Editing in C-SPY windows	344

C-SPY Debugger main window	344
Disassembly window	346
Memory window	348
Fill dialog box	351
Memory Save dialog box	352
Memory Restore dialog box	353
Symbolic Memory window	354
Register window	356
Watch window	356
Locals window	358
Auto window	359
Live Watch window	359
Quick Watch window	360
Statics window	360
Select Statics dialog box	362
Call Stack window	363
Terminal I/O window	364
Code Coverage window	365
Profiling window	367
Stack window	368
Symbols window	372
C-SPY menus	373
Debug menu	373
Disassembly menu	378
General options	379
Target	379
Processor variant	379
Endian mode	380
FPU	380
Output	381
Output file	381
Output directories	381

Library Configuration	382
Library	382
Configuration file	382
Library low-level interface implementation	383
Library Options	384
Printf formatter	384
Scanf formatter	384
Buffered terminal output	384
MISRA C	385
Compiler options	387
Multi-file compilation	387
Language	388
Language	388
Require prototypes	389
Language conformance	389
Plain 'char' is	389
Enable multibyte support	390
Code	390
Generate interwork code	390
Processor mode	390
Optimizations	391
Optimizations	391
Output	392
Generate debug information	393
Code section name	393
List	394
Output list file	394
Output assembler file	394
Preprocessor	395
Ignore standard include directories	395
Additional include directories	395
Preinclude file	396
Defined symbols	396

Preprocessor output to file	396
Diagnostics	396
Enable remarks	397
Suppress these diagnostics	397
Treat these as remarks	397
Treat these as warnings	398
Treat these as errors	398
Treat all warnings as errors	398
MISRA C	398
Extra Options	399
Use command line options	399
Assembler options	401
Language	401
User symbols are case sensitive	401
Enable multibyte support	401
Macro quote characters	402
Allow alternative register names, mnemonics and operands	402
Output	403
Generate debug information	403
List	404
Include header	404
Include listing	404
Include cross-reference	405
Lines/page	405
Tab spacing	405
Preprocessor	405
Ignore standard include directories	406
Additional include directories	406
Defined symbols	406
Diagnostics	407
Max number of errors	407
Extra Options	408
Use command line options	408

Converter options	409
Output	409
Promable output format	409
Output file	409
Custom build options	411
Custom Tool Configuration	411
Build actions options	413
Build Actions Configuration	413
Pre-build command line	413
Post-build command line	413
Linker options	415
Config	415
Linker configuration file	415
Configuration file symbol definitions	416
Linker configuration file editor	416
Library	417
Automatic runtime library selection	417
Additional libraries	417
Override default program entry	417
Input	418
Keep symbols	418
Raw binary image	418
Output	419
Output file	419
Include debug information in output	419
List	420
Generate linker map file	420
Generate log	420
#define	421
Defined symbols	421
Diagnostics	421
Enable remarks	422

Suppress these diagnostics	422
Treat these as remarks	422
Treat these as warnings	423
Treat these as errors	423
Treat all warnings as errors	423
Checksum	423
Fill unused code memory	424
Extra Options	425
Use command line options	425
Library builder options	427
Output	427
Debugger options	429
Setup	429
Driver	429
Run to	430
Setup macros	430
Device description file	430
Download	431
Extra Options	431
Use command line options	431
Plugins	431
The C-SPY Command Line Utility—cspybat	433
Using C-SPY in batch mode	433
Invocation syntax	433
Output	434
Using an automatically generated batch file	434
C-SPY command line options	434
Descriptions of C-SPY command line options	438
C-SPY® macros reference	459
The macro language	459
Macro functions	459

Predefined system macro functions	459
Macro variables	460
Macro statements	461
Formatted output	462
Setup macro functions summary	464
C-SPY system macros summary	465
Description of C-SPY system macros	467
Index	493

Tables

1: Typographic conventions used in this guide	xlii
2: Naming conventions used in this guide	xliii
3: File types	22
4: General settings for project1	33
5: Compiler options for project1	34
6: Compiler options for project2	52
7: Project options for Embedded C++ tutorial	56
8: Interrupts dialog box	63
9: Breakpoints dialog box	65
10: General options for a library project	70
11: Command shells	80
12: iarbuild.exe command line options	95
13: C-SPY assembler symbols expressions	128
14: Handling name conflicts between hardware registers and assembler labels	128
15: Project options for enabling profiling	158
16: Project options for enabling code coverage	160
17: Description of Simulator menu commands	166
18: Pipeline window information	168
19: Trace window columns	170
20: Trace toolbar commands	171
21: Toolbar buttons in the Trace Expressions window	173
22: Function buttons in the Memory Access Setup dialog box	178
23: Memory Access types	181
24: Breakpoint conditions	182
25: Memory Access types	183
26: Interrupt statuses	187
27: Characteristics of a forced interrupt	192
28: Description of the Interrupt Log window	194
29: Timer interrupt settings	196
30: Differences between available C-SPY drivers	200
31: Available quickstart reference information	200

32: Commands on the GDB Server menu	219
33: Commands on the J-Link menu	225
34: Commands on the RDI menu	230
35: Commands on the JTAG menu	233
36: Catching exceptions	235
37: Commands on the RDI menu	236
38: Trace window columns	241
39: Trace window columns when using SWO	241
40: Trace toolbar commands	242
41: Breakpoint conditions	246
42: Memory Access types	247
43: Data access types	253
44: CPU modes	254
45: Break conditions	254
46: Function buttons in the Flash Loader Overview dialog box	257
47: IDE menu bar	264
48: Workspace window context menu commands	268
49: Description of source code control commands	269
50: Description of source code control states	270
51: Description of commands on the editor window context menu	276
52: Editor keyboard commands for insertion point navigation	278
53: Editor keyboard commands for scrolling	279
54: Editor keyboard commands for selecting text	279
55: Information in Source Browser window	280
56: Source Browser window context menu commands	281
57: Breakpoints window context menu commands	283
58: Breakpoint conditions	285
59: Log breakpoint conditions	286
60: Location types	287
61: File menu commands	292
62: Edit menu commands	294
63: Find dialog box options	297
64: Replace dialog box options	298
65: Incremental Search function buttons	301

66: View menu commands	302
67: Project menu commands	304
68: Argument variables	306
69: Configurations for project dialog box options	307
70: New Configuration dialog box options	308
71: Description of Create New Project dialog box	309
72: Project option categories	309
73: Description of the Batch Build dialog box	311
74: Description of the Edit Batch Build dialog box	312
75: Tools menu commands	313
76: Project IDE options	326
77: Register Filter options	332
78: Configure Tools dialog box options	334
79: Command shells	336
80: Window menu commands	339
81: Editing in C-SPY windows	344
82: C-SPY menu	344
83: Disassembly window toolbar	346
84: Disassembly context menu commands	347
85: Memory window operations	349
86: Commands on the memory window context menu	350
87: Fill dialog box options	351
88: Memory fill operations	352
89: Symbolic Memory window toolbar	354
90: Symbolic memory window columns	354
91: Commands on the Symbolic Memory window context menu	355
92: Watch window context menu commands	357
93: Effects of display format setting on different types of expressions	358
94: Symbolic memory window columns	361
95: Statics window context menu commands	362
96: Profiling window columns	368
97: Stack window columns	370
98: Symbols window columns	372
99: Commands on the Symbols window context menu	373

100: Debug menu commands	373
101: Log file options	377
102: Description of Disassembly menu commands	378
103: Assembler list file options	404
104: Linker checksum algorithms	424
105: C-SPY driver options	430
106: cspybat parameters	433
107: Examples of C-SPY macro variables	460
108: C-SPY setup macros	464
109: Summary of system macros	465
110: __cancelInterrupt return values	467
111: __disableInterrupts return values	468
112: __driverType return values	469
113: __emulatorSpeed return values	469
114: __enableInterrupts return values	471
115: __evaluate return values	471
116: __hwReset return values	472
117: __hwReset return values	473
118: __openFile return values	478
119: __openFile return values	478
120: __readFile return values	480
121: __setCodeBreak return values	484
122: __setDataBreak return values	485
123: __setSimBreak return values	486
124: __sourcePosition return values	487

Figures

1: Directory structure	19
2: Create New Project dialog box	30
3: Workspace window	30
4: New Workspace dialog box	31
5: Adding files to project1	32
6: Setting general options	33
7: Setting compiler options	34
8: Compilation message	35
9: Workspace window after compilation	36
10: Setting the option Scan for Changed Files	37
11: Linker options dialog box for project1	38
12: The C-SPY Debugger main window	42
13: Stepping in C-SPY	43
14: Using Step Into in C-SPY	43
15: Inspecting variables in the Auto window	44
16: Watching variables in the Watch window	45
17: Setting breakpoints	46
18: Debugging in disassembly mode	47
19: Monitoring memory	47
20: Displaying memory contents as 16-bit units	48
21: Output from the I/O operations	49
22: Reaching program exit in C-SPY	49
23: Assembler settings for creating a list file	53
24: Project2 output in terminal I/O window	54
25: Setting a breakpoint in CPPTutor.cpp	56
26: Inspecting the function calls	57
27: Printing Fibonacci sequences	58
28: Specifying setup macro file	62
29: Inspecting the interrupt settings	64
30: Register window	66
31: Printing the Fibonacci values in the Terminal I/O window	67

32: IAR Embedded Workbench IDE window	76
33: Configure Tools dialog box	79
34: Customized Tools menu	80
35: Examples of workspaces and projects	82
36: Displaying a project in the Workspace window	86
37: Workspace window—an overview	87
38: General options	92
39: Editor window	100
40: Parentheses matching in editor window	103
41: Editor window status bar	103
42: Editor window code template menu	104
43: Specifying external command line editor	106
44: External editor DDE settings	107
45: C-SPY and target systems	112
46: C-SPY highlighting source location	124
47: Viewing assembler variables in the Watch window	133
48: Breakpoint icons	136
49: Breakpoint Usage dialog box	140
50: Zones in C-SPY	143
51: Memory window	145
52: Stack window	146
53: Register window	147
54: Register Filter page	148
55: Macro Configuration dialog box	153
56: Quick Watch window	155
57: Profiling window	158
58: Graphs in Profiling window	159
59: Function details window	159
60: Code Coverage window	161
61: Simulator menu	166
62: Pipeline Trace window	167
63: Memory Access Configuration window	168
64: Memory access costs dialog box	169
65: Trace window	170

66: Trace toolbar	171
67: Function Trace window	172
68: Trace Expressions window	173
69: Find In Trace window	174
70: Find in Trace dialog box	175
71: Memory Access Setup dialog box	177
72: Edit Memory Access dialog box	179
73: Data breakpoints dialog box	181
74: Immediate breakpoints page	183
75: Breakpoint Usage dialog box	184
76: Simulated interrupt configuration	186
77: Simulation states - example 1	187
78: Simulation states - example 2	188
79: Interrupt Setup dialog box	189
80: Edit Interrupt dialog box	190
81: Forced Interrupt window	192
82: Interrupt Log window	194
83: C-SPY Angel debug monitor communication overview	202
84: C-SPY GDB Server communication overview	203
85: C-SPY ROM-monitor communication overview	205
86: C-SPY J-Link communication overview	206
87: C-SPY Macraigor communication overview	209
88: C-SPY RDI communication overview	210
89: Debugger startup when debugging code in flash	211
90: Debugger startup when debugging code in RAM	212
91: C-SPY Download options	215
92: C-SPY Angel options	216
93: GDB Server options	218
94: The GDB Server menu	218
95: IAR C-SPY ROM-monitor options	219
96: C-SPY J-Link/J-Trace Setup options	221
97: C-SPY J-Link/J-Trace Connection options	224
98: The J-Link menu	225
99: SWO Setup dialog box	226

100: C-SPY LMI FTDI Setup options	229
101: The LMI FTDI menu	230
102: C-SPY Macraigor options	231
103: The Macraigor JTAG menu	233
104: C-SPY RDI options	234
105: The RDI menu	235
106: C-SPY Third-Party Driver options	237
107: Trace Setup dialog box	238
108: Trace Save dialog box	240
109: ETM Trace View window	240
110: Trace toolbar	242
111: Breakpoints options	244
112: Code breakpoints page	245
113: Data breakpoints dialog box	247
114: Data Log breakpoints dialog box	249
115: Breakpoint Usage dialog box	250
116: The Vector Catch dialog box	251
117: JTAG Watchpoints dialog box	252
118: Flash Loader Overview dialog box	257
119: Flash Loader Configuration dialog box	258
120: IAR Embedded Workbench IDE window	264
121: IDE toolbar	265
122: IAR Embedded Workbench IDE window status bar	266
123: Workspace window	266
124: Workspace window context menu	268
125: Source Code Control menu	269
126: Select Source Code Control Provider dialog box	271
127: Check In Files dialog box	272
128: Check Out File dialog box	273
129: Editor window	274
130: Editor window tab context menu	275
131: Editor window context menu	276
132: Source Browser window	280
133: Source Browser window context menu	281

134: Breakpoints window	282
135: Breakpoints window context menu	282
136: Code breakpoints page	284
137: Log breakpoints page	285
138: Enter Location dialog box	287
139: Build window (message window)	288
140: Build window context menu	288
141: Find in Files window (message window)	289
142: Find in Files window context menu	289
143: Tool Output window (message window)	290
144: Tool Output window context menu	290
145: Debug Log window (message window)	290
146: Debug Log window context menu	291
147: File menu	292
148: Edit menu	294
149: Find in Files dialog box	299
150: Incremental Search dialog box	300
151: Template dialog box	301
152: View menu	302
153: Project menu	304
154: Configurations for project dialog box	307
155: New Configuration dialog box	308
156: Create New Project dialog box	309
157: Batch Build dialog box	311
158: Edit Batch Build dialog box	312
159: Tools menu	313
160: Common Fonts options	314
161: Key Bindings options	315
162: Language options	316
163: Editor options	317
164: Configure Auto Indent dialog box	319
165: External Editor options	320
166: Editor Setup Files options	322
167: Editor Colors and Fonts options	323

168: Messages option	324
169: Message dialog box containing a Don't show again option	325
170: Project options	326
171: Source Code Control options	327
172: Debugger options	328
173: Stack options	330
174: Register Filter options	332
175: Terminal I/O options	333
176: Configure Tools dialog box	334
177: Customized Tools menu	335
178: Filename Extensions dialog box	336
179: Filename Extension Overrides dialog box	337
180: Edit Filename Extensions dialog box	337
181: Configure Viewers dialog box	338
182: Edit Viewer Extensions dialog box	338
183: Window menu	339
184: Embedded Workbench Startup dialog box	340
185: C-SPY debug toolbar	345
186: C-SPY Disassembly window	346
187: Disassembly window context menu	347
188: Memory window	349
189: Memory window context menu	350
190: Fill dialog box	351
191: Memory Save dialog box	352
192: Memory Restore dialog box	353
193: Symbolic Memory window	354
194: Symbolic Memory window context menu	355
195: Register window	356
196: Watch window	357
197: Watch window context menu	357
198: Locals window	358
199: Auto window	359
200: Live Watch window	359
201: Quick Watch window	360

202: Statics window	361
203: Statics window context menu	361
204: Select Statics dialog box	362
205: Call Stack window	363
206: Call Stack window context menu	363
207: Terminal I/O window	364
208: Ctrl codes menu	364
209: Input Mode dialog box	365
210: Code Coverage window	365
211: Code coverage context menu	366
212: Profiling window	367
213: Profiling context menu	367
214: Stack window	369
215: Stack window context menu	370
216: Symbols window	372
217: Symbols window context menu	372
218: Debug menu	373
219: Autostep settings dialog box	375
220: Macro Configuration dialog box	376
221: Log File dialog box	377
222: Terminal I/O Log File dialog box	378
223: Disassembly menu	378
224: Target options	379
225: Output options	381
226: Library Configuration options	382
227: Library Options page	384
228: Multi-file Compilation	387
229: Compiler language options	388
230: Compiler code options	390
231: Compiler optimizations options	391
232: Compiler output options	392
233: Compiler list file options	394
234: Compiler preprocessor options	395
235: Compiler diagnostics options	397

236: Extra Options page for the compiler	399
237: Assembler language options	401
238: Choosing macro quote characters	402
239: Assembler output options	403
240: Assembler list file options	404
241: Assembler preprocessor options	405
242: Assembler diagnostics options	407
243: Extra Options page for the assembler	408
244: Converter output file options	409
245: Custom tool options	411
246: Build actions options	413
247: Linker configuration options	415
248: Linker configuration file editor	416
249: Library page	417
250: Linker input file options	418
251: Linker output file options	419
252: Linker diagnostics options	420
253: Linker defined symbols options	421
254: Linker diagnostics options	422
255: Linker checksum and fill options	423
256: Extra Options page for the linker	425
257: Library builder output options	427
258: Generic C-SPY options	429
259: Extra Options page for C-SPY	431
260: C-SPY plugin options	432

Preface

Welcome to the *IAR Embedded Workbench® IDE User Guide*. The purpose of this guide is to help you fully utilize 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

You should 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 a 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 start by reading *Part 1. Product overview* to give you an overview of the tools and the functions that the IDE can offer.

If you already have had some experience using IAR Embedded Workbench, but need refreshing on how to work with the IAR development tools, *Part 2. Tutorials* is a good place to begin. The process of managing projects and building, as well as editing, can be found in *Part 3. Project management and building*, page 73, whereas information about how to use C-SPY can be found in *Part 4. Debugging*, page 109.

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 **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

Below 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 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 development tools.

Part 2. Tutorials

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

- *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 the compiler can be used 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 you can add an interrupt handler to the project and how this interrupt can be simulated 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 helps you handle different versions of your applications.
- *Building* discusses the process of building your application.
- *Editing* contains detailed descriptions about 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 setup, start, and configure the debugger to reflect the target hardware.
- *Executing your application* describes how you 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 different methods for monitoring variables and expressions.
- *Using breakpoints* describes the breakpoint system and the different 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, for what purposes these features can be used, and how to use them.

- *Analyzing your application* presents facilities for analyzing your application.

Part 5. The 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 these 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, code, optimizations, 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 Reference Guide*.

All of these guides are delivered in hypertext PDF or HTML format on the installation media. Some of them are also delivered as printed books. 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**, contains information and news about the ARM cores.
- The IAR Systems web site, **www.iar.com**, holds application notes and other product information.
- The web site **www.SevensAndNines.com**, maintained by IAR Systems, provides an online user community and resource site for ARM developers.
- 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 5.0\arm\doc`.

TYPOGRAPHIC CONVENTIONS

This guide uses the following typographic conventions:





Style	Used for
computer	<ul style="list-style-type: none">• Source code examples and file paths.• Text on the command line.• Binary, hexadecimal, and octal numbers.
<i>parameter</i>	A placeholder for an actual value used as a parameter, for example <i>filename.h</i> where <i>filename</i> represents the name of the file.
[option]	An optional part of a command, where [] is part of the described syntax.
{option}	A mandatory part of a command, where { } is part of the described syntax.
[option]	An optional part of a command.
{option}	A mandatory part of a command.
a b c	Alternatives in a command.
bold	Names of menus, menu commands, buttons, and dialog boxes that appear on the screen.
<i>italic</i>	<ul style="list-style-type: none">• A cross-reference within this guide or to another guide.• Emphasis.
...	An ellipsis indicates that the previous item can be repeated an arbitrary number of times.
	Identifies instructions specific to the IAR Embedded Workbench® IDE interface.
	Identifies instructions specific to the command line interface.
	Identifies helpful tips and programming hints.
	Identifies warnings.

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:

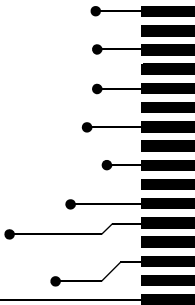
Brand name	Generic term
IAR Embedded Workbench® for ARM	IAR Embedded Workbench®
IAR Embedded Workbench® IDE for ARM	the IDE
IAR C-SPY® Debugger for ARM	C-SPY, the debugger
IAR C/C++ Compiler™ for ARM	the compiler
IAR Assembler™ for ARM	the assembler
IAR ILINK™ Linker	ILINK, the linker
IAR DLIB Library™	the DLIB library

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
- IAR C-SPY® Debugger, a state-of-the-art high-level language debugger.

IAR Embedded Workbench is available for a large number of 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 thus a significant reduction of the development time can be achieved by using the IAR Systems tools. We call this concept “Different Architectures. One Solution.”

If you want detailed information about supported target processors, contact your software distributor or your IAR 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 can be 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 different 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. The following 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. The following 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). There is also help and hypertext PDF versions of the user documentation 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, corrections can be made 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.

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 functions provided by the target system, 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 provided by 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 calls—inside expressions, as well as function calls being 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 host PC 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 can be provided by IAR Systems, as well as by third-party suppliers. Example 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 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. The following operating systems are currently supported:

- IAR PowerPac RTOS
- CMX CMX-RTX and CMX-Tiny+ real-time operating systems
- Micrium μ C/OS-II
- Express Logic ThreadX
- RTX 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, as well as 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. The 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 the following 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
- Luminary FTDI JTAG interface (for Cortex devices only).

Contact your software distributor or IAR 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 111. In the following sections you can find general descriptions of the different drivers.

IAR C-SPY SIMULATOR

The C-SPY simulator driver simulates the functions of the target processor entirely in software. With this driver, the program logic can be debugged 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. The 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 J-Link 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 J-Link 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 RAVEN, WIGGLER, 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 or Ethernet
- 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.

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/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).
- Placement of classes in different memory types
- 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.

There are several mechanisms available for customizing the runtime environment and the runtime libraries. 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 the following 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.

DOCUMENTATION

The assembler is documented in the *ARM® IAR Assembler Reference Guide*.

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, there are various tools 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 different 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 `x:\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.

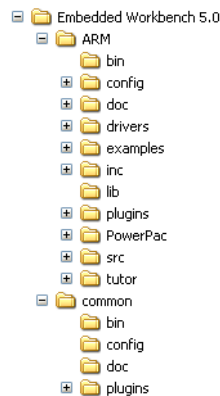


Figure 1: Directory structure

Note: The installation path can be different from the one shown above depending on previously installed IAR products, and on your preferences.

THE ARM DIRECTORY

The `arm` directory contains all product-specific subdirectories.

The `arm\bin` directory

The `arm\bin` subdirectory contains executable files for ARM-specific components, such as the compiler, the assembler, the linker, and the C-SPY® drivers. This directory also contains executable files for the GNU binary utilities.

The `arm\config` directory

The `arm\config` subdirectory contains files used for configuring the development environment and projects, for example:

- Linker configuration files (*.icf)
- C-SPY device description files (*.ddf)
- Device selection files (*.i79, *.menu)
- Flash loader applications for different devices (*.out)
- Syntax coloring configuration files (*.cfg)
- Project templates for both application and library projects (*.ewp), and for the library projects, the corresponding library configuration files.

The `arm\doc` directory

The `arm\doc` subdirectory contains release notes with recent additional information about the ARM tools. We recommend that you read all of these files. The directory also contains online hypertext versions in hypertext PDF format of this user guide, and of the ARM reference guides, as well as online help files (*.chm). Documentation for the GNU binary utilities is available in html format.

The `arm\drivers` directory

The `arm\drivers` subdirectory contains low-level device drivers, typically USB drivers, required by the C-SPY drivers.

The `arm\examples` directory

The `arm\examples` subdirectory contains files related to example projects, which can be opened from the **Startup Screen** dialog box.

The arm\inc directory

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 defining special function registers (SFRs); these files are used by both the compiler and the assembler.

The arm\lib directory

The `arm\lib` subdirectory holds prebuilt libraries and the corresponding library configuration files, used by the compiler.

The arm\plugins directory

The `arm\plugins` subdirectory contains executable files and description files for components that can be loaded as plugin modules.

The arm\powerpac directory

The `arm\powerpac` subdirectory contains files related to the add-on product IAR PowerPac.

The arm\src directory

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.

The arm\tutor directory

The `arm\tutor` subdirectory contains the files used for the tutorials in this guide.

THE COMMON DIRECTORY

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

The common\bin directory

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.

The common\config directory

The `common\config` subdirectory contains files used by IDE for holding settings in the development environment.

The common\doc directory

The `common\doc` subdirectory contains readme files 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. The directory also contains an online version in PDF format of the *IAR Linker and Library Tools Reference Guide*.

The common\plugins directory

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.

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:

Ext.	Type of file	Output from	Input to
out	Target application	ILINK	EPROM, C-SPY, etc.
asm	Assembler source code	Text editor	Assembler
bat	Windows command batch file	C-SPY	Windows
c	C source code	Text editor	Compiler
cfg	Syntax coloring configuration	Text editor	IDE
chm	Online help system	--	IDE
cpp	C++ source code	Text editor	Compiler
out	Target application with debug information	ILINK	C-SPY and other symbolic debuggers
dat	Macros for formatting of STL containers	IDE	IDE
dbgt	Debugger desktop settings	C-SPY	C-SPY
ddf	Device description file	Text editor	C-SPY
dep	Dependency information	IDE	IDE
dni	Debugger initialization file	C-SPY	C-SPY

Table 3: File types

Ext.	Type of file	Output from	Input to
ewd	Project settings for C-SPY	IDE	IDE
ewp	IAR Embedded Workbench project (current version)	DE	IDE
ewplugin	IDE description file for plugin modules	--	IDE
eww	Workspace file	IDE	IDE
fmt	Formatting information for the Locals and Watch windows	IDE	IDE
h	C/C++ or assembler header source	Text editor	Compiler or assembler #include
helpfiles	Help menu configuration file	Text editor	IDE
html, htm	HTML document	Text editor	IDE
i	Preprocessed source	Compiler	Compiler
i79	Device selection file	Text editor	IDE
icf	Linker configuration file	Text editor	ILINK linker
inc	Assembler header source	Text editor	Assembler #include
ini	Project configuration	IDE	—
log	Log information	IDE	—
lst	List output	Compiler and assembler	—
mac	C-SPY macro definition	Text editor	C-SPY
menu	Device selection file	Text editor	IDE
pbd	Source browse information	IDE	IDE
pbi	Source browse information	IDE	IDE
pew	IAR Embedded Workbench project (old project format)	IDE	IDE
prj	IAR Embedded Workbench project (old project format)	IDE	IDE
o	Object module	Compiler and assembler	ILINK
s	ARM assembler source code	Text editor	Assembler

Table 3: File types (Continued)

Ext.	Type of file	Output from	Input to
vsp	visualSTATE project files	IAR visualSTATE Designer	IAR visualSTATE Designer and IAR Embedded Workbench IDE
wsdt	Workspace desktop settings	IDE	IDE
xcl	Extended command line	Text editor	Assembler, compiler, linker

Table 3: 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 336.



On the command line, you can override the default filename extension by including an explicit extension when specifying 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

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

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. You should 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 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. There is also 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 ON THE WEB

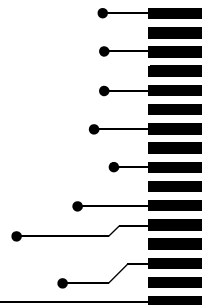
The latest news from IAR Systems can be found at the web site **www.iar.com**, available from the **Help** menu in the IDE. Visit it for information about:

- 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:

- 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.





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.

The development cycle continues in the next chapter, see *Debugging using the IAR C-SPY® Debugger*, page 41.

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.

We recommend that you create a specific directory where you can store all your project files. In this tutorial we call the directory `projects`. 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.

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

CREATING A WORKSPACE WINDOW

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

- 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.

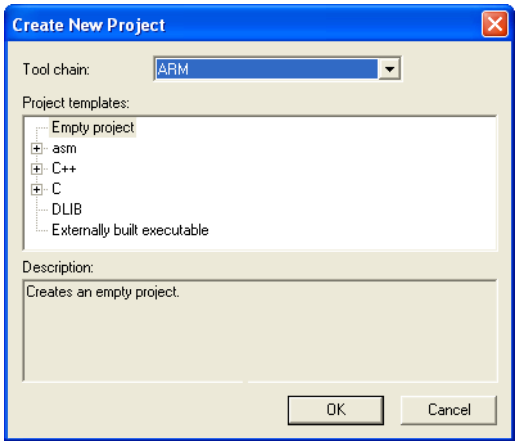


Figure 2: Create New Project dialog box

- 2 Make sure the **Tool chain** is set to **ARM**, 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.

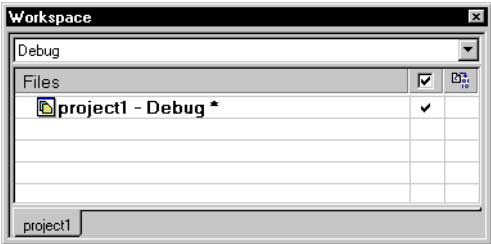


Figure 3: Workspace window

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 calls 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 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.

- 5 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.

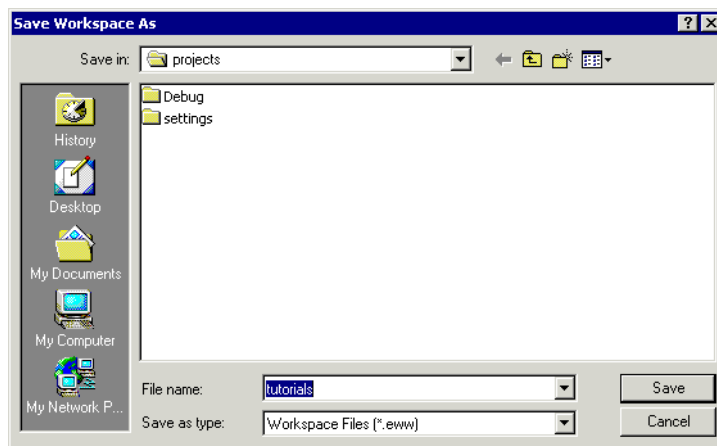


Figure 4: New Workspace dialog box

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.

ADDING 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 there are only two files in this project there is no need for creating a group. For more information about how to create complex project structures, see the chapter *Managing projects*.

- 1 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.

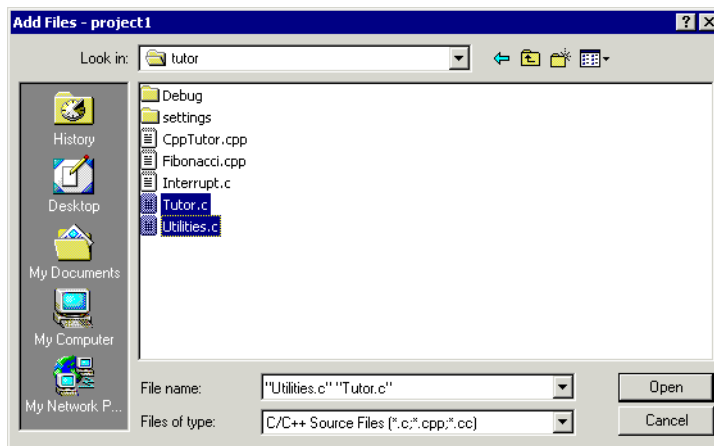


Figure 5: Adding files to `project1`

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**.

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

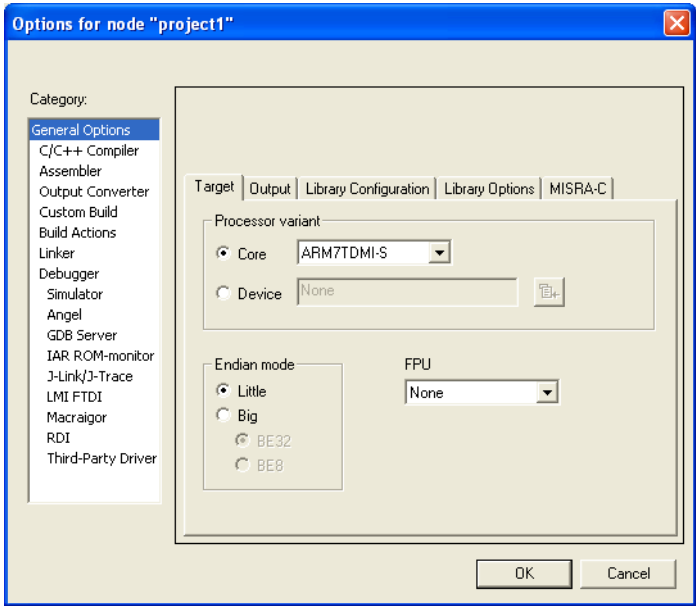


Figure 6: Setting general options

Verify the following settings:

Page	Setting
Target	Core: ARM7TDMI
Output	Output file: Executable
Library Configuration	Library: Normal
Library Configuration	Library low-level interface implementation: Semihosted

Table 4: General settings for project1

Then set up the compiler options for the project.

- 2 Select **C/C++ Compiler** in the **Category** list to display the compiler option pages.

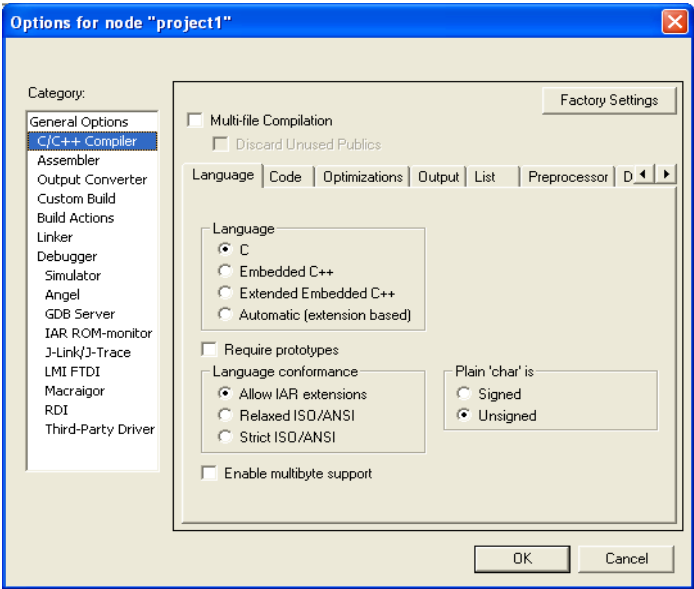


Figure 7: Setting compiler options

- 3 Verify the following settings:

Page	Setting
Optimizations	Level: None (Best debug support)
Output	Generate debug information
List	Output list file Assembler mnemonics

Table 5: Compiler options for project1

- 4 Click **OK** to set the options you have specified.

Note: It is possible to customize the amount of information to be displayed in the Build messages window. In this tutorial, the default setting is not used. Thus, the contents of the Build messages window on your screen might differ from the screen shots.

The project is now ready to be built.

Compiling and linking the application

You can now compile and link the application. You should also create a compiler list file and a linker map file and view both of them.

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 in 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 will be displayed in the Build messages window.

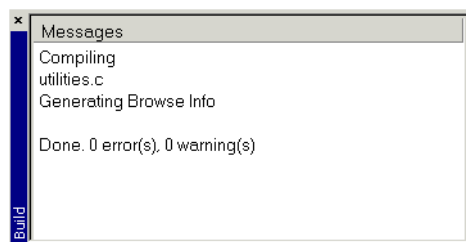


Figure 8: Compilation message

- 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 directories **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 will be 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 will be used as input to the IAR C-SPY® Debugger. Note that this directory will be 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, IAR Embedded Workbench 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.

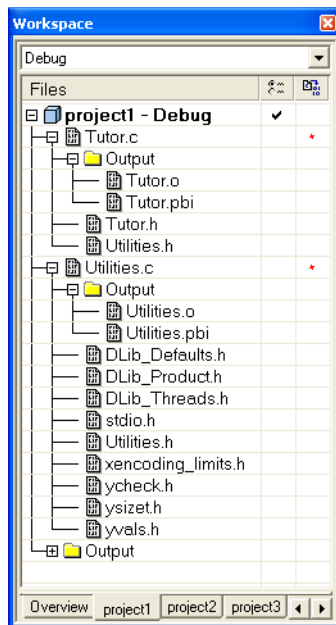


Figure 9: Workspace window after compilation

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.

- I 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 different 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.

- 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. Click the **OK** button.

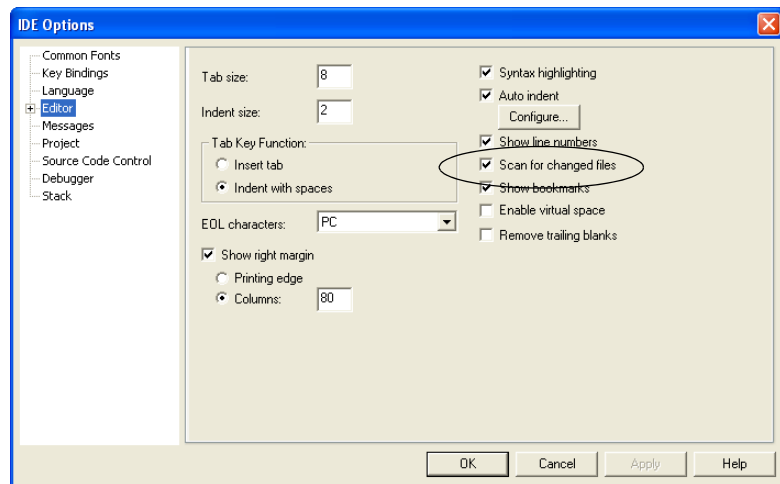


Figure 10: Setting the option *Scan for Changed Files*

- 3 Select the file `Utilities.c` in the Workspace window. Open the **C/C++ Compiler** options dialog box by right-clicking on the selected file in the Workspace window. Click the **Optimizations** tab and select the **Override inherited settings** option. Choose **High** from the **Level** drop-down list. Click **OK**.

Notice that the options override on the file node is indicated 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.

- I Select the project folder icon **project1 - Debug** in the Workspace window and choose **Project>Options**. Then select **Linker** in the **Category** list to display the linker option pages.

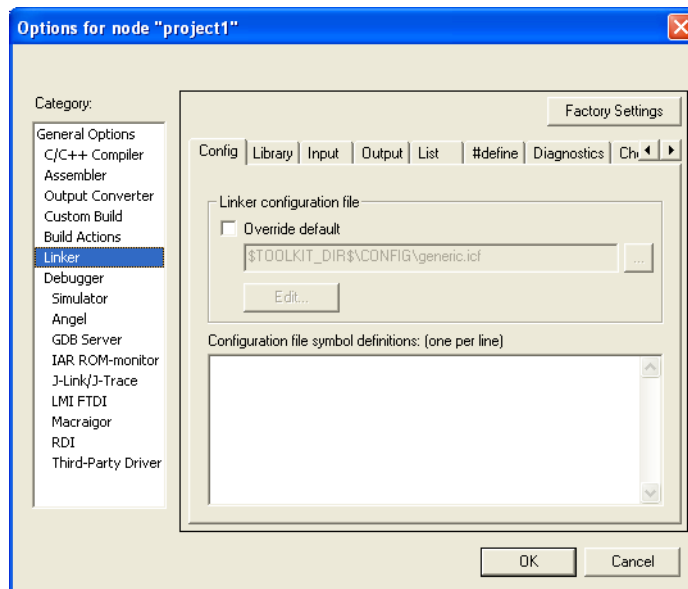


Figure 11: Linker options dialog box for project1

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 need to convert the file. You can use the converted provided with IAR Embedded Workbench, see *Converter options*, page 409.

Linker configuration file

Program code and data are placed in memory according to the configuration specified in the linker configuration file. It is important to be familiar with its syntax for how sections are placed in memory. You can 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 `src\examples`.

In this tutorial you will use the default linker configuration file, which you can see on the **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 **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 **List** tab and select the option **Generate linker map file**.

- 2 Click **OK** to save the linker options.

Now you should link the object file, to generate code that can be debugged.

- 3 Choose **Project>Make**. The progress will as usual be displayed in the Build messages window. The result of the linking is a code file `project1.out` with debug information and a map file `project1.map`.

VIEWING THE MAP FILE

Examine the file `project1.map` to see how the sections were placed in memory.

You can read more about the map file in the *IAR C/C++ Development Guide for ARM®*.

The `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 product package you have installed, C-SPY might or might not be included. The tutorials assume that you are using the C-SPY Simulator.

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.

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.



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 77.

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.

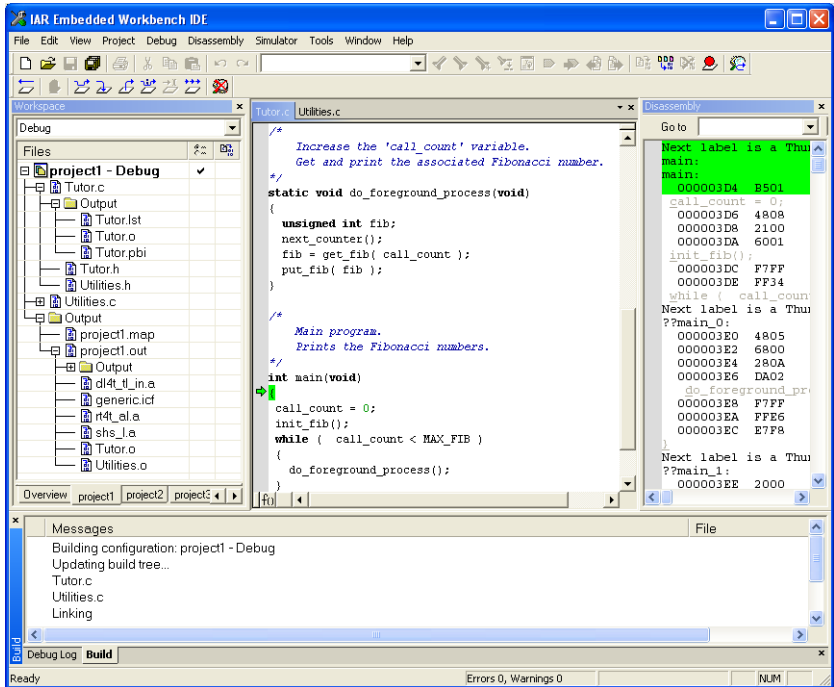


Figure 12: The C-SPY Debugger main window

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.



Alternatively, click the **Step Over** button on the toolbar.

The current position should be the call to the `init_fib` function.

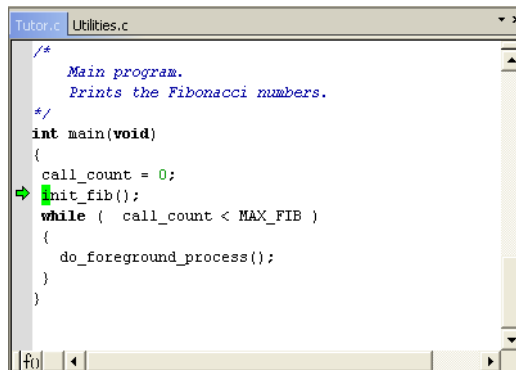


Figure 13: Stepping in C-SPY

- 3 Choose **Debug>Step Into** to step into the function `init_fib`.



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 122.

When **Step Into** is executed you will notice that the active window changes to `Utilities.c` as the function `init_fib` is located in this file.

- 4 Use the **Step Into** command until you reach the `for` loop.

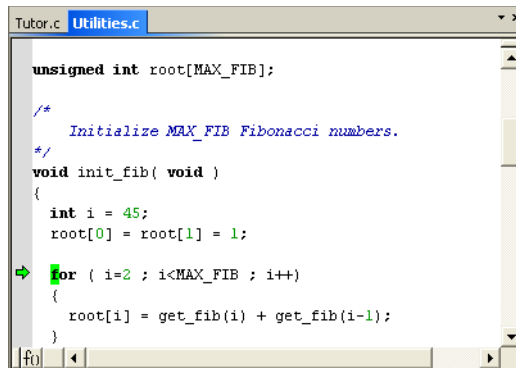


Figure 14: Using Step Into in C-SPY

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

INSPECTING VARIABLES

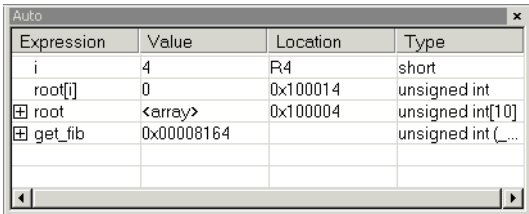
C-SPY allows you to watch variables or expressions in the source code, so that you can keep track of their values as you execute your application. You can look at a variable in a number of ways; for example by pointing at it in the source window with the mouse pointer, or by opening one of the Locals, Watch, Live Watch, or Auto windows. 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.



Expression	Value	Location	Type
i	4	R4	short
root[i]	0	0x100014	unsigned int
root	<array>	0x100004	unsigned int[10]
get_fib	0x00008164		unsigned int (...)

Figure 15: 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 the following 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 `root` array in the `init_fib` function, then drag it to the Watch window.

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

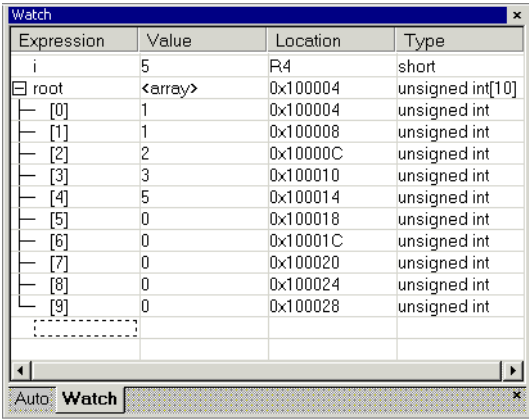


Figure 16: Watching variables in the Watch window

- 6 Execute some more steps to see how the values of `i` and `root` change.
- 7 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 different breakpoints, see *The breakpoint system*, page 135.

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 statement `root[i]` using the following 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 statement. The statement will be highlighted and there will be a **red dot** in the margin to show that there is a breakpoint there.

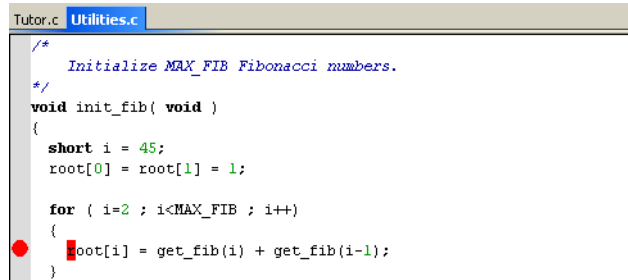


Figure 17: Setting breakpoints

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 `root` expression and the Debug Log window will contain information about the breakpoint.

- 3 Select the breakpoint and choose **Edit>Toggle Breakpoint** 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 and choose **Code Coverage>Enable** and then **Code Coverage>Show** from the context menu in the Disassembly window.

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

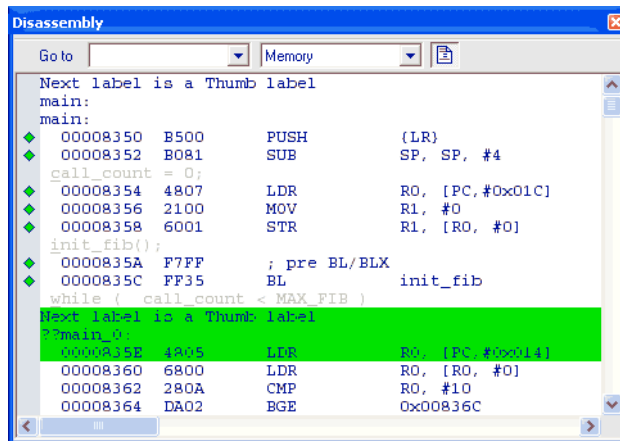


Figure 18: Debugging in disassembly mode

MONITORING MEMORY

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

- 1 Choose **View>Memory** to open the Memory window.
- 2 Make the Utilities.c window active and select `root`. Then drag it from the C source window to the Memory window.

The memory contents in the Memory window corresponding to `root` will be selected.

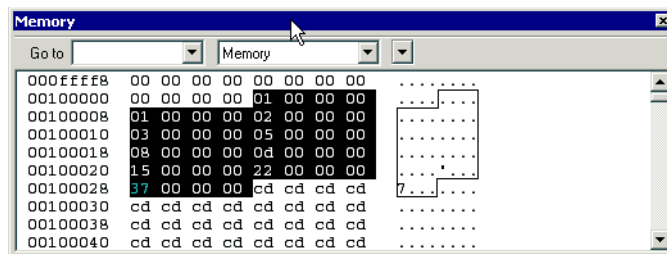


Figure 19: Monitoring memory

- 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.

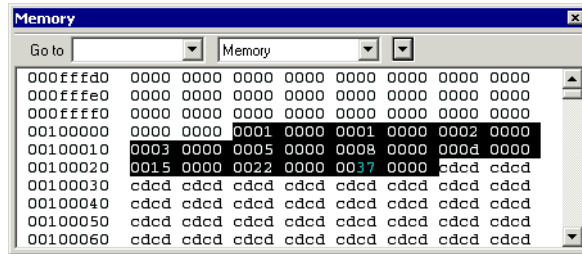


Figure 20: Displaying memory contents as 16-bit units

If not all of the memory units have been initialized by the `init_fib` function of the C application yet, continue to step over and you will notice how the memory contents will be updated.

You can change the memory contents by editing 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 need 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 **Semihosted** or the **IAR breakpoint** option. This means that some low-level routines will be linked that direct `stdin` and `stdout` to the Terminal I/O window, see *Linking the application*, page 38.

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

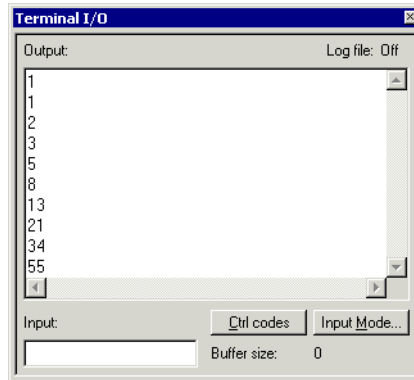


Figure 21: Output from the I/O operations

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

REACHING PROGRAM EXIT

- I 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.

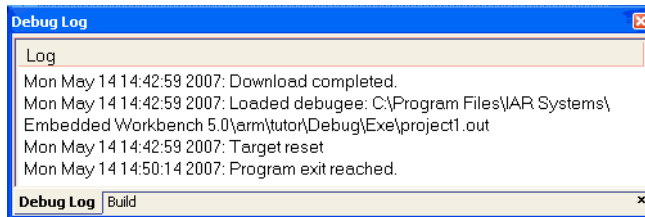


Figure 22: Reaching program exit in C-SPY

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 may 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 need to 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.

Examining the calling convention

When writing an assembler routine that will be called from a C routine, it is necessary to be aware of the calling convention used by the compiler. By creating skeleton code in C and letting 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 workspace `tutorials` used in previous tutorials, 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 the following settings:

Page	Option
Optimizations	Level: None (Best debug support)
List	Output assembler file
	Include source
	Include call frame information (deselected).

Table 6: Compiler options for project2

- Note:** In this example it is necessary to use a low optimization level when compiling the code to show local and global variable accesses. If a higher level of optimization is used, the required references to local variables can be removed. 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.
- 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 file `Write.s`.

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**.

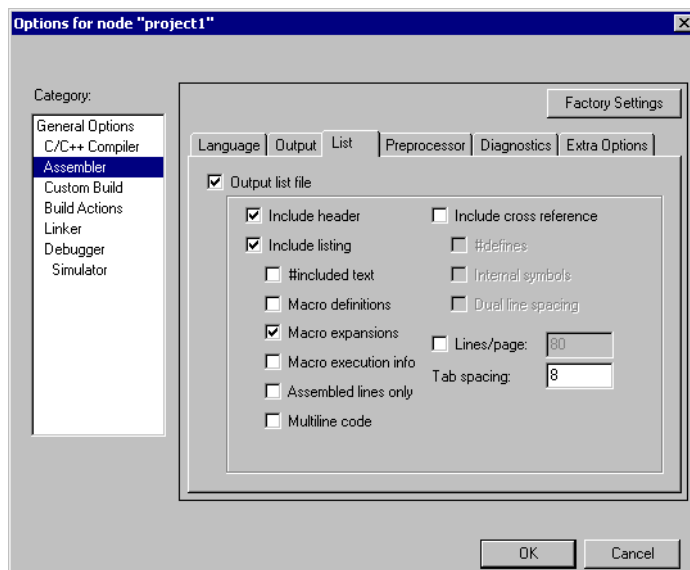


Figure 23: Assembler settings for creating a list file

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` will be 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 in the previous tutorial, but with a `*` before every character written.

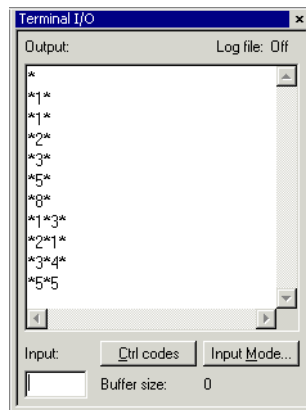


Figure 24: Project2 output in terminal I/O window

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.

Creating a C++ application

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

- `Fibonacci.cpp` creates 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 the following options are selected:

Category	Page	Option
General Options	Target	ARM7TDMI
C/C++ Compiler	Language	Embedded C++

Table 7: Project options for Embedded C++ tutorial

All you need to do to switch to the Embedded 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 the following line:

```
fibonacci fib1;
```

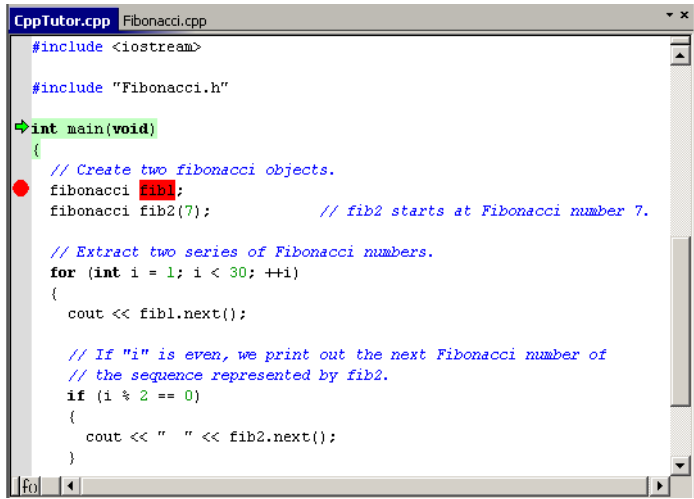


Figure 25: Setting a breakpoint in CppTutor.cpp

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:

```
cout << fib1.next();
```

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

- 6 Use the **Go to function** button in the lower left corner of the editor window to find and go to the function `nth` by double-clicking the function name. Set a breakpoint on the function call `nth(n-1)` at the line



```
value = nth(n-1) + nth(n-2);
```

- 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. By adding 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>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. Set the value in the **Skip count** text box to 4 and click **OK**.

Close the dialog box.

Looking at the function calls

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

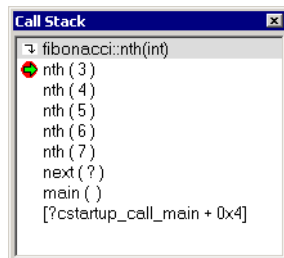


Figure 26: Inspecting the function calls

There are five instances of the function `nth` displayed on the call stack. Because the Call Stack window displays the values of the function parameters, you can see the different values of `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.

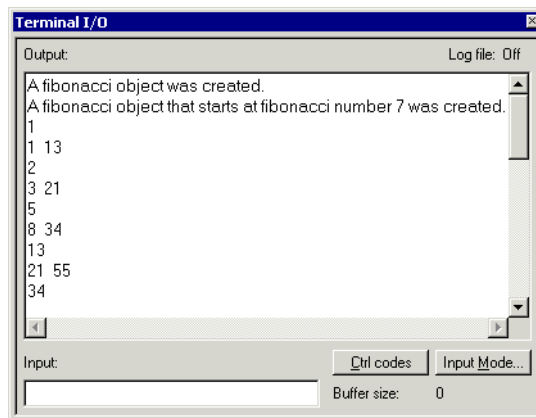


Figure 27: Printing Fibonacci sequences

Simulating an interrupt

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

This tutorial will show how the compiler keywords `__irq` and `__arm` can be used. The tutorial will also show how an interrupt can be simulated 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.

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), `UARTBRTHR` (in header files `__UARTBRTHR`). It will then print the value. The main program enables interrupts and starts printing periods (.) in the foreground process while waiting for interrupts.

WRITING AN INTERRUPT HANDLER

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

```
/* define the IRQ handler */
__irq __arm void irqHandler( 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 and pragma directives used in this tutorial, see the *IAR C/C++ Development Guide for ARM®*.

SETTING UP THE PROJECT

- 1 Add a new project—`project4`—to the workspace `tutorials` used in previous tutorials.
- 2 Add the files `Utilities.c` and `Interrupt.c` to it.
- 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** from the **Core** drop-down menu.
- 4 Select the **C/C++ Compiler** category, and click the **Code** tab. Select the option **Generate interwork code**.

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 will be read from the file `InputData.txt`, which contains the Fibonacci series. You will set an *immediate read breakpoint* on the UART receive register, `UARTBRTHR`, 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 will read `UARTBRTHR` and the breakpoint will be triggered, the `Access` macro function will be executed and the Fibonacci values will be fed into the UART receive register.

The immediate read breakpoint will trigger the break *before* the processor reads the `UARTBRTHR` 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 195.

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`, 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, will be opened:

```
_fileHandle = __openFile(
    "$TOOLKIT_DIR$\tutor\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 ) )
    {
        UATRBRTHR = _fibValue;
    }
}
```

You will have to 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`

Alternatively, use an argument variable to specify the path:

`$TOOLKIT_DIR$\tutor\SetupSimple.mac`

See *Argument variables summary*, page 306, for details.

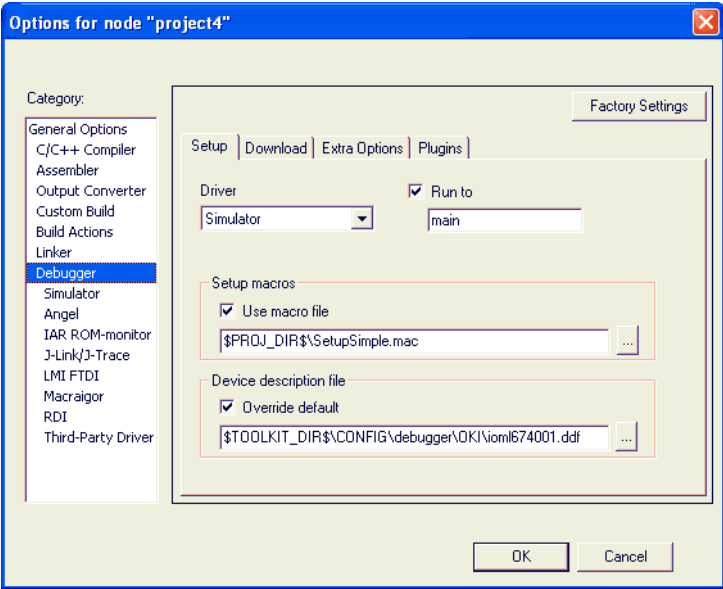


Figure 28: Specifying setup macro file

- 3 Set the **Device description file** option to `iom1674001.ddf`. This file makes it possible to view the value of `UARTBRTHR` in the Register window and provides the interrupt definitions that are needed by the interrupt system.
- 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` project.

The `Interrupt.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 and make the following settings for your interrupt:

Setting	Value	Description
Interrupt	IRQ	Specifies which interrupt to use;
Description	As is	The interrupt definition that the simulator uses to be able to simulate the interrupt correctly.
First activation	4000	Specifies the first activation moment for the interrupt. The interrupt is activated when the cycle counter has passed this value.
Repeat Interval	2000	Specifies the repeat interval for the interrupt, measured in clock cycles.
Hold time	Infinite	Hold time, not used here.

Table 8: Interrupts dialog box

Setting	Value	Description
Probability %	100	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.
Variance %	0	Time variance, not used here.

Table 8: Interrupts dialog box (Continued)

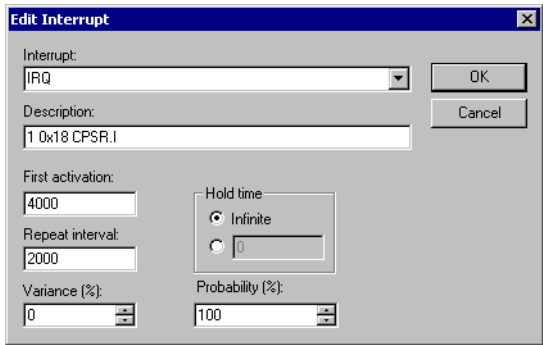


Figure 29: Inspecting the interrupt settings

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.

- 2 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 67.

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 `UARTBRTHR` 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.
- 2 Add the following parameters for your breakpoint.

Setting	Value	Description
Break at	UARTBRTHR	Receive buffer address.
Access Type	Read	The breakpoint type (Read or Write)
Action	Access ()	The macro connected to the breakpoint.

Table 9: Breakpoints dialog box

During execution, when C-SPY detects a read access from the `UARTBRTHR` 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 `UARTBRTHR`. C-SPY will then resume the simulation by reading the receive buffer value in `UARTBRTHR`.

- 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 67.

Simulating the interrupt

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

EXECUTING THE APPLICATION

- 1 Step through the application and stop when it reaches the `while` loop, where the application waits for input.
 - 2 In the `Interrupt.c` source window, locate the function `irqHandler`.
 - 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 **Edit>Breakpoints**.
- 4 The Register window lets you monitor and modify the contents of the processor registers.

To inspect the contents of the serial communication port receive register UARTRBRTHR, choose **View>Register** to open the Register window. Choose UART from the drop down list.

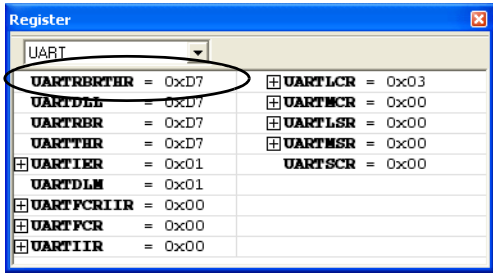


Figure 30: Register window

- 5** Run your application by choosing **Debug>Go** or clicking the **Go** button on the toolbar. The application should stop in the interrupt function.

Note how the contents of UARTRBRTHR has been updated.

- 6** 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.

- 7** Click **Go** again in order 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 `exit` label and stop.

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

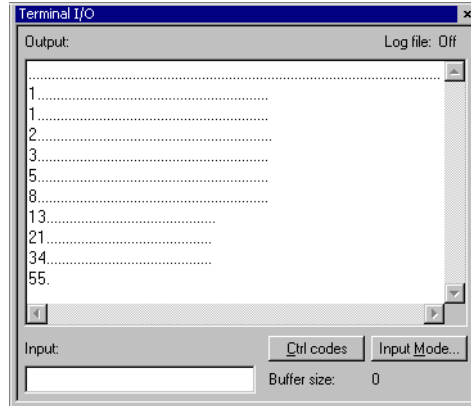


Figure 31: Printing the Fibonacci values in the Terminal I/O window

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` is extended with system macro calls for setting the breakpoint and specifying the interrupt:

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

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

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

By replacing the file `SetupSimple.mac`, used in the previous tutorial, with the file `SetupAdvanced.mac`, setting the breakpoint and defining the interrupt will be automatically performed at C-SPY 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` you should remove the previously defined breakpoint and interrupt.

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.

Using libraries

If you are working on a large project you will soon accumulate a collection of useful modules containing one or more routines that can 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 GNU utility `ar` 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 with the product.

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.

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 the following settings:

Page	Option
Output	Output file: Library
Library Configuration	Library: None

Table 10: General options for a library project

Note that **Library Builder** appears in the list of categories, which means that the GNUutility `ar` is added to the build tool chain. It is not necessary to set any `ar`-specific options for this tutorial.

Click **OK**.

- 4 Choose **Project>Make**.

The library output file `tutor_library.a` has now been created.

USING THE LIBRARY IN YOUR APPLICATION PROJECT

You can 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**.



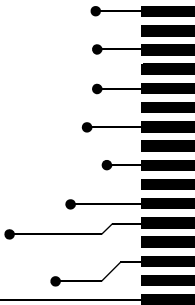
2 Click **Make** to build your project.

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 *GNU Binary Utils* documentation available on the Help menu.

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 96.

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 different components.

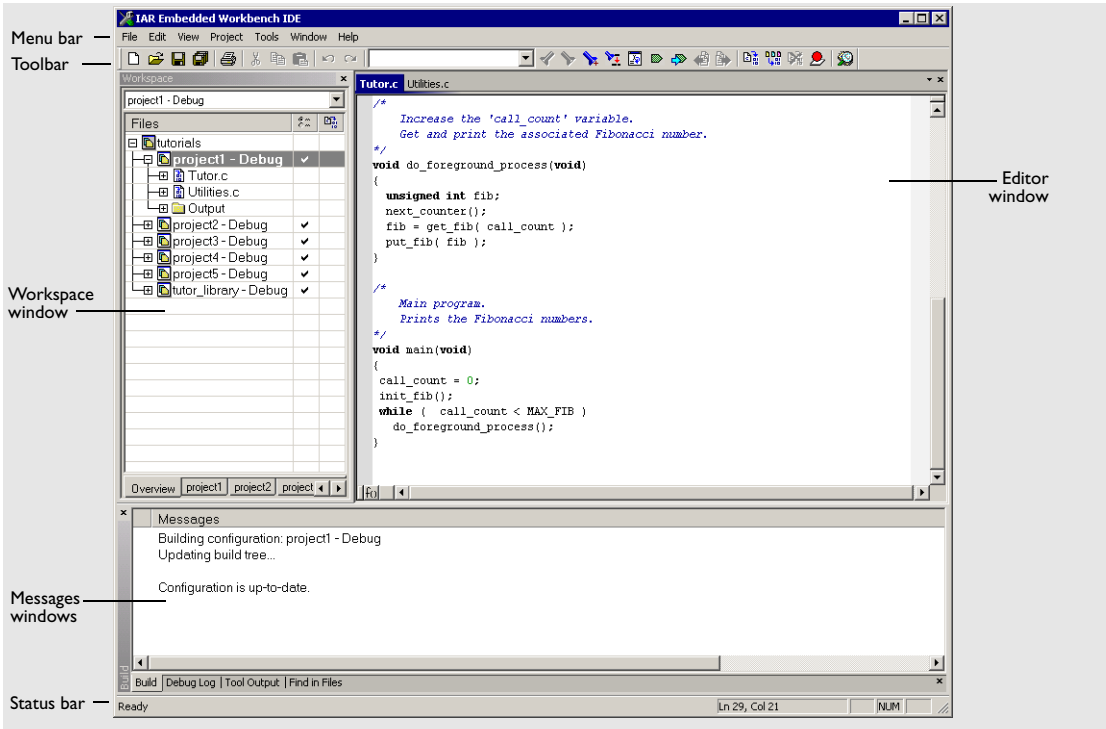


Figure 32: IAR Embedded Workbench IDE window

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

RUNNING THE IDE

Click the **Start** button on the 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 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 will be 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 99.

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 313. You can also find further information related to customizing the editor in the section *Customizing the editor environment*, page 105. 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.

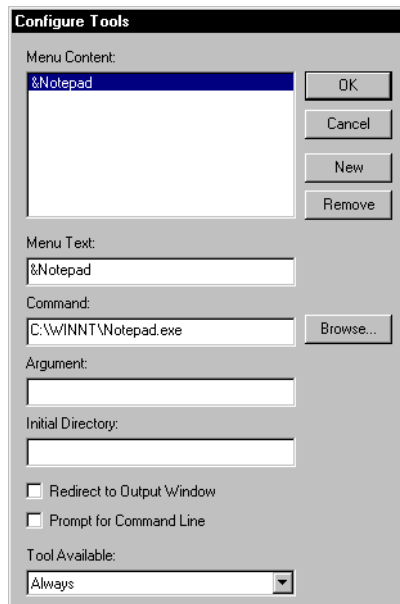


Figure 33: *Configure Tools* dialog box

For reference information about this dialog box, see *Configure Tools dialog box*, page 334.

Note: It is not possible to use the **Configure Tools** dialog box to extend the tool chain in the IDE, see *The tool chain*, page 75.

After you have entered the appropriate information and clicked **OK**, the menu command you have specified is displayed on the **Tools** menu.

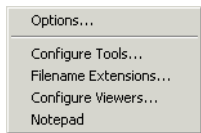


Figure 34: Customized Tools menu

Note: If you intend to add an external tool to the standard build tool chain, see *Extending the tool chain*, page 96.

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.

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

Type one of the following command shells in the **Command** text box:

System	Command shell
Windows 2000/XP/Vista	cmd.exe (recommended) or command.com

Table 11: 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 a number of 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 different 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—will be developed, requiring one development team each (team A and B). Because the two applications are related, parts of the source code can be shared between the applications. 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.

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

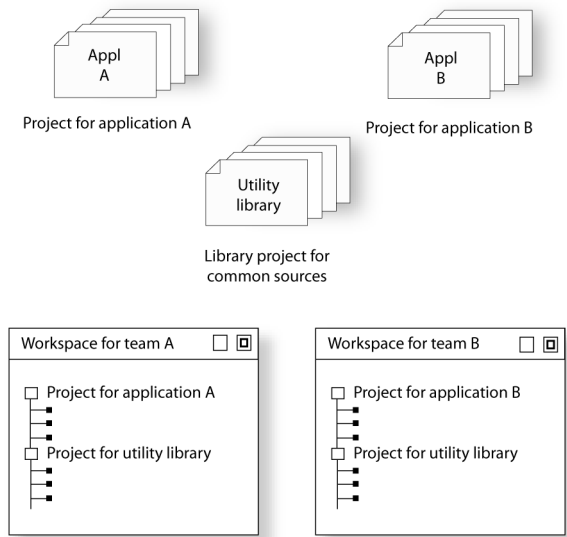


Figure 35: Examples of workspaces and projects

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 `NDEBUG` is defined, which means the application will not contain any asserts.

Additional build configurations can 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, appropriate source files can be excluded from the build configuration. The following 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.

Note: The settings for a build configuration can affect which include files that will be used during 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.

CREATING AND MANAGING WORKSPACES

This section describes the overall procedure for creating the workspace, projects, groups, files, and build configurations. The **File** menu provides the commands for creating workspaces. The **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. There are template projects 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 29.

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* will be added to that group. Source files dropped outside the project tree—on the Workspace window background—will be added to the active project.

Source file paths

The IDE supports relative source file paths to a certain degree.

- 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 `EW_DIR`. The argument variable `EW_DIR` is only used if the path refers to a file located in subdirectory to `EW_DIR` and the distance from `EW_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 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 117.

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 Choose which project you want to view by clicking its tab at the bottom of the Workspace window.

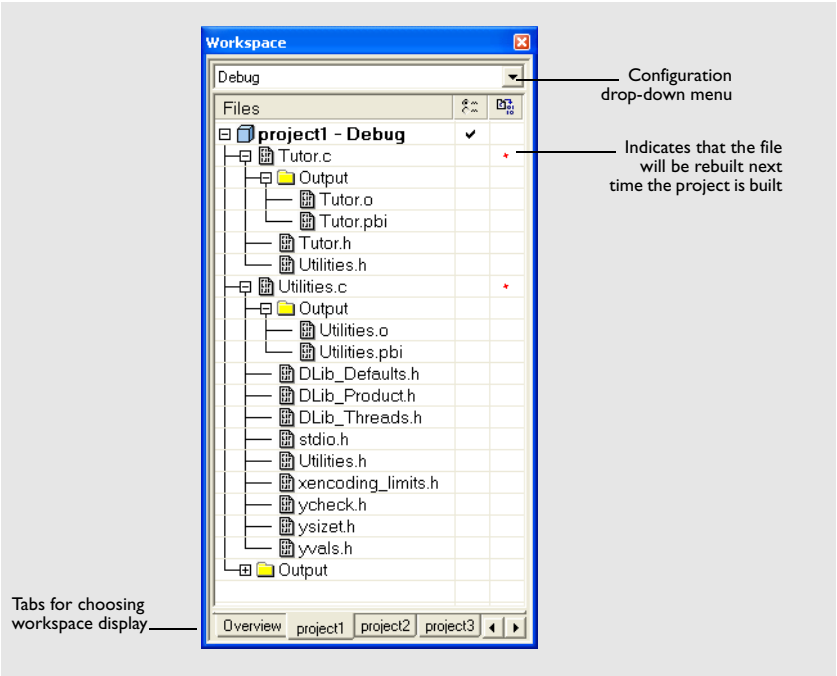


Figure 36: Displaying a project in the Workspace window

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.

- 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.

The project and build configuration you have selected are displayed highlighted in the Workspace window. It is the project and build configuration that is selected from the drop-down list that will be 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.

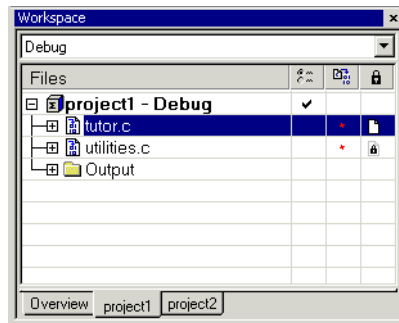


Figure 37: Workspace window—an overview

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 280.

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, there are three alternative methods that you can use:

- 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. It is important to keep this in mind when reading the description below.

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 there is at least one SCC client application 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 will be 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
- there is a new version of a file 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 270.

For reference information about the commands available for accessing the SCC system, see *Source Code Control menu*, page 269.

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 *Terminal I/O options*, page 333.

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 the following 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 use the IAR Embedded Workbench IDE for building projects, it is also possible to use the command line utility `iarbuild.exe` for building projects.

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.

It is possible to override project level settings by selecting the required item, for instance a specific group of files, and selecting 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 file level (file level override), no options on higher levels that operate on files will affect that file.

Using the Options dialog box

The **Options** dialog box—available by choosing **Project>Options**—provides options for the building 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.

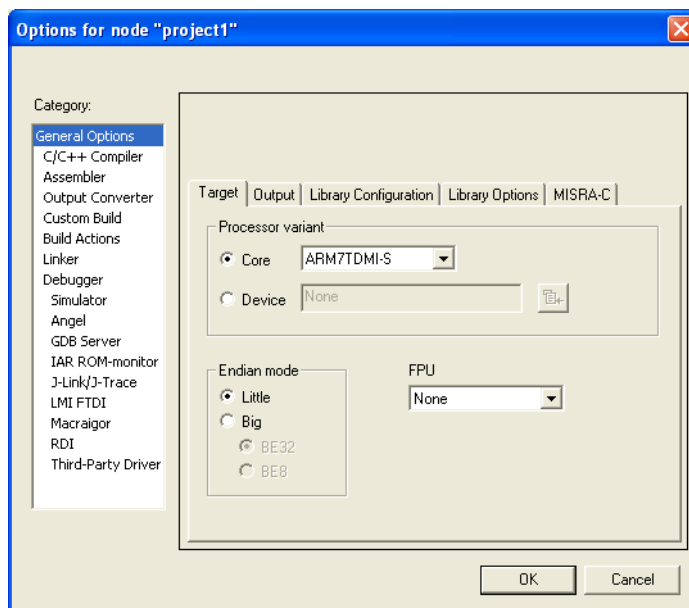


Figure 38: General options

The **Category** list allows you to select which building tool to set options for. The tools 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** will be 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 there are two sets of factory settings available: Debug and Release. Which one that will be used depends on your build configuration; see *New Configuration dialog box*, page 308.

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 336.

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 304.

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 several 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 311.

USING PRE- AND POST-BUILD ACTIONS

If necessary, you can specify pre-build and post-build actions that you want to take place 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 413.



Using pre-build actions for time stamping

Pre-build actions can be used for embedding a time stamp for the build in the resulting binary file. To achieve this, 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. The regular build process then always must 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 will be 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. So if there are errors in your source code, 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 288.

BUILDING FROM THE COMMAND LINE

It is possible to build the project from the command line by using 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]
```

Parameter	Description
<code>project.ewp</code>	Your IAR Embedded Workbench project file.
<code>-clean</code>	Removes any intermediate and output files.
<code>-build</code>	Rebuilds and relinks all files in the current build configuration.
<code>-make</code>	Brings the current build configuration up to date by compiling, assembling, and linking only the files that have changed since the last build.
<code>configuration</code>	The name of the configuration you want to build, which can either be one of the predefined configurations Debug or Release, or a name that you define yourself. For more information about build configurations, see <i>Projects and build configurations</i> , page 83.
<code>-log errors</code>	Displays build error messages.
<code>-log warnings</code>	Displays build warning and error messages.
<code>-log info</code>	Displays build warning and error messages, and messages issued by the <code>#pragma message</code> preprocessor directive.
<code>-log all</code>	Displays all messages generated from the build, for example compiler sign-on information and the full command line.

Table 12: `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 by IAR Systems). You can make these tools execute each time specific files in your project have changed.

By specifying 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 411, 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, as well as the name of the output files generated by the external tool. Note that it is possible to use argument variables for substituting file paths.

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

It is possible to 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.

- I 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 will be expanded to:

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

Note the usage of *argument variables*. For further details of these variables, see *Argument variables summary*, page 306.

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 there are any additional files used by the external tool 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**.

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*. You can have several editor windows open at the same time.

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.

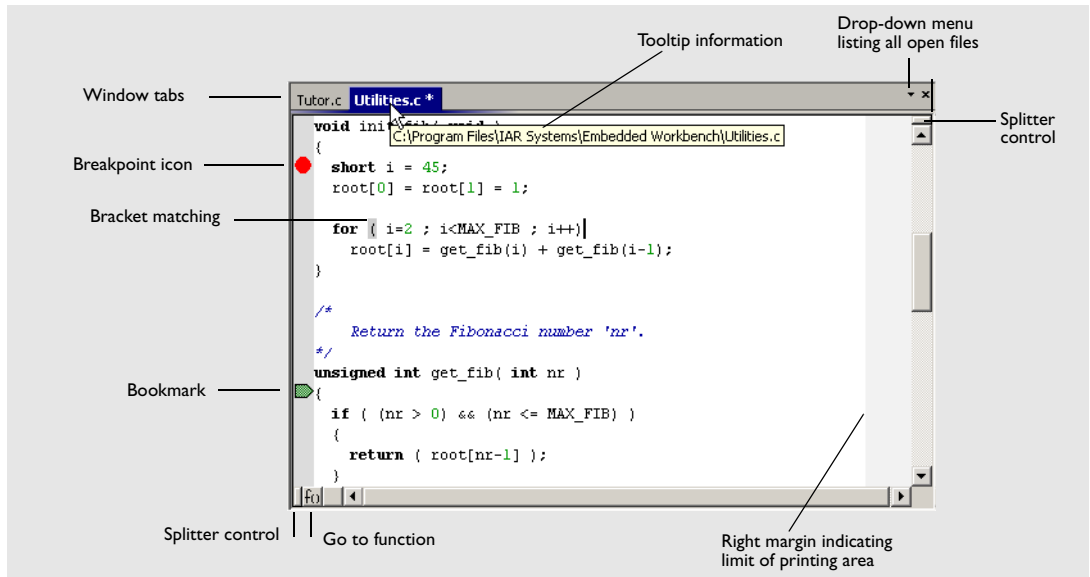


Figure 39: Editor window

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, as well as commands for moving files between different editor windows. For reference information about each command on the menu, see *Window menu*, page 339. For reference information about the editor window, see *Editor window*, page 274.



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.

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 by using 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 294.

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 278.

To change the default shortcut key bindings, choose **Tools>Options**, and click the **Key Bindings** tab. For further details, see *Key Bindings options*, page 315.

Splitting the editor window into panes

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

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 323.

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 **Edit>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 323.
- 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 different 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 a number of 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 319.

Matching brackets and parentheses

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

```
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:

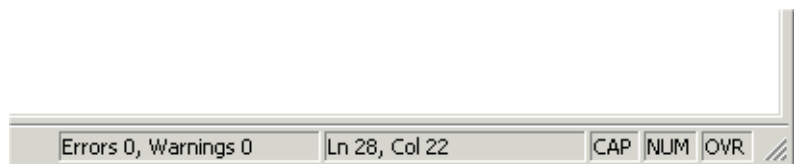


Figure 41: Editor window status bar

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, there are a few example templates 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 in your source code

To insert a code template in your source code, place the insertion point at the location where you want the template to be inserted and choose **Edit>Insert Template**. This command displays a list in the editor window from which you can choose a code template.

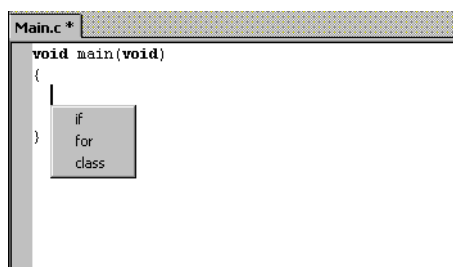


Figure 42: Editor window code template menu

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 will be used by default if code templates are enabled. To use your own template file, follow the procedure described in *Enabling code templates*, page 103.

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.

NAVIGATING IN AND BETWEEN FILES

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

- 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 294.

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 313.

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`.

You can send an argument to the external editor by typing 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).

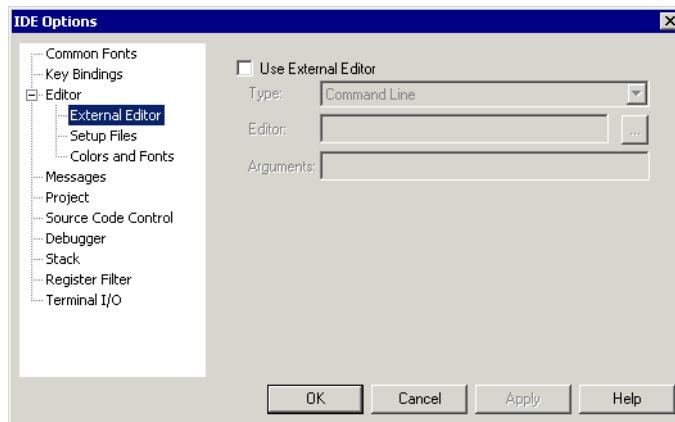


Figure 43: Specifying external command line editor

- 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 CommandString

DDE-Topic CommandString

as in the following example, which applies to Codewright®:

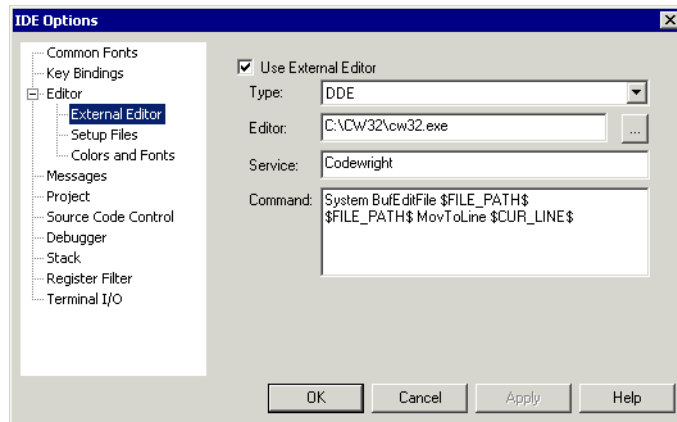


Figure 44: External editor DDE settings

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.

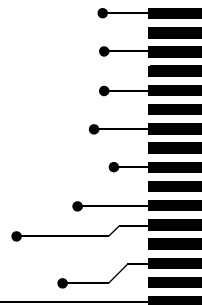
When you open a file by double-clicking it in the Workspace window, the file will be opened by the external editor.

Variables can be used in the arguments. For more information about the argument variables that are available, see *Argument variables summary*, page 306.

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.





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 C-SPY environment is presented, followed by a description of how to setup, 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, such information can be found 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

C-SPY can be used for debugging either a software target system or a hardware target system.

The following figure shows an overview of C-SPY and possible target systems.

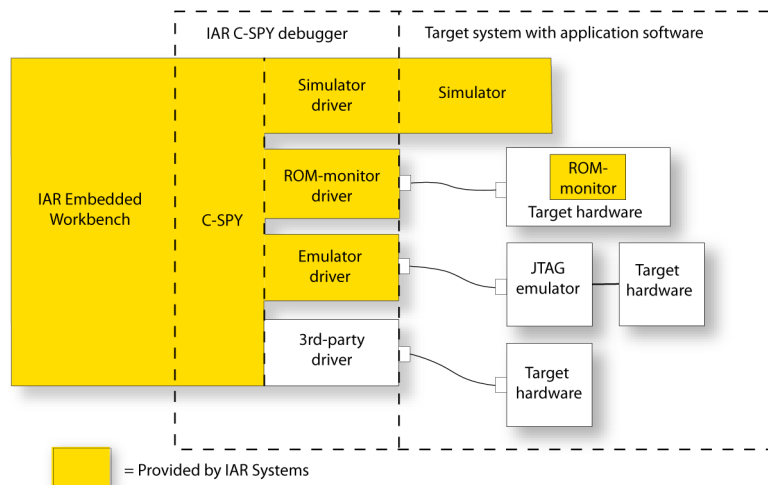


Figure 45: C-SPY and target systems

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 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 by choosing the appropriate driver from within the IDE.

For an overview of the general features of C-SPY, see *IAR C-SPY Debugger*, page 6. In that chapter you can also find an overview of the functionality provided by each driver. Contact your software distributor or IAR 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

It is possible to 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 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 will be 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. It is also possible to 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.

Reference information about each item specific to C-SPY can be found in the chapter *C-SPY® reference*, page 343.

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 328.

For information about how to configure the debugger to reflect the target hardware, see *Adapting C-SPY to target hardware*, page 118.

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 also need to set hardware-specific options. For information about these options, see *Part 7. Reference information* 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 will be executed prior to 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 there are no breakpoints available when C-SPY starts, a warning message appears notifying 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, by using setup macro functions and system macros. Thus, by loading 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 150. 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 handles several of the target-specific adaptations by using device description files. 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 118. 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, as well as by third-party suppliers. Contact your software distributor or IAR representative, or visit the IAR Systems web site, for information about available modules.

For information about how to load plugin modules, see *Plugins*, page 432.

The C-SPY RTOS awareness plugin modules

You can load plugin modules for real-time operating systems plugin modules supported for the IAR Embedded Workbench version you are using, you can load one for use with C-SPY. 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 can start it.



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

It is also possible to 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.

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 it can be easily inspected. 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.

The information printed in the file is by default 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 378.

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 handles several of the target-specific adaptations by using device description files provided with the product. They contain device-specific information such as definitions of peripheral registers, 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 115.

Registers

For each device there is 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 different 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 147.

Modifying a device description file

There is normally no need to modify the device description file. However, if the predefinitions 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 are 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 functions `__writeMemory32()` will perform the necessary initialization of the memory controller.

The following example illustrates a macro used to setup the memory controller and remap 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, 0xffe0000c, "Memory");

    // unused
    __writeMemory32(0x40000000, 0xffe00010, "Memory");

    // unused
    __writeMemory32(0x50000000, 0xffe00014, "Memory");
}
```

```

// unused
__writeMemory32(0x60000000, 0xffe00018, "Memory");

// unused
__writeMemory32(0x70000000, 0xffe0001c, "Memory");

// REMAP command
__writeMemory32(0x00000001, 0xffe00020, "Memory");

// standard read
__writeMemory32(0x00000006, 0xffe00024, "Memory");
}

```

Note that the setup macro `execUserReset()` may 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 setup 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 153. 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 41.

Executing

C-SPY provides a flexible range of features for executing your application. You can find commands for executing on the **Debug** menu as well as on the toolbar.

STEP

C-SPY allows slightly more precise stepping than most other debuggers in that it is not line-oriented but statement-oriented. This means that you can follow the flow of execution precisely even if there are multiple statements on the same line. Use any of the following step commands:

- Step Into
- Step Over
- Step Out.

Consider the following example, and assume that the previous step has taken you to the call `f(3)` as highlighted in green (and also indicated by a green arrow in the left margin of the text editor):

```
void f(int n)
{
    int i;
    for (i = start(n); is_valid(i); i = next(i))
    {
        process(i);
    }
}
...
f(3);
var += 1;
```

Step Into will execute the current statement, but will stop inside the first function called. In this case, **Step Into** would take you directly inside the `f(int n)` function, as shown below:

```
void f(int n)
{
    int i;
    for (i = start(n); is_valid(i); i = next(i))
    {
        process(i);
    }
}
...
f(3);
var += 1;
```

Step Over will execute the current statement without stopping inside called functions. The command would first take you to the beginning of the `for` statement:

```
for (i = start(n); is_valid(i); i = next(i))
{
    process(i);
}
```

Repeating **Step Over** will follow the flow of execution as follows:

```
for (i = start(n); is_valid(i); i = next(i))
{
    process(i);
}
```

```
for (i = start(n); is_valid(i); i = next(i))
{
    process(i);
}
```

```
for (i = start(n); is_valid(i); i = next(i))
{
    process(i);
}
```

The sequence is repeated until the loop terminates. At any time, you can use the **Step Out** command function to directly take you back to the calling function:

```
void f(int n)
{
    int i;
    for (i = start(n); is_valid(i); i = next(i))
    {
        process(i);
    }
}
...
f(3);
var += 1;
```

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.

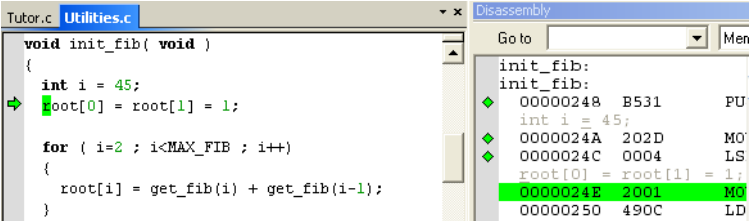


Figure 46: C-SPY highlighting source location

Occasionally, you will notice that a statement in the source window is highlighted using a pale variant of the normal highlight color. This happens when the program counter is at an assembler instruction which is part of a source statement but not exactly at its beginning. 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 there is a special kind of breakpoint to facilitate 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. It is also possible to 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 different 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. You can stop the application execution by clicking the **Break** button, alternatively by choosing 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, there are situations where 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 will be 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 364.

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 need 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 can be useful in two different contexts:

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

To use this window, you need to build your application with the **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 365.

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 379.

Working with variables and expressions

This chapter defines the variables and expressions used in C-SPY®. It also demonstrates the different 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:

```
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`:

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

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 R0–R14, 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 118.

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

Example	What it does
<code>#pc++</code>	Increments the value of the program counter.
<code>myptr = #label7</code>	Sets <code>myptr</code> to the integral address of <code>label7</code> within its zone.

Table 13: C-SPY assembler symbols expressions

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

Example	What it does
<code>#pc</code>	Refers to the program counter.
<code>#`pc`</code>	Refers to the assembler label <code>pc</code> .

Table 14: 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 147.

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 150.

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 assigns both its value and type.

For details of C-SPY macro variables and how to use them, see *The macro language*, page 459.

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:

```
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 there is no need for the compiler 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 will be 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 131.
- **The Trace system**, see *Using the trace system*, page 131.



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

For reference information about the different windows, see *C-SPY windows*, page 343.

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.

USING THE TRACE SYSTEM

A *trace* is a recorded sequence of events in the target system, typically executed machine instructions. Depending on what C-SPY driver you are using, additional types of trace data can be recorded. For example, read and write accesses to memory, as well as the values of C-SPY expressions.

By using the trace system, you can trace the program flow up to a specific state, for instance an application crash, and use the trace information to locate the origin of the problem. Trace information can be useful for locating programming errors that have irregular symptoms and occur sporadically. Trace information can also be useful as test documentation.

The trace system is not supported by all C-SPY drivers. For detailed information about the trace system and the components provided by the C-SPY driver you are using, see the corresponding driver documentation in *Simulator-specific debugging*, page 165 and *Hardware-specific debugging*, page 213, respectively.

Which trace system functionality that is provided depends on the C-SPY driver you are using. However, for all C-SPY drivers that support the trace system, the Trace window, the Find in Trace window, and the **Find in Trace** dialog box are always available. You can save the trace information to a file to be analyzed later.

The Trace window and its browse mode

The type of information that is displayed in the Trace window depends on the C-SPY driver you are using. The different trace data is displayed in separate columns, but the **Trace** column is always available if the driver you are using supports the trace system. The corresponding source code can also be shown.

You can follow the execution history by simply looking and scrolling 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 window by 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. Double-click again to leave browse mode.

Searching in the trace data

You can perform advanced searches in the recorded trace data. 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.

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 treats, by default, all data located at assembler labels as variables of type `int`. However, in the Watch, Quick Watch, and Live Watch windows, you can select a different interpretation to better suit the declaration of the variables.

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

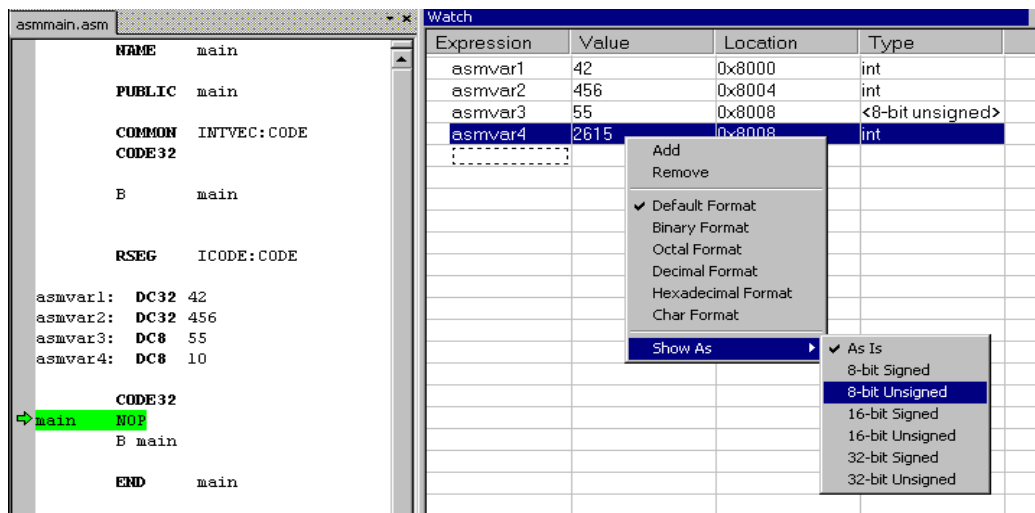


Figure 47: Viewing assembler variables in the Watch window

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

Using breakpoints

This chapter describes the breakpoint system and different ways to create and monitor breakpoints.

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.

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. It is also possible to 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. C-SPY provides different ways of defining breakpoints.

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

Defining breakpoints

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. For more details, see *Breakpoints window*, page 282.

Breakpoints are set with a higher precision than single lines, in analogy with the step mechanism; for more details about the step precision, see *Step*, page 122.

You can set a breakpoint in various ways; by using:

- The **Toggle Breakpoint** command
- The Memory window
- The Breakpoints dialog box
- Predefined system macros
- The editor window, see *Editor window*, page 274.

The different methods allow different levels of complexity and automation.

TOGGLING A SIMPLE CODE BREAKPOINT

Toggleing 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.



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:

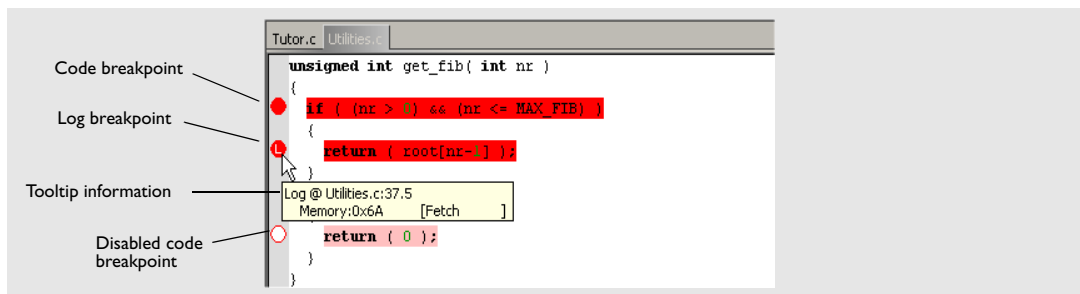


Figure 48: Breakpoint icons



If the breakpoint icon does not appear, make sure the option **Show bookmarks** is selected, see *Editor options*, page 317.



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.

SETTING A BREAKPOINT IN THE MEMORY WINDOW

It is possible to 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 by using the Breakpoints window, which is available from the **View** menu. The breakpoints you set in this window will be triggered for both read and write access. 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 THE DIALOG BOX

The advantage of using the 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.

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.

To modify an existing breakpoint:

- 1 Choose **View>Breakpoints** to open the Breakpoints window.
- 2 In the Breakpoints window, select the breakpoint you want to modify and right-click to open the context menu.
- 3 On the context menu, choose **Edit**.

A breakpoint dialog box appears. Specify the breakpoint settings and click **OK**. The breakpoint will be displayed in the Breakpoints window.

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

For reference information about code and log breakpoints, see *Code breakpoints dialog box*, page 283 and *Log breakpoints dialog box*, page 285, respectively. For details about any additional breakpoint types, see the driver-specific documentation.



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 will be evaluated and since it is not true execution will continue.

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

```
__var my_counter;

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

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

DEFINING BREAKPOINTS USING SYSTEM MACROS

You can define breakpoints not only by using the **Breakpoints** 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 by 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 it is still possible to 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 will be removed when you exit the debug session.

The following 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 153.

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 282.

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.

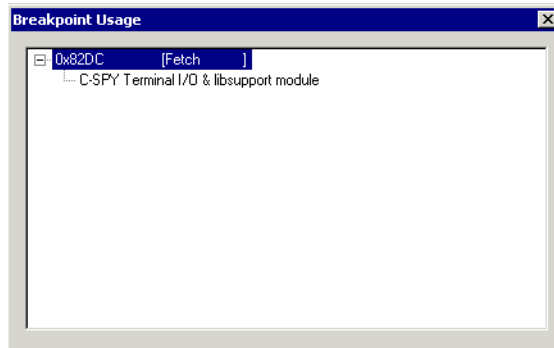


Figure 49: Breakpoint Usage dialog box

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 **Breakpoints** 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, the **Breakpoint Usage** dialog box can be useful for:

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

For information about the available number of 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

There are several consumers of breakpoints in a debugger system.

User breakpoints—the breakpoints you define by using the **Breakpoints** 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:

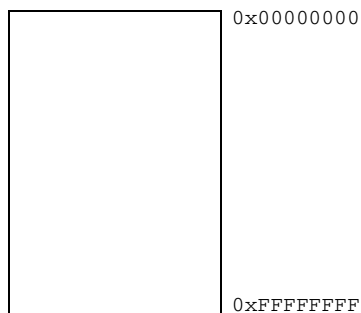
- Choose **Tools>Options>Stack**.
- Deselect the **Stack pointer(s) not valid until program reaches: label** option.

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. There are four memory zones: `Memory`, `Memory8`, `Memory16`, and `Memory32`, which all of them cover the whole ARM memory range.



Default zone `Memory`

Figure 50: Zones in C-SPY

Memory zones are used in several contexts, perhaps most importantly in the Memory and Disassembly windows. The **Zone** box in these windows allows you 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.
- **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.

You can easily view the memory contents for a specific variable by dragging the variable to the Memory window or the Symbolic memory window. The memory area where the variable is located will appear.

USING THE MEMORY WINDOW

The Memory window gives an up-to-date display of a specified area of memory and allows you to edit it.

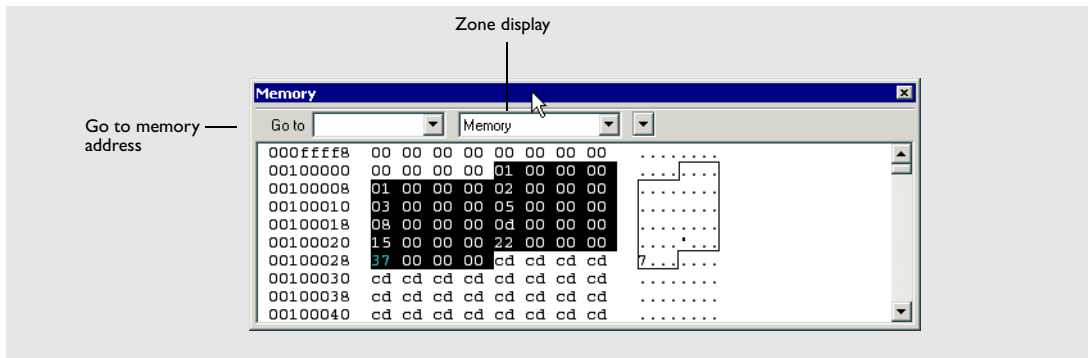


Figure 51: Memory window

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.

For reference information, see *Memory window*, page 349. See also *Setting a breakpoint in the Memory window*, page 137.

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.

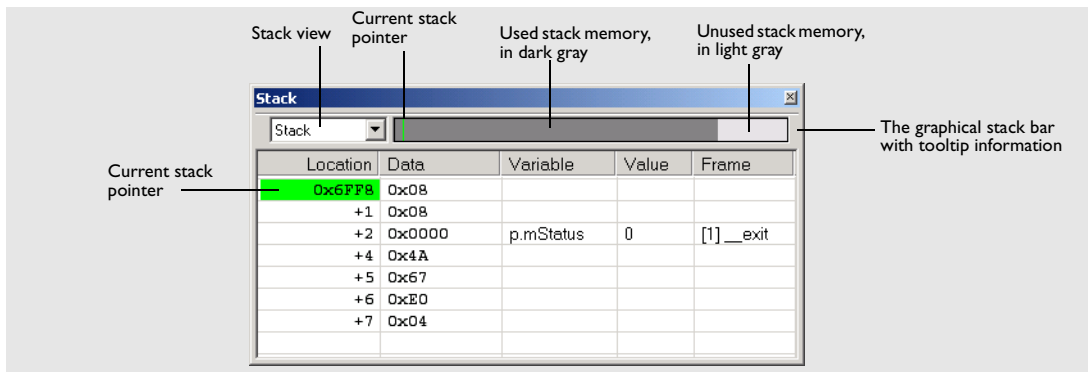


Figure 52: Stack window

For detailed reference information about the Stack window, and the method used for computing the stack usage and its limitations, see *Stack window*, page 369. For reference information about the options specific to the window, see *Stack options*, page 330.



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.

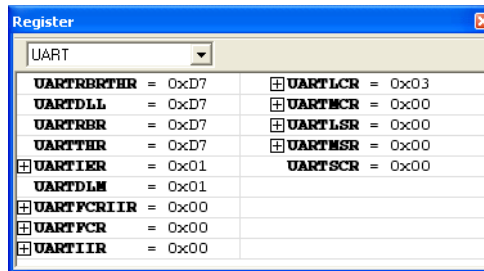


Figure 53: Register window

Every time C-SPY stops, a value that has changed since the last stop is highlighted. To edit the contents of a register, click it, and modify the value. Some registers can be expanded to show individual bits or subgroups of bits.

You can change the display format by changing the **Base** setting on the **Register Filter** page—available by choosing **Tools>Options**.

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 is only one register group in the debugger: **CPU Registers**.

In addition to the **CPU Registers** there are additional register groups 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 115.

The available register groups will be 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 C-SPY is running.

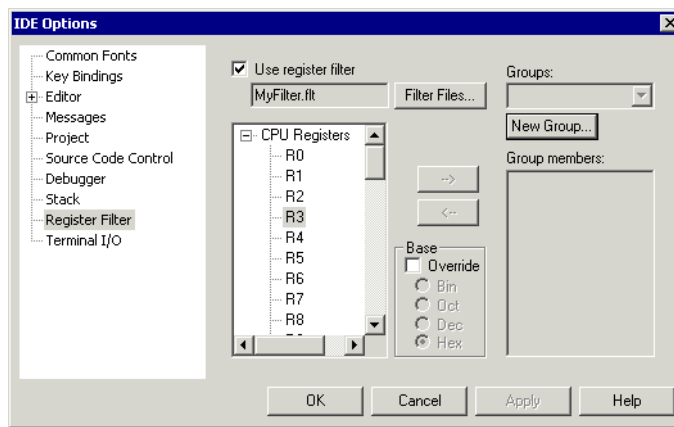


Figure 54: Register Filter page

For reference information about this dialog box, see *Register Filter options*, page 332.

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

C-SPY macros can be used solely 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 by using 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. You can set up your simulator environment automatically by writing a macro file and executing it, for instance when you start C-SPY. Another advantage is that the debug session will be documented, and if there are several engineers 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 459.

Example

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

```
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

It is possible to 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. An example of a C-SPY setup macro file `SetupSimple.mac` can be found 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 153. 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 will be 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 464.

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 macro function `execUserPreload()` is suitable. For an example, see *Remapping memory*, page 119.

Using C-SPY macros

If you decide to use C-SPY macros, you first need to create a macro file in which you define your macro functions. C-SPY needs to 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 need to list all available macro functions as well as 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 by using 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.
- A file containing macro function definitions can be registered 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 482.
- 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 will be 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 will be deactivated when you exit the debug session, and will not automatically be registered at the next debug session.

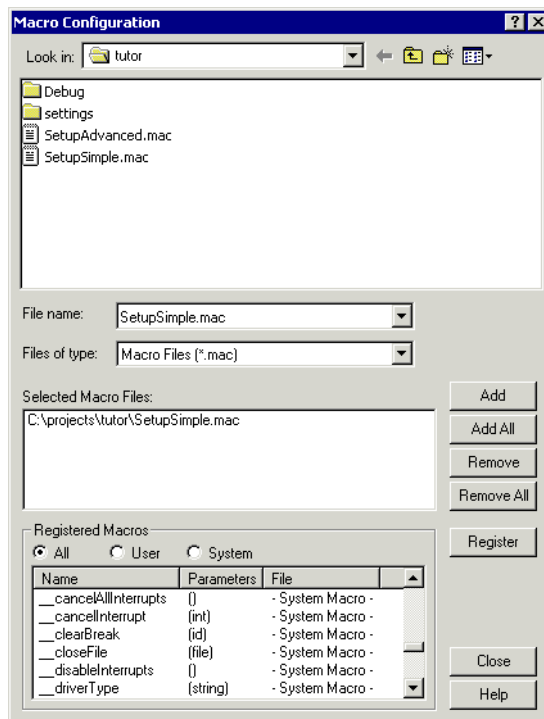


Figure 55: Macro Configuration dialog box

For reference information about this dialog box, see *Macro Configuration dialog box*, page 376.

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. You achieve this by specifying 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 define the macro functions by using the setup macro function names 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:

```
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 the following simple macro function which checks the status of a watchdog timer interrupt enable bit:

```
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.

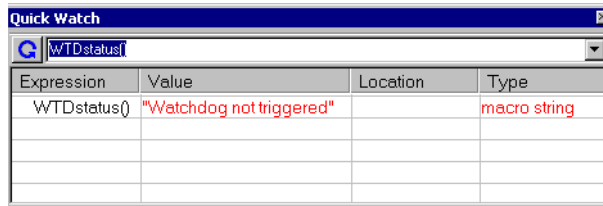


Figure 56: Quick Watch window

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:

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

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

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

The `__message` statement will log messages to the Log window.

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

- 3 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 462.

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.

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 368.

USING THE PROFILER

Before you can use the Profiling window, you must build your application using the following options:

Category	Setting
C/C++ Compiler	Output>Generate debug information
Linker	Output>Include debug information in output
Debugger	Plugins>Profiling

Table 15: 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 by right-clicking 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.

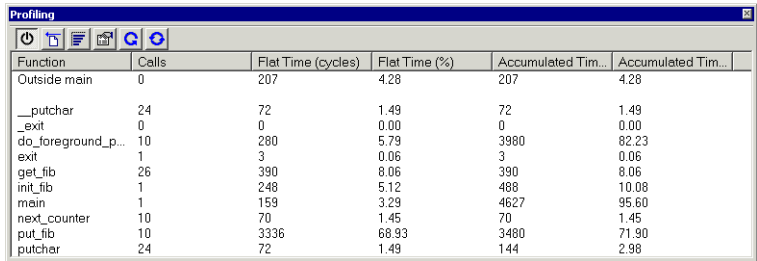


Figure 57: Profiling window

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.

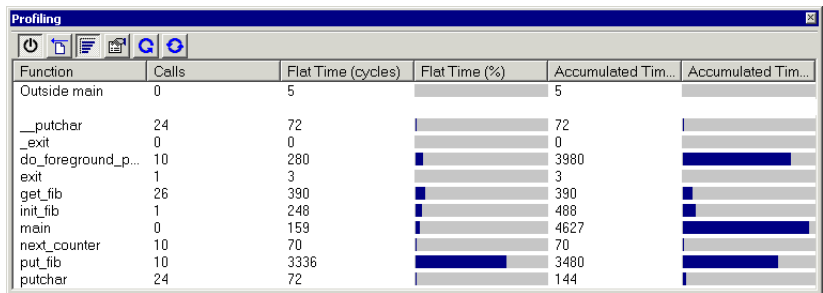


Figure 58: Graphs in Profiling window



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:

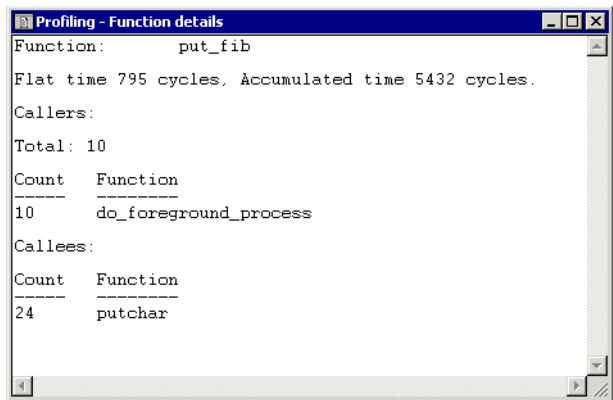


Figure 59: Function details window

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 will be saved to a file.

Code coverage

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, as well as 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 366.

Before using the Code Coverage window you must build your application using the following options:

Category	Setting
C/C++ Compiler	Output>Generate debug information
Linker	Output>Include debug information in output
Debugger	Plugins>Code Coverage

Table 16: 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 and click **Activate** to switch on the code coverage analyzer. The following window will be displayed:

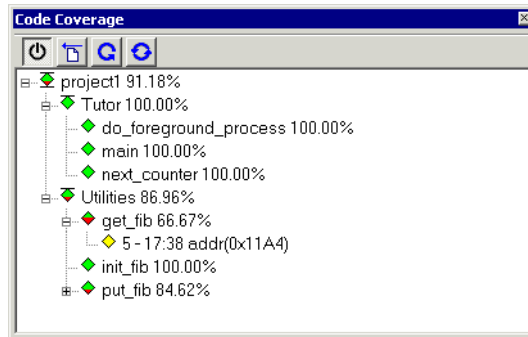


Figure 60: Code Coverage window

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.

The following 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 needs to 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 have been compiled with debug information. Thus, startup code, exit code and library code will not be displayed in the window. Furthermore, coverage information for statements in inlined functions will not be displayed. Only the statement containing the inlined function call will be marked as executed.

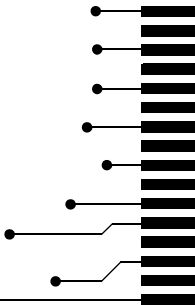
Producing reports

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 will be saved to a file.

Part 5. The 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, as well as 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 the program logic can be debugged long before any hardware is available. As 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 in the menu bar.

SIMULATOR MENU

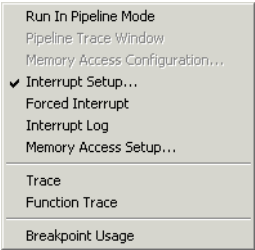


Figure 61: Simulator menu

The **Simulator** menu contains the following commands:

Menu command	Description
Run in Pipeline Mode	Toggles pipeline mode on or off. When the simulator runs in pipeline mode, it performs a cycle accurate simulation of the ARM7TDMI/ARM9TDMI pipeline. If you switch from pipeline mode to normal mode, the switch is not completely performed until the pipeline is flushed, which happens when a branch instruction is executed.
Pipeline Trace Window	Displays the Pipeline Trace window which shows the stages in the pipeline for each clock cycle and the assembler instruction handled in each stage; see <i>Pipeline Trace window</i> , page 167. This window is only available when C-SPY is running in pipeline mode.
Memory Access Configuration	Displays the Memory Access Configuration window where you can specify the number of clock cycles needed to access a specific part of the memory; see <i>Memory Access Configuration</i> , page 168. This window is only available when C-SPY is running in pipeline mode.
Interrupt Setup	Displays a dialog box to allow you to configure C-SPY interrupt simulation; see <i>Interrupt Setup dialog box</i> , page 189.
Forced Interrupts	Displays a window from which you can trigger an interrupt; see <i>Forced interrupt window</i> , page 192.
Interrupt Log	Displays a window which shows the status of all defined interrupts; see <i>Interrupt Log window</i> , page 194.

Table 17: Description of Simulator menu commands

Menu command	Description
Memory Access Setup	Displays a dialog box to simulate memory access checking by specifying memory areas with different access types; see <i>Memory Access setup dialog box</i> , page 177.
Trace	Opens the Trace window with the recorded trace data; see <i>Trace window</i> , page 170.
Function Trace	Opens the Function Trace window with the trace data for which functions were called or returned from; see <i>Function Trace window</i> , page 172.
Breakpoint Usage	Displays the Breakpoint Usage dialog box which lists all active breakpoints; see <i>Breakpoint Usage dialog box</i> , page 184.

Table 17: Description of Simulator menu commands

PIPELINE TRACE WINDOW

The Pipeline Trace window—available from the **Simulator** menu—shows the stages in the ARM core pipeline for each clock cycle and the assembler instruction handled in each stage. This allows accurate simulation of each instruction and of the pipeline flow. Simulation in pipeline mode is supported for ARM architecture 4 cores.

The Pipeline Trace window is only available when C-SPY is running in pipeline mode, and the information is only displayed when C-SPY is single-stepping in disassembly mode. To enable the Pipeline Trace window, choose **Simulator>Run In Pipeline Mode**. If you switch from pipeline mode to normal mode, the switch is not completely performed until the pipeline is flushed, which happens when a branch instruction is executed.

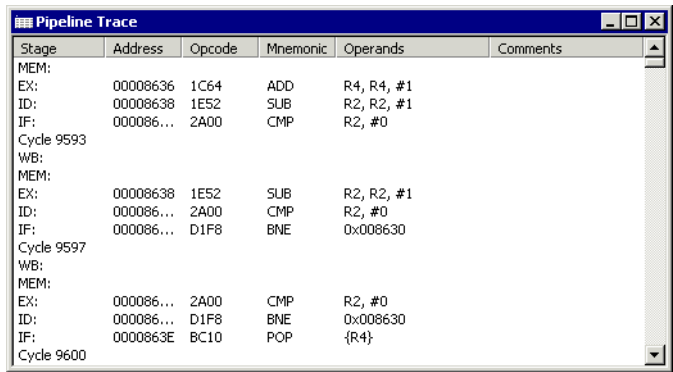


Figure 62: Pipeline Trace window

The Pipeline window shows:

Column	Description
Stage	The stages for each clock cycle.
Address	The address where the assembler instruction originates.
Opcode	Operation code for the assembler instruction.
Mnemonic	The mnemonic for the assembler instruction.
Operands	Operands for the instruction.
Comments	The absolute address of a jump.

Table 18: Pipeline window information

When a jump instruction is executed, some already fetched and decoded instructions will be removed from the pipeline.

Note that simulation in pipeline mode slows down the simulation speed.

MEMORY ACCESS CONFIGURATION

The Memory Access Configuration window—available from the **Simulator** menu—lets you specify the number of clock cycles needed to access a specific region of the address space. This is used in pipeline mode to customize a memory configuration by specifying access times and bus width. The Memory Access Configuration window is only available when C-SPY is running in pipeline mode.

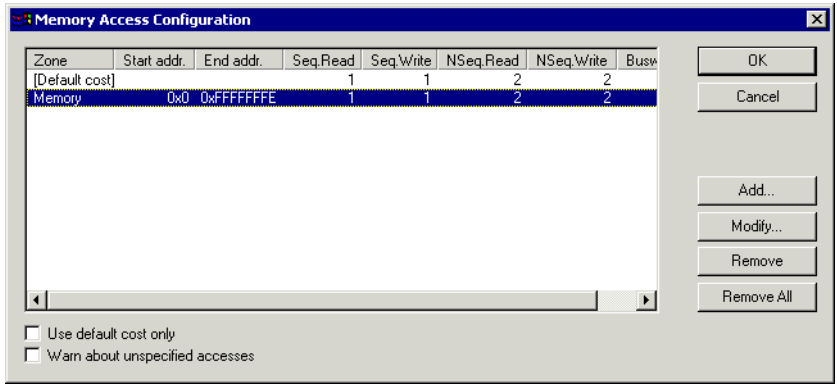


Figure 63: Memory Access Configuration window

The option **Use default cost only** disables all specified access costs and uses the default cost. Select the option **Warn about unspecified accesses** to make the debugger issue the warning Address without a specified cost was accessed for any memory access to an address outside specified access costs. This option is disabled when the **Use default cost only** option is selected.

To define a memory access region, click **Add**. This will open the **Memory access costs** dialog box.

MEMORY ACCESS COSTS DIALOG BOX

In the **Memory access costs** dialog box—available from the Memory Access Configuration window—you can define memory access regions.

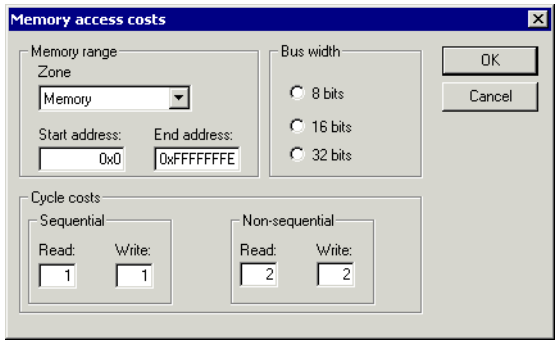


Figure 64: Memory access costs dialog box

Specify memory **Zone**, **Start address** and **End address**. Then set the number of read and write cycles required for memory access in the **Cycle costs** text boxes. You can choose between:

- | | |
|----------------|---|
| Sequential | The access time for succeeding consecutive accesses, for example access to byte 2,3,4 of a word access to byte-wide memory. |
| Non-sequential | The access time for access to a random location within this memory range. |

Finally, specify the **Bus width** and click **OK** to save the changes.

To edit an existing memory access region, select it in the list and click **Modify** to display or edit its settings, or **Remove** to delete it. To remove all memory access regions, click **Remove All**.

Using the trace system in the simulator

In the C-SPY Simulator, a *trace* is a recorded sequence of executed machine instructions. In addition, you can record the values of C-SPY expressions by selecting the expressions in the Trace Expressions window. The Function Trace window only shows trace data corresponding to calls to and returns from functions, whereas the Trace window displays all instructions.

For more detailed information about using the common features in the trace system, see *Using the trace system*, page 131.

TRACE WINDOW

The Trace window—available from the **Simulator** menu—displays a recorded sequence of executed machine instructions. In addition, the window can display trace data for expressions.

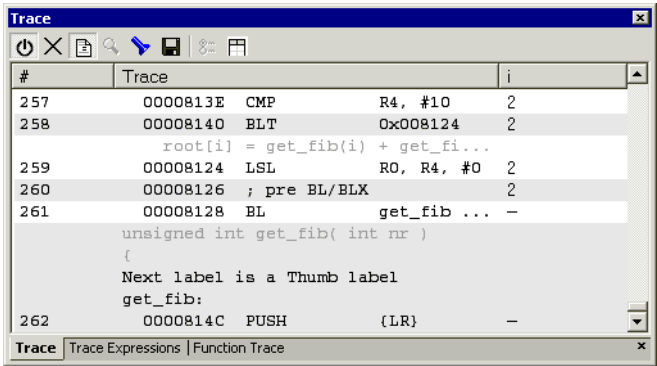


Figure 65: Trace window

C-SPY generates trace information based on the location of the program counter.

The Trace window contains the following columns:

Trace window column	Description
#	A serial number for each row in the trace buffer. Simplifies the navigation within the buffer.
Trace	The recorded sequence of executed machine instructions. Optionally, the corresponding source code can also be displayed.

Table 19: Trace window columns

Trace window column	Description
Expression	Each expression you have defined to be displayed appears in a separate column. Each entry in the expression column displays the value <i>after</i> executing the instruction on the same row. You specify the expressions for which you want to record trace information in the Trace Expressions window; see <i>Trace Expressions window</i> , page 173.

Table 19: Trace window columns (Continued)

For more information about using the trace system, see *Using the trace system*, page 131.

TRACE TOOLBAR

The Trace toolbar is available in the Trace window and in the Function trace window:

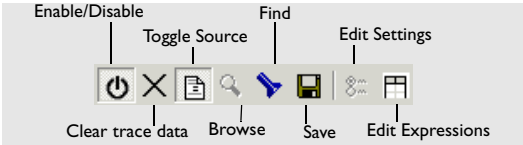


Figure 66: Trace toolbar

The following function buttons are available on the toolbar:








Toolbar button	Description
	Enable/Disable Enables and disables tracing. This button is not available in the Function trace window.
	Clear trace data Clears the trace buffer. Both the Trace window and the Function trace window are cleared.
	Toggle Source Toggles the Trace column between showing only disassembly or disassembly together with corresponding source code.
	Browse Toggles browse mode on and off for a selected item in the Trace column. For more information about browse mode, see <i>The Trace window and its browse mode</i> , page 132.
	Find Opens the Find In Trace dialog box where you can perform a search; see <i>Find in Trace dialog box</i> , page 175.
	Save Opens a standard Save dialog box where you can save the recorded trace information to a text file, with tab-separated columns.
	Edit Settings This button is not enabled in the C-SPY Simulator.

Table 20: Trace toolbar commands


Toolbar button	Description
	Opens the Trace Expressions window; see <i>Trace Expressions window</i> , page 173.

Table 20: Trace toolbar commands (Continued)

FUNCTION TRACE WINDOW

The Function Trace window—available from the **Simulator** menu—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.

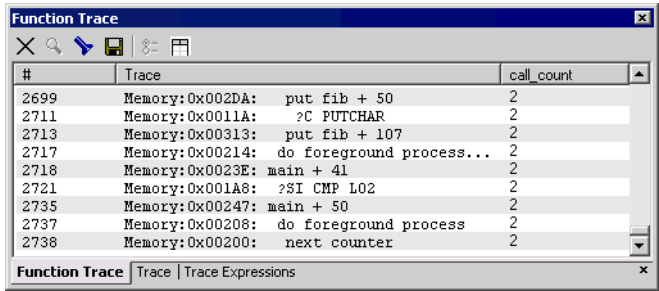


Figure 67: Function Trace window

For information about the toolbar, see *Trace toolbar*, page 171.

For more information about using the trace system, see *Using the trace system*, page 131.

TRACE EXPRESSIONS WINDOW

In the Trace Expressions window—available from the Trace window toolbar—you can specify specific expressions for which you want to record trace information.

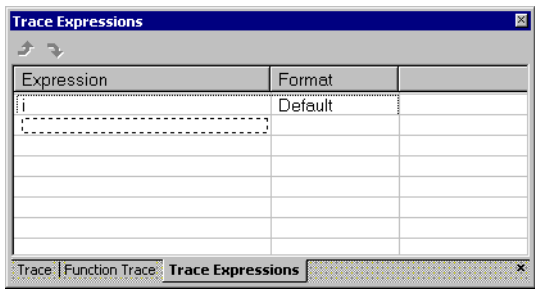


Figure 68: Trace Expressions window

In the **Expression** column, you specify any expression you want to be recorded. You can specify any expression that can be evaluated, such as variables and registers.

The **Format** column shows which display format is used for each expression.

Each row in this window will appear as an extra column in the Trace window.

For more information about using the trace system, see *Using the trace system*, page 131.

Use the toolbar buttons to change the order between the expressions:

Toolbar button	Description
Arrow up	Moves the selected row up
Arrow down	Moves the selected row down

Table 21: Toolbar buttons in the Trace Expressions window

FIND IN TRACE WINDOW

The Find In Trace window—available from the **View>Messages** menu—displays the result of searches in the trace data.

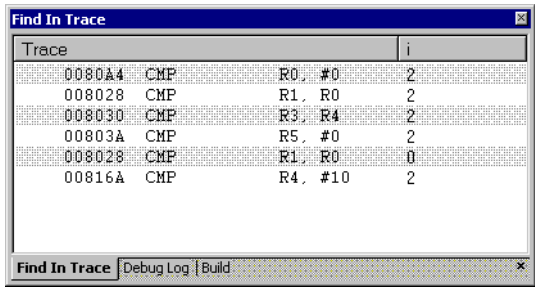


Figure 69: Find In Trace window

The Find in Trace window looks like 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.

You specify the search criteria in the **Find In Trace** dialog box. For information about how to open this dialog box, see *Find in Trace dialog box*, page 175.

For more information about using the trace system, see *Using the trace system*, page 131.

FIND IN TRACE DIALOG BOX

Use the **Find in Trace** dialog box—available by choosing **Edit>Find and Replace>Find** or from the Trace window toolbar—to specify the search criteria for advanced searches in the trace data. Note that the **Edit>Find and Replace>Find** command is context-dependent. It displays the **Find in Trace** dialog box if the Trace window is the current window or the **Find** dialog box if the editor window is the current window.

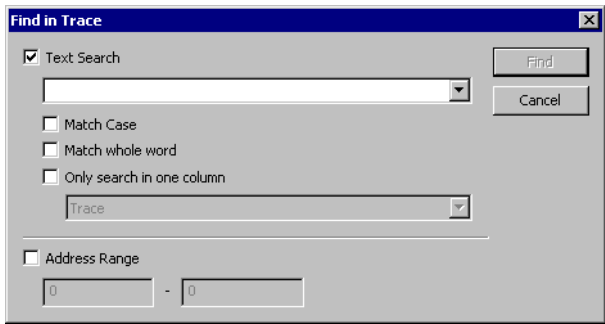


Figure 70: Find in Trace dialog box

The search results are displayed in the Find In Trace window—available by choosing the **View>Messages** command, see *Find In Trace window*, page 174.

In the **Find in Trace** dialog box, you specify the search criteria with the following settings:

Text search

A text field where you type the string you want to search for. Use the following options to fine-tune the search:

- | | |
|----------------------------------|---|
| Match Case | Searches only for occurrences that exactly match the case of the specified text. Otherwise specifying <code>int</code> will also find <code>INT</code> and <code>Int</code> . |
| Match whole word | Searches only for the string when it occurs as a separate word. Otherwise <code>int</code> will also find <code>print</code> , <code>sprintf</code> and so on. |
| Only search in one column | Searches only in the column you selected from the drop-down menu. |

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 will be searched for within the address range.

For more information about using the trace system, see *Using the trace system*, page 131.

Memory access checking

C-SPY can simulate different 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. It is not possible to map two different access types to the same memory area. You can choose between checking access type violation or checking 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. In other words, the dialog box displays the memory access setup that will be used during the simulation.

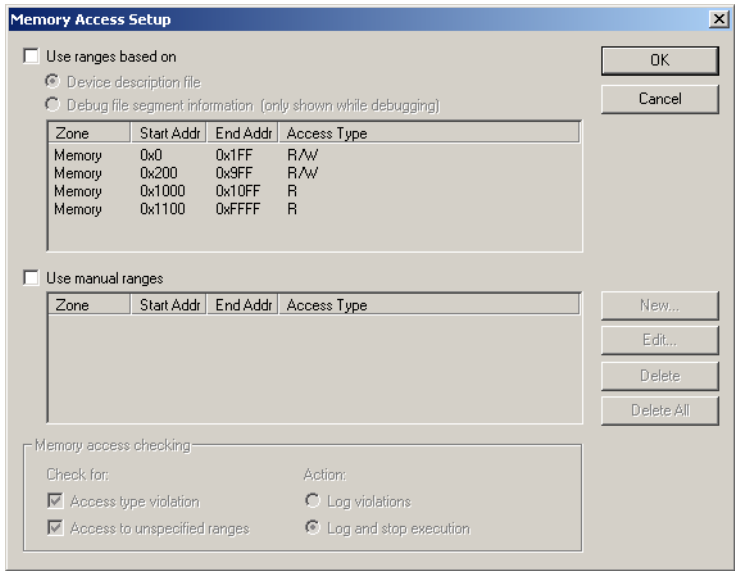


Figure 71: Memory Access Setup dialog box

Note: If you enable both the **Use ranges based on** and the **Use manual ranges** option, memory accesses will be checked for all defined ranges.

For information about the columns and the properties displayed, see *Edit Memory Access dialog box*, page 179.

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 will be loaded from the device description file
- **Debug file segment information**, which means the properties will be 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 179.

The ranges you define manually are saved between debug sessions.

Memory access checking

Use the **Check for** options to specify what to check for. Choose between:

- Access type violation
- Access to unspecified ranges.

Use the **Action** options to specify the action to be performed if there is an access violation. 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 the following buttons:

Button	Description
OK	Standard OK.
Cancel	Standard Cancel.
New	Opens the Edit Memory Access dialog box, where you can specify a new memory range and attach an access type to it; see <i>Edit Memory Access dialog box</i> , page 179.
Edit	Opens the Edit Memory Access dialog box, where you can edit the selected memory area. See <i>Edit Memory Access dialog box</i> , page 179.
Delete	Deletes the selected memory area definition.
Delete All	Deletes all defined memory area definitions.

Table 22: 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.

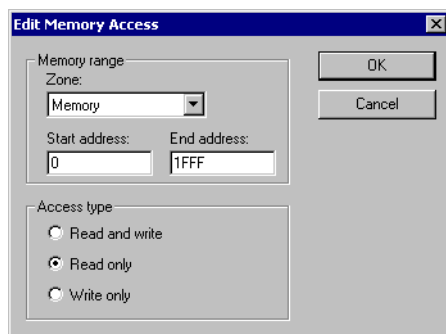


Figure 72: Edit Memory Access dialog box

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 <i>Memory addressing</i> , page 143.
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**. It is not possible to 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.

For information about the breakpoint system, see the chapter *Using breakpoints* in this guide. For detailed information about code breakpoints, see *Code breakpoints dialog box*, page 283.

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 will be 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; by using:

- A dialog box, see *Data breakpoints dialog box*, page 180
- A system macro, see `__setDataBreak`, page 485
- The Memory window, see *Setting a breakpoint in the Memory window*, page 137
- The editor window, see *Editor window*, page 274.

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.

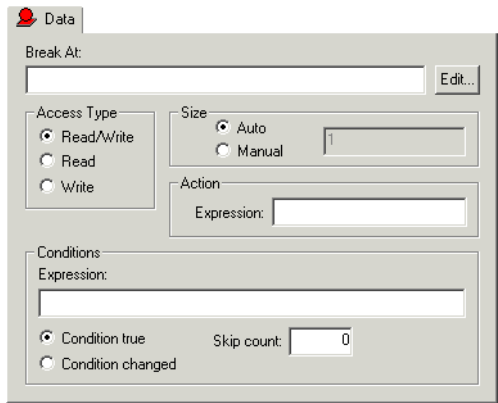


Figure 73: Data breakpoints dialog box

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 287.

Access Type

Use the options in the **Access Type** area to specify the type of memory access that triggers data or immediate breakpoints.

Memory Access type	Description
Read/Write	Read or write from location.
Read	Read from location.
Write	Write to location.

Table 23: 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 182.)

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 the size can be specified:

- **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.

Conditions	Description
Expression	A valid expression conforming to the C-SPY expression syntax.
Condition true	The breakpoint is triggered if the value of the expression is true.
Condition changed	The breakpoint is triggered if the value of the expression has changed since it was last evaluated.
Skip count	The number of times that the breakpoint must be fulfilled before a break occurs (integer).

Table 24: Breakpoint conditions

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 183
- A system macro, see `__setSimBreak`, page 486.

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.

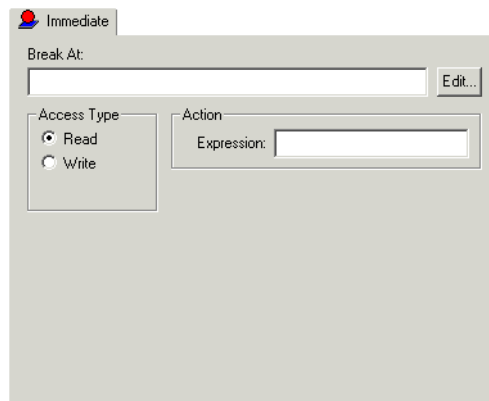


Figure 74: Immediate breakpoints page

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 287.

Access Type

Use the options in the **Access Type** area to specify the type of memory access that triggers data or immediate breakpoints.

Memory Access type	Description
Read	Read from location.
Write	Write to location.

Table 25: Memory Access types

Note: Immediate breakpoints do not stop execution at all; they only suspend it temporarily. See *Using breakpoints in the simulator*, page 179.

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.

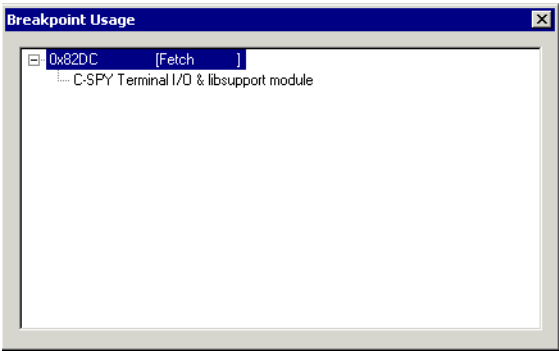


Figure 75: Breakpoint Usage dialog box

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 139.

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. It is possible to configure the interrupt simulation system so that it resembles your hardware interrupt system. By using simulated interrupts in conjunction with C-SPY macros and breakpoints, you can compose a complex simulation of, for instance, interrupt-driven peripheral devices. Having simulated interrupts also lets you test the logic of your interrupt service routines.

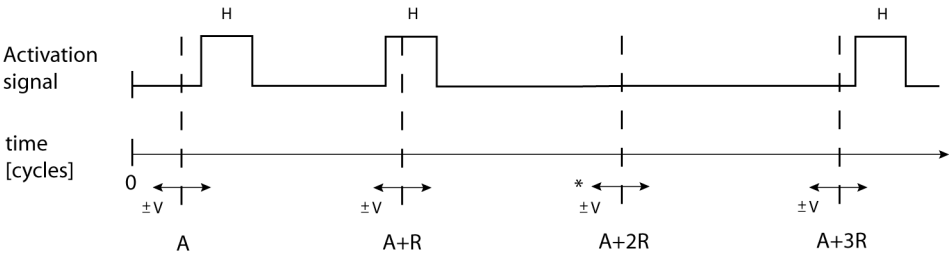
The interrupt system has the following features:

- Simulated interrupt support for the ARM core
- Single-occasion or periodical interrupts based on the cycle counter
- Predefined interrupts for different 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 it can be turned 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 by 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*.



* If probability is less than 100%, some interrupts may be omitted.

A = Activation time

R = Repeat interval

H = Hold time

V = Variance

Figure 76: 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 can be used for locating timing problems in your application. The **Interrupt Setup** dialog box displays the available status information. For an interrupt, the following statuses can be displayed: *Idle*, *Pending*, *Executing*, *Executed*, *Removed*, or *Expired*.

Status	Description
Idle	Interrupt activation signal is low (deactivated).
Pending	Interrupt activation signal is active, but the interrupt has not been acknowledged yet by the interrupt handler.
Executing	The interrupt is currently being serviced, that is the interrupt handler function is executing.
Executed	This is a single-occasion interrupt and it has been serviced.
Removed	The interrupt has been removed by the user, but because the interrupt is currently executing it is visible in the Interrupt Setup dialog box until it is finished.
Expired	This is a single-occasion interrupt which was not serviced while the interrupt activation signal was active.

Table 26: 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:

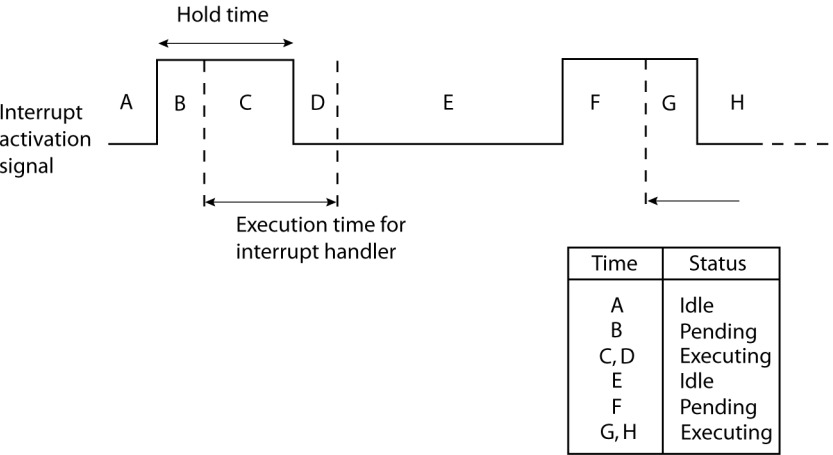


Figure 77: Simulation states - example 1

Note: The interrupt activation signal—also known as the pending bit—will be automatically deactivated the moment the interrupt is acknowledged by the interrupt handler.

If the interrupt repeat interval is shorter than the execution time, and the interrupt is re-entrant (or non-maskable), the status information at different times can look like this:

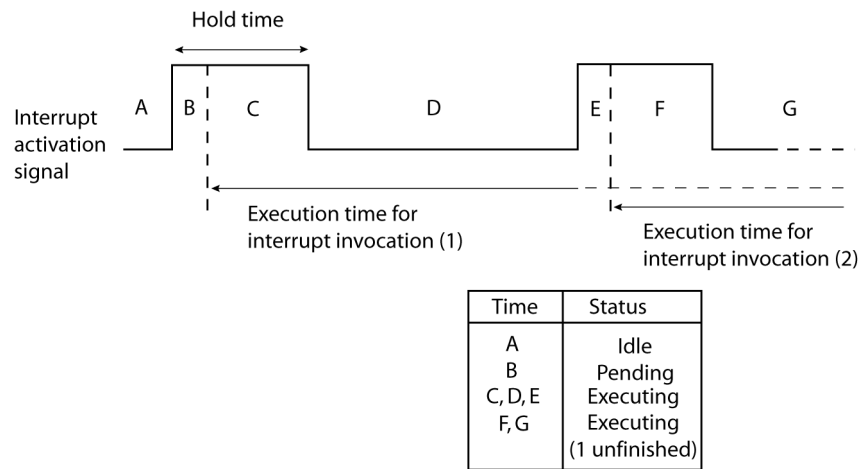


Figure 78: Simulation states - example 2

In this case, the execution time of the interrupt handler is too long compared to the repeat interval, which might indicate that you should rewrite your interrupt handler and make it faster, or that you should specify a longer repeat interval for the interrupt simulation system.

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.

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 derivatives, 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` directory. The default settings will be 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 118 .

INTERRUPT SETUP DIALOG BOX

The **Interrupt Setup** dialog box—available by choosing **Simulator>Interrupt Setup**—lists all defined interrupts.

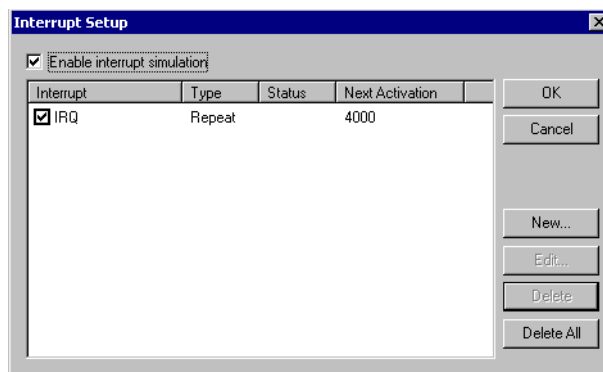


Figure 79: Interrupt Setup dialog box

The option **Enable interrupt simulation** enables or disables interrupt simulation. If the interrupt simulation is disabled, the definitions remain but no interrupts will be 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 the following information:

Interrupt	Lists all interrupts.
Type	Shows the type of the interrupt. The type can be Forced , Single , or Repeat .
Status	Shows the status of the interrupt. The status can be Idle , Removed , Pending , Executing , or Expired .
Next Activation	Shows the next activation time in cycles.

Note: For repeatable interrupts there might be additional information in the **Type** column about how many interrupts of the same type that is 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.

Only non-forced interrupts may be edited or removed.

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.

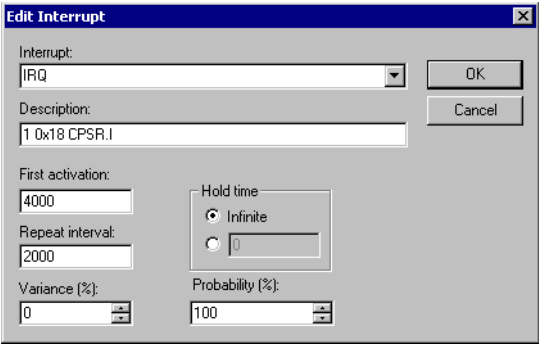


Figure 80: Edit Interrupt dialog box

For each interrupt you can set the following options:

Interrupt	A drop-down list containing all available interrupts. Your selection will automatically update the Description box. The list is populated with entries from the device description file that you have selected.
Description	Contains the description of the selected interrupt, if available. The description is retrieved from the selected device description file and consists of a string describing the vector address, priority, enable bit, and pending bit, separated by space characters. For interrupts specified using the system macro <code>__orderInterrupt</code> , the Description box will be 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 may 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.
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.

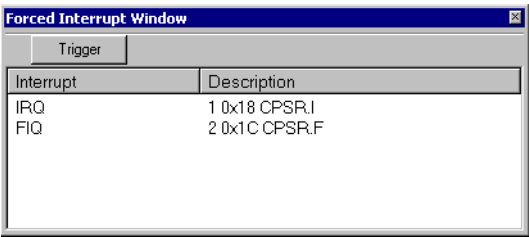


Figure 81: Forced Interrupt window

To force an interrupt, the interrupt simulation system must be enabled. To enable the interrupt simulation system, see *Interrupt Setup dialog box*, page 189.

The Forced Interrupt window lists all available interrupts and their definitions. The description field consists of a string describing the vector address, priority, enable bit, and pending bit, separated by space characters.

By selecting an interrupt and clicking the **Trigger** button, an interrupt of the selected type is generated.

A triggered interrupt will have the following characteristics:

Characteristics	Settings
First Activation	As soon as possible (0)
Repeat interval	0
Hold time	Infinite
Variance	0%
Probability	100%

Table 27: 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. By writing a macro function containing definitions for the simulated interrupts you can automatically execute the functions 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 193.

For detailed reference information about each macro, see *Description of C-SPY system macros*, page 467.

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 153.

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 there are too many interrupts executing simultaneously, a warning might be issued.

To avoid these problems, you can use the

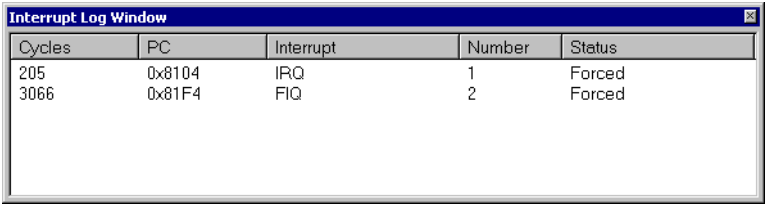
`__popSimulatorInterruptExecutingStack` macro to inform the interrupt simulation system that the interrupt handler has finished executing, as if the normal instruction used for returning from an interrupt handler was executed. You can use the following procedure:

- 1 Set a code breakpoint on the instruction that returns from the interrupt function.
- 2 Specify the `__popSimulatorInterruptExecutingStack` macro as a condition to the breakpoint.

When the breakpoint is triggered, the macro is executed and then the application continues to execute automatically.

INTERRUPT LOG WINDOW

The **Interrupt Log** window—available from the **Simulator** menu—displays runtime information about the interrupts that you have activated in the **Interrupts** dialog box or forced via the **Forced Interrupt** window. The information is useful for debugging the interrupt handling in the target system.



Cycles	PC	Interrupt	Number	Status
205	0x8104	IRQ	1	Forced
3066	0x81F4	FIQ	2	Forced

Figure 82: Interrupt Log window

The columns contain the following information:

Column	Description
Cycles	The point in time, measured in cycles, when the event occurred.
PC	The value of the program counter when the event occurred.
Interrupt	The interrupt as defined in the device description file.
Number	A unique number assigned to the interrupt. The number is used for distinguishing between different interrupts of the same type.
Status	Shows the status of the interrupt, which can be Triggered, Forced, Executing, Finished, or Expired. <ul style="list-style-type: none">• Triggered: The interrupt has passed its activation time.• Forced: The same as Triggered, but the interrupt has been forced from the Forced Interrupt window.• Executing: The interrupt is currently executing.• Finished: The interrupt has been executed.• Expired: The interrupt hold time has expired without the interrupt being executed.

Table 28: Description of the Interrupt Log window

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

Note: If the window becomes full of entries, the first entries will be erased.

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.

```
/* Enables use of extended keywords */
#pragma language=extended

#include <intrinsics.h>
#include <arm_interrupt.h>
#include <oki/ioml674001.h>
#include <stdio.h>

unsigned int ticks = 0;

/* Installs the function 'function' at the vector address
'vector'. A branch instruction to the function will be placed
at the vector address, the old contents at the vector location
will be returned by install_handler and can be used to chain
another handler.*/
unsigned int install_handler (unsigned int *vector,
                             unsigned int function)
{
    unsigned int vec, old_vec;
    vec = ((function - (unsigned int)vector - 8) >> 2);
    old_vec = *vector;
    vec |= 0xea000000; /* add opcode for B instruction */
    *vector = vec;
    old_vec &= ~0xea000000;
    old_vec = ( old_vec << 2 ) + (unsigned int)vector + 8;
    return(old_vec);
}

/* IRQ handler */
__irq __arm void irqHandler(void)
{
    /* We use only system timer interrupts, so we do not need
    to check the interrupt source. */
    ticks += 1;
    TMOVFR_bit.OVF = 1; /* Clear system timer overflow flag */
}
```

```
int main( void )
{
    /* IRQ setup code */
    install_handler(irqvec, (unsigned int)irqHandler);
    __enable_interrupt();
    /* Timer setup code */
    ILC0_bit.ILR0 = 4; /* System timer interrupt priority */
    TMRLR_bit.TMRLR = 1E5; /* System timer reload value */
    TMEN_bit.TCEN = 1; /* Enable system timer */
    while (ticks < 100);
    printf("Done\n");
}
```

To simulate and debug an interrupt, perform the following 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 TimerInterrupt example, verify the following settings:

Option	Settings
Interrupt	IRQ
First Activation	4000
Repeat interval	2000
Hold time	0
Probability %	100
Variance %	0

Table 29: Timer interrupt settings

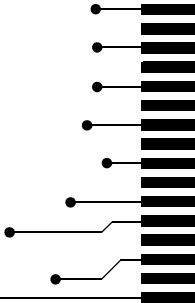
Click **OK**.

- 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.

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
- Using flash loaders.





Introduction to C-SPY® hardware debugger systems

This chapter introduces you to the IAR C-SPY hardware debugger systems and how they differ from the IAR C-SPY Simulator.

The chapters specific to C-SPY 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. For a quick introduction, see *Part 2. Tutorials*.

Please note that additional features may have been added to the software after this guide was printed. The release notes contain the latest information.

The IAR C-SPY hardware debugger systems

The IAR C-SPY Debugger 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.

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

- Simulator
- RDI (Remote Debug Interface)
- J-Link/J-Trace JTAG interface
- GDB Server
- 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).

For further details about the concepts that are related to the IAR C-SPY Debugger, see *Debugger concepts*, page 111.

DIFFERENCES BETWEEN THE C-SPY DRIVERS

The following table summarizes the key differences between the C-SPY drivers:

Feature	Simulator	Angel	GDB Server	IAR ROM-monitor	J-Link/J-Trace	LMI FTDI	Mac-raigor	RDI
Data breakpoints	x				x 2)	x 4)	x 2)	
Code breakpoints	x	x	x	x	x 2)	x 4)	x 2)	x
Execution in real time		x	x	x	x	x	x	x
Zero memory footprint	x				x	x	x	x
Simulated interrupts	x							
Real interrupts		x	x	x	x	x	x	x
Live Watch	x				x 6)			
Cycle counter	x							
Code coverage	x				x 5)			
Data coverage	x							
Profiling	x	x 1)	x 1) 3)	x 1)	x 1) 3)	x 1) 3)	x 1) 3)	x 1) 3)

Table 30: 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 160.
- 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 228.

Contact your software distributor or IAR representative for information about available C-SPY drivers. Below are general descriptions of the different drivers.

Getting started

The following documents containing information about how to set up various debugging systems are available in the `arm\doc` subdirectory:

File	Debugger system
<code>rdi_quickstart.htm</code>	Quickstart reference for RDI-controlled JTAG debug interfaces
<code>gdbserver_quickstart.htm</code>	Quickstart reference for a GDB Server using OpenOCD together with STR9-comStick
<code>angel_quickstart.htm</code>	Quickstart reference for Angel ROM-monitors and JTAG interfaces
<code>iar_rom_quickstart.htm</code>	Quickstart reference for IAR and OKI ROM-monitor

Table 31: Available quickstart reference information

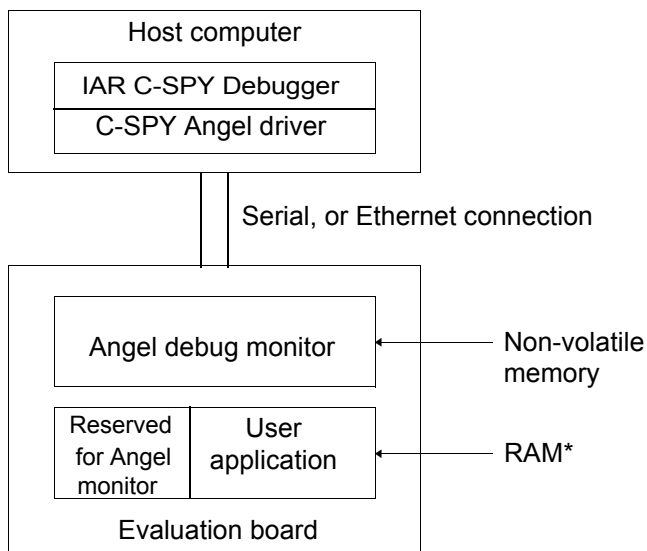
The IAR C-SPY Angel debug monitor driver

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.

Figure 83: C-SPY Angel debug monitor communication overview

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 any of the GDB Server-based JTAG solutions available, 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 204.

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 218.

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.

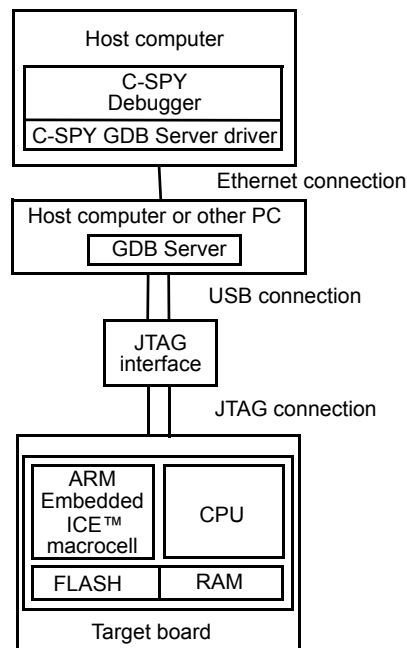


Figure 84: C-SPY GDB Server communication overview

CONFIGURING THE OPENOCD SERVER

Follow these instructions to configure the OpenOCD Server:

- 1 Install IAR Embedded Workbench for ARM.
- 2 Download OpenOCD (Open On-Chip Debugger) from <http://www.yagarto.de> or <http://openocd.berlios.de/web> and install the package.

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.

- 3 Start the OpenOCD server from a command line window and specify the configuration file `str912_comStick.cfg`, available in the ARM\examples\ST\STR9-comStick directory in your IAR Embedded Workbench installation. For example:

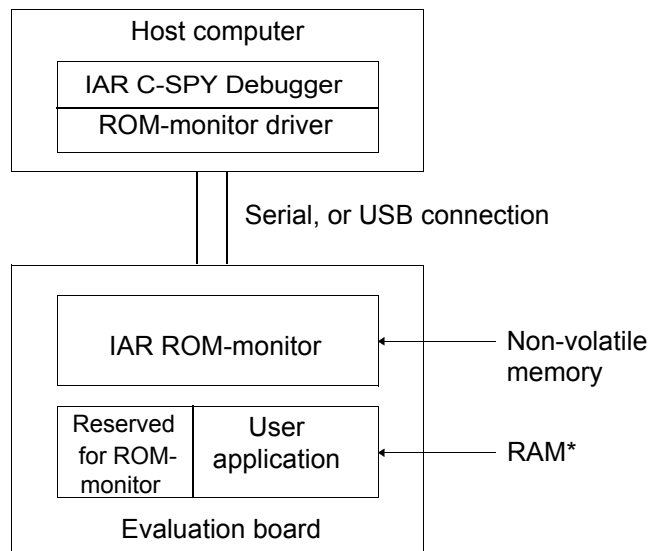
```
"C:\Program Files\openocd-2007re204\bin\openocd-ftd2xx" --file
"C:\Program Files\IAR Systems\Embedded Workbench
5.0\arm\examples\ST\STR91x\STR9-comStick\str912_comStick.cfg"
```

- 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 ADμC7xxx, 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 for Analog Devices AD μ C7xxx, the code that you want to debug can be located in flash memory. To maintain debug functionality, the ROM-monitor may simulate some instructions, for example when single stepping.



* For some ROM-monitors, the user application can be located in flash memory.

Figure 85: C-SPY ROM-monitor communication overview

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 ARM IAR J-Link driver, C-SPY can connect to the IAR J-Link JTAG interface and the IAR J-Trace JTAG interface. JTAG is a standard on-chip debug connection available on most ARM processors.

Before you can use the J-Link/J-Trace JTAG interface over the USB port, the Segger J-Link/J-Trace USB driver must be installed; see *Installing the J-Link USB driver*, page 206. 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 225.

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.

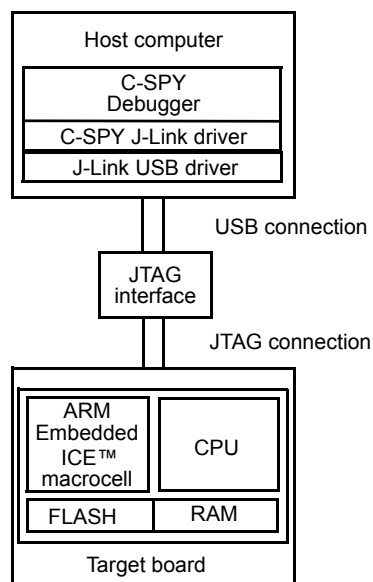


Figure 86: C-SPY J-Link communication overview

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.

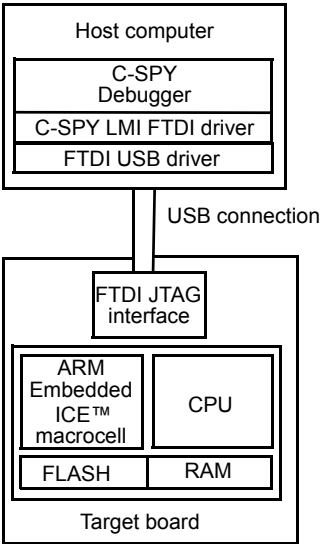
Note that J-Link will continuously blink until the USB driver has established contact with the J-Link interface. When contact has been established, J-Link will start with a slower blink to indicate that it is alive.

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 230.



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\FTDI` 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 RAVEN, WIGGLER, 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 233.

The C-SPY Macraigor driver communicates with the JTAG interface module over a parallel, serial, or Ethernet connection. The JTAG interface module, in turn, communicates with the JTAG module on the hardware.

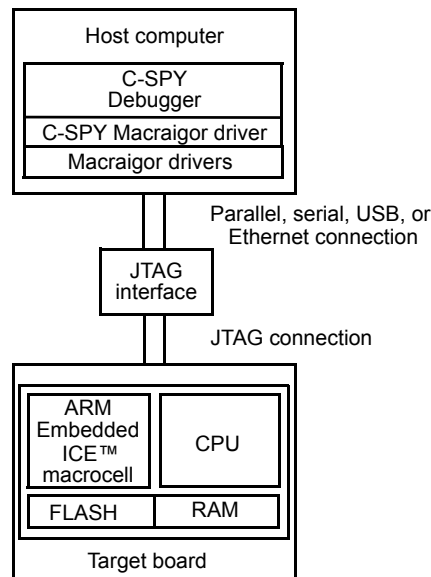


Figure 87: C-SPY Macraigor communication overview

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 233. 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.

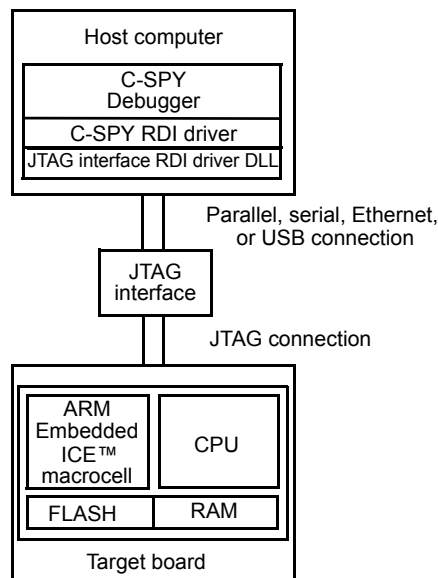


Figure 88: C-SPY RDI communication overview

For further information, see the `rdi_quickstart.htm` file, or refer to the manufacturer's documentation.

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

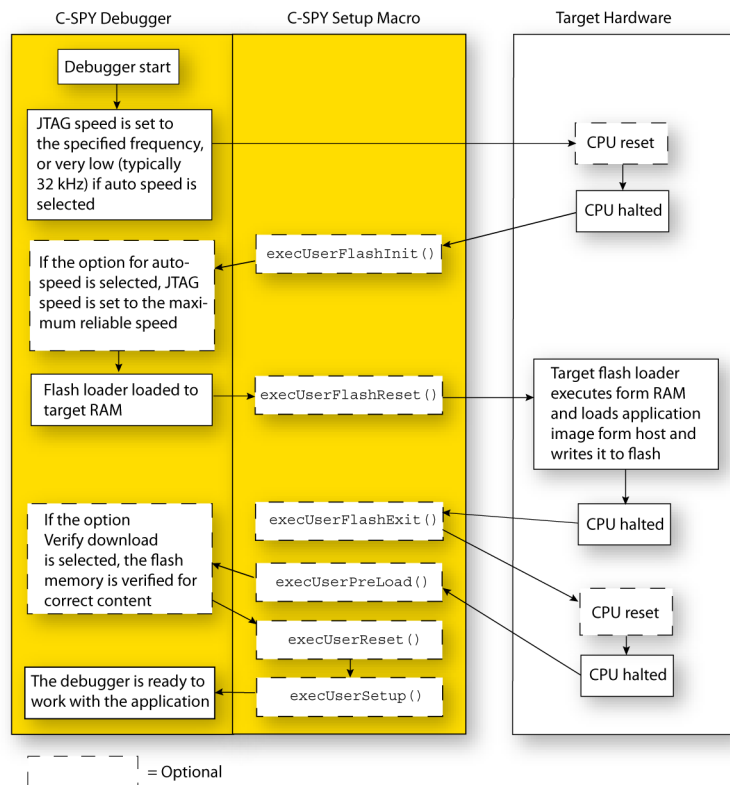


Figure 89: Debugger startup when debugging code in flash

DEBUGGING CODE IN RAM

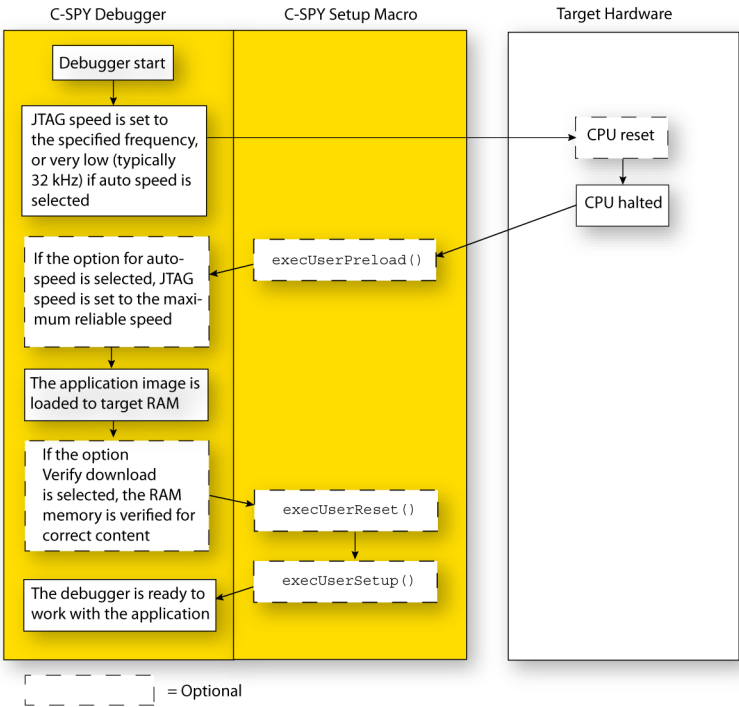


Figure 90: Debugger startup when debugging code in RAM

Hardware-specific debugging

This chapter describes the options and settings needed for using the C-SPY hardware debugger systems. You will also get information about how to use the JTAG watchpoints and information about the trace system and breakpoints. This 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 ROM-monitor driver
- Debugging using the IAR C-SPY GDB Server 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 third-party drivers
- Using the trace system in hardware debugger systems
- Using breakpoints in the hardware debugger systems.

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:

- I 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
- 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 **Extra Options** and **Plugins**, see *Debugger options*, page 429.

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 215
- *Angel*, page 216
- *GDB Server*, page 218
- *IAR ROM-monitor*, page 219
- For J-Link/J-Trace, see *Setup*, page 221 and *Connection*, page 224
- *Macraigor*, page 231
- *RDI*, page 233
- *Third-Party Driver*, page 236.

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.

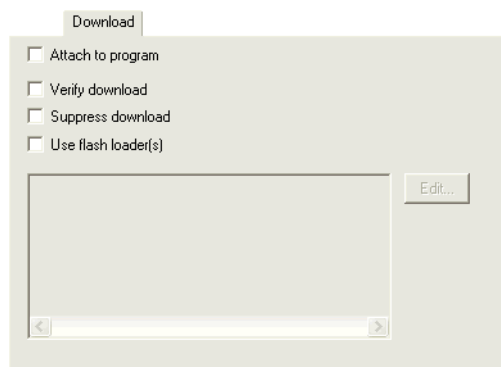


Figure 91: C-SPY Download options

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. 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, you

cannot build the debugged application with the linker option **With runtime control modules** 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 will be used as default. Press the **Edit** button to open the **Flash Loader Overview** dialog box.

To read more about flash loaders, see *Using flash loaders*, page 255.

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.

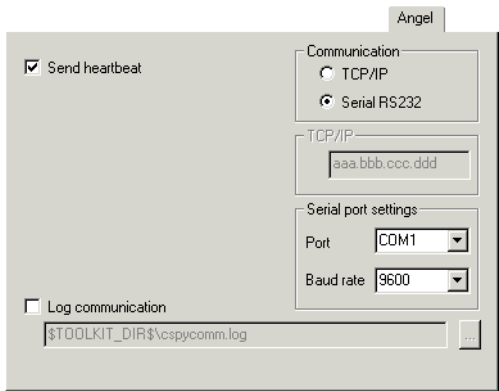


Figure 92: C-SPY Angel options

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** combo box.

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 218,
- *Using breakpoints in the hardware debugger systems*, page 243
- *The GDB Server menu*, page 218.

GDB SERVER

This section describes the options that specify the GDB Server. In the IAR Embedded Workbench IDE, choose **Project>Options**, select the **GDB Server** category, and click the **GDB Server** tab.



Figure 93: GDB Server options

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.

THE GDB SERVER MENU

When you are using the C-SPY GDB Server driver, the additional menu **GDB Server** appears in C-SPY.



Figure 94: The GDB Server menu

The following commands are available on the **GDB Server** menu:

Menu command	Description
Breakpoint Usage	Opens the Breakpoint Usage dialog box to list all active breakpoints; see <i>Breakpoint Usage dialog box</i> , page 250.

Table 32: 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 AD μ C7xxx, 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.

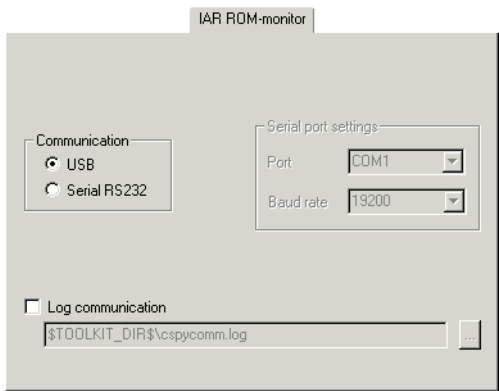


Figure 95: IAR C-SPY ROM-monitor options

Communication

Use this option to select the ROM-monitor communication link. RS232 serial port connection and USB connection are supported.

Serial port settings

Use the **Port** combo box 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** combo box. 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 JTAG interface, you should be familiar with the following details:

- The C-SPY options specific to the J-Link/J-Trace JTAG interface, see *Setup*, page 221, *Connection*, page 224, and *Using breakpoints in the hardware debugger systems*, page 243
- *The J-Link menu*, page 225
- *Using the trace system in hardware debugger systems*, page 237
- *Live watch and use of DCC*, page 228
- *Using breakpoints in the hardware debugger systems*, page 243
- *Using JTAG watchpoints*, page 251.

SETUP

This section describes the options that specify the J-Link/J-Trace interface. In the IAR Embedded Workbench IDE, choose **Project>Options**, select the **J-Link/J-Trace** category, and click the **Setup** tab.

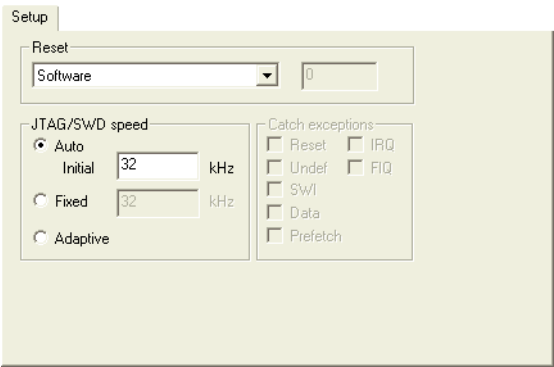


Figure 96: C-SPY J-Link/J-Trace Setup options

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 only	The core is reset via the VECTRESET bit; the peripheral units are not affected.
Core and peripherals	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.

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:

Hardware, halt after delay (ms)	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.
Hardware, halt using Breakpoint	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.
Hardware, halt at 0	Hardware reset. The processor is halted by placing a breakpoint at zero. Note that this is not supported by all ARM microcontrollers.
Hardware, halt using DBGRQ	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.
Software	Software reset. Sets PC to the program entry address.
Software, Analog devices	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>Target page.
Hardware, NXP LPC	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>Target page.
Hardware, Atmel AT91SAM7	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>Target page.

For more details about the different reset strategies, see the *J-Link / J-Trace User's Guide*.

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 **Flash download** is selected, also once after flash download, see Figure 89, *Debugger startup when debugging code in flash*, page 211, and Figure 90, *Debugger startup when debugging code in RAM*, page 212.



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 119.

JTAG speed

Use the JTAG speed options to set the JTAG communication speed in kHz.

Auto

If you use the **Auto** option, the J-Link interface 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 may 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*.

Catch exceptions

For details about this option, see *Breakpoints on vectors*, page 250.

CONNECTION

In the IAR Embedded Workbench IDE, choose **Project>Options**, select the **J-Link/J-Trace** category, and click the **Connection** tab.

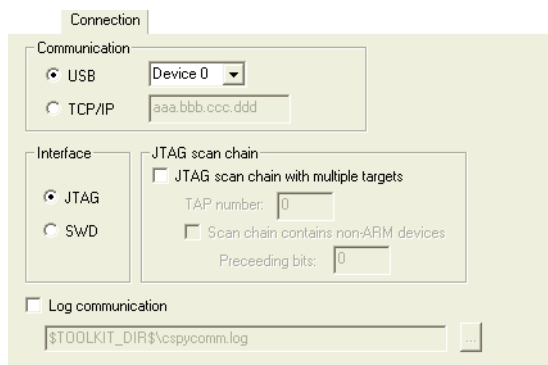


Figure 97: C-SPY J-Link/J-Trace Connection options

Communication

Use this option to select the communication link. 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 Setup dialog box*, page 226.

JTAG scan chain

If there are more than one device on the JTAG scan chain, enable the **JTAG scan chain with multiple targets** option, and specify **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.

THE J-LINK MENU

When you are using the C-SPY J-Link driver, the additional menu **J-Link** appears in C-SPY.

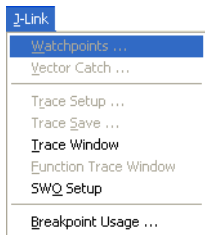


Figure 98: The J-Link menu

The following commands are available on the **J-Link** menu:

Menu command	Description
Watchpoints	Opens a dialog box for setting watchpoints, see <i>JTAG watchpoints dialog box</i> , page 252.
Vector Catch	Opens a dialog box for setting a breakpoint directly on a vector in the interrupt vector table, see <i>Breakpoints on vectors</i> , page 250. Note that this command is not available for all ARM cores.
Trace Setup	Opens the Trace Setup dialog box to configure capturing trace information; see <i>Trace Setup dialog box</i> , page 238.

Table 33: Commands on the J-Link menu

Menu command	Description
Trace Save	Opens the Trace Save dialog box to save the captured trace data to a file; see <i>Trace Save dialog box</i> , page 240.
Trace Window	Opens the Trace window to display the captured trace data; see <i>Trace window</i> , page 240.
Function Trace Window	Opens the Function Trace window to display a subset of the trace data displayed in the Trace window; see <i>Function Trace window</i> , page 172.
SWO Setup	Opens the SWO Setup dialog box; see <i>SWO Setup dialog box</i> , page 226.
Breakpoint Usage	Opens the Breakpoint Usage dialog box to list all active breakpoints; see <i>Breakpoint Usage dialog box</i> , page 250. Note that this command is only available when the SWD interface is used.

Table 33: Commands on the J-Link menu (Continued)

SWO SETUP DIALOG BOX

Use the **SWO Setup** dialog box—available from the **J-Link** menu—to set up the serial-wire output communication channel.

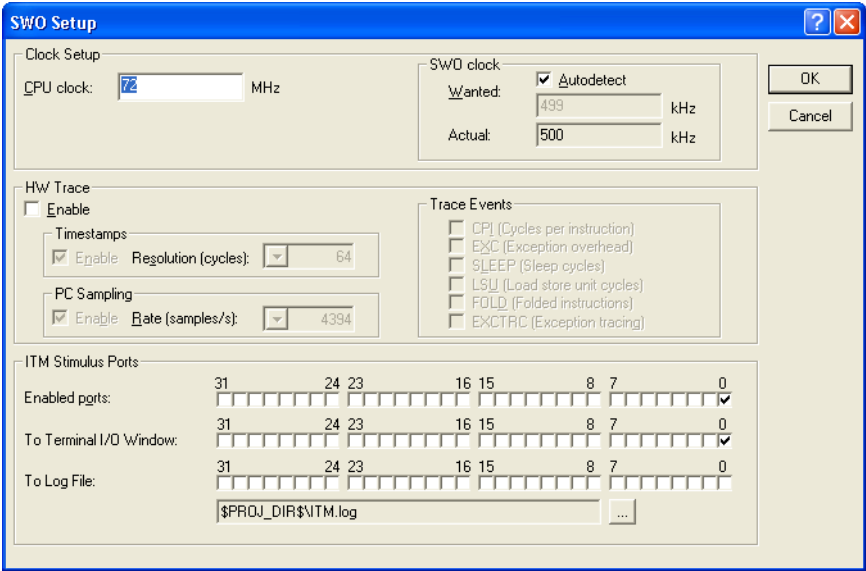


Figure 99: SWO Setup dialog box

Note: This dialog box requires the SWD interface between the J-Link debug probe and the target system, see *Interface*, page 224.

CPU clock

Use this option to specify the clock frequency of the internal processor clock, `HCLK`, given in MHz. The value can have decimals.

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.

The clock frequency actually used is displayed in the **Actual** text box.

HW Trace

Use this option to specify the hardware trace information to be displayed in the Trace window. Choose between:

- Timestamps

Enables timestamps for every ITM stimulus or HW trace packet, or group of packets, that is sent over the SWO communication channel. Use the resolution drop-down list to choose the resolution of the timestamp value.
- PC Sampling

Enables sampling of the program counter register, `PC`, at regular intervals. 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 probably not work because the SWO communication channel is not fast enough to handle that much data.
- Trace Events

Enables exception tracing or cycle counters that can be used for special types of profiling. Some of these events can overflow the SWO communication channel and thus do not work if the SWO frequency is too low. For more information about trace events, see the Cortex-M documentation from ARM Ltd.

ITM Stimulus Ports

The ITM Stimulus Ports are used for sending data from your application to the debugger host without stopping 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 which ports that will send data to the Terminal I/O window.
To Log File	Specifies which ports that will send 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.

LIVE WATCH AND USE OF DCC

The following possibilities for using live watch apply:

For Cortex-3

Access to memory or setting breakpoints is always possible during execution. DCC 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 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 `DCC_Process.c` and `DCC_HandleDataAbort.s` located in `arm\src\debugger\dcc` to your project and call the `DCC_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 `DCC_HandleDataAbort` handler.

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 221 and *Breakpoints options*, page 243
- The **LMI FTDI** menu, see *The LMI FTDI menu*, page 230
- The breakpoint system, see *Using breakpoints in the hardware debugger systems*, page 243.

SETUP

This section describes the options that specify the J-Link/J-Trace interface. In the IAR Embedded Workbench IDE, choose **Project>Options**, select the **LMI FTDI** category, and click the **Setup** tab.

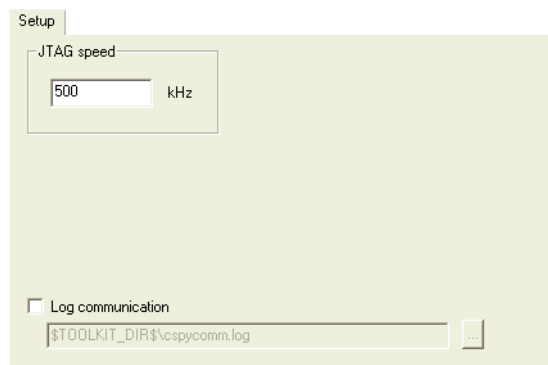


Figure 100: C-SPY LMI FTDI Setup options

JTAG 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.

THE LMI FTDI MENU

When you are using the C-SPY LMI FTDI driver, the additional menu **LMI FTDI** appears in C-SPY.



Figure 101: The LMI FTDI menu

The following commands are available on the **LMI FTDI** menu:

Menu command	Description
Breakpoint Usage	Opens the Breakpoint Usage dialog box to list all active breakpoints; see <i>Breakpoint Usage dialog box</i> , page 250.

Table 34: Commands on the RDI 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 231 and *Breakpoints options*, page 243
- *The Macraigor JTAG menu*, page 233
- *Using breakpoints in the hardware debugger systems*, page 243
- *Using JTAG watchpoints*, page 251.

MACRAIGOR

In the IAR Embedded Workbench IDE, choose **Project>Options** and click the **Macraigor** tab in the **Debugger** category.

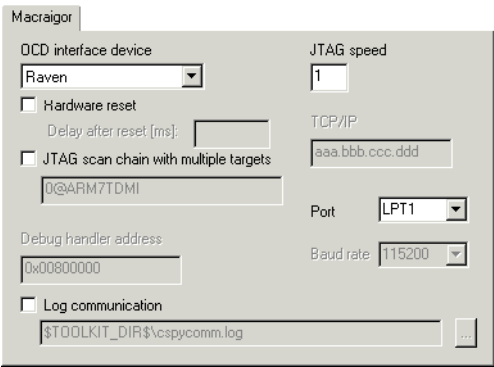


Figure 102: C-SPY Macraigor options

OCD interface device

Select the device corresponding to the hardware interface you are using. Supported Macraigor JTAG interfaces are Macraigor **Raven**, **Wiggler**, and **mpDemon**.

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 speed setting 1 usually works fine on Wiggler and Raven. The mpDemon interface may 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 may in some cases map its single parallel port to LPT2 or LPT3. If possible, configure the parallel port in EPP mode since 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 **Flash download** is selected, also once after flash download, see Figure 89, *Debugger startup when debugging code in flash*, page 211, and Figure 90, *Debugger startup when debugging code in RAM*, page 212.



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 119.

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# 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.

THE MACRAIGOR JTAG MENU

When you are using the Macraigor driver, the additional menu **JTAG** appears in C-SPY.



Figure 103: The Macraigor JTAG menu

The following commands are available on the **JTAG** menu:

Menu command	Description
Watchpoints	Opens a dialog box for setting watchpoints, see <i>JTAG watchpoints dialog box</i> , page 252.
Vector Catch	Opens a dialog box for setting a breakpoint directly on a vector in the interrupt vector table, see <i>Breakpoints on vectors</i> , page 250. Note that this command is not available for all ARM cores.

Table 35: Commands on the JTAG menu

Debugging using the RDI driver

To use C-SPY for the RDI interface, you should be familiar with the following details:

- The C-SPY options that specify the RDI interface, see *RDI*, page 233
- The RDI menu, see *RDI menu*, page 236
- The ETM trace mechanism, see *Using the trace system in hardware debugger systems*, page 237.

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 MultiICE JTAG interface.

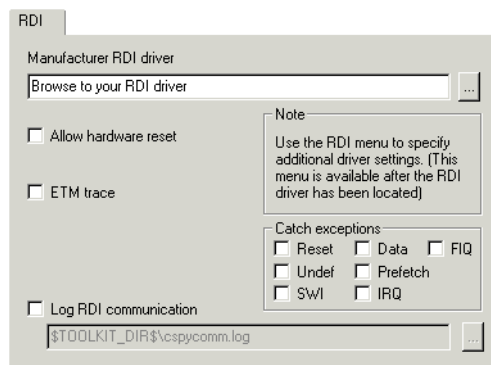


Figure 104: C-SPY RDI options

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 119.

Note: To use 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:

- **Trace Window**, see *Trace window*, page 240
- **Trace Setup**, see *Trace Setup dialog box*, page 238
- **Trace Save**, see *Trace Save dialog box*, page 240.

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:

Exception	Description
Reset	Reset
Undef	Undefined instruction
SWI	Software interrupt
Prefetch	Prefetch abort (instruction fetch memory fault)
Data	Data abort (data access memory fault)
IRQ	Normal interrupt
FIQ	Fast interrupt

Table 36: 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.

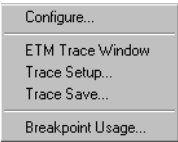


Figure 105: The RDI menu

The following commands are available on the **RDI** menu:

Menu command	Description
Configure	Opens a dialog box that originates from the RDI driver vendor. For information about details in this dialog box, refer to the driver documentation.
ETM Trace Window	Opens the Trace window to display the captured trace data; see <i>Trace window</i> , page 240.
Trace Setup	Opens the Trace Setup dialog box to configure the ETM trace; see <i>Trace Setup dialog box</i> , page 238.
Trace Save	Opens the Trace Save dialog box to save the captured trace data to a file; see <i>Trace Save dialog box</i> , page 240.
Breakpoint Usage	Opens the Breakpoint Usage dialog box to list all active breakpoints; see <i>Breakpoint Usage dialog box</i> , page 250.

Table 37: 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 not need any specific settings for your project.

Debugging using third-party drivers

It is possible to load other debugger drivers than those supplied with 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.

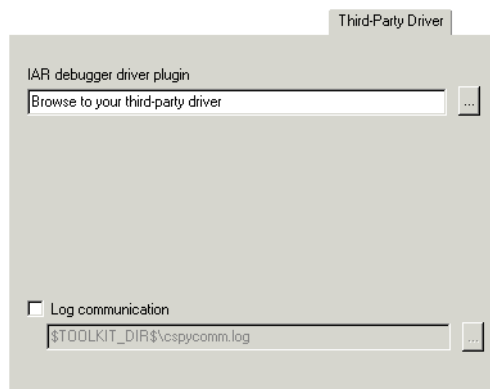


Figure 106: C-SPY Third-Party Driver options

IAR debugger driver plugin

Type or paste 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.

Using the trace system in hardware debugger systems

The Trace system is supported by the J-Trace driver, the J-Link driver for devices that have support for ETB (Embedded Trace Buffer), and RDI drivers where the connected hardware supports ETM.

Note: All trace features described here can be used by both of these systems, unless otherwise specifically noted.

This section describes the:

- *Trace Setup dialog box*, page 238
- *Trace Save dialog box*, page 240
- *Trace window*, page 240

- *Trace toolbar*, page 242.

For more detailed information about using the common features in the trace system, see *Using the trace system*, page 131.

TRACE SETUP DIALOG BOX

Use the **Trace Setup** dialog box—available from the driver-specific menu—to configure the trace system.

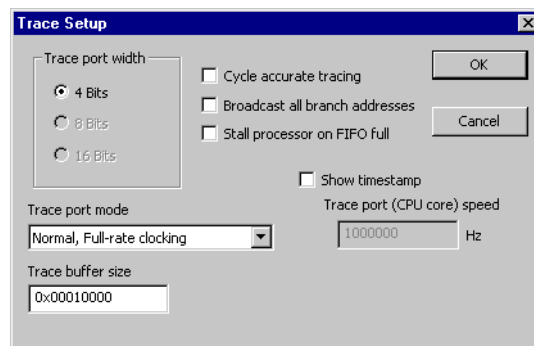


Figure 107: Trace Setup dialog box

Note: This dialog box looks slightly different for the RDI drivers.

Trace port width

The trace bus width can be set to 4, 8, or 16 bits. The value must correspond with what is supported by the hardware.

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, alternatives are available depending on support for them on the device.

Trace buffer size

Use the text box to specify the size of the trace buffer. By default, the size of trace frames is 64 Kbytes and the maximum size is 1 Mbyte.

One trace frame corresponds to 2 bytes of the J-Trace 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 will increase.

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 will increase.

Stall processor on FIFO full

The trace FIFO buffer might in some situations get 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 gets full.

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 field.

Note: The **Show timestamp** option is only available for the J-Trace driver.

TRACE SAVE DIALOG BOX

Use the **Trace Save** dialog box—available from the driver-specific menu—to save the captured trace data, as it is displayed in the Trace Window, to a file.

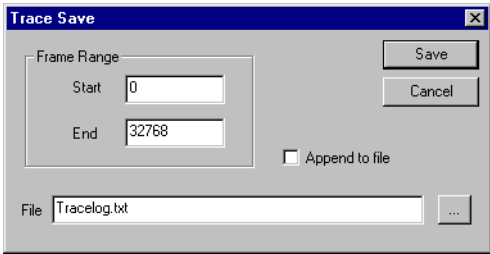


Figure 108: Trace Save dialog box

Frame Range

To save a range of frames to a file, specify a start frame and an end frame.

Append to file

Appends the trace data to an existing file.

File

Use this text box to locate a file for the trace data.

TRACE WINDOW

The Trace window—available from the driver-specific menu—displays the captured trace data.

Trace						
Index	Frame	Address	Opcode	Trace	Comment	
000000	000000				Execution stopped/sta...	
000001	000001	0x08000000	E59FF018	LDR	PC, [PC, #+2...	
000002	000011	0x080000A4	E10F0000	MRS	R0, CPSR	
000003	000015	0x080000A8	E3C0001F	BIC	R0, R0, #0x1F	
000004	000016	0x080000AC	E3800012	ORR	R0, R0, #0x12	
000005	000017	0x080000B0	E121F000	MSR	CPSR_c, R0	
000006	000018	0x080000B4	E59FD014	LDR	SP, [PC, #+2...	
000007	000019	0x080000B8	E3C0001F	BIC	R0, R0, #0x1F	
000008	000020	0x080000BC	E380001F	ORR	R0, R0, #0x1F	
000009	000021	0x080000C0	E121F000	MSR	CPSR_c, R0	
000010	000022	0x080000C4	E59FD008	LDR	SP, [PC, #+8...	
000011	000023	0x080000C8	E59F0008	LDR	R0, [PC, #+8...	

Figure 109: ETM Trace View window

Note: For RDI drivers, you must select the **ETM trace** option available on the **RDI** options page to display this window. For RDI drivers, this window looks slightly different.

The Trace window contains the following columns:

Trace window column	Description
Index	A serial number for each row in the trace buffer. Simplifies the navigation within the buffer.
Frame	The position in the number of trace data frames that have been displayed. When tracing in cycle accurate mode, the value corresponds to the number of elapsed cycles. This column is only available for the J-Trace driver.
Time	When tracing in non-cycle accurate mode, the value displays the time instead of cycles.This column is only available for the J-Trace driver and when the option Show timestamp is selected.
Address	The address of the executed instruction.
Opcode	The operation code of the executed instruction.
Trace	The recorded sequence of executed machine instructions. Optionally, the corresponding source code can also be displayed.
Comment	This column is only available for the J-Trace driver.

Table 38: Trace window columns

J-Link/J-Trace specials

If you are using the SWD interface between the J-Link debug probe and the target system, you can use the serial-wire output (SWO) communication channel for trace data. For information about how to set up for trace data over the SWO channel, see *SWO Setup dialog box*, page 226.

In that case, the Trace window will display trace information in these columns:

Trace window column	Description
Index	An index number for each row in the trace buffer. Simplifies the navigation within the buffer.
SWO Packet	The contents of the captured SWO packet.
Cycles	The approximate number of cycles from the start of the execution until the event.
Event	The type of the event for the captured SWO packet.
Value	The event value, if any.

Table 39: Trace window columns when using SWO

Trace window column	Description
Trace	The recorded sequence of executed machine instructions. Optionally, the corresponding source code can also be displayed. If the event is a PC value, the instruction is displayed in the Trace column.
Comment	Complementary information.

Table 39: Trace window columns when using SWO (Continued)

TRACE TOOLBAR

The following toolbar is available in the Trace window:

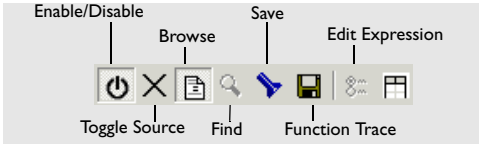


Figure 110: Trace toolbar

The following function buttons are available on the toolbar:

Toolbar button	Description
Enable/Disable	Enables and disables tracing.
Toggle Source	Toggles the Trace column between showing only disassembly or disassembly together with corresponding source code.
Browse	Toggles browse mode on and off for a selected item in the Trace column. For more information about browse mode, see <i>The Trace window and its browse mode</i> , page 132.
Find	Opens the Find In Trace dialog box where you can perform a search; see <i>Find In Trace window</i> , page 174.
Save	Opens a standard Save dialog box where you can save the recorded trace information to a text file, with tab-separated columns.
Function Trace	Opens the Function Trace window, see <i>Function Trace window</i> , page 172. This button is not available for the RDI driver.
Edit Expressions	Opens the Trace Setup dialog box, see <i>Trace Setup dialog box</i> , page 238. This button is not available for the RDI driver.

Table 40: Trace toolbar commands

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 243
- *Breakpoints options*, page 243
- *Code breakpoints dialog box*, page 245
- *Data breakpoints dialog box*, page 246
- *Breakpoint Usage dialog box*, page 250
- *Breakpoints on vectors*, page 250
- *Setting breakpoints in `__ramfunc` declared functions*, page 251.

For information about the different methods for setting breakpoints and the facilities for monitoring breakpoints, see *Using breakpoints*, page 135.

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 141.

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 331.

BREAKPOINTS OPTIONS

For the following hardware debugger systems it is possible to set some driver-specific breakpoint options:

- 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.

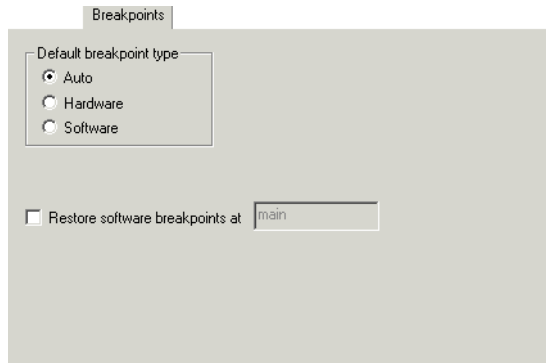


Figure 111: Breakpoints options

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. |
| 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. |

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.

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 `initialize` by `copy` for code in the linker configuration file or if you have any `__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.

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>Log** 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.

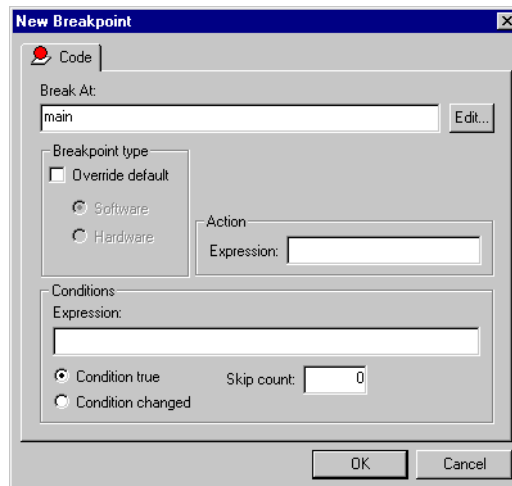


Figure 112: Code breakpoints page

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 287.

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.

Conditions	Description
Expression	A valid expression conforming to the C-SPY expression syntax.
Condition true	The breakpoint is triggered if the value of the expression is true.
Condition changed	The breakpoint is triggered if the value of the expression has changed since it was last evaluated.
Skip count	The number of times that the breakpoint must be fulfilled before a break occurs (integer).

Table 41: 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>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** breakpoints dialog box appears.

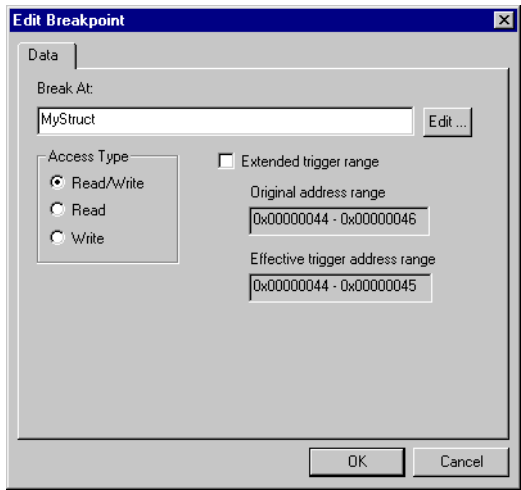


Figure 113: Data breakpoints dialog box

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 287.

Access Type

Use the options in the **Access Type** area to specify the type of memory access that triggers data breakpoints.

Memory Access type	Description
Read/Write	Read or write from location.

Table 42: Memory Access types

Memory Access type	Description
Read	Read from location.
Write	Write to location.

Table 42: Memory Access types (Continued)

Note: Data breakpoints never stop execution within a single instruction. They are recorded and reported after the instruction is executed.

Extended trigger 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.

DATA LOG BREAKPOINTS DIALOG BOX

Data Log breakpoints are triggered when data is accessed at the specified location. If you have set a log on a specific address or a range, a log message will be printed in the Trace window for each access to that location. However, this requires that you have set up for trace data in the SWO Setup dialog box, see *SWO Setup dialog box*, page 226.

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.

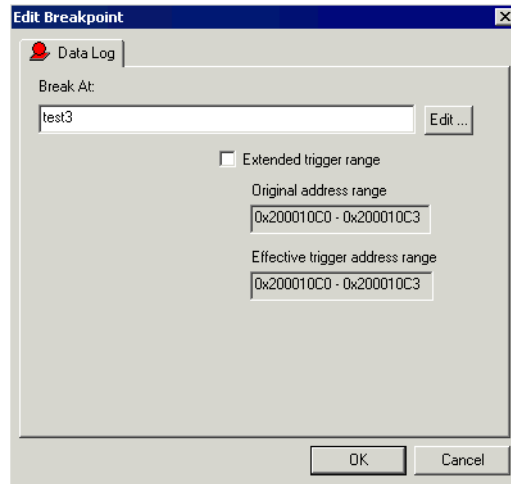


Figure 114: Data Log breakpoints dialog box

Note: Setting Data Log breakpoints is possible only for Cortex-M with SWO using the J-Link debug probe.

Extended trigger 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.

BREAKPOINT USAGE DIALOG BOX

The **Breakpoint Usage** dialog box—available from the driver-specific menu—lists all active breakpoints.

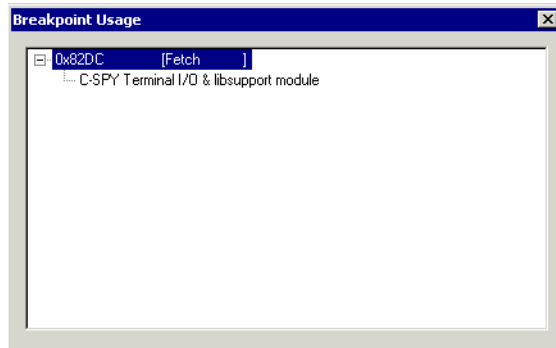


Figure 115: Breakpoint Usage dialog box

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 139.

BREAKPOINTS ON VECTORS

For ARM9 devices, it is possible to set a breakpoint directly on a vector in the interrupt vector table, without using a hardware breakpoint. First, you must 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. Start C-SPY.

To set the breakpoint directly on a vector in the interrupt vector table, choose the **Vector Catch** command from the **J-Link** menu.

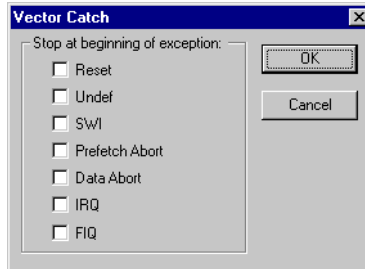


Figure 116: The Vector Catch dialog box

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 221 and *RDI*, page 233.

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.

In addition, breakpoints in `__ramfunc` declared functions added from the editor have to be disabled prior to invoking C-SPY and disabled 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 **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 127.

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.

Access Type

Use these options to define the access type of the data to watch for:

Type	Description
Any	Matches any access type
OP Fetch	Operation code (instruction) fetch
Read	Data read
Write	Data write
R/W	Data read or write

Table 43: 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 127.

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. **Any** means that the state is ignored.

Mode

Use these options to define the CPU mode that must be active for a match:

Mode	Description
User	The CPU must run in USER mode
Non User	The CPU must run in one of the SYSTEM SVC, UND, ABORT, IRQ or FIQ modes
Any	The CPU mode is ignored

Table 44: CPU modes

Break Condition

Use these options to specify how to use the defined watchpoints:

Break condition	Description
Normal	The two watchpoints are used individually (OR).
Range	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.
Chain	A trigger of watchpoint 1 will arm watchpoint 0. A program break will then occur when watchpoint 0 is triggered.

Table 45: Break conditions

For example, to cause a trigger for accesses in the range 0x20–0xFF:

- 1 Set **Break Condition** to **Range**.
- 2 Set watchpoint 0’s address value to 0 and mask to 0xFF.
- 3 Set watchpoint 1’s address value to 0 and mask to 0x1F.

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 C-SPY 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, and click the **Edit** button.
- 4 The **Flash Loader Overview** dialog box lists all currently available flash loaders; see *Flash Loader Overview dialog box*, page 257. 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 different flash loader options, see *Flash Loader Configuration dialog box*, page 258.

Setting up the target system using a C-SPY macro file

You can use a C-SPY macro to set up the target system before loading the flash loader to RAM. One example when this is useful is for targets where the RAM is not functional after reset; the macro is used for setting up the necessary registers for proper RAM operation.

The following criteria must be met for a macro function to be executed before downloading the flash loader:

- The macro file must be loaded in the same directory as the flash loader
- The macro file must have the filename extension `mac`
- The name of the macro file must be the same as the flash loader
- The setup macro function `execUserFlashInit` must be defined in the macro file. This macro function is called from the debugger before the flash loader is loaded into RAM. Note that debugging while when the flash loader is running as an application, the setup macro `execUserPreload` must be used instead of `execUserFlashInit`.

THE FLASH LOADING MECHANISM

When the **Use flash loader(s)** option is selected and one or several flash loaders have been configured, the following steps will be 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 opens the file holding the application code.
- 4** The flash loader reads the application code and programs it into flash memory.
- 5** The flash loader terminates.
- 6** C-SPY switches context to the user application.

The steps 1 to 5 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.

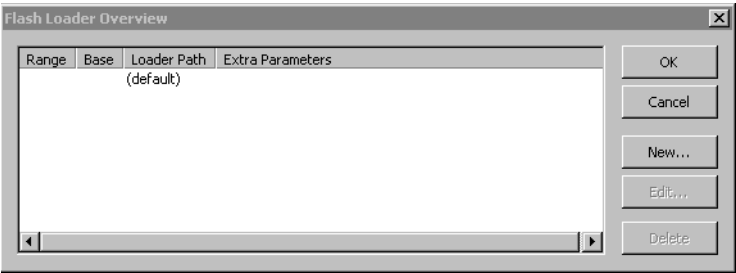


Figure 118: Flash Loader Overview dialog box

The following function buttons are available:

Button	Description
OK	The selected flash loader(s) will be used for downloading your application to memory.
Cancel	Standard cancel.
New	Opens the Flash Loader Configuration dialog box where you can specify what flash loader to use; see <i>Flash Loader Configuration dialog box</i> , page 258.
Edit	Opens the Flash Loader Configuration dialog box where you can modify the settings for the selected flash loader; see <i>Flash Loader Configuration dialog box</i> , page 258.
Delete	Deletes the selected flash loader configuration.

Table 46: 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.

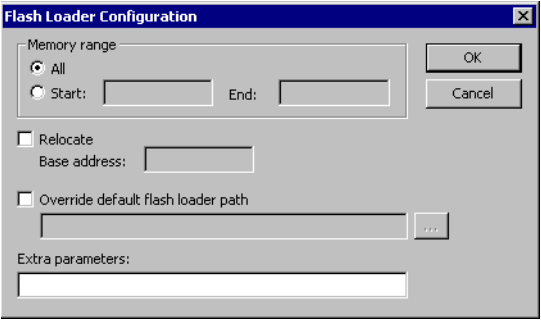


Figure 119: Flash Loader Configuration dialog box

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 add an offset to the location of the application as specified in the linker configuration file. It can sometimes be necessary to override the address and start flashing at a different location in the address space. This can, for example, be necessary for devices that remap the location of the flash memory.

Use the **Offset** text box to specify a numeric value for the new base address. You can use the following numeric formats:

123456	Decimal numbers
0x123456	Hexadecimal numbers
0123456	Octal numbers

Override default flash loader path

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 flash loader path** and specify the path to the flash loader you want to use. A browse button is available for your convenience.

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 any additional flash loaders and flash loader options to those described here, see the release notes delivered with your IAR Embedded Workbench installation.

Atmel AT91EBxx flash loader, Atmel AT91SAM7AI-Ek, Atmel AT91SAM7A2-Ek

<code>--user</code>	Allows programming the flash while the board jumper is in the <i>USER</i> position (address line A20 inverted). Without this argument, the board jumper must be in the <i>STD</i> position for proper flash programming operation.
---------------------	--

Freescal MAC71x1 flash loader

<code>--clock value</code>	Passes the clock frequency to the flash loader; <i>value</i> is the clock speed in kHz. The default clock frequency value is 8000 kHz.
----------------------------	--

Nohau LPC2800 flash loader

<code>--clock value</code>	Passes the clock frequency to the flash loader; <i>value</i> is the CKL speed in Hz. The default clock frequency value is 12 MHz.
----------------------------	---

Nohau LPC2888 flash loader

<code>--clock value</code>	Passes the clock frequency to the flash loader; <i>value</i> is the CKL speed in Hz. The default clock frequency value is 12 MHz.
----------------------------	---

NXP LPC flash loader

`--clock value` Passes the clock frequency to the flash loader; *value* is the CCLK speed in kHz. The default clock frequency value is 14,746 kHz.

Olimex LPCH 288x flash loader

`--clock value` Passes the clock frequency to the flash loader; *value* is the CKL speed in Hz. The default clock frequency value is 12000000 Hz.

Phytec LPC3180 flash loader

`--nand_verify_dis` Speeds up programming speed by skipping Reed–Solomon and compare error checks.

Texas Instruments TMS470 flash loader

`--clock value` Passes the clock frequency to the flash loader; *value* is the CCLK speed in kHz. The default clock frequency covers 12,000 kHz < x < 14,000 kHz.

The flash keys are located in the memory area 0x1FF0–0x1FFF. There is a set of arguments for handling this memory area and the keys located there:

`--allownewkeys` To avoid any keys being accidentally overwritten, the flash loader will by default issue an error whenever there is a write access to the memory range 0x1FF0–0x1FFF. However, in some situations you might want to write to this area; use this option to make that possible.

`--flashkey0 value` Use these options to unlock a flash memory that has previously been protected with a flash key; *value* must be the original flash key value.

`--flashkey1 value`

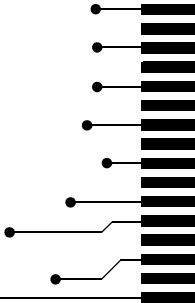
`--flashkey2 value`

`--flashkey3 value`

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. Information about how to best use the IDE for your purposes can be found in parts 3 to 7 in this guide. This chapter contains the following sections:

- *Windows*, page 263
- *Menus*, page 291.

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. Reference information about these windows can be found in the chapter *C-SPY® reference* in this guide.

IAR EMBEDDED WORKBENCH IDE WINDOW

The figure shows the main window of the IDE and its different components. The window might look different depending on which plugin modules you are using.

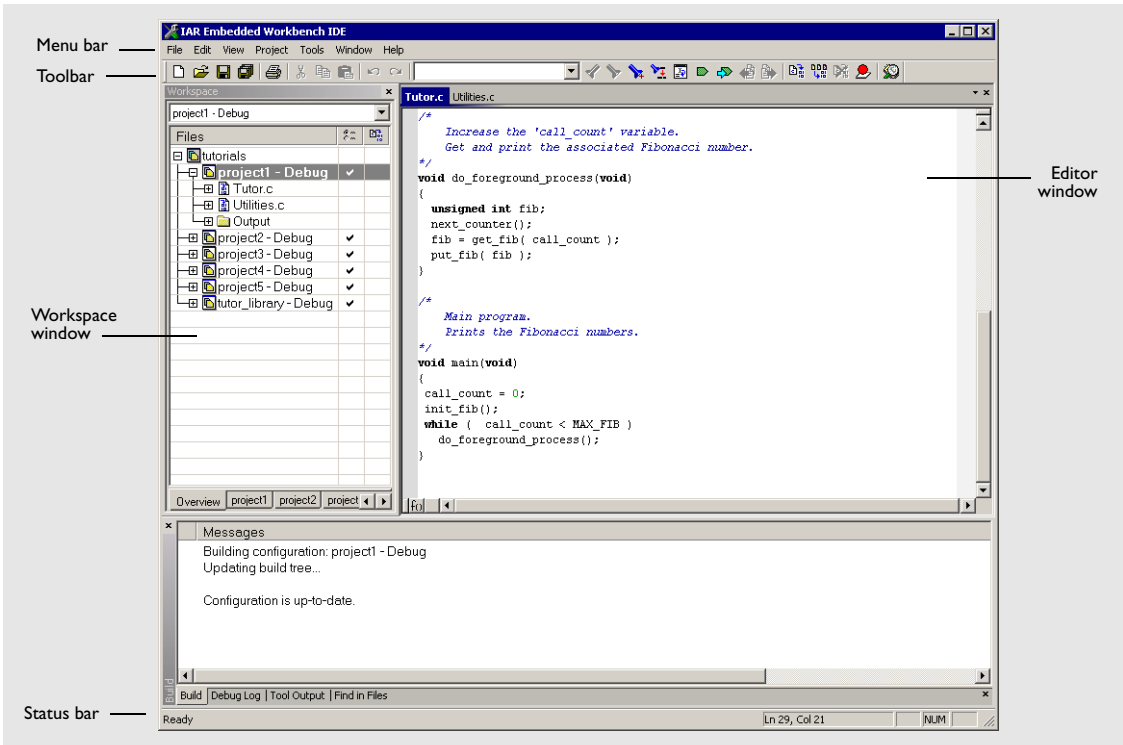


Figure 120: IAR Embedded Workbench IDE window

Each window item is explained in greater detail in the following sections.

Menu bar

Gives access to the IDE menus.

Menu	Description
File	The File menu provides commands for opening source and project files, saving and printing, and exiting from the IDE.
Edit	The Edit menu provides commands for editing and searching in editor windows and for enabling and disabling breakpoints in C-SPY.

Table 47: IDE menu bar

Menu	Description
View	Use the commands on the View menu to open windows and decide which toolbars to display.
Project	The Project menu provides commands for adding files to a project, creating groups, and running the IAR Systems tools on the current project.
Tools	The Tools menu is a user-configurable menu to which you can add tools for use with the IDE.
Window	With the commands on the Window menu you can manipulate the IDE windows and change their arrangement on the screen.
Help	The commands on the Help menu provide help about the IDE.

Table 47: IDE menu bar (Continued)

For reference information for each menu, see *Menus*, page 291.

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.

You can display a description of any button by pointing to it with the mouse button. When a command is not available, the corresponding toolbar button will be dimmed, and you will not be able to click it.

This figure shows the menu commands corresponding to each of the toolbar buttons:

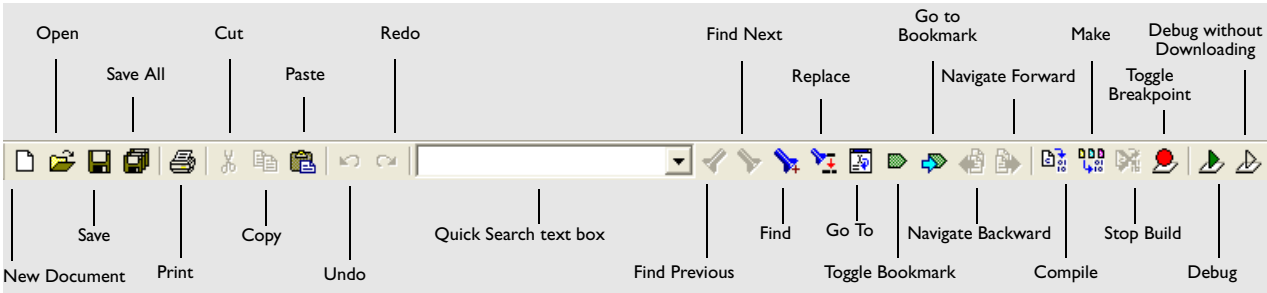
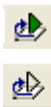


Figure 121: IDE toolbar



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.

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.

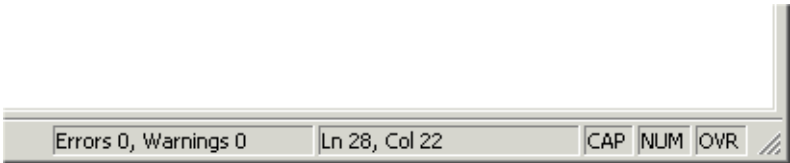


Figure 122: IAR Embedded Workbench IDE window status bar

WORKSPACE WINDOW

The Workspace window, available from the **View** menu, is where you can access your projects and files during the application development.

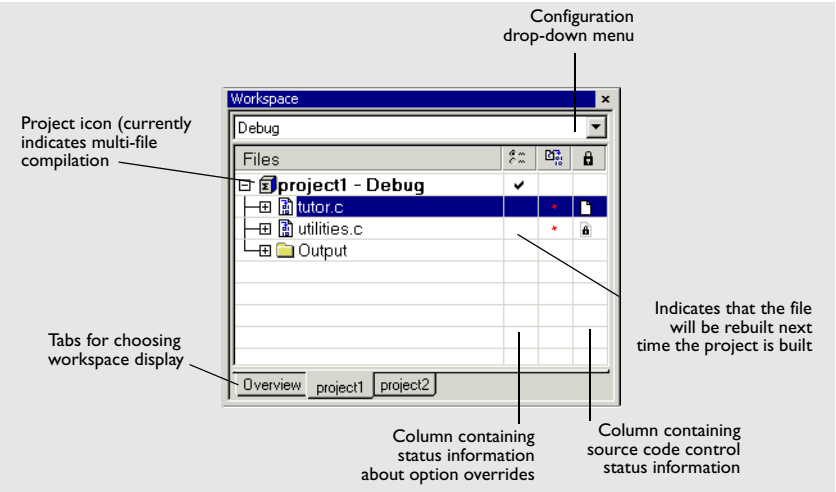


Figure 123: Workspace window

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 in 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.



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
Red check mark	Include the following sentence for multi-file compile only, in that case, remove the 1st sentence There are local settings/overrides for this file/group, but they are either identical with the inherited settings or they will be ignored because of use of multi-file compilation, which means the overrides are superfluous.



The column that contains build status information can have one of three icons for each file in the project:

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 different source code control icons, see *Source code control states*, page 270.

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.

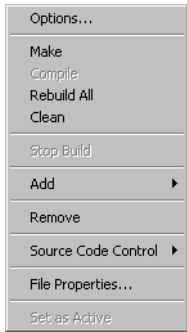


Figure 124: Workspace window context menu

The following commands are available on the context menu:

Menu command	Description
Options	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.
Make	Brings the current target up to date by compiling, assembling, and linking only the files that have changed since the last build.
Compile	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.
Rebuild All	Recompiles and relinks all files in the selected build configuration.
Clean	Deletes intermediate files.
Stop Build	Stops the current build operation.
Add>Add Files	Opens a dialog box where you can add files to the project.
Add>Add "filename"	Adds the indicated file to the project. This command is only available if there is an open file in the editor.
Add>Add Group	Opens a dialog box where you can add new groups to the project.
Remove	Removes selected items from the Workspace window.
Source Code Control	Opens a submenu with commands for source code control, see <i>Source Code Control menu</i> , page 269.

Table 48: Workspace window context menu commands

Menu command	Description
File Properties	Opens a standard File Properties dialog box for the selected file.
Set as Active	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.

Table 48: Workspace window context menu commands (Continued)

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.

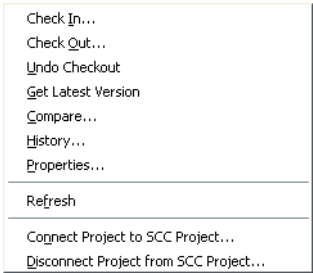


Figure 125: Source Code Control menu

For more information about interacting with an external source code control system, see *Source code control*, page 88.

The following commands are available on the submenu:

Menu command	Description
Check In	Opens the Check In Files dialog box where you can check in the selected files; see <i>Check In Files dialog box</i> , page 272. 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.
Check Out	Checks out the selected file or files. Depending on the SCC system you are using, a dialog box may appear; see <i>Check Out Files dialog box</i> , page 273. 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.

Table 49: Description of source code control commands

Menu command	Description
Undo Check out	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.
Get Latest Version	Replaces the selected files with the latest archived version.
Compare	Displays—in a SCC-specific window—the differences between the local version and the most recent archived version.
History	Displays SCC-specific information about the revision history of the selected file.
Properties	Displays information available in the SCC system for the selected file.
Refresh	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.
Connect Project to SCC Project	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.
Disconnect Project From SCC Project	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.

Table 49: Description of source code control commands (Continued)

Source code control states

Each source code-controlled file can be in one of several states.





SCC state	Description
	Checked out to you. The file is editable.
	Checked out to you. The file is editable and you have modified the file.
 (grey padlock)	Checked in. In many SCC systems this means that the file is write-protected.
 (grey padlock)	Checked in. There is a new version available in the archive.

Table 50: Description of source code control states



SCC state	Description
 (red padlock)	Checked out exclusively to another user. In many SCC systems this means that you cannot check out the file.
 (red padlock)	Checked out exclusively to another user. There is a new version available in the archive. In many SCC systems this means that you cannot check out the file.

Table 50: Description of source code control states (Continued)

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 there are several SCC systems from different vendors available. Use this dialog box to choose the SCC system you want to use.

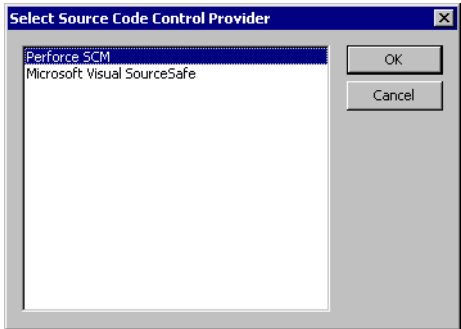


Figure 126: Select Source Code Control Provider dialog box

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.

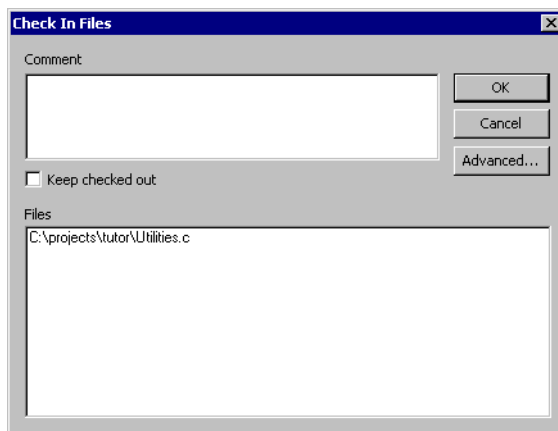


Figure 127: Check In Files dialog box

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 File** 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.

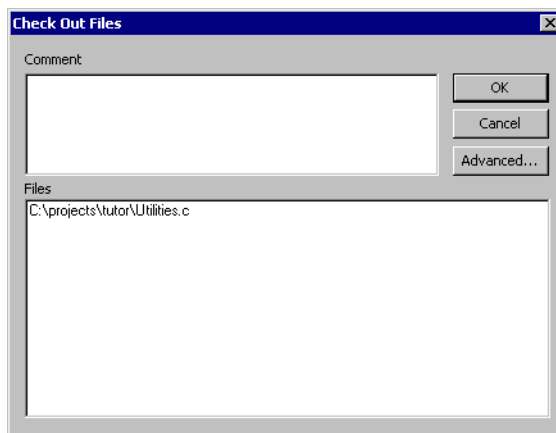


Figure 128: Check Out File dialog box

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 depends on other currently open windows.

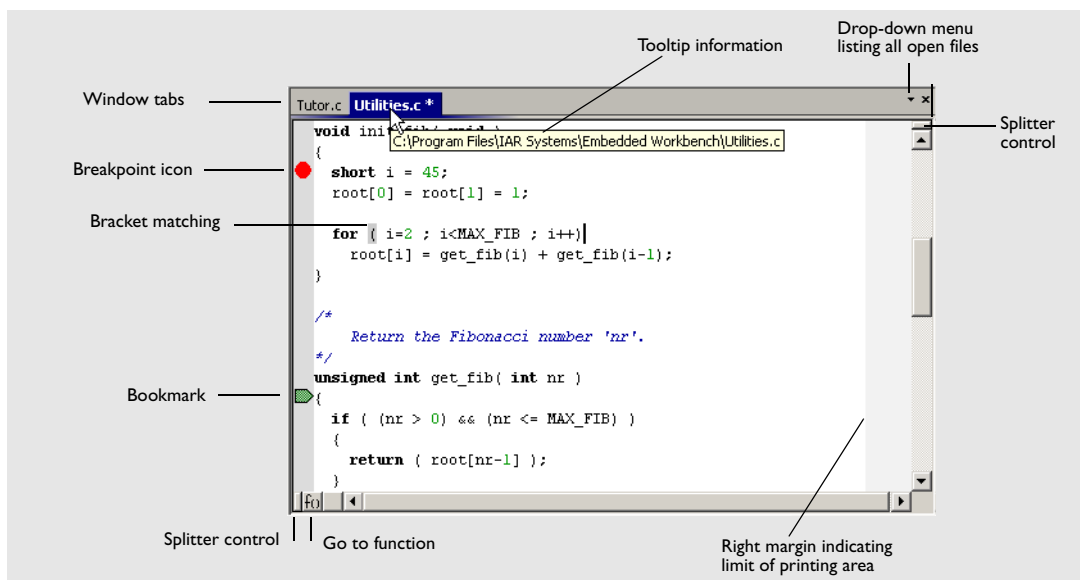


Figure 129: Editor window

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 99.

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, as well as commands for moving files between the different editor windows.

Go to function



With the **Go to function** button in the bottom left-hand corner of the editor window you can display all functions in the C or C++ editor window. You can then choose to go directly to one of them.

Editor window tab context menu

The context menu that appears if you right-click on a tab in the editor window provides access to commands for saving and closing the file.

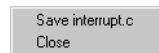


Figure 130: Editor window tab context menu

Editor window context menu

The context menu available in the editor window provides convenient access to several commands.

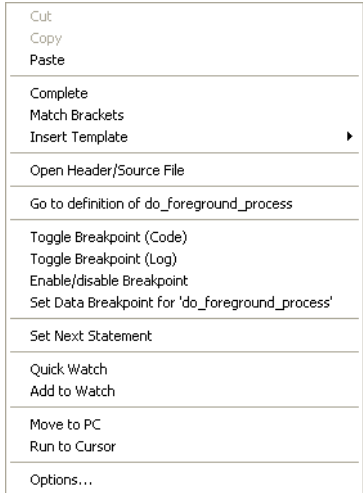


Figure 131: Editor window context menu

Note: The contents of this menu are dynamic, which means it may contain other commands than in this figure. All available commands are described in Table 51, *Description of commands on the editor window context menu*.

The following commands are available on the editor window context menu:

Menu command	Description
Cut, Copy, Paste	Standard window commands.
Complete	Attempts to complete the word you have begun to type, basing the guess on the contents of the rest of the editor document.
Match Brackets	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.

Table 51: Description of commands on the editor window context menu

Menu command	Description
Insert Template	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 <i>Template dialog box</i> , page 301. For information about using code templates, see <i>Using and adding code templates</i> , page 103.
Open "header.h"	Opens the header file "header.h" in an editor window. This menu command is only available if the insertion point is located on an <code>#include</code> line when you open the context menu.
Open Header/Source File	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 <code>#include</code> line when you open the context menu. This command is also available from the File>Open menu.
Go to definition	Shows the declaration of the symbol where the insertion point is placed.
Check In	Commands for source code control; for more details, see <i>Source Code Control menu</i> , page 269. 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.
Check Out	
Undo Checkout	
Toggle Breakpoint (Code)	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 <i>Code breakpoints dialog box</i> , page 283.
Toggle Breakpoint (Log)	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 <i>Log breakpoints dialog box</i> , page 285.
Enable/disable Breakpoint	Toggles a breakpoint between being disabled, but not actually removed—making it available for future use—and being enabled again.
Set Data Breakpoint for variable	Toggles a data breakpoint on variables with static storage duration. Requires support in the C-SPY driver you are using.
Set Next Statement	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.
Quick Watch	Opens the Quick Watch window, see <i>Quick Watch window</i> , page 360. This menu command is only available when you are using the debugger.

Table 51: Description of commands on the editor window context menu (Continued)

Menu command	Description
Add to Watch	Adds the selected symbol to the Watch window. This menu command is only available when you are using the debugger.
Move to PC	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.
Run to Cursor	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.
Options	Displays the IDE Options dialog box, see <i>Tools menu</i> , page 313.

Table 51: 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 the following keys and key combinations for moving the insertion point:

To move the insertion point	Press
One character left	Arrow left
One character right	Arrow right
One word left	Ctrl+Arrow left
One word right	Ctrl+Arrow right
One line up	Arrow up
One line down	Arrow down
To the start of the line	Home
To the end of the line	End
To the first line in the file	Ctrl+Home
To the last line in the file	Ctrl+End

Table 52: Editor keyboard commands for insertion point navigation

Use the following keys and key combinations for scrolling text:

To scroll	Press
Up one line	Ctrl+Arrow up
Down one line	Ctrl+Arrow down
Up one page	Page Up
Down one page	Page Down

Table 53: Editor keyboard commands for scrolling

Use the following key combinations for selecting text:

To select	Press
The character to the left	Shift+Arrow left
The character to the right	Shift+Arrow right
One word to the left	Shift+Ctrl+Arrow left
One word to the right	Shift+Ctrl+Arrow right
To the same position on the previous line	Shift+Arrow up
To the same position on the next line	Shift+Arrow down
To the start of the line	Shift+Home
To the end of the line	Shift+End
One screen up	Shift+Page Up
One screen down	Shift+Page Down
To the beginning of the file	Shift+Ctrl+Home
To the end of the file	Shift+Ctrl+End

Table 54: 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.

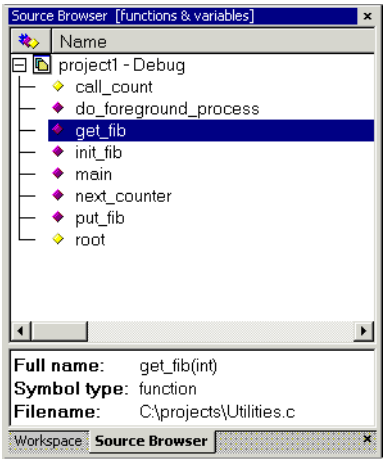


Figure 132: Source Browser window

The window consists of two separate panes. The top pane displays the names of global symbols and functions defined in the project.

Each row is prefixed with an icon, which corresponds to the *Symbol type* classification, see Table 55, *Information in Source Browser window*. By clicking in the window header, you can sort the symbols either by name or by symbol type.

In the top pane you can also access a context menu; see *Source Browser window context menu*, page 281.

For a symbol selected in the top pane, the bottom pane displays the following information:

Type of information	Description
Full name	Displays the unique name of each element, for instance <i>classname::membername</i> .
Symbol type	Displays the symbol type for each element: enumeration, enumeration constant, class, typedef, union, macro, field or variable, function, template function, template class, and configuration.
Filename	Specifies the path to the file in which the element is defined.

Table 55: Information in Source Browser window

For further details about how to use the Source Browser window, see *Displaying browse information*, page 87.

Source Browser window context menu

Right-clicking in the Source Browser window displays a context menu with convenient access to several commands.

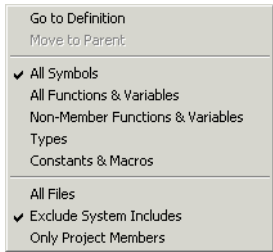


Figure 133: Source Browser window context menu

The following commands are available on the context menu:

Menu command	Description
Go to Definition	The editor window will display the definition of the selected item.
Move to Parent	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.
All Symbols	Type filter; all global symbols and functions defined in the project will be displayed.
All Functions & Variables	Type filter; all functions and variables defined in the project will be displayed.
Non-Member Functions & Variables	Type filter; all functions and variables that are not members of a class will be displayed
Types	Type filter; all types such as structures and classes defined in the project will be displayed.
Constants & Macros	Type filter; all constants and macros defined in the project will be displayed.
All Files	File filter; symbols from all files that you have explicitly added to your project and all files included by them will be displayed.

Table 56: Source Browser window context menu commands

Menu command	Description
Exclude System Includes	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.
Only Project Members	File filter; symbols from all files that you have explicitly added to your project will be displayed, but no include files.

Table 56: Source Browser window context menu commands (Continued)

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.

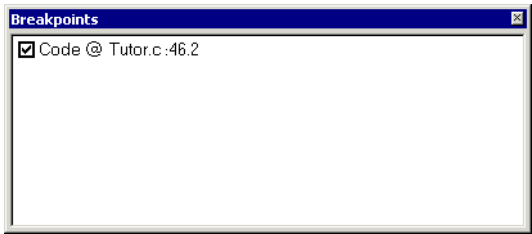


Figure 134: Breakpoints window

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.

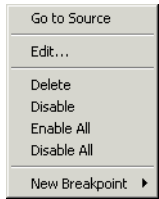


Figure 135: Breakpoints window context menu

The following commands are available on the context menu:

Menu command	Description
Go to Source	Moves the insertion point to the location of the breakpoint, if the breakpoint has a source location. Double-click a breakpoint in the Breakpoints window to perform the same command.
Edit	Opens the Edit Breakpoint dialog box for the selected breakpoint.
Delete	Deletes the selected breakpoint. Press the Delete key to perform the same command.
Enable	Enables the selected breakpoint. The check box at the beginning of the line will be selected. You can also perform the command by manually selecting the check box. This command is only available if the selected breakpoint is disabled.
Disable	Disables the selected breakpoint. The check box at the beginning of the line will be cleared. You can also perform this command by manually deselecting the check box. This command is only available if the selected breakpoint is enabled.
Enable All	Enables all defined breakpoints.
Disable All	Disables all defined breakpoints.
New Breakpoint	Displays a submenu where you can open the New Breakpoint dialog box for the available breakpoint types. All breakpoints you define using the New Breakpoint dialog box are preserved between debug sessions. In addition to code and log breakpoints—see <i>Code breakpoints dialog box</i> , page 283 and —other types of breakpoints might be available depending on the C-SPY driver you are using. For information about driver-specific breakpoint types, see the driver-specific debugger documentation.

Table 57: 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.

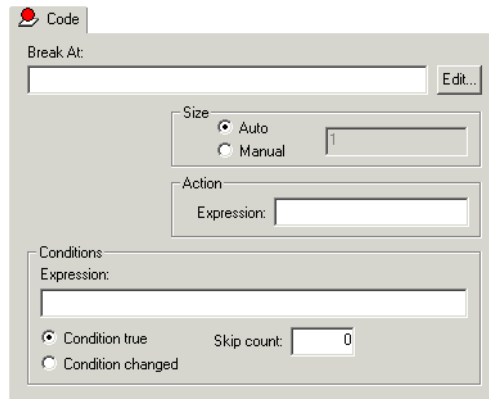


Figure 136: Code breakpoints page

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 287.

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 the size can be specified:

- **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.

Conditions	Description
Expression	A valid expression conforming to the C-SPY expression syntax.
Condition true	The breakpoint is triggered if the value of the expression is true.
Condition changed	The breakpoint is triggered if the value of the expression has changed since it was last evaluated.
Skip count	The number of times that the breakpoint must be fulfilled before a break occurs (integer).

Table 58: 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.

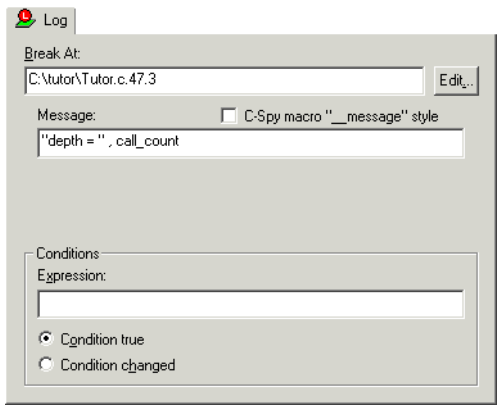


Figure 137: Log breakpoints page

The quickest—and typical—way to set a log breakpoint is by choosing **Toggle Breakpoint (Log)** from the context menu available by right-clicking in either the editor or the Disassembly window. For more information about how to set breakpoints, see *Defining breakpoints*, page 135.

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 287.

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 462.

Conditions

You can specify simple and complex conditions.

Conditions	Description
Expression	A valid expression conforming to the C-SPY expression syntax.
Condition true	The breakpoint is triggered if the value of the expression is true.
Condition changed	The breakpoint is triggered if the value of the expression has changed since it was last evaluated.

Table 59: 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.

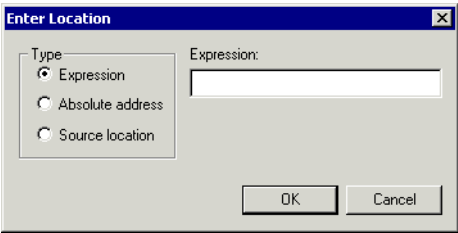


Figure 138: Enter Location dialog box

You can choose between these locations and their possible settings:

Location type	Description/Examples
Expression	Any expression that evaluates to a valid address, such as a function or variable name. Code breakpoints are set on functions and data breakpoints are set on variable names. For example, <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> .
Absolute Address	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.
Source Location	A location in the C source code using the syntax: <code>{file path}.row.column</code> . File specifies the filename and full path. Row specifies the row in which you want the breakpoint. Column specifies the column in which you want the breakpoint. 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> .

Table 60: Location types

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 263.

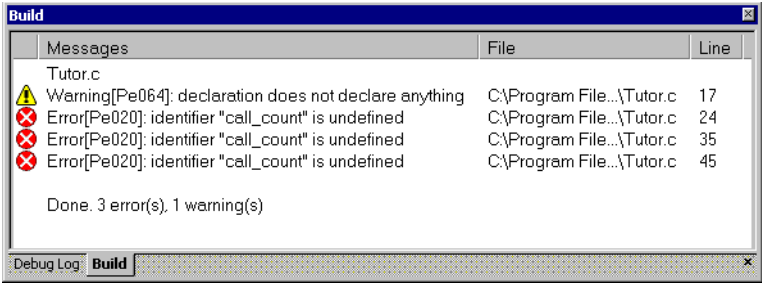


Figure 139: Build window (message window)

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.

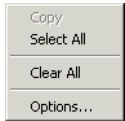


Figure 140: Build window context menu

The **Options** command opens the **Messages** page of the **IDE options** dialog box. On this page you can set options related to messages; see *Messages options*, page 324.

FIND IN FILES WINDOW

The 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 263.

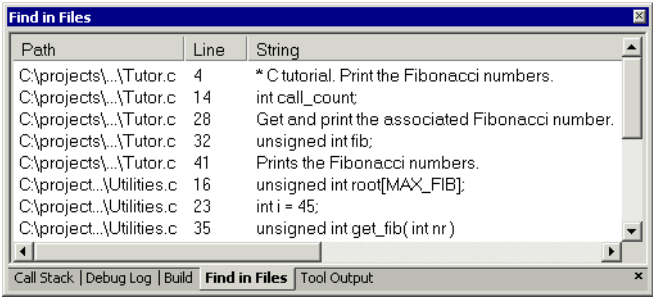


Figure 141: Find in Files window (message window)

Double-clicking an entry in the page 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.

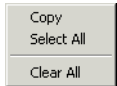


Figure 142: Find in Files window context menu

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 334. When opened, this window is by default grouped together with the other message windows, see *Windows*, page 263.

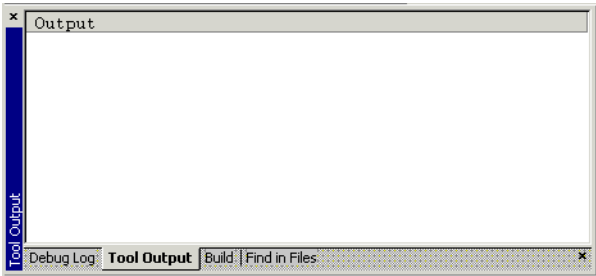


Figure 143: Tool Output window (message window)

Right-clicking in the Tool Output window displays a context menu which allows you to copy, select, and clear the contents of the window.

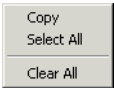


Figure 144: Tool Output window context menu

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 263.

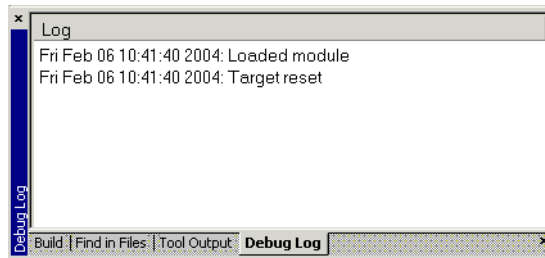


Figure 145: Debug Log window (message window)

Right-clicking in the Tool Output window displays a context menu which allows you to copy, select, and clear the contents of the window.

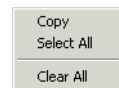


Figure 146: Debug Log window context menu

Menus

The following 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. Reference information about these menus can be found in the chapter *C-SPY® reference*, page 343.

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 allow you to open one by selecting its name from the menu.

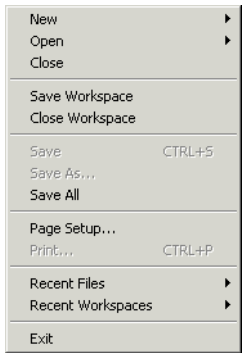


Figure 147: File menu

The following commands are available on the **File** menu:





	Menu command	Shortcut	Description
	New	CTRL+N	Displays a submenu with commands for creating a new workspace, or a new text file.
	Open>File	CTRL+O	Displays a submenu from which you can select a text file or an HTML document to open.
	Open>Workspace		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.
	Open>Header/Source File	CTRL+SHIFT+H	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.
	Close		Closes the active window. You will be given the opportunity to save any files that have been modified before closing.

Table 61: File menu commands



	Menu command	Shortcut	Description
	Open Workspace		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.
	Save Workspace		Saves the current workspace file.
	Close Workspace		Closes the current workspace file.
	Save	CTRL+S	Saves the current text file or workspace file.
	Save As		Displays a dialog box where you can save the current file with a new name.
	Save All		Saves all open text documents and workspace files.
	Page Setup		Displays a dialog box where you can set printer options.
	Print	CTRL+P	Displays a dialog box where you can print a text document.
	Recent Files		Displays a submenu where you can quickly open the most recently opened text documents.
	Recent Workspaces		Displays a submenu where you can quickly open the most recently opened workspace files.
	Exit		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.

Table 61: File menu commands (Continued)

EDIT MENU

The **Edit** menu provides several commands for editing and searching.

Undo	Ctrl+Z
Redo	Ctrl+Y
Cut	Ctrl+X
Copy	Ctrl+C
Paste	Ctrl+V
Paste Special...	
Select All	Ctrl+A
Find and Replace	▶
Navigate	▶
Code Templates	▶
Next Error/Tag	F4
Previous Error/Tag	Shift+F4
Complete	Ctrl+Space
Match Brackets	Ctrl+B
Auto Indent	Ctrl+T
Block Comment	Ctrl+K
Block Uncomment	Ctrl+Shift+K
Toggle Breakpoint	F9
Enable/Disable Breakpoint	Ctrl+F9

Figure 148: Edit menu






	Menu command	Shortcut	Description
	Undo	CTRL+Z	Undoes the last edit made to the current editor window.
	Redo	CTRL+Y	Redoes the last Undo in the current editor window. You can undo and redo an unlimited number of edits independently in each editor window.
	Cut	CTRL+X	The standard Windows command for cutting text in editor windows and text boxes.
	Copy	CTRL+C	The standard Windows command for copying text in editor windows and text boxes.
	Paste	CTRL+V	The standard Windows command for pasting text in editor windows and text boxes.
	Paste Special		Provides you with a choice of the most recent contents of the clipboard to choose from when pasting in editor documents.
	Select All	CTRL+A	Selects all text in the active editor window.

Table 62: Edit menu commands






	Menu command	Shortcut	Description
	Find and Replace>Find	CTRL+F	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.
	Find and Replace>Find Next	F3	Finds the next occurrence of the specified string.
	Find and Replace>Find Previous	SHIFT+F3	Finds the previous occurrence of the specified string.
	Find and Replace>Find Next (Selected)	CTRL+F3	Searches for the next occurrence of the currently selected text or the word currently surrounding the insertion point.
	Find and Replace>Find Previous (Selected)	CTRL+SHIFT+F3	Searches for the previous occurrence of the currently selected text or the word currently surrounding the insertion point.
	Find and Replace>Replace	CTRL+H	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.
	Find and Replace>Find in Files		Displays a dialog box where you can search for a specified string in multiple text files; see <i>Find in Files dialog box</i> , page 299.
	Find and Replace>Incremental Search	CTRL+I	Displays a dialog box where you can gradually fine-tune or expand the search by continuously changing the search string.
	Navigate>Go To	CTRL+G	Displays a dialog box where you can move the insertion point to a specified line and column in the current editor window.
	Navigate>Toggle Bookmark	CTRL+F2	Toggles a bookmark at the line where the insertion point is located in the active editor window.
	Navigate>Go to Bookmark	F2	Moves the insertion point to the next bookmark that has been defined with the Toggle Bookmark command.

Table 62: Edit menu commands (Continued)

Menu command	Shortcut	Description
Navigate> Navigate Backward	ALT+Left arrow	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.
Navigate> Navigate Forward	ALT+Right arrow	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.
Navigate> Go to Definition	F12	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 <i>Project options</i> , page 326.
Code Templates> Insert Template	CTRL+ SHIFT+ SPACE	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 <i>Template dialog box</i> , page 301. For information about using code templates, see <i>Using and adding code templates</i> , page 103.
Code Templates> Edit Templates		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 <i>Using and adding code templates</i> , page 103.
Next Error/Tag	F4	If there is a list of error messages or the results from a Find in Files search in the Messages window, this command will display the next item from that list in the editor window.
Previous Error/Tag	SHIFT+F4	If there is a list of error messages or the results from a Find in Files search in the Messages window, this command will display the previous item from that list in the editor window.
Complete	CTRL+ SPACE	Attempts to complete the word you have begun to type, basing the guess on the contents of the rest of the editor document.
Auto Indent	CTRL+T	Indents one or several lines you have selected in a C/C++ source file. To configure the indentation, see <i>Configure Auto Indent dialog box</i> , page 319.

Table 62: Edit menu commands (Continued)

Menu command	Shortcut	Description
Match Brackets		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.
Block Comment	CTRL+K	Places the C++ comment character sequence <code>//</code> at the beginning of the selected lines.
Block Uncomment	CTRL+K	Removes the C++ comment character sequence <code>//</code> from the beginning of the selected lines.
Toggle Breakpoint	F9	Toggles a breakpoint at the statement or instruction that contains or is located near the cursor in the source window. This command is also available as an icon button in the debug bar.
Enable/Disable Breakpoint	CTRL+F9	Toggles a breakpoint between being disabled, but not actually removed—making it available for future use—and being enabled again.

Table 62: Edit menu commands (Continued)

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.


Option	Description
Find What	Selects the text to search for.
Match Whole Word Only	Searches the specified text only if it occurs as a separate word. Otherwise specifying <code>int</code> will also find <code>print</code> , <code>sprintf</code> etc. This option is only available when you search in an editor window.
Match Case	Searches only occurrences that exactly match the case of the specified text. Otherwise specifying <code>int</code> will also find <code>INT</code> and <code>Int</code> . This option is only available when you search in an editor window.
Search as Hex	Searches for the specified hexadecimal value. This option is only available when you search in the Memory window.
 Find Next	Finds the next occurrence of the selected text.
Find Previous	Finds the previous occurrence of the selected text.
Stop	Stops an ongoing search. This button is only available during a search in the Memory window.

Table 63: Find dialog box options

Replace dialog box

The **Replace** dialog box is available from the **Edit** menu.

Option	Description
Find What	Selects the text to search for.
Replace With	Selects the text to replace each found occurrence in the Replace With box.
Match Whole Word Only	Searches the specified text only if it occurs as a separate word. Otherwise <code>int</code> will also find <code>print</code> , <code>sprintf</code> etc. This checkbox is not available when you perform the search in the Memory window.
Match Case	Searches only occurrences that exactly match the case of the specified text. Otherwise specifying <code>int</code> will also find <code>INT</code> and <code>Int</code> . This checkbox is not available when you perform the search in the Memory window.
Search as Hex	Searches for the specified hexadecimal value. This checkbox is only available when you perform the search in the Memory window.
Find Next	Searches the next occurrence of the text you have specified.
Replace	Replaces the searched text with the specified text.
Replace All	Replaces all occurrences of the searched text in the current editor window.

Table 64: 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.

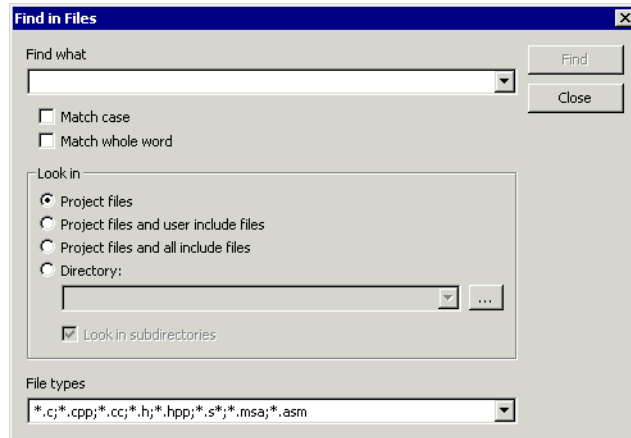


Figure 149: Find in Files dialog box

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. There are two options for fine-tuning the search:

- | | |
|-------------------------|---|
| Match case | Searches only for occurrences that exactly match the case of the specified text. Otherwise specifying <code>int</code> will also find <code>INT</code> and <code>Int</code> . |
| Match whole word | Searches only for the string when it occurs as a separate word. Otherwise <code>int</code> will also find <code>print</code> , <code>sprintf</code> and so on. |

Look in

The options in the **Look in** area lets you specify which files you want to search in for a specified string. Choose between:

Project files	The search will be performed in all files that you have explicitly added to your project.
Project files and user include files	The search will be performed in all files that you have explicitly added to your project and all files included by them, except the include files in the IAR Embedded Workbench installation directory.
Project files and all include files	The search will be performed in all project files that you have explicitly added to your project and all files included by them.
Directory	The search will be 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 will be 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.

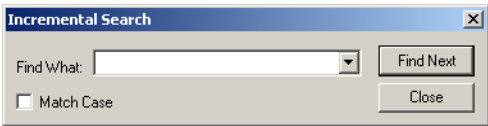


Figure 150: Incremental Search dialog box

Find What

Type the string to search for. The search will be 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 will be 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

Function button	Description
Find Next	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 .
Close	Closes this dialog box.

Table 65: 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.

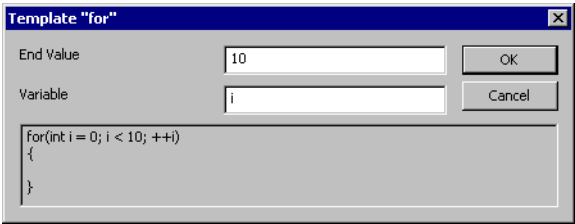


Figure 151: Template dialog box

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 103.

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.



Figure 152: View menu

Menu command	Description
Messages	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.
Workspace	Opens the current Workspace window.
Source Browser	Opens the Source Browser window.
Breakpoints	Opens the Breakpoints window.
Toolbars	The options Main and Debug toggle the two toolbars on and off.
Status bar	Toggles the status bar on and off.

Table 66: View menu commands

Menu command	Description
Debugger windows	During a debugging session, the different 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 Profiling window Stack window For descriptions of these windows, see <i>C-SPY windows</i> , page 343.

Table 66: View menu commands (Continued)

PROJECT MENU

The **Project** menu provides commands for working with workspaces, projects, groups, and files, as well as specifying options for the build tools, and running the tools on the current project.

Add Files...	
Add Group...	
Import File List...	
Edit Configurations...	
Remove	
Create New Project...	
Add Existing Project...	
Options...	ALT+F7
Source Code Control	▶
Make	F7
Compile	CTRL+F7
Rebuild All	
Clean	
Batch build...	F8
Stop Build	CTRL+BREAK
Download and Debug	CTRL+D
Debug without Downloading	
Make & Restart Debugger	SHIFT+F3
Restart Debugger	CTRL+F3

Figure 153: Project menu

Menu Command	Description
Add Files	Displays a dialog box that where you can select which files to include to the current project.
Add Group	Displays a dialog box where you can create a new group. The Group Name text box specifies the name of the new group. The Add to Target list selects the targets to which the new group should be added. By default the group is added to all targets.
Import File List	Displays a standard Open dialog box where you can import information about files and groups from projects created using another IAR tool chain. To import information from project files which have one of the older filename extensions <code>pew</code> or <code>prj</code> you must first have exported the information using the context menu command Export File List available in your own IAR Embedded Workbench.

Table 67: Project menu commands



Menu Command		Description
Edit Configurations		Displays the Configurations for project dialog box, where you can define new or remove existing build configurations.
Remove		In the Workspace window, removes the selected item from the workspace.
Create New Project		Displays a dialog box where you can create a new project and add it to the workspace.
Add Existing Project		Displays a dialog box where you can add an existing project to the workspace.
Options Alt+F7		Displays the Options for node 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.
Source Code Control		Opens a submenu with commands for source code control, see <i>Source Code Control menu</i> , page 269.
	Make F7	Brings the current build configuration up to date by compiling, assembling, and linking only the files that have changed since the last build.
	Compile Ctrl+F7	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 Compile command is only enabled if every file in the selection is individually suitable for the command. You can also select a <i>group</i> , 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.
Rebuild All		Rebuilds and relinks all files in the current target.
Clean		Removes any intermediate files.
Batch BuildF8		Displays a dialog box where you can configure named batch build configurations, and build a named batch.
Stop BuildCtrl+Break		Stops the current build operation.

Table 67: Project menu commands (Continued)





	Menu Command	Description
	Download and Debug Ctrl+D	Downloads the application and starts C-SPY so that you can debug the project object file. If necessary, a make will be performed before running C-SPY to ensure the project is up to date. This command is not available during debugging.
	Debug without Downloading	Starts C-SPY so that you can debug the project object file. This menu command is a short cut for the Suppress Download option available on the Download page. This command is not available during debugging.
	Make & Restart Debugger	Stops C-SPY, makes the active build configuration, and starts the debugger again; all in a single command. This command is only available during debugging.
	Restart Debugger	Stops C-SPY and starts the debugger again; all in a single command. This command is only available during debugging.

Table 67: Project menu commands (Continued)

Argument variables summary

Variables can be used for paths and arguments. The following argument variables can be used:

Variable	Description
\$CUR_DIR\$	Current directory
\$CUR_LINE\$	Current line
\$CONFIG_NAME\$	The name of the current build configuration, for example Debug or Release.
\$EW_DIR\$	Top directory of IAR Embedded Workbench, for example c:\program files\iar systems\embedded workbench 5.n
\$EXE_DIR\$	Directory for executable output
\$FILE_BNAME\$	Filename without extension
\$FILE_BPATH\$	Full path without extension
\$FILE_DIR\$	Directory of active file, no filename
\$FILE_FNAME\$	Filename of active file without path
\$FILE_PATH\$	Full path of active file (in Editor, Project, or Message window)
\$LIST_DIR\$	Directory for list output
\$OBJ_DIR\$	Directory for object output

Table 68: Argument variables

Variable	Description
\$PROJ_DIR\$	Project directory
\$PROJ_FNAME\$	Project file name without path
\$PROJ_PATH\$	Full path of project file
\$TARGET_DIR\$	Directory of primary output file
\$TARGET_BNAME\$	Filename without path of primary output file and without extension
\$TARGET_BPATH\$	Full path of primary output file without extension
\$TARGET_FNAME\$	Filename without path of primary output file
\$TARGET_PATH\$	Full path of primary output file
\$TOOLKIT_DIR\$	Directory of the active product, for example c:\program files\iar systems\embedded workbench 5.n\arm
\$_ENVVAR_\$	The environment variable <i>ENVVAR</i> . Any name within \$_ and _\$ will be expanded to that system environment variable.

Table 68: Argument variables (Continued)

Configurations for project dialog box

In the **Configuration for project** dialog box—available by choosing **Project>Edit Configurations**—you can define new build configurations for the selected project; either entirely new, or based on a previous project.

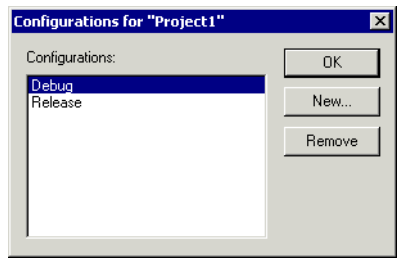


Figure 154: Configurations for project dialog box

The dialog box contains the following:

Operation	Description
Configurations	Lists existing configurations, which can be used as templates for new configurations.
New	Opens a dialog box where you can define new build configurations.
Remove	Removes the configuration that is selected in the Configurations list.

Table 69: 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.

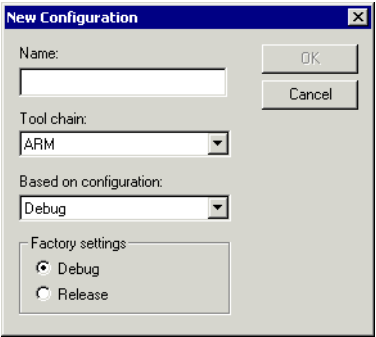


Figure 155: New Configuration dialog box

The dialog box contains the following:

Item	Description
Name	The name of the build configuration.
Tool chain	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.
Based on configuration	A currently defined build configuration that you want the new configuration to be based on. The new configuration will inherit the project settings as well as 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.
Factory settings	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.

Table 70: New Configuration dialog box options

Create New Project dialog box

The **Create New Project** dialog box is available from the **Project** menu, and lets you create a new project based on a template project. There are template projects available for C/C++ applications, assembler applications, and library projects. You can also create your own template projects.

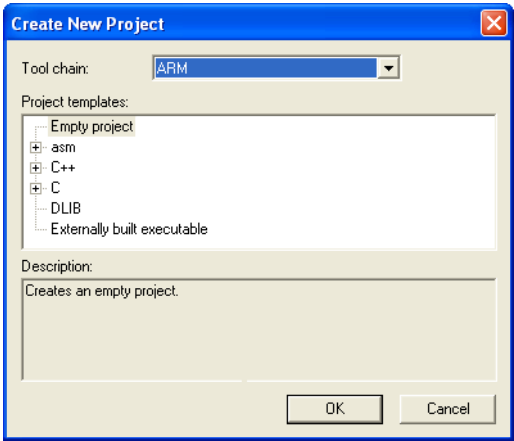


Figure 156: Create New Project dialog box

The dialog box contains the following:

Item	Description
Tool chain	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.
Project templates	Lists all available template projects that you can base a new project on.

Table 71: Description of Create New Project dialog box

Options 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 the following options:

Category	Description
General Options	General options
C/C++ Compiler	IAR C/C++ Compiler options
Assembler	IAR Assembler options
Converter	Options for converting ELF output to Motorola or Intex-standard.
Custom Build	Options for extending the tool chain
Build Actions	Options for pre-build and post-build actions
Linker	IAR ILINK Linker options. This category is available for application projects.
Library Builder	Library builder options. This category is available for library projects.
Debugger	IAR C-SPY Debugger options
Simulator	Simulator-specific options

Table 72: 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.

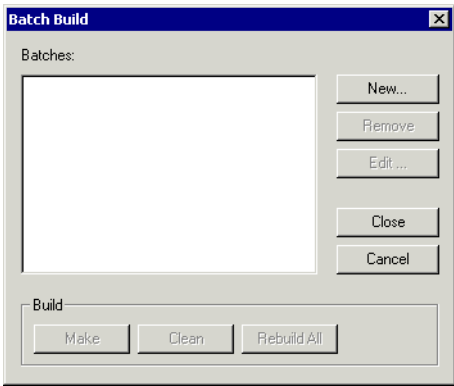


Figure 157: Batch Build dialog box

The dialog box contains the following:

Item	Description
Batches	Lists all currently defined batches of build configurations.
New	Displays the Edit Batch Build dialog box, where you can define new batches of build configurations.
Remove	Removes the selected batch.
Edit	Displays the Edit Batch Build dialog box, where you can modify already defined batches.
Build	Consists of the three build commands Make , Clean , and Rebuild All .

Table 73: 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.

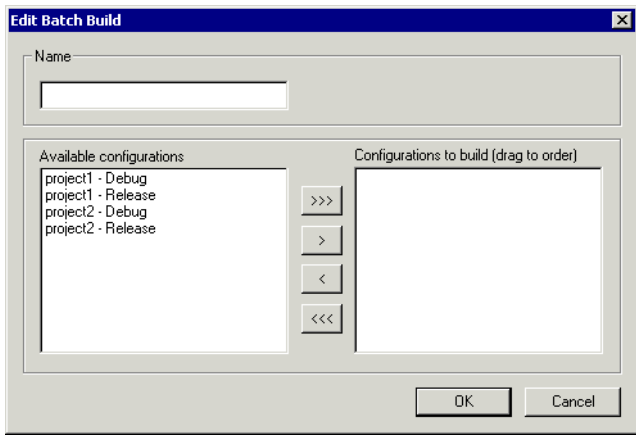


Figure 158: Edit Batch Build dialog box

The dialog box contains the following:

Item	Description
Name	The name of the batch.
Available configurations	Lists all build configurations that are part of the workspace.
Configurations to build	Lists all the build configurations you select to be part of a named batch.

Table 74: 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 334.

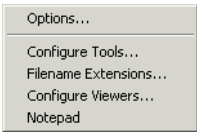


Figure 159: Tools menu

Tools menu commands

Menu command	Description
Options	Displays the IDE Options dialog box where you can customize the IDE. In the left side of the dialog box, select a category and the corresponding options are displayed in the right side of the dialog box. Which categories that are available in this dialog box depends on your IDE configuration, and whether the IDE is in a debugging session or not.
Configure Tools	Displays a dialog box where you can set up the interface to use external tools.
Filename Extensions	Displays a set of dialog boxes where you can define the filename extensions to be accepted by the build tools.
Configure Viewers	Displays a dialog box where you can configure viewer applications to open documents with.
Notepad	User-configured. This is an example of a user-configured addition to the Tools menu.

Table 75: 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.

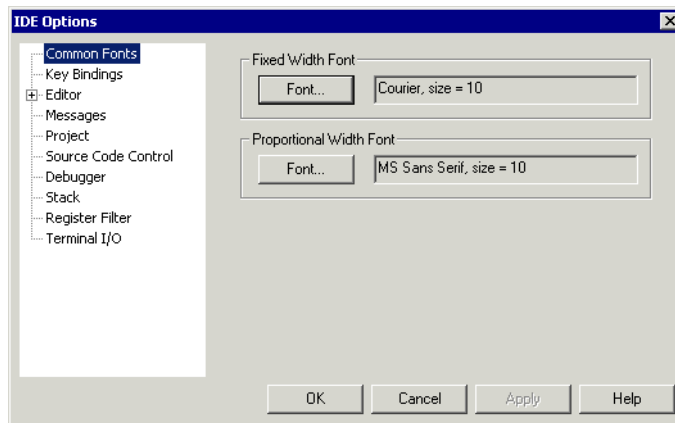


Figure 160: Common Fonts options

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 323.

KEY BINDINGS OPTIONS

Use the **Key Bindings** options—available by choosing **Tools>Options**—to customize the shortcut keys used for the IDE menu commands.

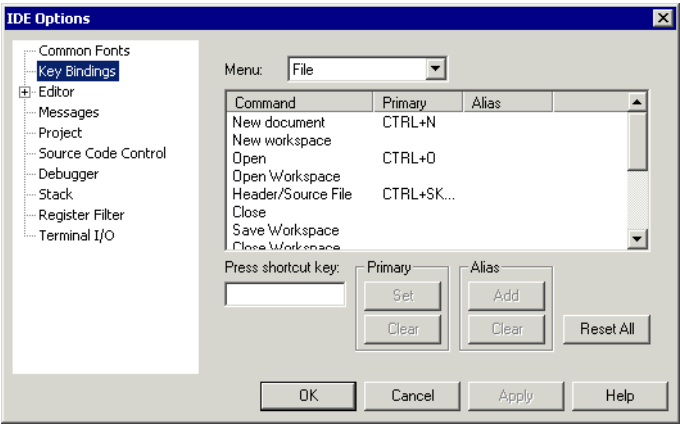


Figure 161: Key Bindings options

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 below.

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. It is not possible to 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.

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.

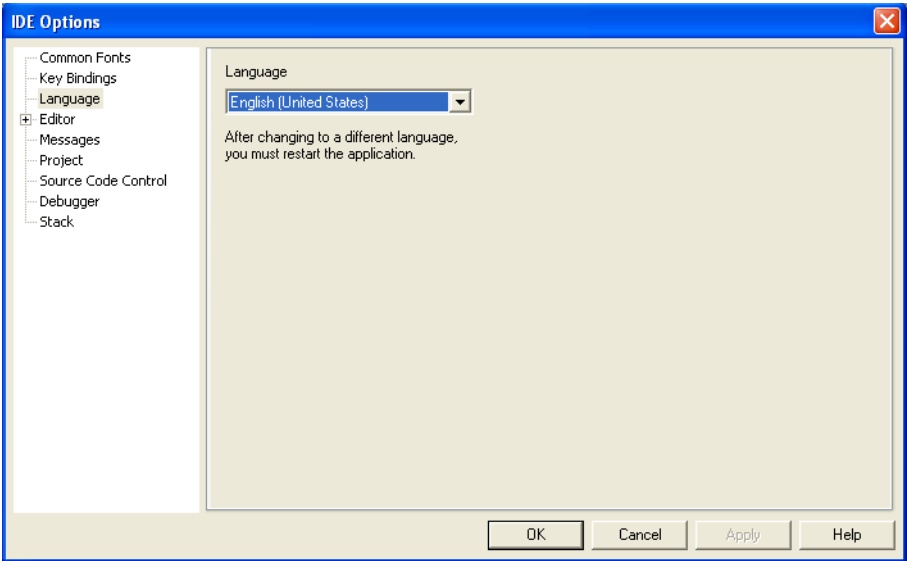


Figure 162: Language options

Language

Use the drop-down list to choose the language to be used.

In the IDE, **English (United States)** and **Japanese** are 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.

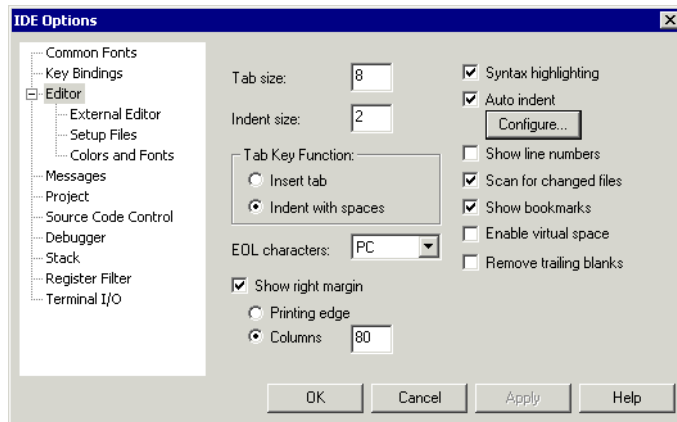


Figure 163: Editor options

For more information about the IAR Embedded Workbench IDE Editor and how it can be used, see *Editing*, page 99.

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 will be 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 323, and *Syntax coloring*, page 101.

Auto Indent

Use this option to ensure that when you press Return, the new line will automatically be indented. For C/C++ source files, indentation will be 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 319. 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 check if files have been modified by some other tool. In that case the files will be automatically reloaded. If a file has been modified in the IDE, you will be prompted first.

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.

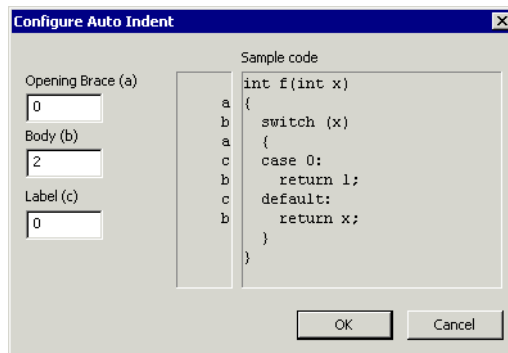


Figure 164: Configure Auto Indent dialog box

To read more about indentation, see *Automatic text indentation*, page 102.

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.

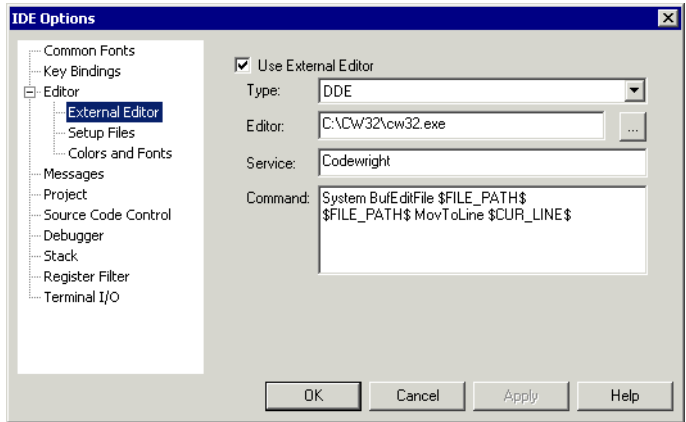


Figure 165: External Editor options

See also *Using an external editor*, page 106.

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 321.

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 321.

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 321.

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: Variables can be used in arguments. See *Argument variables summary*, page 306, 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.

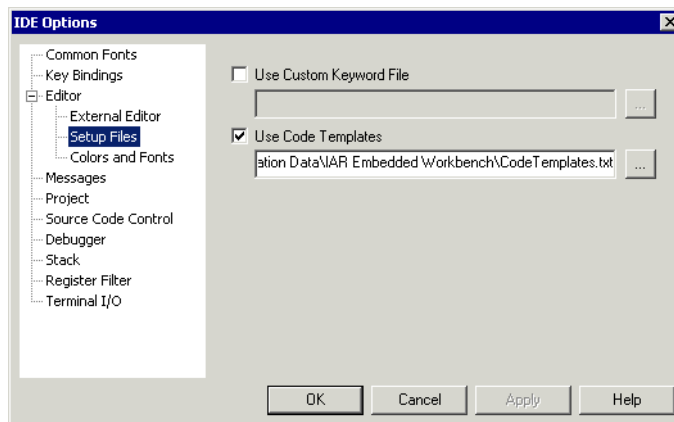


Figure 166: Editor Setup Files options

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 101.

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 103.

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.

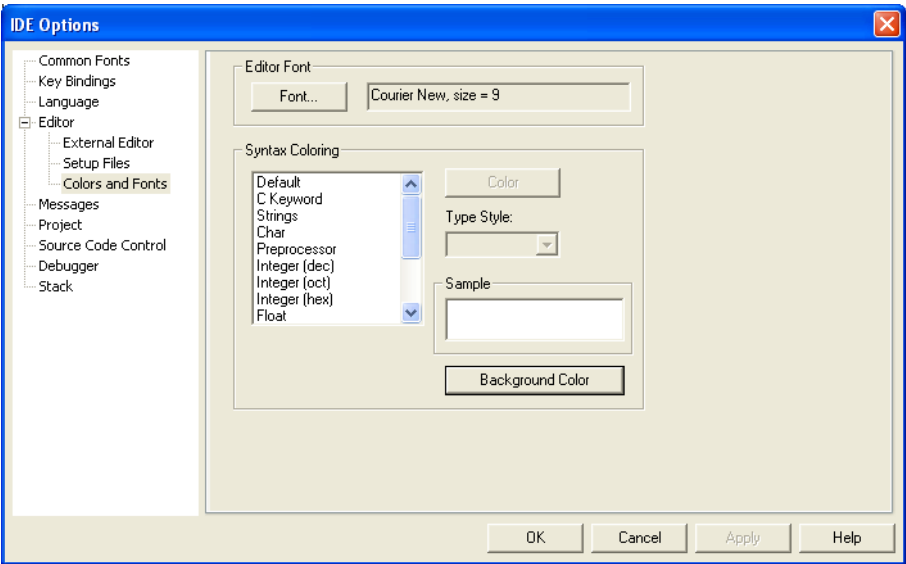


Figure 167: Editor Colors and Fonts options

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 color and type style for selected elements. The elements you can customize are: C or C++, compiler keywords, assembler keywords, and user-defined keywords. Use the following options:

Scroll-bar list	Lists the possible items for which you can specify font and style of syntax.
Color	Provides a list of colors to choose from for the selected element.
Type Style	Provides a list of type styles to choose from.
Sample	Displays the current setting.

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.

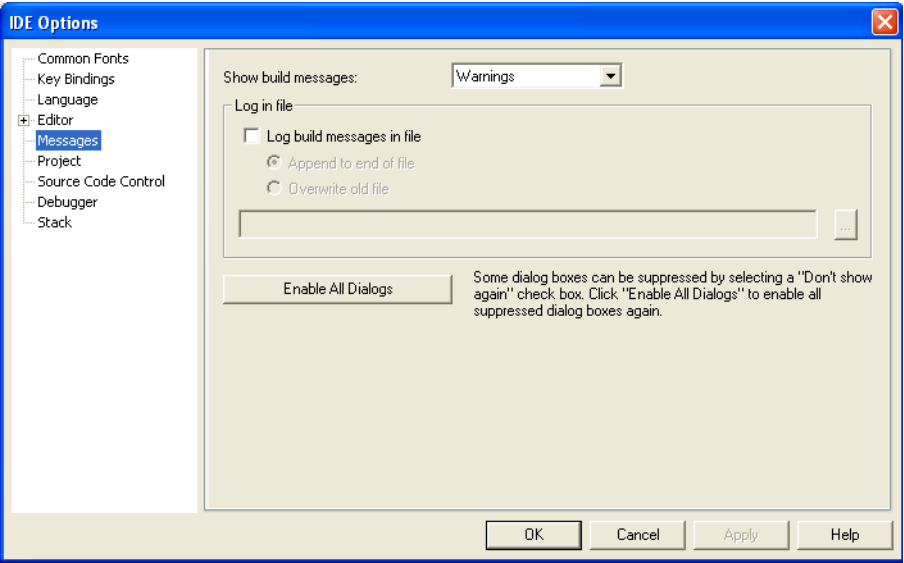


Figure 168: Messages option

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.

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.

Some dialog boxes can be suppressed by selecting a **Don't show again** check box, for example:

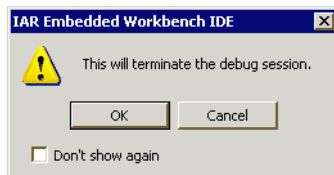


Figure 169: Message dialog box containing a Don't show again option

PROJECT OPTIONS

Use the **Project** options—available by choosing **Tools>Options**—to set options for the **Make** and **Build** commands.

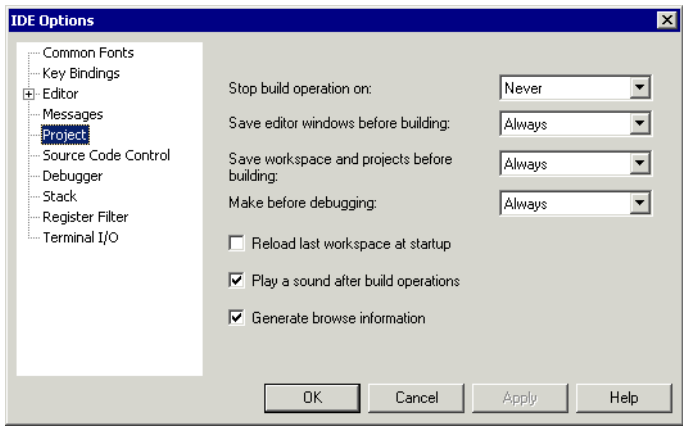


Figure 170: Project options

The following options are available:

Option	Description
Stop build operation on	Specifies when the build operation should stop. Never: Do not stop. Warnings: Stop on warnings and errors. Errors: Stop on errors.
Save editor windows before building	Always: Always save before Make or Build. Ask: Prompt before saving. Never: Do not save.
Save workspace and projects before building	Always: Always save before Make or Build. Ask: Prompt before saving. Never: Do not save.
Make before debugging	Always: Always perform the Make command before debugging. Ask: Always prompt before performing the Make command. Never: Do not perform the Make command before debugging.

Table 76: Project IDE options

Option	Description
Reload last workspace at startup	Select this option if you want the last active workspace to load automatically the next time you start IAR Embedded Workbench.
Play a sound after build operations	Plays a sound when the build operations are finished.
Generate browse information	Enables the use of the Source Browser window, see <i>Source Browser window</i> , page 280.

Table 76: Project IDE options (Continued)

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.

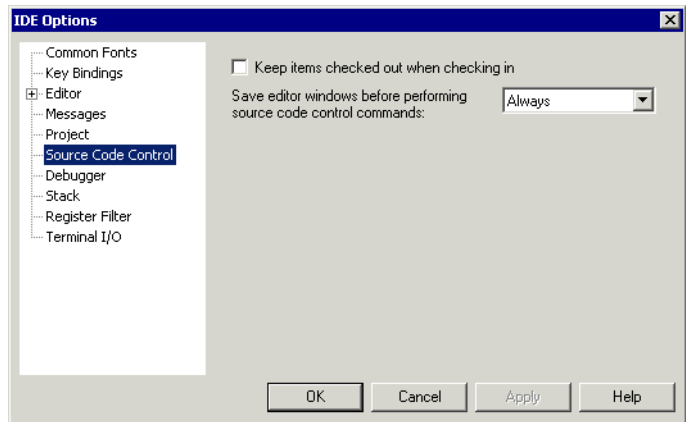


Figure 171: Source Code Control options

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 272.

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.

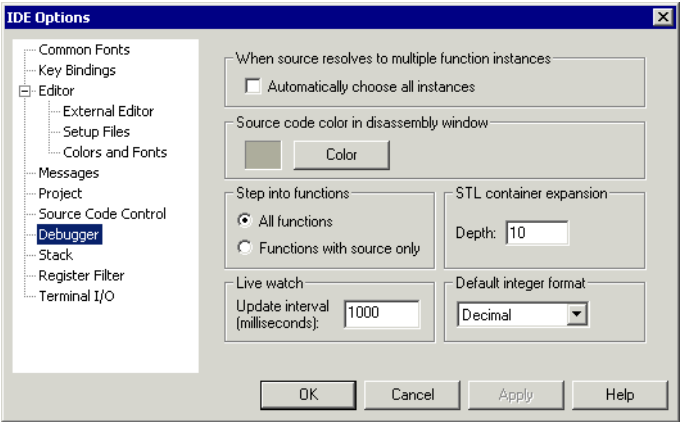


Figure 172: Debugger options

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.

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 in, for example, the Watch window. Additional elements can be shown by clicking the expansion arrow.

Live watch

The **Update interval** value decides how often the C-SPY Live Watch window is updated during execution.

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.

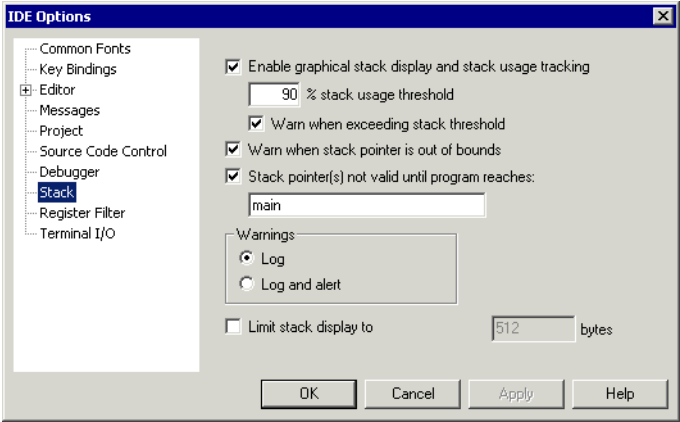


Figure 173: Stack options

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 370.

% 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 take place. 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. By using 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 the following 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 147.

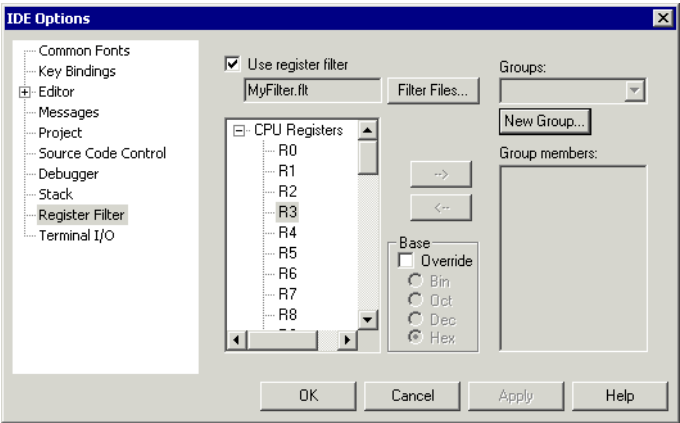


Figure 174: Register Filter options

The following options are available:

Option	Description
Use register filter	Enables the usage of register filters.
Filter Files	Displays a dialog box where you can select or create a new filter file.
Groups	Lists available groups in the register filter file, alternatively displays the new register group.
New Group	The name for the new register group.
Group members	Lists the registers selected from the register scroll bar window.
Base	Changes the default integer base.

Table 77: 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.

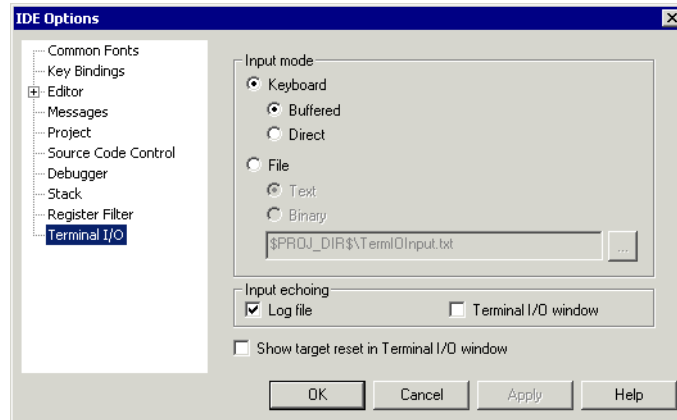


Figure 175: Terminal I/O options

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.

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.

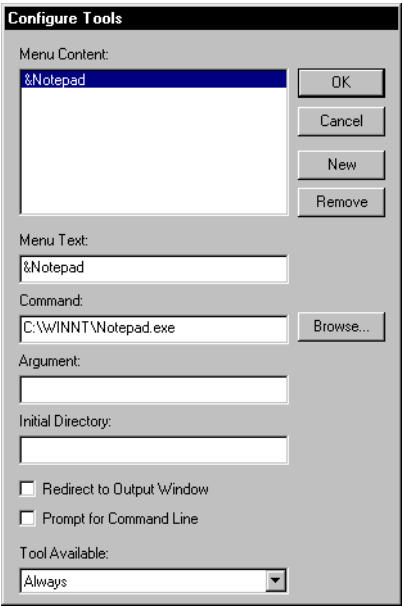


Figure 176: Configure Tools dialog box

Note: If you intend to add an external tool to the standard build tool chain, see *Extending the tool chain*, page 96.

The following options are available:

Option	Description
Menu Content	Lists all available user defined menu commands.
Menu Text	Specifies the text for the menu command. By adding the sign &, the following letter, N in this example, will then appear as the mnemonic key for this command. The text you type in this field will be reflected in the Menu Content field.

Table 78: Configure Tools dialog box options

Option	Description
Command	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.
Argument	Optionally type an argument for the command.
Initial Directory	Specifies an initial working directory for the tool.
Redirect to Output window	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 <i>require</i> user input or make special assumptions regarding the console that they execute in, will <i>not</i> work at all if launched with this option.
Prompt for Command Line	Displays a prompt for the command line argument when the command is chosen from the Tools menu.
Tool Available	Specifies in which context the tool should be available, only when debugging or only when not debugging.

Table 78: Configure Tools dialog box options (Continued)

Note: Variables can be used in the arguments, allowing 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.

You can remove a command from the **Tools** menu by selecting it in this list and clicking **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.

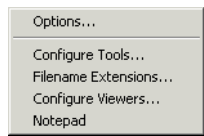


Figure 177: Customized Tools menu

Specifying command line commands or batch files

Command line commands or batch files need to be run from a command shell, so to add these to the **Tools** menu you need to specify an appropriate command shell in the **Command** text box. These are the command shells that can be entered as commands:

System	Command shell
Windows 2000/XP/Vista	cmd.exe (recommended) or command.com

Table 79: Command shells

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.

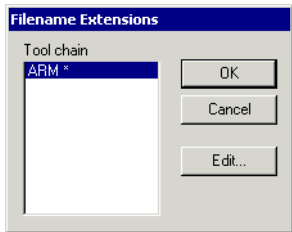


Figure 178: Filename Extensions dialog box

Note the * sign which indicates that there are 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.

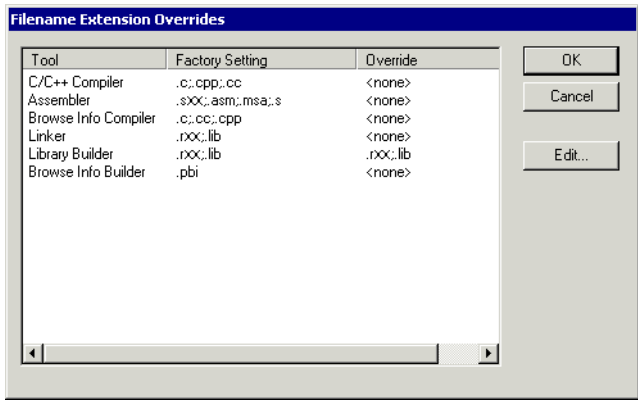


Figure 179: Filename Extension Overrides dialog box

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.

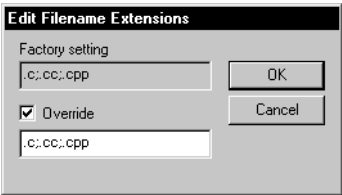


Figure 180: Edit Filename Extensions dialog box

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 will be 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.

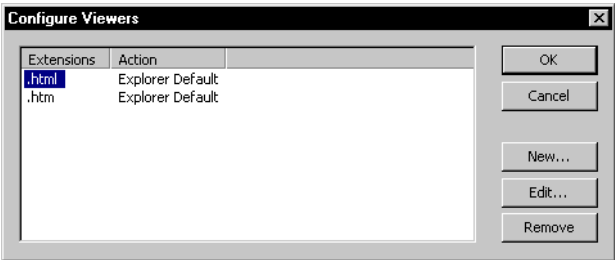


Figure 181: Configure Viewers dialog box

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.



Figure 182: Edit Viewer Extensions dialog box

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.

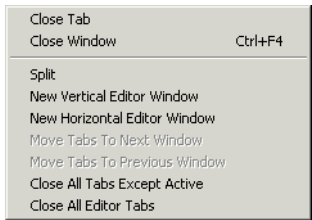


Figure 183: Window menu

The following commands are available on the Window menu:

Menu command		Description
Close Tab		Closes the active tab.
Close Window	CTRL+F4	Closes the active editor window.
Split		Splits an editor window horizontally or vertically into two, or four panes, to allow you to see more parts of a file simultaneously.
New Vertical Editor Window		Opens a new empty window next to current editor window.
New Horizontal Editor Window		Opens a new empty window under current editor window.
Move Tabs To Next Window		Moves all tabs in current window to next window.
Move Tabs To Previous Window		Moves all tabs in current window to previous window.
Close All Tabs Except Active		Closes all the tabs except the active tab.
Close All Editor Tabs		Closes all tabs currently available in editor windows.

Table 80: 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.

EMBEDDED WORKBENCH STARTUP DIALOG BOX

The **Embedded Workbench Startup** dialog box—available from the **Help** menu—provides easy access to ready-made example workspaces that can be built and executed *out of the box* for a smooth development startup.

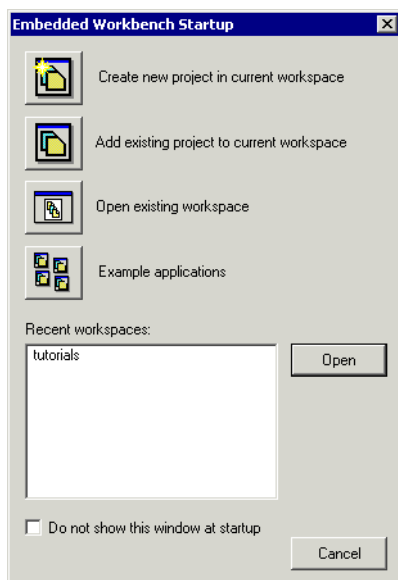


Figure 184: Embedded Workbench Startup dialog box

Create new project in current workspace

Use this option to create a new project in your current workspace.

Add existing project to current workspace

Use this option to add an existing project to your current workspace.

Open existing workspace

Use this option to open an existing workspace.

Note: Do not use this option to open an existing workspace which is part of your product installation, because that might overwrite the original files. Instead, use the option **Example applications**.

Example applications

Use this option to open the **Example Applications** dialog box. In this dialog box you can choose an example application which is part of your product installation. Click **Open** to first choose a destination directory for the project and then to open it. Select **Do not prompt for working copy directory** if you do not want to be prompted for a destination directory. In this case, the example application will be copied to the `My Documents\IAR Embedded Workbench\arm\Example Applications` directory.

Recent workspace

In the list of workspaces, select a recently used workspace and click **Open** to open it. If this is the first time you open your IAR Embedded Workbench, the list will be empty.

Do not show this window at startup

Use this option if you do not want the **Embedded Workbench Startup** dialog box to be automatically displayed when you start IAR Embedded Workbench. If you have selected this option, you can still open the dialog box from the **Help** menu.

Do not show the Information Center at startup

Use this option if you do not want the Information Center to be automatically displayed in the editor window when you start your IAR Embedded Workbench. If you have selected this option, you can still access the Information Center from the **Help** menu.

The Information Center provides convenient access to useful information.

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 343
- *C-SPY menus*, page 374.

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
- Stack window
- Symbols window.

Additional windows will be 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 the following keyboard keys to edit the contents of these windows:

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

Table 81: Editing in C-SPY windows

C-SPY DEBUGGER MAIN WINDOW

When you start the debugger, the following 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 in the debug toolbar. The following menus are available when C-SPY is running:

Menu	Description
Debug	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.
Disassembly	The Disassembly menu provides commands for controlling the disassembly processor mode.

Table 82: C-SPY menu

Menu	Description
Simulator	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.

Table 82: C-SPY menu (Continued)

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.

You can display a description of any button by pointing to it with the mouse pointer. When a command is not available the corresponding button will be dimmed and you will not be able to select it.

The following diagram shows the command corresponding to each button:

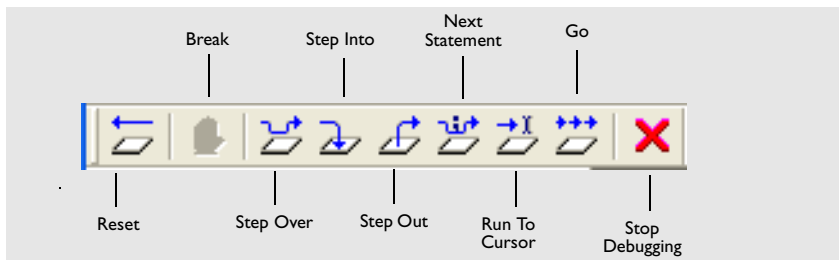


Figure 185: C-SPY debug toolbar

DISASSEMBLY WINDOW

The C-SPY Disassembly window—available from the **View** menu—shows the application being debugged as disassembled application code.

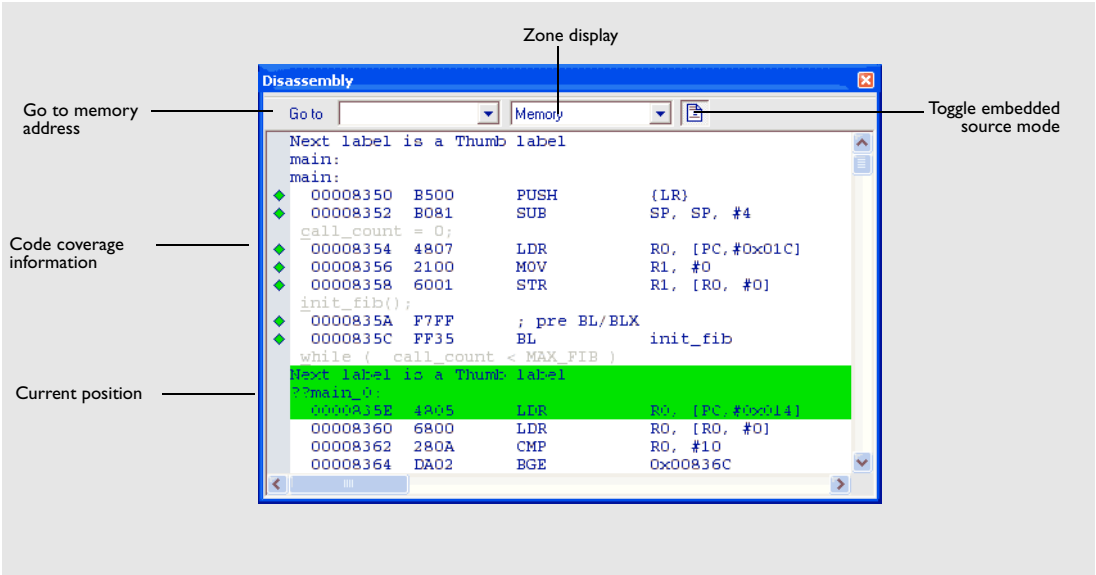


Figure 186: C-SPY Disassembly window

Note that it is possible to disassemble in different modes, see *Disassembly menu*, page 379.

Toolbar

At the top of the window you can find a toolbar.

Operation	Description
Go to	The location you want to view. This can be a memory address, or the name of a variable, function, or label.
Zone display	Lists the available memory zones to display. Read more about Zones in the section <i>Memory addressing</i> , page 143.
Toggle Mixed-Mode	Toggles between showing only disassembled code or disassembled code together with the corresponding source code. Source code requires that the corresponding source file has been compiled with debug information.

Table 83: Disassembly window toolbar

The display area

The current position—highlighted in green—indicates the next assembler instruction to be executed. You can move the cursor to any line in the Disassembly window by clicking on the line. Alternatively, you can 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.

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

Clicking the right mouse button in the Disassembly window displays a context menu which gives you access to some commands.

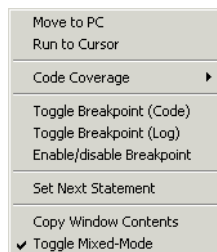


Figure 187: Disassembly window context menu

The following commands are available on the menu:

Menu command	Description
Move to PC	Displays code at the current program counter location.
Run to Cursor	Executes the application from the current position up to the line containing the cursor.
Code Coverage	Opens a submenu with commands for controlling code coverage.
Enable	Enable toggles code coverage on and off.
Show	Show toggles between displaying and hiding code coverage. Executed code is indicated by a green diamond.
Clear	Clear clears all code coverage information.

Table 84: Disassembly context menu commands

Menu command	Description
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 <i>Code breakpoints dialog box</i> , page 283.
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 <i>Log breakpoints dialog box</i> , page 285.
Enable/Disable Breakpoint	Enables and Disables a breakpoint.
Set Next Statement	Sets program counter to the location of the insertion point.
Copy Window Contents	Copies the selected contents of the Disassembly window to the clipboard.
Toggle Mixed-Mode	Toggles between showing only disassembled code or disassembled code together with the corresponding source code. Source code requires that the corresponding source file has been compiled with debug information.

Table 84: Disassembly context menu commands (Continued)

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 different memory or register zones, or monitor different parts of the memory.

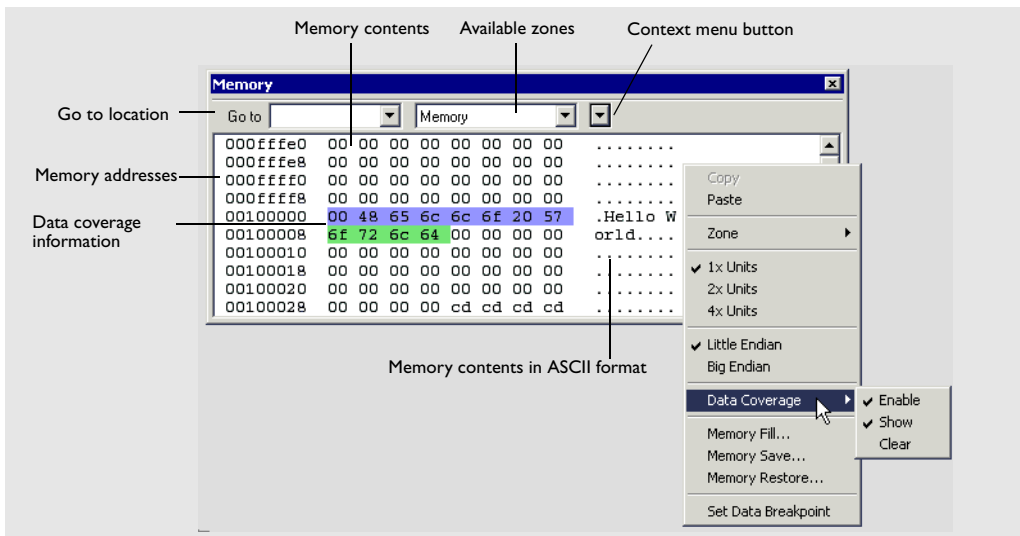


Figure 188: Memory window

Toolbar

At the top of the window you can find a toolbar:

Operation	Description
Go to	The location you want to view. This can be a memory address, or the name of a variable, function, or label.
Zone display	Lists the available memory zones to display. Read more about Zones in section <i>Memory addressing</i> , page 143.
Context menu button	Displays the context menu, see <i>Memory window context menu</i> , page 350.

Table 85: Memory window operations

The display area

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 the following 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.

Memory window context menu

The following context menu is available in the Memory window:



Figure 189: Memory window context menu

The following commands are available on the menu:

Menu command	Description
Copy, Paste	Standard editing commands.
Zone	Lists the available memory zones to display. Read more about Zones in <i>Memory addressing</i> , page 143.

Table 86: Commands on the memory window context menu

Menu command	Description
x1, x2, x4 Units	Switches between displaying the memory contents in units of 8, 16, or 32 bits
Little Endian	Switches between displaying the contents in big-endian or little-endian order.
Big Endian	
Data Coverage	
Enable	Enable toggles data coverage on and off.
Show	Show toggles between showing and hiding data coverage.
Clear	Clear clears all data coverage information.
Memory Fill	Displays the Fill dialog box, where you can fill a specified area with a value, see <i>Fill dialog box</i> , page 351.
Memory Save	Displays the Memory Save dialog box, where you can save the contents of a specified memory area to a file, see <i>Memory Save dialog box</i> , page 352.
Memory Restore	Displays the Memory Restore dialog box, where you can load the contents of a file in Intex-hex or Motorola s-record format to a specified memory zone, see <i>Memory Restore dialog box</i> , page 353.
Set Data Breakpoint	Sets breakpoints directly in the Memory window. The breakpoint is not highlighted; you can see, edit, and remove it in the Breakpoints dialog box. The breakpoints you set in this window will be triggered for both read and write access.

Table 86: Commands on the memory window context menu (Continued)

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.

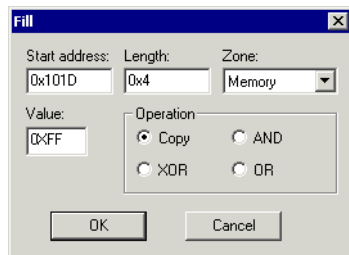


Figure 190: Fill dialog box

Options

Option	Description
Start Address	Type the start address—in binary, octal, decimal, or hexadecimal notation.
Length	Type the length—in binary, octal, decimal, or hexadecimal notation.
Zone	Select memory zone.
Value	Type the 8-bit value to be used for filling each memory location.

Table 87: Fill dialog box options

These are the available memory fill operations:

Operation	Description
Copy	The Value will be copied to the specified memory area.
AND	An AND operation will be performed between the Value and the existing contents of memory before writing the result to memory.
XOR	An XOR operation will be performed between the Value and the existing contents of memory before writing the result to memory.
OR	An OR operation will be performed between the Value and the existing contents of memory before writing the result to memory.

Table 88: 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.

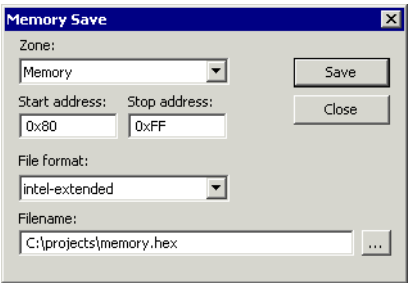


Figure 191: Memory Save dialog box

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.

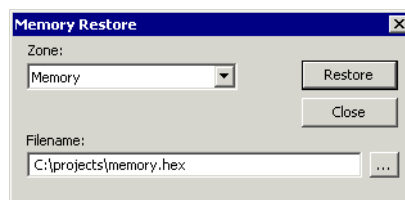


Figure 192: Memory Restore dialog box

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.

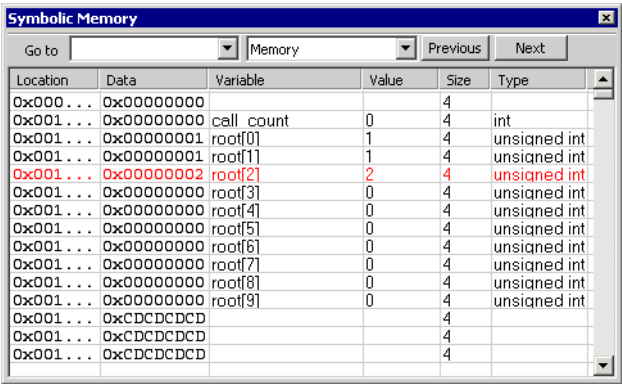


Figure 193: Symbolic Memory window

Toolbar

At the top of the window there is a toolbar:

Operation	Description
Go to	The memory location or symbol you want to view.
Zone display	Lists the available memory zones to display. To read more about zones, see <i>Memory addressing</i> , page 143.
Previous	Jumps to the previous symbol.
Next	Jumps to the next symbol.

Table 89: Symbolic Memory window toolbar

The display area

The display area displays the memory space, where information is provided in the following columns:

Column	Description
Location	The memory address.
Data	The memory contents in hexadecimal format. The data is grouped according to the size of the symbol. This column is editable.
Variable	The variable name; requires that the variable has a fixed memory location. Local variables are not displayed.
Value	The value of the variable. This column is editable.
Type	The type of the variable.

Table 90: Symbolic memory window columns

There are several different ways to navigate within the memory space:

- Text that is dropped in the window will be 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

The following context menu is available in the Symbolic Memory window:

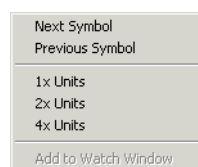


Figure 194: Symbolic Memory window context menu

The following commands are available on the context menu:

Menu command	Description
Next Symbol	Jumps to the next symbol.
Previous Symbol	Jumps to the previous symbol.

Table 91: Commands on the Symbolic Memory window context menu

Menu command	Description
x1, x2, x4 Units	Switches between displaying the memory contents in units of 8, 16, or 32 bits. This applies only to rows which do not contain a variable.
Add to Watch Window	Adds the selected symbol to the Watch window.

Table 91: Commands on the Symbolic Memory window context menu (Continued)

REGISTER WINDOW

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 sub-groups of bits.

You can open several instances of this window, which is very convenient if you want to keep track of different register groups.

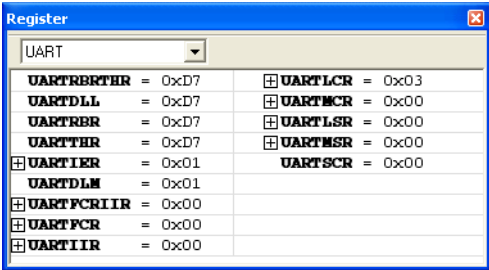


Figure 195: Register window

You can select which register group to display in the Register window using the drop-down list. To define application-specific register groups, see *Defining application-specific groups*, page 148.

WATCH WINDOW

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.

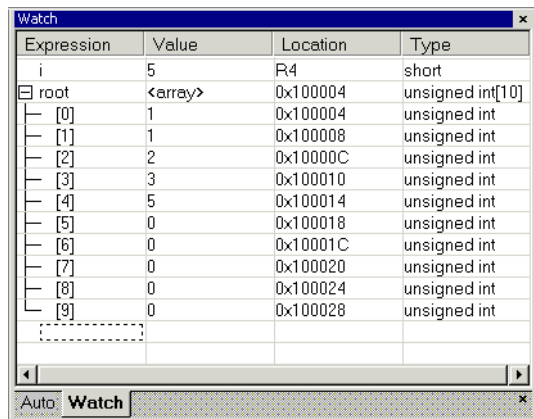


Figure 196: Watch window

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

The following context menu is available in the Watch window:

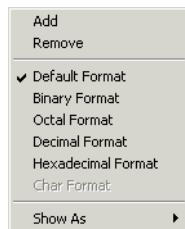


Figure 197: Watch window context menu

The menu contains the following commands:

Menu command	Description
Add, Remove	Adds or removes the selected expression.
Default Format, Binary Format, Octal Format, Decimal Format, Hexadecimal Format, Char Format	Changes the display format of expressions. The display format setting affects different types of expressions in different ways, see Table 93, <i>Effects of display format setting on different types of expressions</i> . Your selection of display format is saved between debug sessions.
Show As	Provides a submenu with commands for changing the default type interpretation of variables. The commands on this submenu are mainly useful for assembler variables—data at assembler labels—as these are by default displayed as integers. For more information, see <i>Viewing assembler variables</i> , page 132.

Table 92: Watch window context menu commands

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

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

Table 93: 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.

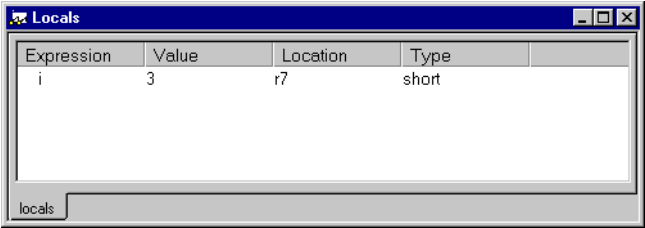


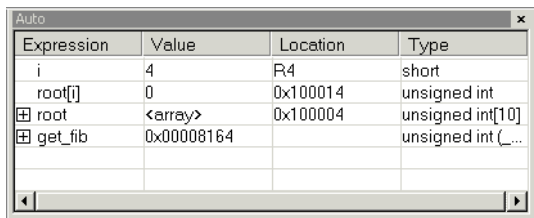
Figure 198: Locals window

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 357.

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.



Expression	Value	Location	Type
i	4	R4	short
root[i]	0	0x100014	unsigned int
root	<array>	0x100004	unsigned int[10]
get_fib	0x00008164		unsigned int (...)

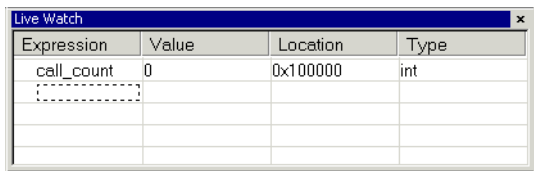
Figure 199: Auto window

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 357.

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.



Expression	Value	Location	Type
call_count	0	0x100000	int

Figure 200: 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, as well as commands for changing the default type interpretation of variables. For information about these commands, see *Watch window context menu*, page 357.

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.

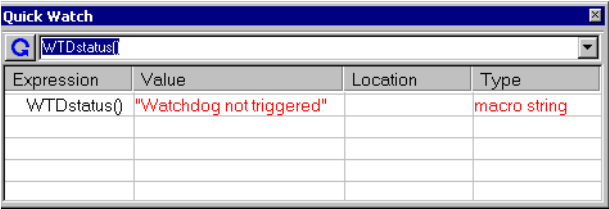


Figure 201: Quick Watch window



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 131 and *Executing macros using Quick Watch*, page 154.

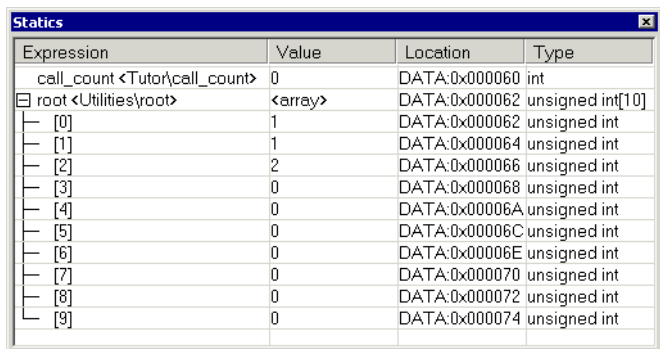
Quick Watch window context menu

The context menu available in the Quick Watch window provides commands for changing the display format of expressions, as well as commands for changing the default type interpretation of variables. For information about these commands, see *Watch window context menu*, page 357.

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.



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

Figure 202: Statics window

The display area

The display area shows the values of variables with static storage duration, where information is provided in the following columns:

Column	Description
Expression	The name of the variable. The base name of the variable is followed by the full name, which includes module, class, or function scope. This column is not editable.
Value	The value of the variable. Values that have changed are highlighted in red. This column is editable.
Location	The location in memory where this variable is stored.
Type	The data type of the variable.

Table 94: Symbolic memory window columns

Statics window context menu

The following context menu is available in the Statics window:

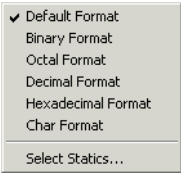


Figure 203: Statics window context menu

The menu contains the following commands:

Menu command	Description
Default Format, Binary Format, Octal Format, Decimal Format, Hexadecimal Format, Char Format	Changes the display format of expressions. The display format setting affects different types of expressions in different ways, see Table 93, <i>Effects of display format setting on different types of expressions</i> . Your selection of display format is saved between debug sessions.
Select Statics	Displays a dialog box where you can select a subset of variables to be displayed in the Statics window, see <i>Select Statics dialog box</i> , page 363.

Table 95: 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.

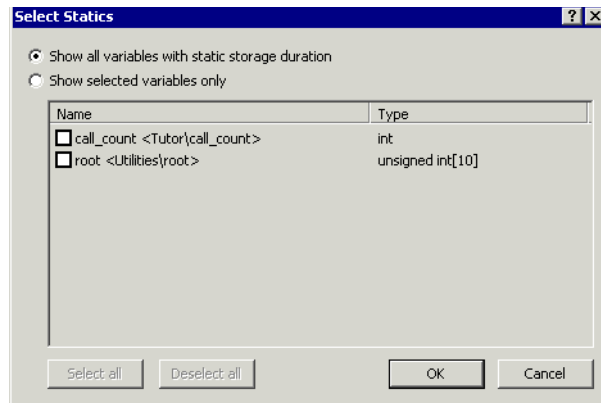


Figure 204: Select Statics dialog box

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, 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.

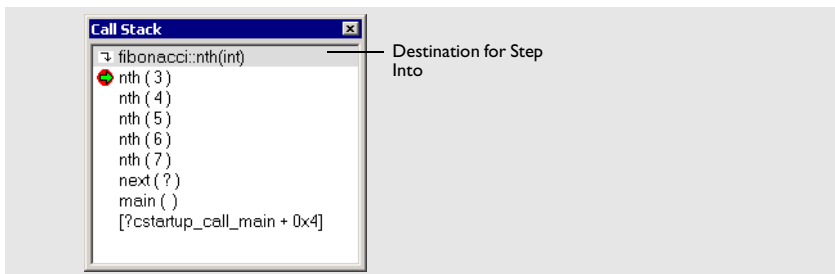


Figure 205: Call Stack window

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 by right-clicking in the Call Stack window provides the following commands:

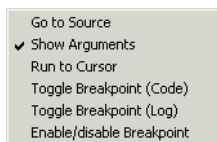


Figure 206: Call Stack window context menu

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 need to 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.

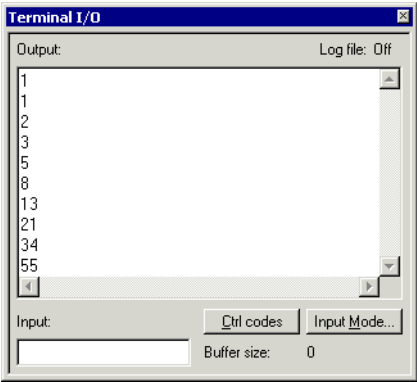


Figure 207: Terminal I/O window

Clicking the **Ctrl codes** button opens a menu with submenus for input of special characters, such as EOF (end of file) and NUL.

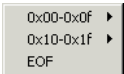


Figure 208: Ctrl codes menu

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.

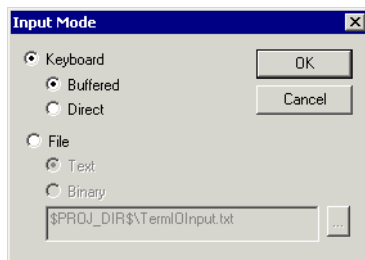


Figure 209: Input Mode dialog box

For reference information about the options available in the dialog box, see *Terminal I/O options*, page 333.

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, as well as 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.

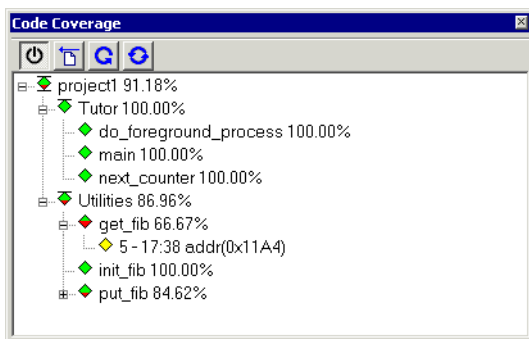


Figure 210: Code Coverage window

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.

Code coverage commands

In addition to the commands available as icon buttons in the toolbar, clicking the right mouse button in the Code Coverage window displays a context menu that gives you access to these and some extra commands.

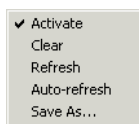






Figure 211: Code coverage context menu

You can find the following commands on the menu:

	Activate	Switches code coverage on and off during execution.
	Clear	Clears the code coverage information. All step points are marked as not executed.
	Refresh	Updates the code coverage information and refreshes the window. All step points that has been executed since the last refresh are removed from the tree.
	Auto-refresh	Toggles the automatic reload of code coverage information on and off. When turned on, the code coverage information is reloaded automatically when C-SPY stops at a breakpoint, at a step point, and at program exit.
	Save As	Saves the current code coverage information in a text file.

The following 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.

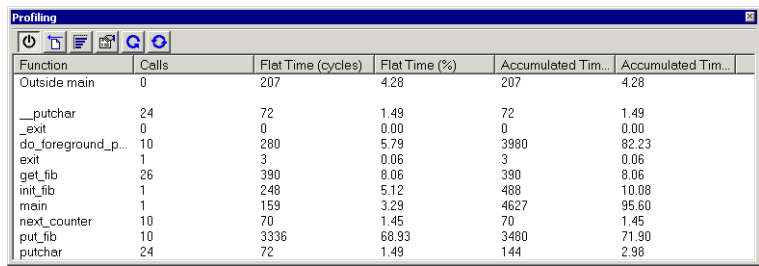
For step point lines, the information displayed is the column number range and the row number of the step point in the source window, followed by the address of the step point.

```
<column start>--<column end>:<row>.
```

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.



Function	Calls	Flat Time (cycles)	Flat Time (%)	Accumulated Tim...	Accumulated Tim...
Outside main	0	207	4.28	207	4.28
__putchar	24	72	1.49	72	1.49
_exit	0	0	0.00	0	0.00
do_foreground_p...	10	280	5.79	3980	82.23
exit	1	3	0.06	3	0.06
get_fib	26	390	8.06	390	8.06
init_fib	1	248	5.12	488	10.08
main	1	159	3.29	4627	95.60
next_counter	10	70	1.45	70	1.45
put_fib	10	3336	68.93	3480	71.90
putchar	24	72	1.49	144	2.98

Figure 212: 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:

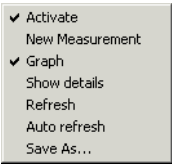








Figure 213: Profiling context menu

You can find the following commands on the menu:

	Activate	Toggles profiling on and off during execution.
	New measurement	Starts a new measurement. By clicking the button, the values displayed are reset to zero.
	Graph	Displays the percentage information for Flat Time and Accumulated Time as graphs (bar charts) or numbers.
	Show details	Shows more detailed information about the function selected in the list. A window is opened showing information about callers and callees for the selected function.
	Refresh	Updates the profiling information and refreshes the window.
	Auto refresh	Toggles the automatic update of profiling information on and off. When turned on, the profiling information is updated automatically when C-SPY stops at a breakpoint, at a step point, and at program exit.
	Save As	Saves the current profiling information in a text file.

Profiling columns

The Profiling window contains the following columns:

Column	Description
Function	The name of each function.
Calls	The number of times each function has been called.
Flat Time	The total time spent in each function in cycles or as a percentage of the total number of cycles, excluding all function calls made from that function.
Accumulated Time	Time spent in each function in cycles or as a percentage of the total number of cycles, including all function calls made from that function.

Table 96: Profiling window columns

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.

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.

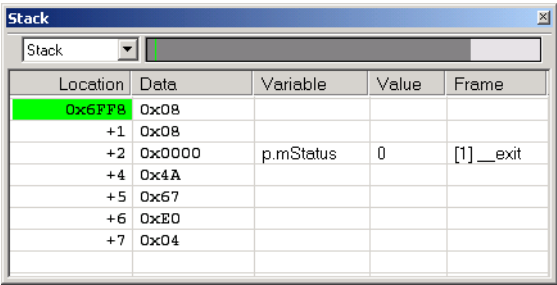


Figure 214: Stack window

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 330.

The Stack window columns

The main part of the window displays the contents of stack memory in the following columns:

Column	Description
Location	Displays the location in memory. The addresses are displayed in increasing order. The address referenced by the stack pointer, in other words the top of the stack, is highlighted in a green color.
Data	Displays the contents of the memory unit at the given location. From the Stack window context menu, you can select how the data should be displayed; as a 1-, 2-, or 4-byte group of data.
Variable	Displays the name of a variable, if there is a local variable at the given location. Variables are only displayed if they are declared locally in a function, and located on the stack and not in registers.
Value	Displays the value of the variable that is displayed in the Variable column.
Frame	Displays the name of the function the call frame corresponds to.

Table 97: Stack window columns

The Stack window context menu

The following context menu is available if you right-click in the Stack window:

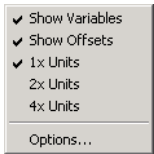


Figure 215: Stack window context menu

The following commands are available in the context window:

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 <i>Stack options</i> , page 330.

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 455.

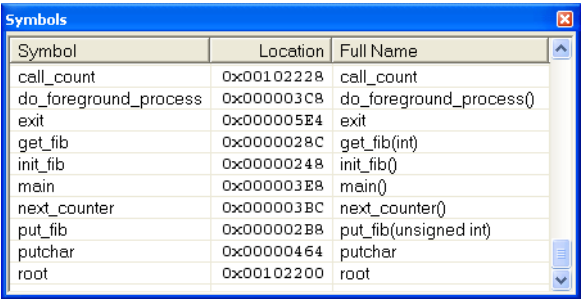
Syntax

```
--proc_stack_mode stackstart, stackend
```

The parameters *stackstart* and *stackend* follow the standard C-SPY expression syntax. White space 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.



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

Figure 216: Symbols window

The display area

The display area lists the symbols, where information is provided in the following columns:

Column	Description
Symbol	The symbol name.
Location	The memory address.
Full Name	The symbol name; often the same as the contents of the Symbol column but differs for example for C++ member functions.

Table 98: Symbols window columns

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

Symbols window context menu

The following context menu is available in the Symbols window:

Functions
Variables
Labels

Figure 217: Symbols window context menu

The following commands are available on the menu:

Menu command	Description
Function	Toggles the display of function symbols in the list.
Variables	Toggles the display of variables in the list.
Labels	Toggles the display of labels in the list.

Table 99: 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 will be available depending on which C-SPY driver you are using. For information about driver-specific menus, see the chapter *Hardware-specific debugging*, page 213.

DEBUG MENU

The **Debug** menu provides commands for executing and debugging your application. Most of the commands are also available as toolbar buttons.

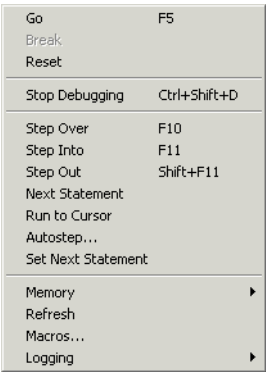


Figure 218: Debug menu


Menu Command	Description
 Go F5	Executes from the current statement or instruction until a breakpoint or program exit is reached.

Table 100: Debug menu commands









Menu Command		Description
	Break	Stops the application execution.
	Reset	Resets the target processor.
	Stop Debugging Ctrl+Shift+D	Stops the debugging session and returns you to the project manager.
	Step Over F10	Executes the next statement, function call, or instruction, without entering C or C++ functions or assembler subroutines.
	Step Into F11	Executes the next statement or instruction, entering C or C++ functions or assembler subroutines.
	Step Out Shift+F11	Executes from the current statement up to the statement after the call to the current function.
	Next Statement	Executes directly to the next statement without stopping at individual function calls.
	Run to Cursor	Executes from the current statement or instruction up to a selected statement or instruction.
	Autostep	Displays the Autostep settings dialog box which lets you customize and perform autostepping.
	Set Next Statement	Moves the program counter directly to where the cursor is, without executing any source code. Note, however, that this creates an anomaly in the program flow and might have unexpected effects.
	Memory>Save	Displays the Memory Save dialog box, where you can save the contents of a specified memory area to a file, see <i>Memory Save dialog box</i> , page 352.
	Memory>Restore	Displays the Memory Restore dialog box, where you can load the contents of a file in Intex-extended or Motorola s-record format to a specified memory zone, see <i>Memory Restore dialog box</i> , page 353.
	Refresh	Refreshes the contents of all debugger windows. Because window updates are automatic, this is needed only in unusual situations, such as when target memory is modified in ways C-SPY cannot detect. It is also useful if code that is displayed in the Disassembly window is changed.
	Macros	Displays the Macro Configuration dialog box to allow you to list, register, and edit your macro files and functions.

Table 100: Debug menu commands (Continued)

Menu Command	Description
Logging>Set Log file	Displays a dialog box to allow you to log input and output from C-SPY to a file. You can select the type and the location of the log file. You can choose what you want to log: errors, warnings, system information, user messages, or all of these.
Logging> Set Terminal I/O Log file	Displays a dialog box to allow you to log terminal input and output from C-SPY to a file. You can select the destination of the log file.

Table 100: Debug menu commands (Continued)

Autostep settings dialog box

In the **Autostep settings** dialog box—available from the **Debug** menu—you can customize autostepping.

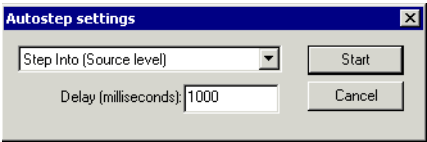


Figure 219: Autostep settings dialog box

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.

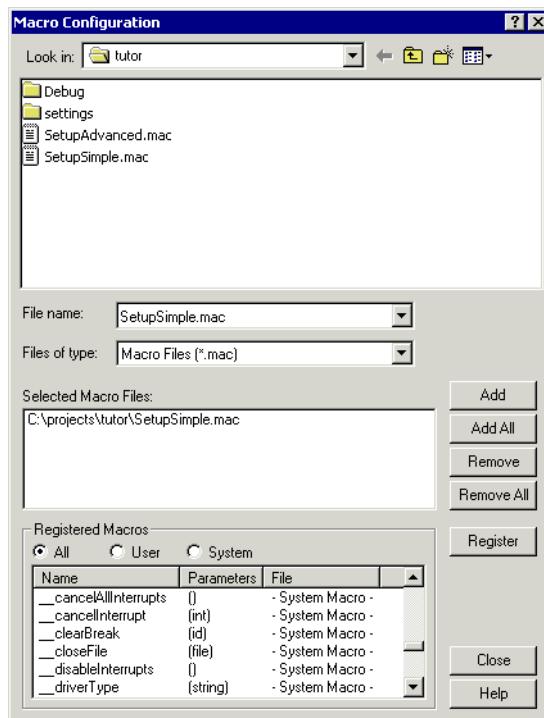


Figure 220: Macro Configuration dialog box

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.

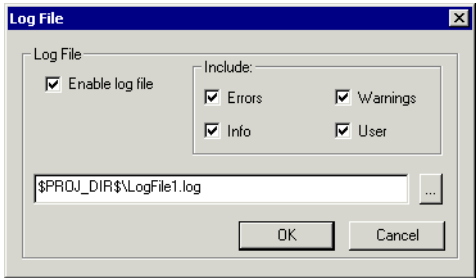


Figure 221: Log File dialog box

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:

Option	Description
Errors	C-SPY has failed to perform an operation.
Warnings	A suspected error.
Info	Progress information about actions C-SPY has performed.
User	Printouts from C-SPY macros, that is, your printouts using the <code>__message</code> statement.

Table 101: Log file options

Click the browse button, to override the default file type and location of the log file. Click **Save** to select the specified file—the default filename extension is `log`.

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.

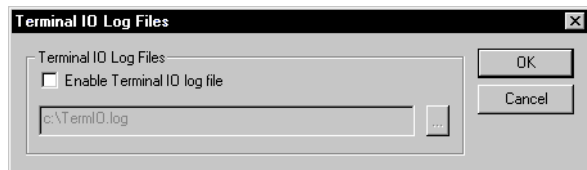


Figure 222: Terminal I/O Log File dialog box

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.

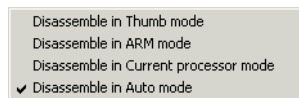


Figure 223: Disassembly menu

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:

Menu command	Description
Disassemble in Thumb mode	Select this option to disassemble your application in Thumb mode.
Disassemble in ARM Mode	Select this option to disassemble your application in ARM mode.
Disassemble in Current processor mode	Select this option to disassemble your application in the current processor mode.
Disassemble in Auto mode	Select this option to disassemble your application in automatic mode. This is the default option.

Table 102: Description of Disassembly menu commands

See also, *Disassembly window*, page 346.

General options

This chapter describes the general options in the IAR Embedded Workbench® IDE.

For information about how options can be set, see *Setting options*, page 91.

Target

The **Target** options specify the processor variant, FPU, and byte order for the IAR C/C++ Compiler and Assembler.

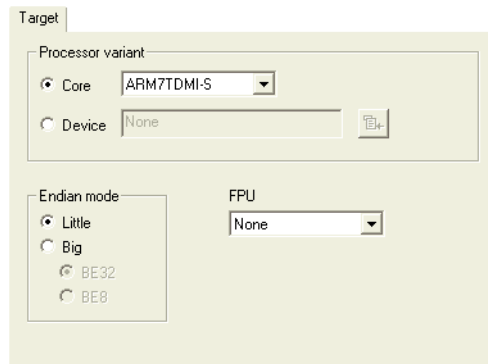


Figure 224: Target options

Note: There are additional target-specific options available on the **Code** page, see *Optimizations*, page 391.

PROCESSOR VARIANT

Choose between the following two options to specify the processor variant:

- | | |
|---------------|---|
| Core | The processor core you are using. For a description of the available variants, see the <i>IAR C/C++ Development Guide for ARM®</i> . |
| Device | The device your are using. The choice of device will automatically determine the default C-SPY® device description file. For information about how to override the default file, see <i>Device description file</i> , page 430. |

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 | <p>The lowest address holds the most significant byte, while the highest address holds the least significant byte.</p> <p>There are two variants of the big-endian mode. Choose:</p> <p>BE8 to make data big endian and code little endian</p> <p>BE32 to make both data and code big endian.</p> |

FPU

Use this option to generate code that carries out floating-point operations using a Vector Floating Point (VFP) coprocessor. By selecting a VFP coprocessor, you will override the use of the software floating-point library for all supported floating-point operations.

Select **VFPv1** support if you have a vector floating-point unit conforming to architecture VFPv1, such as the VFP10 rev 0. Similarly, select **VFPv2** on a system that implements a VFP unit conforming to architecture VFPv2, such as the VFP10 rev 1.

VFP9-S is an implementation of the VFPv2 architecture that can be used with the ARM9E family of CPU cores. Selecting the **VFP9-S** coprocessor is therefore identical to selecting the **VFPv2** architecture.

By selecting **none** (default) the software floating-point library is used.

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.

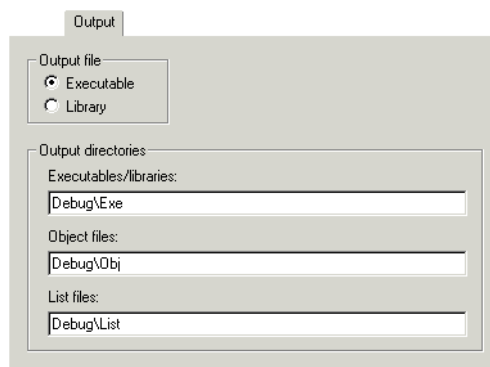


Figure 225: Output options

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 the following 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.

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 Configuration

With the **Library Configuration** options you can specify which library to use.

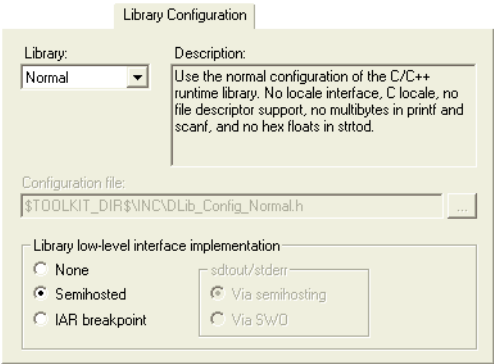


Figure 226: Library Configuration options

For information about the runtime library, library configurations, the runtime environment they provide, and the possible customizations, see *IAR C/C++ Development Guide for ARM®*.

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 the type of low-level interface for I/O to be included in the library.

For Cortex-M, choose between:

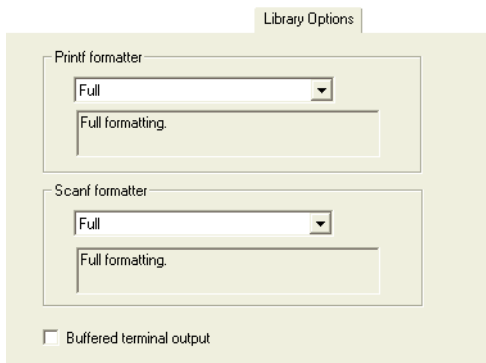
None	No low-level support for I/O available in the libraries. You must provide your own <code>__write</code> function to use the I/O functions part of the library.
Semihosted and with stdout/stderr via semihosting	Semihosted I/O which uses the <code>BKPT</code> instruction.
Semihosted and with stdout/stderr via SWO	Semihosted I/O which uses the <code>BKPT</code> instruction for all functions except for the <code>stdout</code> and <code>stderr</code> output where the <code>SWO</code> interface—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 <code>__write</code> function to use the I/O functions part of the library.
Semihosted	Semihosted I/O which uses the <code>SVC</code> instruction (earlier <code>SWI</code>).
IAR breakpoint	The IAR proprietary variant of semihosting, which does not use the <code>SVC</code> instruction and thus does not need to set a breakpoint on the <code>SVC</code> vector. This is an advantage for applications which require the <code>SVC</code> 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.



The screenshot shows the 'Library Options' dialog box. It contains two sections: 'Printf formatter' and 'Scanf formatter'. Each section has a dropdown menu set to 'Full' and a text box containing 'Full formatting.'. At the bottom, there is a checkbox labeled 'Buffered terminal output' which is currently unchecked.

Figure 227: Library Options page

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**.

For more information about using the stacks and heaps, see the *IAR C/C++ Development Guide for ARM®*.

BUFFERED TERMINAL OUTPUT

Use this option to make output via `stdout` buffered, to increase performance. Note that memory consumption is slightly increased.

MISRA C

Use the options on the **MISRA C** page to control how the IDE checks the source code for deviations from the MISRA C rules. The settings will be used for both the compiler and the linker.

For details about specific option, see the *IAR Embedded Workbench® MISRA C 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 91.

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 will be 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.



Figure 228: Multi-file Compilation

If you use this option, all files included in the selected group will be 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 266.

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.

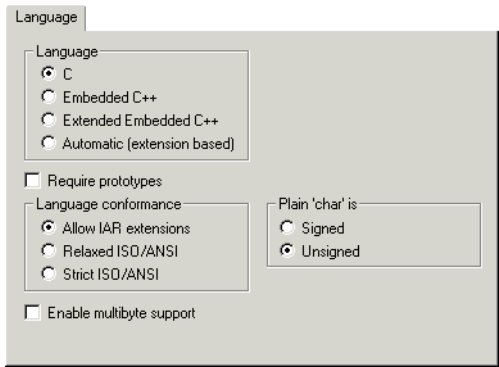


Figure 229: Compiler language options

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 utilized.

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 utilized.

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 will be 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 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.

Code

The **Code** options determine several target-specific settings for code generation.

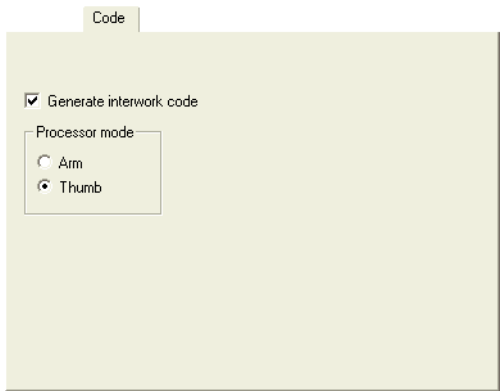


Figure 230: Compiler code options

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. |

Optimizations

The **Optimizations** options determine the type and level of optimization for generation of object code.

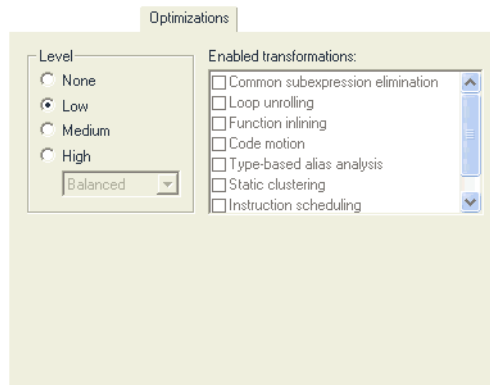


Figure 231: Compiler optimizations options

OPTIMIZATIONS

The compiler supports different levels of optimizations, and for the highest level it is possible to 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

The following 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.

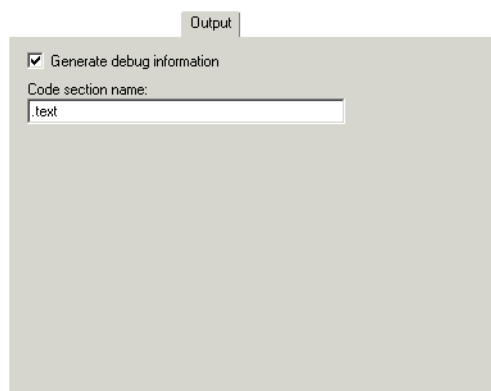


Figure 232: Compiler output options

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 different methods for controlling placement of code, see the *IAR C/C++ Development Guide for ARM®*.

List

The **List** options determine whether a list file is produced, and the information is included in the list file.

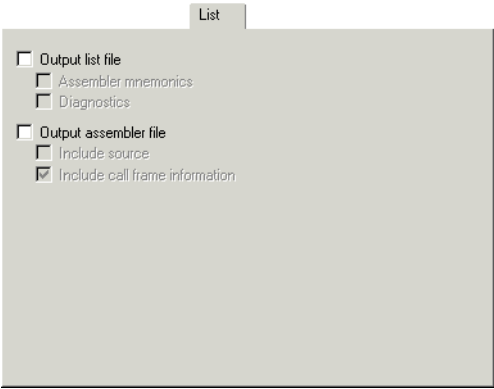


Figure 233: Compiler list file options

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.

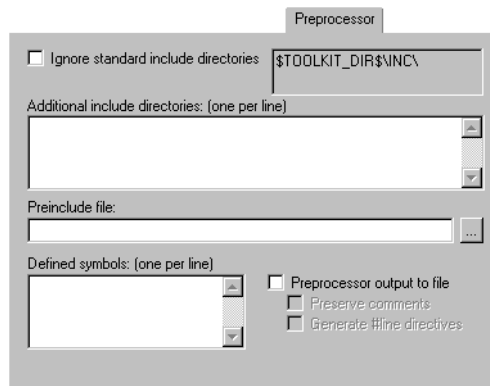


Figure 234: Compiler preprocessor options

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 by default depending on your choice of runtime library.

Type the full file path of your `#include` files.

Note: Any additional directories specified using this option will be 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 306.

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:

```
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:

```
#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.

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.

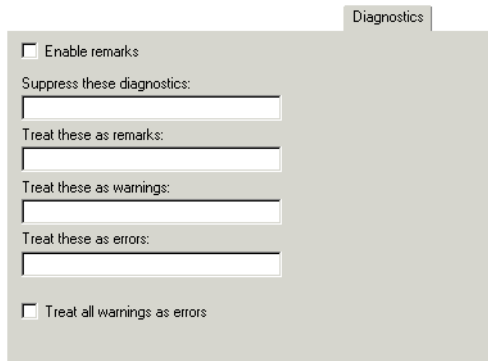


Figure 235: Compiler diagnostics options

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** page of the **General Options** category.

For details about specific option, see the *IAR Embedded Workbench® MISRA C Reference Guide* available from the **Help** menu.

Extra Options

The **Extra Options** page provides you with a command line interface to the compiler.

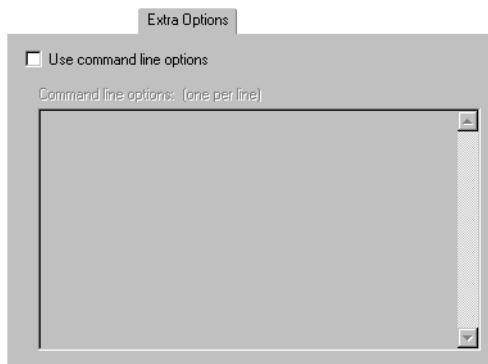


Figure 236: Extra Options page for the compiler

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 91.

Language

The **Language** options control the code generation of the assembler.

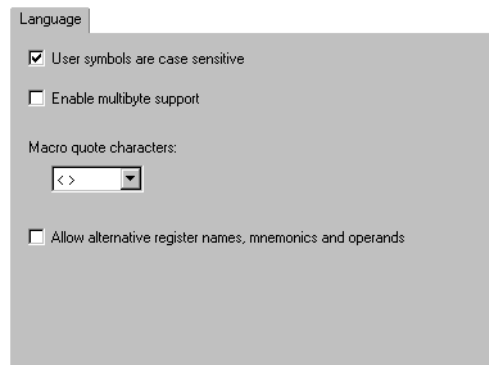


Figure 237: Assembler language options

USER SYMBOLS ARE CASE SENSITIVE

By default, case sensitivity is on. This means that, for example, `LABEL` and `label` refer to different symbols. You can deselect **User symbols are case sensitive** to turn case sensitivity off, in which case `LABEL` and `label` will refer to the same symbol.

ENABLE MULTIBYTE SUPPORT

By default, multibyte characters cannot be used in assembler 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 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:

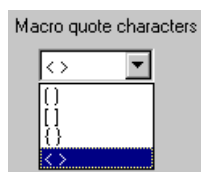


Figure 238: Choosing macro quote characters

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*.

Output

The **Output** options allow you to generate information to be used by a debugger such as the IAR C-SPY® Debugger.



Figure 239: Assembler output options

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.

List

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.

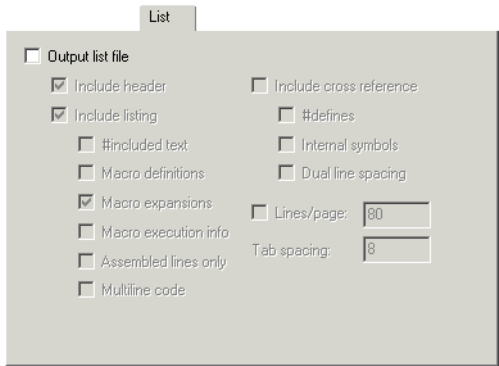


Figure 240: Assembler list file options

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 381, 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:

Option	Description
#included text	Includes #include files in the list file.
Macro definitions	Includes macro definitions in the list file.
Macro expansions	Includes macro expansions in the list file.
Macro execution info	Prints macro execution information on every call of a macro.

Table 103: Assembler list file options

Option	Description
Assembled lines only	Excludes lines in false conditional assembler sections from the list file.
Multiline code	Lists the code generated by directives on several lines if necessary.

Table 103: Assembler list file options (Continued)

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.

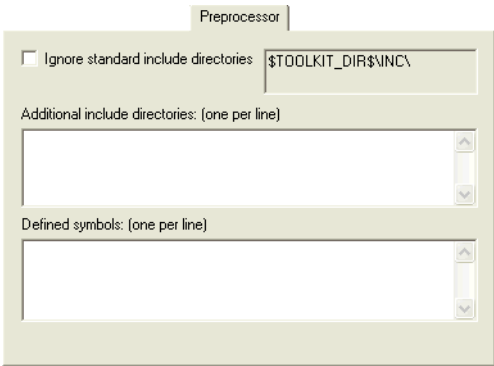


Figure 241: Assembler preprocessor options

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 by default.

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 68, *Argument variables*, page 306.

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.

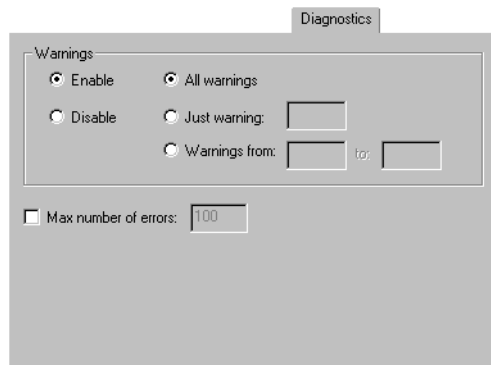


Figure 242: Assembler diagnostics options

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.

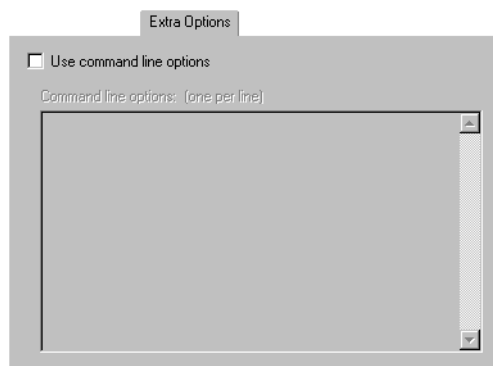


Figure 243: Extra Options page for the assembler

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 91.

Output

The **Output** options are used for specifying details about the promable output format and the level of debugging information included in the output file.



Figure 244: Converter output file options

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 `ar` 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 91.

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:

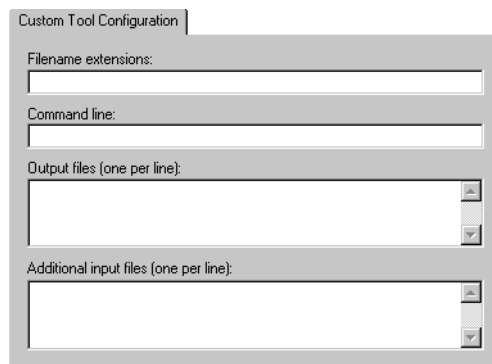


Figure 245: Custom tool options

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 there are any additional files that 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, so-called dependency files, are modified, the need for a rebuild is detected.

For an example, see *Extending the tool chain*, page 96.

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 91.

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.

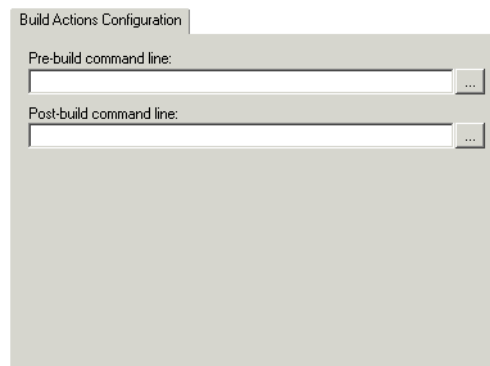


Figure 246: Build actions options

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 91.

Config

With the **Config** options you can specify the path and name of the linker configuration file and define symbols for the configuration file.

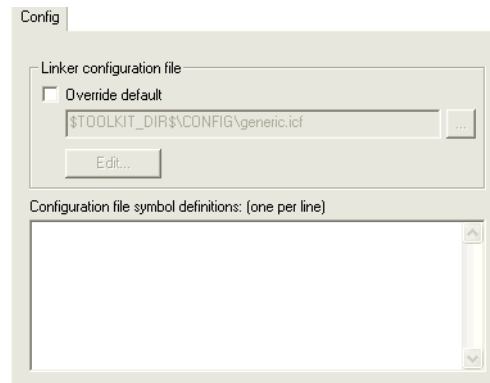


Figure 247: Linker configuration options

LINKER CONFIGURATION FILE

A default linker configuration file is selected automatically based on your project settings. You can override this by selecting the **Override default** option, and then specifying 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.

Click **Edit** to open the **Linker configuration file editor** dialog box, see *Linker configuration file editor*, page 416.

CONFIGURATION FILE SYMBOL DEFINITIONS

Use the text box to define constant configuration symbols to be used in the configuration file. Such symbols have the same effect as symbols defined using the `define symbol` directive in the linker configuration file.

LINKER CONFIGURATION FILE EDITOR

The **Linker configuration file editor** dialog box—available from the linker **Config** page—provides a graphical interface for editing the linker configuration file. On the linker **Config** page you can see the name of the file you are editing. The first time you edit the file after creating a project, a copy of the default template file `generic.icf` will be created.

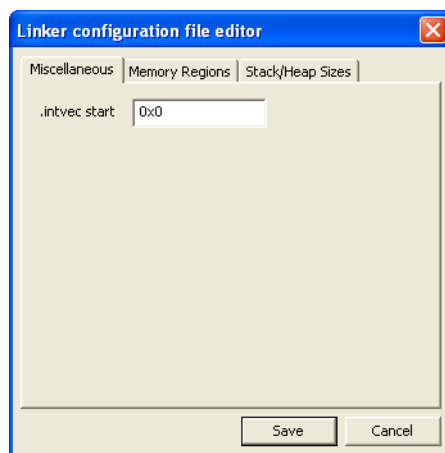


Figure 248: Linker configuration file editor

On the pages **Miscellaneous**, **Memory Regions**, and **Stack/Heap Sizes** you can specify the interrupt vector start address, the start and end addresses for ROM and RAM memory, and the stack and heap sizes that suit your application.

See the *IAR C/C++ Development Guide for ARM®* for more information about the linker configuration file.

Library

With the options on the **Library Usage** page you can make settings for library usage.

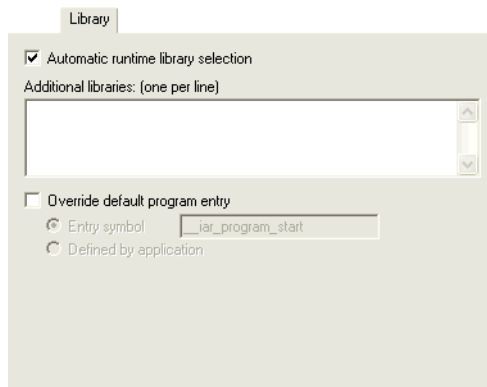


Figure 249: Library page

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. Note that you can only specify one library per line.

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 <code>__iar_program_start</code> 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.

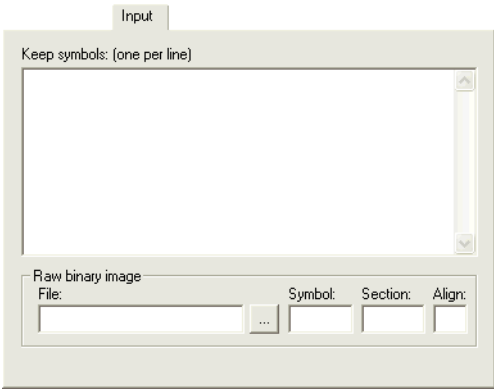


Figure 250: Linker input file options

KEEP SYMBOLS

Normally, the linker keeps a symbol only if it is needed by your application.

Use the text box to specify a symbol, or several symbols one per line, 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 the following 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 will be 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 *LAR C/C++ Development Guide for ARM®*.

Output

The **Output** options are used for specifying details about the output.

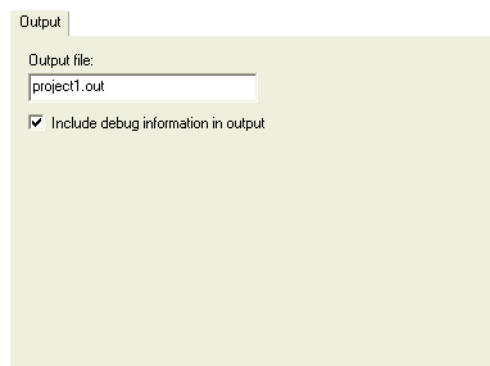


Figure 251: Linker output file options

OUTPUT FILE

Use **Output file** to specify the name of the ILINK output file. If you do not specify a name, the linker will use the project name with the filename extension `out`.

INCLUDE DEBUG INFORMATION IN OUTPUT

Use **Include debug information in output** to make the linker generate an ELF output file including DWARF for debug information.

List

The **List** options determine the generation of an linker listing.

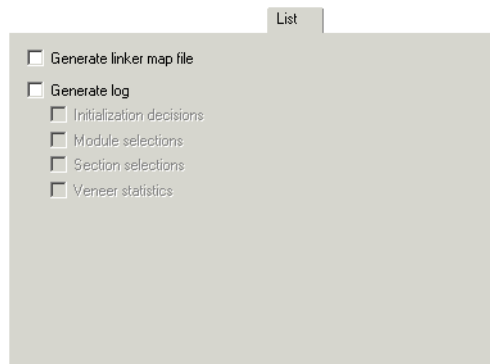


Figure 252: Linker diagnostics options

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:

- Initialization decisions
- Module selections
- Section selections
- Veneer statistics.

#define

You can define symbols with the **#define** option.

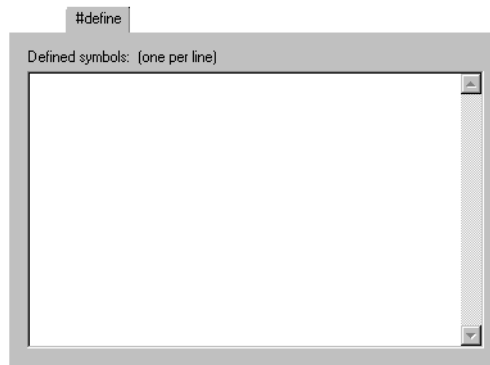


Figure 253: Linker defined symbols options

DEFINED SYMBOLS

Use the text box 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:

```
TESTVER=1
```

Note that there should be no space around the equal sign.

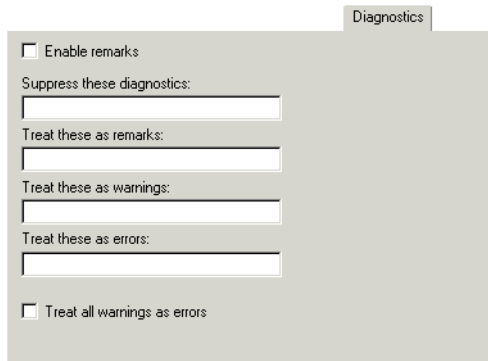
Any number of symbols can be defined in a linker configuration file. The symbol(s) defined in this manner will be located in a special module called `?ABS_ENTRY_MOD`, which is generated by the linker.

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.



The screenshot shows a dialog box titled "Diagnostics". Inside the dialog, there are several options and text input fields:

- ☐ Enable remarks
- Suppress these diagnostics:
[Empty text box]
- Treat these as remarks:
[Empty text box]
- Treat these as warnings:
[Empty text box]
- Treat these as errors:
[Empty text box]
- ☐ Treat all warnings as errors

Figure 254: 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

☒ Fill unused code memory

Fill pattern: 0xFF

Start address: 0x0

End address: 0x0

☒ Generate checksum

Size: 2 bytes

Alignment: 1

☐ Arithmetic sum

☒ CRC16 (0x11021)

☐ CRC32 (0x4C11DB7)

☐ Crc polynomial: 0x11021

Complement: As is

Initial value: 0x0

Bit order: MSB first

Figure 255: Linker checksum and fill options

For more information about filling and checksumming, see the *IAR C/C++ Development 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.

Start 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:

Algorithms	Description
Arithmetic sum	Simple arithmetic sum
CRC16	CRC16, generating polynomial 0x11021 (default)
CRC32	CRC32, generating polynomial 0x104C11DB7
Crc polynomial	CRC with a generating polynomial of the value you enter

Table 104: 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.

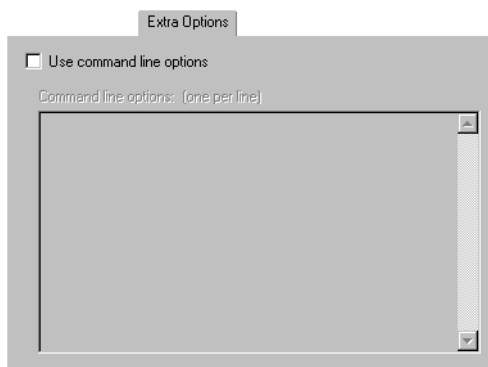


Figure 256: Extra Options page for the linker

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 91.

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.

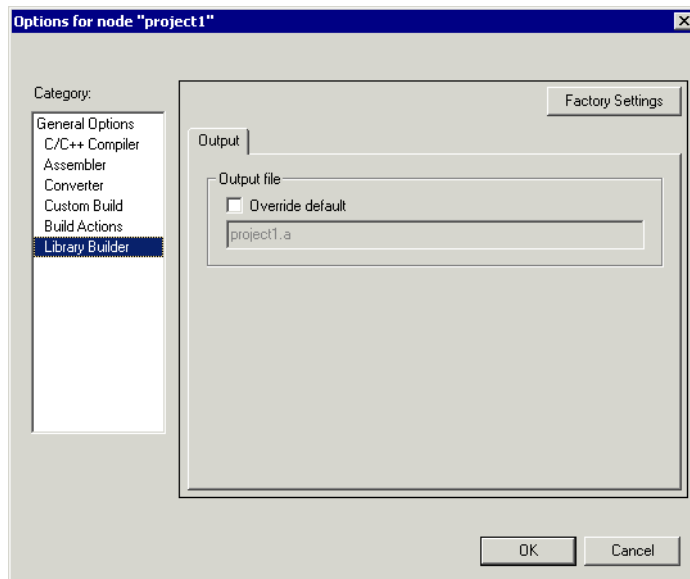


Figure 257: Library builder output options

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 91.

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.

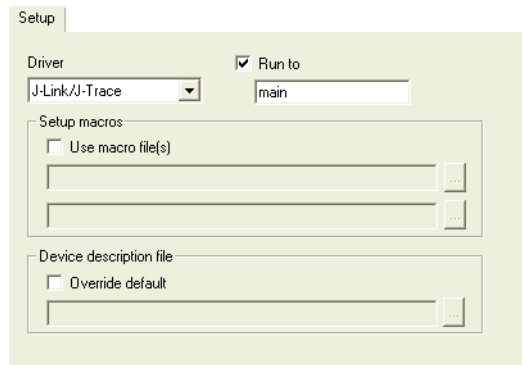


Figure 258: Generic C-SPY options

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.

The following drivers are currently available:

C-SPY driver	Filename
Simulator	armsim.dll
Angel	armangel.dll
GDB Server	armgdbserv.dll
J-Link/J-Trace	armjlink.dll
LMI FTDI	armlmiftdi.dll
Macraigor	armjtag.dll
RDI	armrdi.dll
ROM-monitor for serial port	armrom.dll
ROM-monitor for USB	armromUSB.dll

Table 105: 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 118.

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 213 in *Part 6. C-SPY hardware debugger systems*.

Extra Options

The **Extra Options** page provides you with a command line interface to C-SPY.

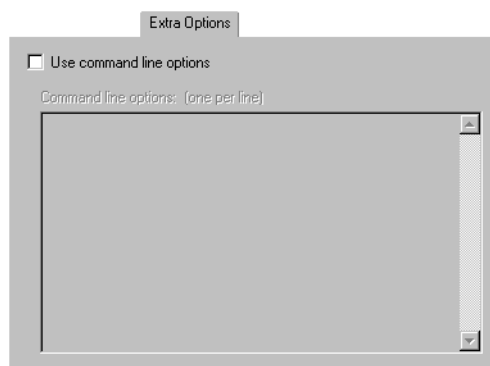


Figure 259: Extra Options page for C-SPY

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, as well as by third-party suppliers. Contact your software distributor or IAR representative, or visit the IAR Systems web site, for information about available modules.

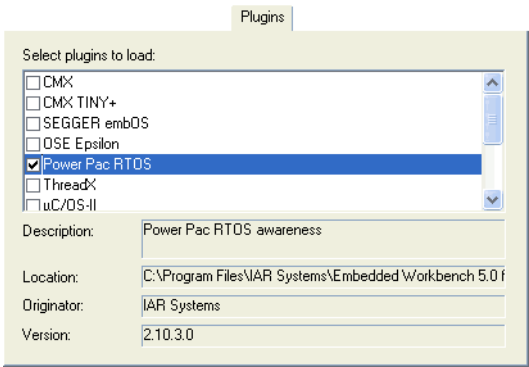


Figure 260: C-SPY plugin options

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 CMX-RTX plugin module can be found in the document `cmx_quickstart.pdf`, delivered with this product. The `uC/OS-II` plugin module is documented in the *uC/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

The IAR C-SPY Debugger can be executed in batch mode by using the C-SPY Command Line Utility—cspybat.exe—which is described in this chapter.

Using C-SPY in batch mode

C-SPY can be executed in batch mode if you use the command line utility `cspybat`, which you can find in the directory `common\bin`.

INVOCATION SYNTAX

The invocation syntax for `cspybat` is:

```
cspybat processor_DLL driver_DLL debug_file [cspybat_options]
      --backend driver_options
```

Note: In those cases where a filename is required—including the DLL files—you are recommended to give a full path to the filename.

Parameters

The parameters are:

Parameter	Description
<code>processor_DLL</code>	The processor-specific DLL file; available in <code>arm\bin</code> .
<code>driver_DLL</code>	The C-SPY driver DLL file; available in <code>arm\bin</code> .
<code>debug_file</code>	The object file that you want to debug (filename extension <code>.out</code>).
<code>cspybat_options</code>	The command line options that you want to pass to <code>cspybat</code> . Note that these options are optional. For information about each option, see <i>Descriptions of C-SPY command line options</i> , page 438.
<code>--backend</code>	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.
<code>driver_options</code>	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 <i>Descriptions of C-SPY command line options</i> , page 438.

Table 106: *cspybat* parameters

Example

The following example starts `cspybat` using the simulator driver:

```
c:\installation_dir\common\bin\cspybat
c:\installation_dir\arm\bin\armproc.dll
c:\installation_dir\arm\bin\armsim.dll c:\proj_dir\myproject.out
--plugin c:\installation_dir\arm\bin\armbat.dll --backend -d
arm7tdmi -B --cpu arm -p
c:\installation_dir\arm\bin\config\debugger\Atmel\ioat91sam7s256.
ddf
```

OUTPUT

When you run `cspybat`, the following type 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 the file can be used from the command line to start `cspybat`. The file also contains information about required modifications.

C-SPY command line options

General `cspybat` options

<code>--backend</code>	Marks the beginning of the parameters to be sent to the C-SPY driver (mandatory).
<code>--cycles</code>	Specifies the maximum number of cycles to run.

<code>--flash_loader</code>	Specifies a flash loader specification xml file.
<code>--generate_sim</code>	Generates a simple binary code file which is used by a flash loader.
<code>--macro</code>	Specifies a macro file to be used.
<code>--plugin</code>	Specifies a plugin file to be used.
<code>--silent</code>	Omits the sign-on message.

Options available for all C-SPY drivers

<code>-B</code>	Enables batch mode (mandatory).
<code>--BE8</code>	Uses the big-endian format BE8. For reference information, see the <i>IAR C/C++ Development Guide for ARM®</i> .
<code>--BE32</code>	Uses the big-endian format BE32. For reference information, see the <i>IAR C/C++ Development Guide for ARM®</i> .
<code>--cpu</code>	Specifies a processor variant. For reference information, see the <i>IAR C/C++ Development Guide for ARM®</i> .
<code>-d</code>	Specifies the C-SPY driver to be used.
<code>--device</code>	Specifies the name of the device.
<code>--drv_attach_to_program</code>	Attaches the debugger to a running application at its current location. For reference information, see <i>Attach to program</i> , page 215.
<code>--drv_catch_exceptions</code>	Makes the application stop for certain exceptions.
<code>--drv_attach_to_program</code>	Attaches the debugger to a running application at its current location. For reference information, see <i>Attach to program</i> , page 215.
<code>--drv_communication</code>	Specifies the communication link to be used.
<code>--drv_communication_log</code>	Creates a log file.
<code>--drv_default_breakpoint</code>	Sets the type of breakpoint resource to be used when setting breakpoints.
<code>--drv_reset_to_cpu_start</code>	Omits setting the PC when starting or resetting the debugger.
<code>--drv_restore_breakpoints</code>	Restores automatically any breakpoints that were destroyed during system startup.

<code>--drv_suppress_download</code>	Suppresses download of the executable image. For reference information, see <i>Suppress download</i> , page 215.
<code>--drv_vector_table_base</code>	Specifies the location of the Cortex-M reset vector and the initial stack pointer value.
<code>--drv_verify_download</code>	Verifies the target program. For reference information, see <i>Verify download</i> , page 215. Available in the drivers for Angel, GDB Server, IAR ROM-monitor, J-Link/J-Trace, LMI FTDI, Macraigor, and RDI.
<code>--endian</code>	Specifies the byte order of the generated code and data. For reference information, see the <i>IAR C/C++ Development Guide for ARM®</i> .
<code>--fpu</code>	Selects the type of floating-point unit. For reference information, see the <i>IAR C/C++ Development Guide for ARM®</i> .
<code>-p</code>	Specifies the device description file to be used.
<code>--proc_stack_stack</code>	Provides information to the C-SPY plugin module about reserved stacks.
<code>--semihosting</code>	Enables semihosted I/O.

Options available for the simulator driver

<code>--mapu</code>	Activates memory access checking.
---------------------	-----------------------------------

Options available for the C-SPY Angel debug monitor driver

<code>--rdi_heartbeat</code>	Makes C-SPY poll your target system periodically. For reference information, see <i>Send heartbeat</i> , page 217.
<code>--rdi_step_max_one</code>	Executes one instruction.

Options available for the C-SPY GDB Server driver

<code>--gdbserv_exec_command</code>	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

<code>--jlink_device_select</code>	Selects a specific device in the JTAG scan chain.
<code>--jlink_exec_command</code>	Calls the <code>__jlinkExecCommand</code> macro after target connection has been established.
<code>--jlink_initial_speed</code>	Sets the initial JTAG communication speed in kHz.
<code>--jlink_interface</code>	Specifies the communication between the J-Link debug probe and the target system.
<code>--jlink_ir_length</code>	Sets the number of IR bits before the ARM device to be debugged.
<code>--jlink_reset_strategy</code>	Selects the reset strategy to be used at debugger startup.
<code>--jlink_speed</code>	Sets the JTAG communication speed in kHz.

Options available for the C-SPY LMI FTDI driver

<code>--lmiftdi_speed</code>	Sets the JTAG communication speed in kHz.
------------------------------	---

Options available for the C-SPY Macraigor driver

<code>--mac_handler_address</code>	Specifies the location of the debug handler used by Intel XScale devices.
<code>--mac_interface</code>	Specifies the communication between the Macraigor debug probe and the target system.
<code>--mac_jtag_device</code>	Selects the device corresponding to the hardware interface.
<code>--mac_multiple_targets</code>	Specifies the device to connect to, if there are more than one device on the JTAG scan chain.
<code>--mac_reset_pulls_reset</code>	Makes C-SPY generate an initial hardware reset.
<code>--mac_set_temp_reg_buffer</code>	Provides the driver with a physical RAM address for accessing the coprocessor.
<code>--mac_speed</code>	Sets the JTAG speed between the JTAG interface and the ARM JTAG ICE port.
<code>--mac_xscale_ir7</code>	Specifies that the XScale ir7 architecture is used.

Options available for the C-SPY RDI driver

<code>--rdi_allow_hardware_reset</code>	Performs a hardware reset.
---	----------------------------

<code>--rdi_driver_dll</code>	Specifies the path to the RDI driver DLL file.
<code>--rdi_use_etm</code>	Enables C-SPY to use and display ETM trace.
<code>--rdi_step_max_one</code>	Executes one instruction.

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	<code>-B</code>
Applicability	All C-SPY drivers.
Description	Use this option to enable batch mode.

--backend

Syntax	<code>--backend {<i>driver options</i>}</code>
Parameters	<i>driver options</i> Any option available to the C-SPY driver you are using.
Applicability	Sent to <code>cspybat</code> (mandatory).
Description	Use this option to send options to the C-SPY driver. All options that follow <code>--backend</code> will be passed to the C-SPY driver, and will not be processed by <code>cspybat</code> itself.

--cycles

Syntax	<code>--cycles <i>cycles</i></code>	
Parameters	<i>cycles</i>	The number of cycles to run.
Applicability	Sent to <i>cspybat</i> .	
Description	Use this option to specify the maximum number of cycles to run. If the target program executes longer than the number of cycles specified, the target program will be aborted. Using this option requires that the C-SPY driver you are using supports a cycle counter, and that it can be sampled while executing.	

-d

Syntax	<code>-d {angel gdbserv generic jlink jtag lmiftdi rdi rom sim}</code>	
Parameters	angel	Specifies the Angel debug monitor driver.
	gdbserv	Specifies the GDB Server driver.
	generic	Specifies third-party driver.
	jlink	Specifies the J-Link/J-Trace driver.
	jtag	Specifies the Macraigor driver.
	lmiftdi	Specifies the LMI FTDI driver.
	rdi	Specifies the RDI driver.
	rom	Specifies the IAR C-SPY ROM-monitor driver.
	sim	Specifies the simulator driver.
Applicability	All C-SPY drivers.	
Description	Use this option to specify the C-SPY driver to be used.	

--device

Syntax	<code>--device=device_name</code>	
Parameters	<i>device_name</i>	The name of the device.
Applicability	All C-SPY drivers.	
Description	Use this option to specify the name of the device, for example, ADuC7030, AT91SAM7S256, LPC2378, STR912FM44, or TMS470R1B1M.	



To set related option, choose:
Project>Options>General Options>Target>Device

--drv_catch_exceptions

Syntax	<code>--drv_catch_exceptions=value</code>	
Parameters	<i>value</i>	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
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.	

Catch exceptions, page 235.



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

--drv_communication

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*

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 GDB Server driver:

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.

Where *connection* is one of these for the C-SPY Macraigor driver:

Via the parallel port to LPT1-LPT3
 Wiggler, Raven, or
 mpDemon

For Wiggler, Raven and *port:baud*
 mpDemon *port* = COM1-COM4
 baud = 9600, 19200, 38400, 57600, or 115200 (default 9600 baud)

For mpDemon 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.


Via USB to usbDemon and USB ports = USB0-USB3
 usb2Demon

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	<code>--drv_communication_log=<i>filename</i></code>	
Parameters	<i>filename</i>	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	<code>--drv_default_breakpoint={0 1 2}</code>	
Parameters	0	Auto (default)
	1	Hardware
	2	Software
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	<i>Default breakpoint type</i> , page 244.



Project>Options>Debugger>Driver>Breakpoints>Default breakpoint type

--drv_reset_to_cpu_start


Syntax	<code>--drv_reset_to_cpu_start</code>
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	<code>--drv_restore_breakpoints=location</code>
Parameters	<i>location</i> 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	<i>Restore software breakpoints at</i> , page 244.
	 Project>Options>Debugger>Driver>Breakpoints>Restore software breakpoints at

--drv_vector_table_base

Syntax	<code>--drv_vector_table_base=expression</code>
Parameters	<div><i>expression</i><div>A label or an address</div></div>
Applicability	<div>The C-SPY GDB Server driver</div> <div>The C-SPY J-Link/J-Trace driver</div> <div>The C-SPY LMI FTDI driver</div> <div>The C-SPY Macraigor driver</div> <div>The C-SPY RDI driver</div> <div>The C-SPY Simulator driver.</div>
Description	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 <code>__vector_table</code> 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	<code>--flash_loader filename</code>
Parameters	<div><i>filename</i><div>The flash loader specification xml file.</div></div>
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 <i>IAR Embedded Workbench flash loader User Guide</i> .

--gdbserv_exec_command

Syntax	--gdbserv_exec_command= " <i>string</i> "
Parameters	<div>"<i>string</i>"<div>String or command sent to the GDB Server; see its documentation for more information.</div></div>
Applicability	The C-SPY GDB Server driver.
Description	Use this option to send strings or commands to the GDB Server.




Project>Options>Debugger>Extra Options

--generate_sim


Syntax	--generate_sim
Applicability	Sent to cspybat.
Description	As part of the flash loading process, a simple code file (filename extension <code>sim</code>) is required. Use this option to automatically generate such a file from the debug file. If the debug file is an ELF file, you should use this option.

--jlink_device_select


Syntax	--jlink_device_select= <i>tap_number</i>
Parameters	<div><i>tap_number</i><div>The TAP position of the device you want to connect to.</div></div>
Applicability	The C-SPY J-Link/J-Trace driver.

Description	If there are more than one device on the JTAG scan chain, use this option to select a specific device,
See also	<i>JTAG scan chain</i> , page 225.
	 Project>Options>Debugger>J-Link/J-Trace>Connection>JTAG scan chain>TAP number

--jlink_exec_command

Syntax	<code>--jlink_exec_command=cmdstr1; cmdstr2; cmdstr3 ...</code>		
Parameters	<table><tr><td><i>cmdstrn</i></td><td>J-Link/J-Trace command string.</td></tr></table>	<i>cmdstrn</i>	J-Link/J-Trace command string.
<i>cmdstrn</i>	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 <code>__jlinkExecCommand</code> macro with one or several command strings, after target connection has been established.		
See also	<i>__jlinkExecCommand</i> , page 473.		
	 Project>Options>Debugger>Extra Options		

--jlink_initial_speed

Syntax	<code>--jlink_initial_speed=speed</code>		
Parameters	<table><tr><td><i>speed</i></td><td>The initial communication speed in kHz. If no speed is specified, 32 kHz will be used as the initial speed.</td></tr></table>	<i>speed</i>	The initial communication speed in kHz. If no speed is specified, 32 kHz will be used as the initial speed.
<i>speed</i>	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	<i>JTAG speed</i> , page 223.		
	 Project>Options>Debugger>J-Link/J-Trace>Setup>JTAG speed>Fixed		

--jlink_interface

Syntax	<code>--jlink_interface={JTAG SWD}</code>	
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 between the J-Link debug probe and the target system.	
See also	<i>Interface</i> , page 224.	



Project>Options>Debugger>J-Link/J-Trace>Connection>Interface

--jlink_ir_length

Syntax	<code>--jlink_ir_length=length</code>	
Parameters	<i>length</i>	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.	
See also	<i>JTAG scan chain</i> , page 225.	



Project>Options>Debugger>J-Link/J-Trace>Connection>JTAG scan chain>Preceding bits

--jlink_reset_strategy

Syntax	<code>--jlink_reset_strategy={delay, 0 strategy}</code>	
Parameters	<i>delay</i>	0–10000. The delay parameter is only used with strategy 0.

	<i>strategy</i>	Strategies for ARM 7/9/11: 0 = Hardware, halt after reset (<i>delay</i> 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 DBGCRQ 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
--	-----------------	---

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	<i>Reset</i> , page 221.



Project>Options>Debugger>J-Link/J-Trace>Setup>Reset

--jlink_speed

Syntax	<code>--jlink_speed={fixed auto adaptive}</code>						
Parameters	<table><tr><td><i>fixed</i></td><td>1-12000</td></tr><tr><td><i>auto</i></td><td>The highest possible frequency for reliable operation (default)</td></tr><tr><td><i>adaptive</i></td><td>For ARM devices that have the RTCK JTAG signal available</td></tr></table>	<i>fixed</i>	1-12000	<i>auto</i>	The highest possible frequency for reliable operation (default)	<i>adaptive</i>	For ARM devices that have the RTCK JTAG signal available
<i>fixed</i>	1-12000						
<i>auto</i>	The highest possible frequency for reliable operation (default)						
<i>adaptive</i>	For ARM devices that have the RTCK JTAG signal available						
Applicability	The C-SPY J-Link/J-Trace driver.						
Description	Use this option to set the JTAG communication speed in kHz.						
See also	<i>JTAG speed</i> , page 223.						



Project>Options>Debugger>J-Link/J-Trace>Setup>JTAG speed

--lmiftdi_speed

Syntax	<code>--lmiftdi_speed=frequency</code>	
Parameters	<i>frequency</i>	The frequency in kHz.
Applicability	The C-SPY LMI FTDI driver.	
Description	Use this option to set the JTAG communication speed in kHz.	
See also	<i>JTAG speed</i> , page 229.	



Project>Options>Debugger>LMI FTDI>Setup>JTAG speed

--mac_handler_address

Syntax	<code>--mac_handler_address=address</code>	
Parameters	<i>address</i>	The start address of the memory area for the debug handler.
Applicability	The C-SPY Macraigor driver	
Description	Use this option to specify the location—the memory address—of the debug handler used by Intel XScale devices.	
See also	<i>Debug handler address</i> , page 232.	



Project>Options>Debugger>Macraigor>Debug handler address

--mac_interface

Syntax	<code>--mac_interface={JTAG SWO}</code>	
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 between the Macraigor debug probe and the target system.



Project>Options>Debugger>Macraigor>Interface

--mac_jtag_device

Syntax	<code>--mac_jtag_device=device</code>	
Parameters	<i>device</i>	The device corresponding to the hardware interface that is used. Choose between Macraigor Raven, Wiggler, mpDemon, usbdaemon, 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	<i>OCD interface device</i> , page 231.	



Project>Options>Debugger>Macraigor>OCD interface device

--mac_multiple_targets

Syntax	<code>--mac_multiple_targets=<0>@dev0, dev1, dev2, dev3, . . .</code>	
Parameters	0	The TAP number of the device to connect to, where 0 connects to the first device, 1 to the second, and so on.
	<i>dev0-devn</i>	The nearest TDO pin on the Macraigor JTAG interface.
Applicability	The C-SPY Macraigor driver.	
Description	If there are 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	<code>--mac_multiple_targets=0@ARM7TDMI, ARM7TDMI</code>	

See also *JTAG scan chain with multiple targets*, page 232.



Project>Options>Debugger>Macraigor>JTAG scan chain with multiple targets

--mac_reset_pulls_reset

Syntax	<code>--mac_reset_pulls_reset=<i>time</i></code>
Parameters	<i>time</i> 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	<i>Hardware reset</i> , page 232.



Project>Options>Debugger>Macraigor>Hardware reset

--mac_set_temp_reg_buffer

Syntax	<code>--mac_set_temp_reg_buffer=<i>address</i></code>
Parameters	<i>address</i> 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	<code>--mac_speed={1-8}</code>	
Parameters	<i>1-8</i>	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	<i>JTAG speed</i> , page 231.	



Project>Options>Debugger>Macraigor>JTAG speed

--mac_xscale_ir7

Syntax	<code>--mac_xscale_ir7</code>	
Applicability	The C-SPY Macraigor driver.	
Description	Use this option to specify that the XScale ir7 core is used, instead of XScale ir5. Note that this option is mandatory when using the XScale ir7 core. These XScale cores are supported by the C-SPY Macraigor driver: Intel XScale Core 1 (5-bit instruction register) Intel XScale Core 2 (7-bit instruction register)	



To set this option, use **Project>Options>Debugger>Extra Options**.

--macro

Syntax	<code>--macro filename</code>	
Parameters	<i>filename</i>	The C-SPY macro file to be used (filename extension <i>mac</i>).
Applicability	Sent to <i>cspybat</i> .	

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	<i>The macro file</i> , page 150

--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 <code>stdout</code> and the execution will stop.
See also	<i>Memory access checking</i> , page 176.



To set related options, choose:
Simulator>Memory Access Setup

-p

Syntax	-p= <i>filename</i>
Parameters	<i>filename</i> 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	<i>Device description file</i> , page 118

--plugin

Syntax	--plugin <i>filename</i>
Parameters	<i>filename</i> The plugin file to be used (filename extension <code>dll</code>).

Applicability	Sent to <code>cspybat</code> .
Description	<p>Certain C/C++ standard library functions, for example <code>printf</code>, can be supported by C-SPY instead of by real hardware devices (for example, the C-SPY Terminal I/O window). To enable such support in <code>cspybat</code>, a dedicated plugin module called <code>armbat.dll</code> located in the <code>arm\bin</code> directory must be used.</p> <p>Use this option to include this plugin during the debug session. This option can be used more than once on the command line.</p> <p>Note: This option can be used for including also other plugin modules, but in that case the module must be able to work with <code>cspybat</code> specifically. This means that the C-SPY plugin modules located in the <code>arm\plugins</code> directory cannot normally be used with <code>cspybat</code>.</p>
See also	<i>The macro file</i> , page 150.

--proc_stack_stack

Syntax	<code>--proc_stack_stack=startaddress,endaddress</code> where <i>stack</i> is one of <code>usr</code> , <code>svc</code> , <code>fiq</code> , <code>und</code> , or <code>abt</code> for ARM7/9/11 and XScale and where <i>stack</i> is one of <code>main</code> , or <code>proc</code> for Cortex-M				
Parameters	<table><tr><td><i>startaddress</i></td><td>The start address of the stack, specified either as a value or as an expression.</td></tr><tr><td><i>endaddress</i></td><td>The end address of the stack, specified either as a value or as an expression.</td></tr></table>	<i>startaddress</i>	The start address of the stack, specified either as a value or as an expression.	<i>endaddress</i>	The end address of the stack, specified either as a value or as an expression.
<i>startaddress</i>	The start address of the stack, specified either as a value or as an expression.				
<i>endaddress</i>	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 <code>cspybat</code> .				
Description	<p>Use this option to provide information about to the C-SPY 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.</p> <p><i>Stack window</i>, page 369.</p>				



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	<i>Allow hardware reset</i> , page 234.



Project>Options>Debugger>RDI>Allow hardware reset

--rdi_driver_dll

Syntax	--rdi_driver_dll <i>filename</i>
Parameters	<div><div><i>filename</i></div><div>The file or path to the RDI driver DLL file.</div></div>
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. <i>Manufacturer RDI driver</i> , page 234.



Project>Options>Debugger>RDI>Manufacturer RDI driver


--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. <i>ETM trace</i> , page 235.




Project>Options>Debugger>RDI>ETM trace

--rdi_step_max_one

Syntax	<code>--rdi_step_max_one</code>	
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	<code>--semihosting={none iar_breakpoint}</code>							
Parameters	<table><tr><td>No parameter</td><td>Use standard semihosting.</td></tr><tr><td>none</td><td>Does not use semihosted I/O.</td></tr><tr><td>iar_breakpoint</td><td>Uses the IAR proprietary semihosting variant.</td></tr></table>		No parameter	Use standard semihosting.	none	Does not use semihosted I/O.	iar_breakpoint	Uses the IAR proprietary semihosting variant.
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	<p>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.</p> <p>To make semihosting work, your application must be linked with a semihosting library.</p>							
See also	<p>The <i>IAR C/C++ Development Guide for ARM®</i> for more information about linking with semihosting.</p>							
	<div> Project>Options>General Options>Library Configuration</div>							

--silent

Syntax	--silent
Applicability	Sent to <i>cspybat</i> .
Description	Use this option to omit the sign-on message.

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 the following 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 467.

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 127.

The syntax for defining one or more macro variables is:

```
__var nameList;
```

where *nameList* is a list of C-SPY variable names separated by commas.

A macro variable defined outside a macro body has global scope, and it exists throughout the whole debugging session. A macro variable defined within a macro body is created when its definition is executed and destroyed on return from the macro.

By default, macro variables are treated as signed integers and initialized to 0. When a C-SPY variable is assigned a value in an expression, it also acquires the type of that expression. For example:

Expression	What it means
myvar = 3.5;	myvar is now type float, value 3.5.
myvar = (int*)i; myvar = (int*)i;	myvar is now type pointer to int, and the value is the same as i.

Table 107: Examples of C-SPY macro variables

In case of a name conflict between a C symbol and a C-SPY macro variable, C-SPY macro variables have a higher precedence than C variables. Note that macro variables are allocated on the debugger host and do not affect your application.

Macro strings

In addition to C types, macro variables can hold values of *macro strings*. Note that macro strings differ from C language strings.

When you write a string literal, such as "Hello!", in a C-SPY expression, the value is a macro string. It is not a C-style character pointer char*, because char* must point to a sequence of characters in target memory and C-SPY cannot expect any string literal to actually exist in target memory.

You can manipulate a macro string using a few built-in macro functions, for example __strFind or __subString. The result can be a new macro string. You can concatenate macro strings using the + operator, for example str + "tail". You can also access individual characters using subscription, for example str[3]. You can get the length of a string using sizeof(str). Note that a macro string is not NULL-terminated.

The macro function `__toString` is used for converting from a NULL-terminated C string in your application (`char*` or `char[]`) to a macro string. For example, assume the following definition of a C string in your application:

```
char const *cstr = "Hello";
```

Then examine the following examples:

```
__var str;           /* A macro variable */
str = cstr           /* str is now just a pointer to char */
sizeof str          /* same as sizeof (char*), typically 2 or 4 */
str = __toString(cstr,512) /* str is now a macro string */
sizeof str          /* 5, the length of the string */
str[1]              /* 101, the ASCII code for 'e' */
str += " World!"    /* str is now "Hello World!" */
```

See also *Formatted output*, page 462.

MACRO STATEMENTS

Statements are expected to behave in the same way as the corresponding C statements would do. The following C-SPY macro statements are accepted:

Expressions

```
expression;
```

For detailed information about C-SPY expressions, see *C-SPY expressions*, page 127.

Conditional statements

```
if (expression)
    statement
```

```
if (expression)
    statement
else
    statement
```

Loop statements

```
for (init_expression; cond_expression; update_expression)
    statement
```

```
while (expression)
    statement
```

```
do
    statement
```

```
while (expression);
```

Return statements

```
return;
```

```
return expression;
```

If the return value is not explicitly set, signed int 0 is returned by default.

Blocks

Statements can be grouped in blocks.

```
{
    statement1
    statement2
    .
    .
    .
    statementN
}
```

FORMATTED OUTPUT

C-SPY provides different methods for producing formatted output:

<code>__message <i>argList</i>;</code>	Prints the output to the Debug Log window.
<code>__fmessage <i>file</i>, <i>argList</i>;</code>	Prints the output to the designated file.
<code>__smessage <i>argList</i>;</code>	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 478.

Examples

Use the `__message` statement, as in the following example:

```
var1 = 42;
var2 = 37;
__message "This line prints the values ", var1, " and ", var2,
" in the Log window.";
```

This should produce the following 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

It is possible to override the default display format of a scalar argument (number or pointer) in *argList* by suffixing 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

The following table summarizes the available setup macro functions:

Macro	Description
<code>execUserPreload</code>	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.
<code>execUserFlashInit</code>	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.
<code>execUserSetup</code>	Called once after the target application is downloaded. Implement this macro to set up the memory map, breakpoints, interrupts, register macro files, etc.
<code>execUserFlashReset</code>	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.
<code>execUserReset</code>	Called each time the reset command is issued. Implement this macro to set up and restore data.
<code>execUserExit</code>	Called once when the debug session ends. Implement this macro to save status data etc.
<code>execUserFlashExit</code>	Called once when the debug session ends. Implement this macro to save status data etc. This macro is useful for flash loader functionality.

Table 108: 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 59.

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

The following table summarizes the pre-defined system macros:

Macro	Description
<code>__cancelAllInterrupts</code>	Cancels all ordered interrupts
<code>__cancelInterrupt</code>	Cancels an interrupt
<code>__clearBreak</code>	Clears a breakpoint
<code>__closeFile</code>	Closes a file that was opened by <code>__openFile</code>
<code>__disableInterrupts</code>	Disables generation of interrupts
<code>__driverType</code>	Verifies the driver type
<code>__emulatorSpeed</code>	Sets the emulator clock frequency
<code>__emulatorStatusCheckOnRead</code>	Enables or disables the verification of the CPSR register after each read operation
<code>__enableInterrupts</code>	Enables generation of interrupts
<code>__evaluate</code>	Interprets the input string as an expression and evaluates it.
<code>__gdbserver_exec_command</code>	Send strings or commands to the GDB Server.
<code>__hwReset</code>	Performs a hardware reset and halt of the target CPU
<code>__hwResetWithStrategy</code>	Performs a hardware reset and halt with delay of the target CPU
<code>__jlinkExecCommand</code>	Sends a low-level command to the J-Link/J-Trace driver.
<code>__jtagCommand</code>	Sends a low-level command to the JTAG instruction register
<code>__jtagCP15IsPresent</code>	Checks if coprocessor CP15 is available
<code>__jtagCP15ReadReg</code>	Returns the coprocessor CP15 register value
<code>__jtagCP15WriteReg</code>	Writes to the coprocessor CP15 register
<code>__jtagData</code>	Sends a low-level data value to the JTAG data register
<code>__jtagRawRead</code>	Returns the read data from the JTAG interface
<code>__jtagRawSync</code>	Writes accumulated data to the JTAG interface
<code>__jtagRawWrite</code>	Accumulates data to be transferred to the JTAG
<code>__jtagResetTRST</code>	Resets the ARM TAP controller via the TRST JTAG signal
<code>__openFile</code>	Opens a file for I/O operations
<code>__orderInterrupt</code>	Generates an interrupt

Table 109: Summary of system macros

Macro	Description
<code>__popSimulatorInterruptExecutingStack</code>	Informs the interrupt simulation system that an interrupt handler has finished executing
<code>__readFile</code>	Reads from the specified file
<code>__readFileByte</code>	Reads one byte from the specified file
<code>__readMemory8,</code> <code>__readMemoryByte</code>	Reads one byte from the specified memory location
<code>__readMemory16</code>	Reads two bytes from the specified memory location
<code>__readMemory32</code>	Reads four bytes from the specified memory location
<code>__registerMacroFile</code>	Registers macros from the specified file
<code>__resetFile</code>	Rewinds a file opened by <code>__openFile</code>
<code>__restoreSoftwareBreakpoint</code>	Restores any breakpoints that were destroyed during system startup.
<code>__setCodeBreak</code>	Sets a code breakpoint
<code>__setDataBreak</code>	Sets a data breakpoint
<code>__setSimBreak</code>	Sets a simulation breakpoint
<code>__sleep</code>	Causes the debugger to sleep a specified amount of time.
<code>__sourcePosition</code>	Returns the file name and source location if the current execution location corresponds to a source location
<code>__strFind</code>	Searches a given string for the occurrence of another string
<code>__subString</code>	Extracts a substring from another string
<code>__toLowerCase</code>	Returns a copy of the parameter string where all the characters have been converted to lower case
<code>__toString</code>	Prints strings
<code>__toUpperCase</code>	Returns a copy of the parameter string where all the characters have been converted to upper case
<code>__writeFile</code>	Writes to the specified file
<code>__writeFileByte</code>	Writes one byte to the specified file
<code>__writeMemory8,</code> <code>__writeMemoryByte,</code> <code>__writeMemory8,</code> <code>__writeMemoryByte</code>	Writes one byte to the specified memory location

Table 109: Summary of system macros (Continued)

Macro	Description
__writeMemory16	Writes a two-byte word to the specified memory location
__writeMemory32	Writes a four-byte word to the specified memory location

Table 109: Summary of system macros (Continued)

Description of C-SPY system macros

This section gives reference information about each of the C-SPY system macros.

__cancelAllInterrupts

Syntax	<code>__cancelAllInterrupts()</code>
Return value	<code>int 0</code>
Description	Cancels all ordered interrupts.
Applicability	This system macro is only available in the C-SPY Simulator.

__cancelInterrupt

Syntax	<code>__cancelInterrupt(<i>interrupt_id</i>)</code>
Parameter	<div><div><code><i>interrupt_id</i></code></div><div>The value returned by the corresponding <code>__orderInterrupt</code> macro call (unsigned long)</div></div>

Return value	<table><tr><th>Result</th><th>Value</th></tr><tr><td>Successful</td><td><code>int 0</code></td></tr><tr><td>Unsuccessful</td><td>Non-zero error number</td></tr></table>	Result	Value	Successful	<code>int 0</code>	Unsuccessful	Non-zero error number
Result	Value						
Successful	<code>int 0</code>						
Unsuccessful	Non-zero error number						

Table 110: `__cancelInterrupt` return values

Description	Cancels the specified interrupt.
Applicability	This system macro is only available in the C-SPY Simulator.

__clearBreak

Syntax	<code>__clearBreak(<i>break_id</i>)</code>	
Parameter	<i>break_id</i>	The value returned by any of the set breakpoint macros
Return value	<code>int 0</code>	
Description	Clears a user-defined breakpoint.	
See also	<i>Defining breakpoints</i> , page 135.	

__closeFile

Syntax	<code>__closeFile(<i>filehandle</i>)</code>	
Parameter	<i>filehandle</i>	The macro variable used as filehandle by the <code>__openFile</code> macro
Return value	<code>int 0</code>	
Description	Closes a file previously opened by <code>__openFile</code> .	

__disableInterrupts

Syntax	<code>__disableInterrupts()</code>							
Return value	<table><tr><th>Result</th><th>Value</th></tr><tr><td>Successful</td><td><code>int 0</code></td></tr><tr><td>Unsuccessful</td><td>Non-zero error number</td></tr></table>		Result	Value	Successful	<code>int 0</code>	Unsuccessful	Non-zero error number
Result	Value							
Successful	<code>int 0</code>							
Unsuccessful	Non-zero error number							

Table 111: __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; one of the following:
"sim" corresponds to the simulator driver
"rom" corresponds to the ROM-monitor driver
"jtag" corresponds to the Macraigor driver
"rdi" corresponds to the RDI driver
"jlink" corresponds to the J-Link/J-Trace driver
"lmiftdi" corresponds to the LMI FTDI driver
"angel" corresponds to the Angel driver
"generic" corresponds to third-party drivers

Return value

Result	Value
Successful	1
Unsuccessful	0

Table 112: __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 a simulator is the current driver, the value 1 is returned. Otherwise 0 is returned.

__emulatorSpeed

Syntax	__emulatorSpeed(<i>speed</i>)					
Parameter	<i>speed</i>	The emulator speed in Hz. Use 0 (zero) to make the speed detected automatically. Use -1 for adaptive speed (only for emulators supporting adaptive speed).				
Return value	<table><tr><th>Result</th><th>Value</th></tr><tr><td>Successful</td><td>The previous speed, or 0 (zero) if unknown</td></tr></table>		Result	Value	Successful	The previous speed, or 0 (zero) if unknown
Result	Value					
Successful	The previous speed, or 0 (zero) if unknown					

Table 113: __emulatorSpeed return values

	Result	Value
	Unsuccessful; the speed is not supported by –1 the emulator	
	Table 113: __emulatorSpeed return values	
Description	Sets the emulator clock frequency. For JTAG interfaces, this is the JTAG clock frequency as seen on the TCK signal.	
Example	<pre>__emulatorSpeed(0)</pre> <p>Sets the emulator speed to be detected automatically.</p>	
Applicability	This system macro is available for the J-Link/J-Trace JTAG interface.	

__emulatorStatusCheckOnRead

Syntax	<pre>__emulatorStatusCheckOnRead(status)</pre>	
Parameter	<i>status</i>	Use 0 to enable checks (default). Use 1 to disable checks.
Return value	<pre>int 0</pre>	
Description	<p>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.</p> <p>Note: Enabling this verification can cause problems with some CPUs, for example if invalid CPSR values are returned. However, if this verification is disabled (<code>SetCheckModeAfterRead = 0</code>), the success of read operations cannot be verified and possible data aborts are not detected.</p>	
Example	<pre>__emulatorStatusCheckOnRead(1)</pre> <p>Disables the checks for data aborts on memory reads.</p>	
Applicability	This system macro is available for the J-Link/J-Trace JTAG interface.	

__enableInterrupts

Syntax `__enableInterrupts()`

Return value

Result	Value
Successful	int 0
Unsuccessful	Non-zero error number

Table 114: __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

<i>string</i>	Expression string
<i>valuePtr</i>	Pointer to a macro variable storing the result

Return value

Result	Value
Successful	int 0
Unsuccessful	int 1

Table 115: __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

The following 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	<code>__gdbserver_exec_command("string")</code>	
Parameter	<code>"string"</code>	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	<code>__hwReset(halt_delay)</code>	
Parameter	<code>halt_delay</code>	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><tr><th>Result</th><th>Value</th></tr><tr><td>Successful. The actual delay value implemented by the emulator</td><td>≥ 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></table>	Result	Value	Successful. The actual delay value implemented by the emulator	≥ 0	Successful. The delay feature is not supported by the emulator	-1	Unsuccessful. Hardware reset is not supported by the emulator	-2
Result	Value								
Successful. The actual delay value implemented by the emulator	≥ 0								
Successful. The delay feature is not supported by the emulator	-1								
Unsuccessful. Hardware reset is not supported by the emulator	-2								

Table 116: __hwReset return values

Description	Performs a hardware reset and halt of the target CPU.	
Example	<code>__hwReset(0)</code>	Resets the CPU and immediately halts it.
Applicability	This system macro is available for all JTAG interfaces.	

__hwResetWithStrategy

Syntax	__hwResetWithStrategy(<i>halt_delay</i> , <i>strategy</i>)	
Parameter	<i>halt_delay</i>	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; only when <i>strategy</i> is set to 0.
	<i>strategy</i>	The reset strategy used for halting the core. Use 0 (zero) to halt after reset. Use 1 to halt with breakpoint at address 0x0. Use 2 for software reset (for Analog devices).

Return value	Result	Value
	Successful. The actual delay value implemented by the emulator	>=0
	Successful. The delay feature is not supported by the emulator	-1
	Unsuccessful. Hardware reset is not supported by the emulator	-2
	Unsuccessful. The reset strategy is not supported by the emulator	-3

Table 117: __hwReset return values

Description	Performs a hardware reset and a halt with delay of the target CPU.
Example	<pre>__hwResetWithStrategy(0,1)</pre> <p>Resets the CPU and halts it using a breakpoint at memory address zero.</p>
Applicability	This system macro is available for the J-Link/J-Trace JTAG interface.

__jlinkExecCommand

Syntax	__jlinkExecCommand(<i>cmdstr</i>)	
Parameter	<i>cmdstr</i>	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 <i>J-Link / J-Trace User's Guide</i> .	
Applicability	This system macro is available for the J-Link/J-Trace JTAG interface.	

__jtagCommand

Syntax	<code>__jtagCommand(ir)</code>	
Parameter	2	SCAN_N
	4	RESTART
	12	INTEST
	14	IDCODE
	15	BYPASS
Return value	<code>int 0</code>	
Description	Sends a low-level command to the JTAG instruction register <code>IR</code> .	
Example	<code>__jtagCommand(14);</code> <code>Id = __jtagData(0,32);</code> Returns the JTAG ID of the ARM target device.	
Applicability	This system macro is available for the J-Link/J-Trace JTAG interface.	

__jtagCP15IsPresent

Syntax	<code>__jtagCP15IsPresent()</code>	
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	<code>__jtagCP15ReadReg(CRn, CRm, op1, op2)</code>	
Parameter	The parameters—registers and operands—of the MRC instruction. For details, see the <i>ARM Architecture Reference Manual</i> . Note that <code>op1</code> 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	<code>__jtagCP15WriteReg(CRn, CRm, op1, op2, value)</code>
Parameter	The parameters—registers and operands—of the MCR instruction. For details, see the <i>ARM Architecture Reference Manual</i> . Note that <code>op1</code> should always be 0. <i>value</i> 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	<code>__jtagData(dr, bits)</code>	
Parameter	<i>dr</i>	32-bit data register value
	<i>bits</i>	Number of valid bits in <i>dr</i> , both for the macro parameter and the return value; starting with the least significant bit (1 . . . 32)
Return value	Returns the result of the operation; the number of bits in the result is given by the <i>bits</i> parameter.	
Description	Sends a low-level data value to the JTAG data register DR. The bit shifted out of DR is returned.	
Example	<pre>__jtagCommand(14); Id = __jtagData(0,32);</pre> 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	<code>__jtagRawRead(<i>bitpos</i>, <i>numbits</i>)</code>	
Parameter	<i>bitpos</i>	The start bit position in the returned JTAG bits to return data from
	<i>numbits</i>	The number of bits to read. The maximum value is 32.
Description	Returns the data read from the JTAG 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	<p>The following piece of pseudocode illustrates how the data is written to the JTAG (on the TMS and TDI pins) and read (from TDO):</p> <pre> __var Id; __var BitPos; /***** * ReadId() */ ReadId() { __message "Reading JTAG Id\n"; __jtagRawWrite(0, 0x1f, 6); /* Goto IDLE via RESET state */ __jtagRawWrite(0, 0x1, 3); /* Enter DR scan chain */ 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"; } </pre>	
Applicability	This system macro is available for the J-Link/J-Trace JTAG interface.	

__jtagRawSync

Syntax	<code>__jtagRawSync()</code>
Return value	<code>int 0</code>
Description	Sends arbitrary data to the JTAG interface. All accumulated bits using <code>__jtagRawWrite</code> 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.

__jtagRawWrite

Syntax `__jtagRawWrite(tdi, tms, numbits)`

Parameter

<i>tdi</i>	The data output to the TDI pin. This data is sent with the least significant bit first.
<i>tms</i>	The data output to the TMS pin, This data is sent with the least significant bit first.
<i>numbits</i>	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 */
```

Returns the JTAG ID of the ARM target device.

Applicability This system macro is available for the J-Link/J-Trace JTAG interface.

__jtagResetTRST

Syntax __jtagResetTRST()

Result	Value
Successful	int 0
Unsuccessful	Non-zero error number

Table 118: __openFile 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.

__openFile

Syntax __openFile(file, access)

Parameters	<div><div>file</div><div>The filename as a string</div></div> <div><div>access</div><div>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</div></div>
------------	--

Result	Value
Successful	The file handle

Table 119: __openFile return values

Result	Value
Unsuccessful	An invalid file handle, which tests as False

Table 119: `__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 (*.pew or *.prj) is located. The argument to <code>__openFile</code> can specify a location relative to this directory. In addition, you can use argument variables such as <code>\$PROJ_DIR\$</code> and <code>\$TOOLKIT_DIR\$</code> in the path argument.
Example	<pre>__var filehandle; /* The macro variable to contain */ /* the file handle */ filehandle = __openFile("Debug\\Exe\\test.tst", "r"); if (filehandle) { /* successful opening */ }</pre>
See also	<i>Argument variables summary</i> , page 306.

__orderInterrupt

Syntax	<code>__orderInterrupt(<i>specification</i>, <i>first_activation</i>, <i>repeat_interval</i>, <i>variance</i>, <i>infinite_hold_time</i>, <i>hold_time</i>, <i>probability</i>)</code>
Parameters	<div> <div><i>specification</i></div> <div>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.</div> </div> <div> <div><i>first_activation</i></div> <div>The first activation time in cycles (integer)</div> </div> <div> <div><i>repeat_interval</i></div> <div>The periodicity in cycles (integer)</div> </div> <div> <div><i>variance</i></div> <div>The timing variation range in percent (integer between 0 and 100)</div> </div> <div> <div><i>infinite_hold_time</i></div> <div>1 if infinite, otherwise 0.</div> </div> <div> <div><i>hold_time</i></div> <div>The hold time (integer)</div> </div> <div> <div><i>probability</i></div> <div>The probability in percent (integer between 0 and 100)</div> </div>
Return value	<p>The macro returns an interrupt identifier (unsigned long).</p> <p>If the syntax of <i>specification</i> is incorrect, it returns -1.</p>

Description	Generates an interrupt.
Applicability	This system macro is only available in the C-SPY Simulator.
Example	<p>The following example generates a repeating interrupt using an infinite hold time first activated after 4000 cycles:</p> <pre>__orderInterrupt("IRQ", 4000, 2000, 0, 1, 0, 100);</pre>

__popSimulatorInterruptExecutingStack

Syntax	<code>__popSimulatorInterruptExecutingStack(void)</code>
Return value	This macro has no return value.
Description	<p>Notifies 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.</p> <p>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.</p>
Applicability	This system macro is only available in the C-SPY Simulator.

__readFile

Syntax	<code>__readFile(<i>file</i>, <i>valuePtr</i>)</code>						
Parameters	<table><tr><td><i>file</i></td><td>A file handle</td></tr><tr><td><i>valuePtr</i></td><td>A pointer to a variable</td></tr></table>	<i>file</i>	A file handle	<i>valuePtr</i>	A pointer to a variable		
<i>file</i>	A file handle						
<i>valuePtr</i>	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>	Result	Value	Successful	0	Unsuccessful	Non-zero error number
Result	Value						
Successful	0						
Unsuccessful	Non-zero error number						
Description	<p>Reads a sequence of hexadecimal digits from the given file and converts them to an unsigned long which is assigned to the <i>value</i> parameter, which should be a pointer to a macro variable.</p>						

Table 120: __readFile return values

Example

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

__readFileByte

Syntax

```
__readFileByte(file)
```

Parameter

file

A file handle

Return value

-1 upon error or end-of-file, otherwise a value between 0 and 255.

Description

Reads one byte from the file *file*.

Example

```
__var byte;
while ( (byte = __readFileByte(myFile)) != -1 )
{
    // Do something with byte
}
```

__readMemory8, __readMemoryByte

Syntax

```
__readMemory8(address, zone)
__readMemoryByte(address, zone)
```

Parameters

address

The memory address (integer)

zone

The memory zone name (string); for a list of available zones, see *Memory addressing*, page 143

Return value

The macro returns the value from memory.

Description

Reads one byte from a given memory location.

Example

```
__readMemory8(0x0108, "Memory");
```

__readMemory16

Syntax	<code>__readMemory16(<i>address</i>, <i>zone</i>)</code>	
Parameters	<i>address</i>	The memory address (integer)
	<i>zone</i>	The memory zone name (string); for a list of available zones, see <i>Memory addressing</i> , page 143
Return value	The macro returns the value from memory.	
Description	Reads a two-byte word from a given memory location.	
Example	<code>__readMemory16(0x0108, "Memory");</code>	

__readMemory32

Syntax	<code>__readMemory32(<i>address</i>, <i>zone</i>)</code>	
Parameters	<i>address</i>	The memory address (integer)
	<i>zone</i>	The memory zone name (string); for a list of available zones, see <i>Memory addressing</i> , page 143
Return value	The macro returns the value from memory.	
Description	Reads a four-byte word from a given memory location.	
Example	<code>__readMemory32(0x0108, "Memory");</code>	

__registerMacroFile

Syntax	<code>__registerMacroFile(<i>filename</i>)</code>	
Parameter	<i>filename</i>	A file containing the macros to be registered (string)
Return value	<code>int 0</code>	

Description	Registers macros from a setup macro file. With this function you can register multiple macro files during C-SPY startup.
Example	<code>__registerMacroFile("c:\\testdir\\macro.mac");</code>
See also	<i>Registering and executing using setup macros and setup files</i> , page 153.

__resetFile

Syntax	<code>__resetFile(filehandle)</code>	
Parameter	<i>filehandle</i>	The macro variable used as filehandle by the <code>__openFile</code> macro
Return value	<code>int 0</code>	
Description	Rewinds a file previously opened by <code>__openFile</code> .	

__restoreSoftwareBreakpoint

Syntax	<code>__restoreSoftwareBreakpoint()</code>	
Return value	<code>int 0</code>	
Description	<p>Restores automatically any breakpoints that were destroyed during system startup.</p> <p>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 <code>initialize by copy</code> directive for code in the linker configuration file or if you have any <code>__ramfunc</code> declared functions in your application. In this case, any breakpoints will be destroyed during the RAM copying when the C-SPY debugger starts.</p> <p>By using the this macro, C-SPY will restore the destroyed breakpoints.</p>	
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

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

Table 121: __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()");
```

The following example sets a code breakpoint on the label `main` in your source:

```
__setCodeBreak("main", 0, "1", "TRUE", "");
```

See also

Defining breakpoints, page 135.

__setDataBreak

Syntax	<code>__setDataBreak(location, count, condition, cond_type, access, action)</code>	
Parameters	<i>location</i>	<p>A string with a location description. This can be either:</p> <p>A <i>source location</i> 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</p> <p>An <i>absolute location</i> on the form <code>zone:hexaddress</code> or simply <code>hexaddress</code> (for example <code>Memory:0x42</code>)</p> <p>An <i>expression</i> whose value designates a location (for example <code>my_global_variable</code>).</p>
	<i>count</i>	The number of times that a breakpoint condition must be fulfilled before a break occurs (integer)
	<i>condition</i>	The breakpoint condition (string)
	<i>cond_type</i>	The condition type; either "CHANGED" or "TRUE" (string)
	<i>access</i>	The memory access type: "R" for read, "W" for write, or "RW" for read/write
	<i>action</i>	An expression, typically a call to a macro, which is evaluated when the breakpoint is detected

Return value

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

Table 122: __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	<pre>__var brk; brk = __setDataBreak("Memory:0x4710", 3, "d>6", "TRUE", "W", "ActionData()"); ...</pre>

```
__clearBreak(brk);
```

See also *Defining breakpoints*, page 135.

__setSimBreak

Syntax `__setSimBreak(location, access, action)`

Parameters

<i>location</i>	A string with a location description. This can be either: A <i>source location</i> on the form {filename}.line.col (for example {D:\\src\\prog.c}.12.9), although this is not very useful for simulation breakpoints. An <i>absolute location</i> on the form zone:hexaddress or simply hexaddress (for example Memory:0xE01E). An <i>expression</i> whose value designates a location (for example my_global_variable).
<i>access</i>	The memory access type: "R" for read or "W" for write
<i>action</i>	An expression, typically a call to a macro function, which is evaluated when the breakpoint is detected

Return value

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

Table 123: __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.

__sleep

Syntax	__sleep(<i>time</i>)	
Parameter	<i>time</i>	The debugger sleep time in microseconds
Return value	int 0	
Description	Causes the debugger to sleep the specified amount of time.	
Example	__sleep(1000000) Causes the debugger to sleep for 1 second.	

__sourcePosition

Syntax	__sourcePosition(<i>linePtr</i> , <i>colPtr</i>)	
Parameters	<i>linePtr</i>	Pointer to the variable storing the line number
	<i>colPtr</i>	Pointer to the variable storing the column number

Return value	<table><tr><th>Result</th><th>Value</th></tr><tr><td>Successful</td><td>Filename string</td></tr><tr><td>Unsuccessful</td><td>Empty (" ") string</td></tr></table>	Result	Value	Successful	Filename string	Unsuccessful	Empty (" ") string
Result	Value						
Successful	Filename string						
Unsuccessful	Empty (" ") string						

Table 124: __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(<i>macroString</i> , <i>pattern</i> , <i>position</i>)	
Parameters	<i>macroString</i>	The macro string to search in
	<i>pattern</i>	The string pattern to search for

	<i>position</i>	The position where to start the search. The first position is 0
Return value	The position where the pattern was found or -1 if the string is not found.	
Description	This macro searches a given string for the occurrence of another string.	
Example	<pre>__strFind("Compiler", "pile", 0) = 3 __strFind("Compiler", "foo", 0) = -1</pre>	
See also	<i>Macro strings</i> , page 460.	

__subString

Syntax	<code>__subString(<i>macroString</i>, <i>position</i>, <i>length</i>)</code>	
Parameters	<i>macroString</i>	The macro string from which to extract a substring
	<i>position</i>	The start position of the substring. The first position is 0.
	<i>length</i>	The length of the substring
Return value	A substring extracted from the given macro string.	
Description	This macro extracts a substring from another string.	
Example	<pre>__subString("Compiler", 0, 2)</pre>	
	The resulting macro string contains Co.	
	<pre>__subString("Compiler", 3, 4)</pre>	
	The resulting macro string contains pile.	
See also	<i>Macro strings</i> , page 460.	

__toLower

Syntax	<code>__toLower(<i>macroString</i>)</code>
Parameter	<i>macroString</i> is any macro string.
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	<pre>__toLower("IAR")</pre> <p>The resulting macro string contains <code>iar</code>.</p> <pre>__toLower("Mix42")</pre> <p>The resulting macro string contains <code>mix42</code>.</p>
See also	<i>Macro strings</i> , page 460.

__toString

Syntax	<code>__toString(C_string, maxlength)</code>	
Parameter	<i>string</i>	Any null-terminated C string
	<i>maxlength</i>	The maximum length of the returned macro string
Return value	Macro string.	
Description	This macro is used for converting C strings (<code>char*</code> or <code>char[]</code>) into macro strings.	
Example	<p>Assuming your application contains the following definition:</p> <pre>char const * hptr = "Hello World!";</pre> <p>the following macro call:</p> <pre>__toString(hptr, 5)</pre> <p>would return the macro string containing <code>Hello</code>.</p>	
See also	<i>Macro strings</i> , page 460.	

__toUpper

Syntax	<code>__toUpper(macroString)</code>	
Parameter	<i>macroString</i> is any macro string.	
Return value	The converted string.	

Description	This macro returns a copy of the parameter <i>macroString</i> where all the characters have been converted to upper case.
Example	<pre>__toUpper("string")</pre> <p>The resulting macro string contains <i>STRING</i>.</p>
See also	<i>Macro strings</i> , page 460.

__writeFile

Syntax	<pre>__writeFile(<i>file</i>, <i>value</i>)</pre>				
Parameters	<table><tr><td><i>file</i></td><td>A file handle</td></tr><tr><td><i>value</i></td><td>An integer</td></tr></table>	<i>file</i>	A file handle	<i>value</i>	An integer
<i>file</i>	A file handle				
<i>value</i>	An integer				
Return value	int 0				
Description	Prints the integer value in hexadecimal format (with a trailing space) to the file <i>file</i> . Note: The <code>__fmessage</code> statement can do the same thing. The <code>__writeFile</code> macro is provided for symmetry with <code>__readFile</code> .				

__writeFileByte

Syntax	<pre>__writeFileByte(<i>file</i>, <i>value</i>)</pre>				
Parameters	<table><tr><td><i>file</i></td><td>A file handle</td></tr><tr><td><i>value</i></td><td>An integer in the range 0-255</td></tr></table>	<i>file</i>	A file handle	<i>value</i>	An integer in the range 0-255
<i>file</i>	A file handle				
<i>value</i>	An integer in the range 0-255				
Return value	int 0				
Description	Writes one byte to the file <i>file</i> .				

__writeMemory8, __writeMemoryByte

Syntax	<pre>__writeMemory8(value, address, zone) __writeMemoryByte(value, address, zone)</pre>	
Parameters	<div> <div><i>value</i></div> <div>The value to be written (integer)</div> </div> <div> <div><i>address</i></div> <div>The memory address (integer)</div> </div> <div> <div><i>zone</i></div> <div>The memory zone name (string); for a list of available zones, see <i>Memory addressing</i>, page 143</div> </div>	
Return value	int 0	
Description	Writes one byte to a given memory location.	
Example	<pre>__writeMemory8(0x2F, 0x8020, "Memory");</pre>	

__writeMemory16

Syntax	<pre>__writeMemory16(value, address, zone)</pre>	
Parameters	<div> <div><i>value</i></div> <div>The value to be written (integer)</div> </div> <div> <div><i>address</i></div> <div>The memory address (integer)</div> </div> <div> <div><i>zone</i></div> <div>The memory zone name (string); for a list of available zones, see <i>Memory addressing</i>, page 143</div> </div>	
Return value	int 0	
Description	Writes two bytes to a given memory location.	
Example	<pre>__writeMemory16(0x2FFF, 0x8020, "Memory");</pre>	

__writeMemory32

Syntax	<pre>__writeMemory32(value, address, zone)</pre>	
Parameters	<div> <div><i>value</i></div> <div>The value to be written (integer)</div> </div>	

	<i>address</i>	The memory address (integer)
	<i>zone</i>	The memory zone name (string); for a list of available zones, see <i>Memory addressing</i> , page 143
Return value	<code>int</code>	0
Description	Writes four bytes to a given memory location.	

Example

```
__writeMemory32(0x5555FFFF, 0x8020, "Memory");
```

A

- Access Type (Breakpoints dialog box) 247
 - data breakpoint. 181
 - immediate breakpoint. 183
- access type (in JTAG Watchpoints dialog box) 253
- Action (Breakpoints dialog box) 246
 - code breakpoint 284
 - data breakpoint. 182
 - immediate breakpoint. 184
- Add existing project to current workspace
(Startup option) 340
- Additional include directories (assembler option) 406
- Additional include directories (compiler option) 395
- Additional libraries (linker option) 417
- address, in JTAG Watchpoints dialog box 253
- Alias (Key bindings option) 316
- Allow alternative register names, mnemonics and operands
(assembler option) 402
- Allow hardware reset (C-SPY RDI option) 234
- Angel driver, features 14
- Angel interface, specifying 216
- Angel (C-SPY options) 216
- application
 - built outside the IDE 117
 - testing 94, 157
- argument variables 335
 - environment variables 307
- in #include file paths
 - assembler 406
 - compiler 395
 - summary 306
- Arguments (External editor option) 321
- ARM code, mixing with Thumb code 390
- Arm (compiler option) 390
- arm (directory) 20
- ar, using for building libraries 69
- asm (filename extension) 22
- assembler
 - command line version 75
 - documentation 25
 - features 16
- assembler comments, text style in editor 101
- assembler directives
 - text style in editor 101
- assembler labels, viewing 132
- assembler list files
 - compiler call frame information, including 394
 - conditional information, specifying 404
 - cross-references, generating 405
 - format 54
 - generating 404
 - header, including 404
 - lines per page, specifying 405
 - tab spacing, specifying 405
- Assembler mnemonics (compiler option) 394
- assembler options 401
 - Allow alternative register names, mnemonics
and operands 402
 - Diagnostics 407
 - Language 401
 - List. 404
 - Output 403
 - Preprocessor. 405
- assembler output, including debug information 403
- assembler preprocessor 405
- assembler symbols
 - defining 406
 - using in C-SPY expressions 128
- assembler variables, viewing 132
- assert, in built applications 83
- assumptions, programming experience xxvii
- Atmel AT91EBxx flash loader 259
- Atmel AT91SAM7A1-Ek flash loader 259
- Atmel AT91SAM7A2-Ek flash loader 259
- Attach to program (C-SPY Download option) 215
- Auto indent (editor option) 318

Auto window	359
context menu	359
Automatic runtime library selection (linker option)	417
Automatic (compiler option)	389
Autostep settings dialog box (Debug menu)	375

B

-B (C-SPY command line option)	438
--backend (C-SPY command line option)	438–439
Background color (IDE Tools option)	324
backtrace information	
generated by compiler	125
viewing in Call Stack window	363
Base (Register filter option)	332
bat (filename extension)	22
Batch Build	93
Batch Build Configuration dialog box (Project menu) ..	312
Batch Build dialog box (Project menu)	311
batch files, specifying in IDE	80, 336
Baud rate (C-SPY Angel option)	217
Baud rate (C-SPY IAR ROM-monitor option)	220
Baud rate (C-SPY Macraigor option)	232
--BE32 (C-SPY command line option)	435
--BE8 (C-SPY command line option)	435
Big endian (C-SPY target option)	380
bin, arm (subdirectory)	20
bin, common (subdirectory)	21
blocks, in C-SPY macros	462
Body (b) (Configure auto indent option)	320
bold style, in this guide	xlii
bookmarks	
adding	105
showing in editor	319
break condition (in JTAG Watchpoints dialog box)	254
Break (button)	125, 345
breakpoint condition, example	138
breakpoint icons	136
Breakpoint type (Breakpoints dialog box)	246

Breakpoint Usage dialog box (Simulator menu)	184, 250
using	140
breakpoints	124
code, example	484
conditional, example	64
connecting a C-SPY macro	155
consumers	141
data	180, 246, 248
example	485
immediate	182
example	65
in Memory window	137
in ramfunc functions	251
in the simulator	179
listing all	140
on vectors, using Macraigor	250
setting	
in memory window	137
using system macros	138
using the dialog box	137
settings	308
single-stepping if not available	115
system, description of	135
toggling	136
viewing	139
Breakpoints dialog box	
Code	245, 283
Data	180, 246
Data Log	248
Immediate	183
Log	285
Breakpoints options (C-SPY options)	243
Breakpoints window (View menu)	282
Breakpoints (J-Link/J-Trace option)	244
Broadcast all branch addresses (Trace Setup option) ...	239
Buffered terminal output (general option)	384
-build (iarbuild command line option)	95
Build Actions	94
Build Actions Configuration (Build Actions options) ...	413

build configuration	
creating	84
definition of	82
Build window context menu	288
Build window (View menu)	288
building	
commands for	93
from the command line	95
options	326
pre and post actions	94
the process	91
byte order, specifying	380

C

C comments, text style in editor	101
C compiler. <i>See</i> compiler	
C function information, in C-SPY	125
C keywords, text style in editor	101
C symbols, using in C-SPY expressions	127
C variables, using in C-SPY expressions	127
c (filename extension)	22
call chain, displaying in C-SPY	125
Call stack information	125
Call Stack window	363
context menu	363
example	63
for backtrace information	125
__cancelAllInterrupts (C-SPY system macro)	467
__cancelInterrupt (C-SPY system macro)	467
Catch exceptions (C-SPY RDI option)	235
category, in Options dialog box	92, 309
cfg (filename extension)	22
characters, in assembler macro quotes	402
Check In Files dialog box	272
Check Out Files dialog box	273
Checksum (linker options)	423
checksum, generating	424
chm (filename extension)	22
-clean (iarbuild command line option)	95
__clearBreak (C-SPY system macro)	468
Close Workspace (File menu)	292
__closeFile (C-SPY system macro)	468
code coverage	
commands	366
context menu	366
using	160
viewing	161
Code Coverage window	365
code generation	
assembler	401
compiler, features	14
code integrity	88
code memory, filling unused	424
Code page (compiler options)	390
Code section name (compiler option)	393
code templates, using in editor	103
code, testing	94
command line options	
specifying in IDE	80, 336
typographic convention	xlii
command prompt icon, in this guide	xlii
Command (External editor option)	321
Common Fonts (IDE Options dialog box)	314
common (directory)	21
communication problem, J-Link	223
Communication (Angel C-SPY option)	217
Communication (C-SPY J-Link option)	224
Communication (OKI ROM-monitor C-SPY option)	220
compiler	
command line version	4, 75
documentation	15, 25
features	14
compiler call frame information	
including in assembler list file	394
compiler diagnostics	394
suppressing	397

compiler list files	
assembler mnemonics, including	394
example	36
generating	394
source code, including	394
compiler options	387
setting in Embedded Workbench, example	33
Code	390
Diagnostics	396
Generate interwork code	390
Language	388
List	394
MISRA C	398
Optimizations	391
Output	392
Preprocessor	395
Processor mode	390
compiler output, including debug information	393
compiler preprocessor	395
compiler symbols, defining	396
computer style, typographic convention	xlii
conditional breakpoints, example	64
conditional statements, in C-SPY macros	461
Conditions (Breakpoints dialog box)	
code breakpoint	285
data breakpoint	182
Conditions (Breakpoints dialog)	246
Config (linker options)	415
Configuration file symbol definitions (linker option)	416
Configuration file (general option)	382
Configurations for project dialog box (Project menu)	307
Configure Auto Indent (IDE Options dialog box)	319
Configure Tools (Tools menu)	334
Configure Viewers dialog box (Tools menu)	338
config, arm (subdirectory)	20
config, common (subdirectory)	22
\$CONFIG_NAME\$ (argument variable)	306
context menu, in windows	130
conventions, used in this guide	xlii
converter options	409
Output file	409
Copy (button)	265
copyright	ii
Core (General option)	379
cpp (filename extension)	22
--cpu (C-SPY command line option)	435
CPU clock (C-SPY J-Link option)	227
CPU mode (in JTAG Watchpoints dialog box)	254
CPU registers, definitions	115
Create New Project dialog box (Project menu)	308
Create new project in current workspace (Startup option)	340
Cross-reference (assembler option)	405
cspybat	433
current position, in C-SPY Disassembly window	347
cursor, in C-SPY Disassembly window	347
\$CUR_DIR\$ (argument variable)	306
\$CUR_LINE\$ (argument variable)	306
custom build	95
using	96
custom tool configuration	96
Custom Tool Configuration (Custom Build options)	411
Cycle accurate tracing (Trace Setup option)	239
C++ comments, text style in editor	101
C++ keywords, text style in editor	101
C++ terminology	xlii
C++ tutorial	55
C-SPY	
characteristics, additional drivers	213, 255
debugger systems	9
overview	112
environment overview	113
IDE reference information	343
overview	5
plugin modules, loading	116
setting up	114
Simulator	165
starting the debugger	116
C-SPY Bat	433

C-SPY command line option options	438
C-SPY Download options	
Attach to program	215
Flash download	216
Suppress download	215
Verify download	215
C-SPY drivers	
Angel	14
J-Link	10
LMI FTDI	11
Macraigor	12
RDI	11
ROM-monitor	13
simulator	165
C-SPY expressions	127
evaluating	131
in C-SPY macros	461
Quick Watch, using	131
Tooltip watch, using	130
Watch window, using	130
C-SPY GDB Server options	
TCPIP address or hostname	218
C-SPY JTAG options	
Log communication	218, 225, 232
OCD interface device	234
C-SPY J-Link options	
Communication	224
CPU clock	227
HW Trace	227
Interface	224
ITM Stimulus Ports	228
JTAG scan chain	225
JTAG speed	223, 229
SWO clock	227
C-SPY Macraigor options	
Debug handler address	232
Hardware reset	232
JTAG scan chain with multiple targets	232
JTAG speed	231
OCD interface device	231
TCP/IP	231
C-SPY macros	149, 459
blocks	462
conditional statements	461
C-SPY expressions	461
dialog box	375
using	152
examples	150
checking status of register	154
checking the status of WDT	154
creating a log macro	155
execUserPreload(), using	119
execUserSetup	61, 67
remapping memory before download	119
executing	152
connecting to a breakpoint	155
using Quick Watch	154
using setup macro and setup file	153
functions	128, 459
loop statements	461
macro statements	461
setup macro file	
definition of	151
executing	153
setup macro function	
definition of	151
summary	464
system macros, summary of	465
using	149
variables	129, 460
C-SPY options	429
Extra Options	431
for the simulator	165
in Options dialog box	310
Plugins	431
Setup	429
C-SPY RDI options	
Allow hardware reset	234

Catch exceptions	235
ETM trace	234
Log RDI communication	235
C-SPY Third-Party Driver options	
IAR debugger driver plugin	237
Log communication	237
C-SPY windows	
Memory Access Configuration	168
Pipeline Trace	167
C-SPYLink	8
C/C++ syntax styles, options	323

D

-d (C-SPY command line option)	439
dat (filename extension)	22
data breakpoints	180, 246, 248
data coverage, in Memory window	350
data specification (in JTAG Watchpoints dialog box)	253
dbgt (filename extension)	22
ddf (filename extension)	22
selecting device description file	115
Debug handler address (C-SPY Macraigor option)	232
debug information	
generating in assembler	403
in compiler, generating	393
Debug Log window context menu	291
Debug Log window (View menu)	290
Debug menu	373
Debug without downloading text box	265
debugger concepts, definitions of	111
debugger drivers	
simulator	165
debugger system overview	112
Debugger (IDE Options dialog box)	328
debugging projects	
externally built applications	117
in disassembly mode, example	46
debugging, RTOS awareness	9

default installation path	19
Default integer format (IDE option)	329
#define statement, in compiler	396
#define (linker option)	421
Defined by application (linker option)	418
Defined symbols (assembler option)	406
Defined symbols (compiler option)	396
Defined symbols (linker option)	421
dep (filename extension)	22
description (interrupt property)	191
development environment, introduction	75
--device (C-SPY command line option)	440
Device description file (C-SPY option)	430
device description files	20, 115
definition of	118
modifying	118
specifying interrupts	479
device selection files	20
Device (General option)	379
diagnostics	
compiler	
including in list file	394
suppressing	397
linker, suppressing	422
Diagnostics (assembler options)	407
Diagnostics (compiler options)	396
Diagnostics (linker option)	421
directories	
arm	20
assembler, ignore standard include	406
common	21
compiler, ignore standard include	395
root	19
directory structure	19
Disable language extensions (compiler option)	389
__disableInterrupts (C-SPY system macro)	468
Disassembly menu	378
disassembly mode debugging, example	46

- Disassembly window 346
 - context menu 347
- Discard Unused Publics (compiler option) 387
- disclaimer ii
- DLIB. 15
 - documentation xli, 25
- DLIB library functions, reference information 100
- dni (filename extension) 22
- Do not show Information Center/rat startup (Startup option) .
341
- Do not show this window at startup (Startup option) 341
- do (macro statement) 461
- dockable windows. 77
- document conventions. xlii
- documentation 19
 - assembler 16
 - compiler 15
 - GNU binary utilities. 20
 - linker 17
 - MISRA C. 25
 - online. 20, 22
 - other guides xli
 - overview xxxviii
 - product. 24
 - runtime libraries. 25
 - this guide xxxvii
- doc, arm (subdirectory) 20
- doc, common (subdirectory) 22
- Download (C-SPY options) 215
- downloading to flash memory 216
- drag-and-drop
 - of files in Workspace window 84
 - text in editor window 101
- Driver (C-SPY option) 429
- drivers, arm (subdirectory) 20
- __driverType (C-SPY system macro) 469
- drv_attach_to_program (C-SPY command line option) . 435
- drv_catch_exceptions (C-SPY command line option) . . 440
- drv_communication (C-SPY command line option). . . . 441
- drv_communication_log (C-SPY command line option) . 443

- drv_default_breakpoint (C-SPY command line option) . 443
- drv_reset_to_cpu_start (C-SPY command line option) . 444
- drv_restore_breakpoints (C-SPY command line option) . 444
- drv_suppress_download (C-SPY command line option) . 436
- drv_vector_table_base (C-SPY command line option) . . 445
- drv_verify_download (C-SPY command line option) . . 436
- Dynamic Data Exchange (DDE). 106
 - calling external editor 321

E

- Edit Filename Extensions dialog box (Tools menu) 337
- Edit Interrupt dialog box (Simulator menu) 190
- Edit Memory Access dialog box. 179
- Edit menu 294
- editing source files 99
- edition, user guide. ii
- editor
 - code templates 103
 - commands 101
 - customizing the environment 105
 - external 106
 - features 5
 - HTML files 274
 - indentation 102
 - keyboard commands 278
 - matching parentheses and brackets 103
 - options 317
 - shortcut to functions. 105, 275
 - splitter controls 275
 - status bar, using in 103
 - using 99
- Editor Colors and Fonts (IDE Options dialog box) 323
- Editor Font (Editor colors and fonts option) 323
- Editor Setup Files (IDE Options dialog box) 322
- editor setup files, options 322
- Editor window 274
 - context menu 276
 - tab, context menu. 275

Editor (External editor option)	321
Editor (IDE Options dialog box)	317
EEC++ syntax (compiler option)	388
ELF, converting from	409
Embedded C++ Technical Committee	xli
Embedded C++ (compiler option)	388
Embedded C++, enabling syntax in compiler	388
Embedded Workbench	
editor	99
exiting from	77
layout	77
main window	76, 264
reference information	263
running	76
Startup dialog box (Help menu)	340
version number, displaying	340
EmbeddedICE macrocell	252
emulator (C-SPY version)	
third-party	4
__emulatorSpeed (C-SPY system macro)	469
__emulatorStatusCheckOnRead (C-SPY system macro)	470
Enable graphical stack display and stack usage	
tracking (Stack option)	330
Enable multibyte support (assembler option)	401
Enable multibyte support (compiler option)	390
Enable remarks (compiler option)	397
Enable remarks (linker option)	422
Enable Virtual Space (editor option)	319
enabled transformations, in compiler	392
__enableInterrupts (C-SPY system macro)	471
End address (linker option)	424
--endian (C-SPY command line option)	436
Endian mode (General option)	380
Enter Location (Breakpoints dialog box)	287
Entry symbol (linker option)	417
environment variables, as argument variables	307
EOL character (editor option)	318
EPI JEENI JTAG interface	216
error messages	
compiler	398
linker	423
ETM trace (C-SPY RDI option)	234
__evaluate (C-SPY system macro)	471
ewd (filename extension)	23
ewp (filename extension)	23
ewplugin (filename extension)	23
eww (filename extension)	23
the workspace file	76
\$EW_DIR\$ (argument variable)	306
Example applications (Startup option)	341
examples	
breakpoints	45
executing up to	46
setting	
using dialog box	64
using macro	67
calling convention, examining	51
compiling	35
C-SPY macros	150
C/C++ and assembler, mixing	52
ddf file, using	63
debugging a program	41
disassembly mode debugging	46
function calls, displaying in C-SPY	63
interrupts	
using macro	67
linking	
a compiler program	38
viewing the map file	39
macros	
checking status of register	154
checking status of WDT	154
creating a log macro	155
for interrupts and breakpoints	67
using Quick Watch	154
Memory window, using	47
memory, monitoring	47
mixing C and assembler	51
performing tasks without stopping execution	138

- project
 - adding files 31
 - creating 29–30
 - reaching program exit 49
 - registers, monitoring 65
 - Scan for Changed Files (editor option), using 37
 - setting project options 32
 - stepping 42
 - Terminal I/O, displaying 49
 - tracing incorrect function arguments 138
 - using libraries 69
 - variables
 - setting a watch point 44
 - watching in C-SPY 44
 - viewing assembler list file 54
 - viewing compiler list files 36
 - workspace, creating a new 29
 - examples, arm (subdirectory) 20
 - execUserExit (C-SPY setup macro) 464
 - execUserFlashExit (C-SPY setup macro) 464
 - execUserFlashInit (C-SPY setup macro) 464
 - execUserFlashReset (C-SPY setup macro) 464
 - execUserPreload (C-SPY setup macro) 464
 - execUserReset (C-SPY setup macro) 464
 - execUserSetup (C-SPY setup macro) 464
 - example 61, 67
 - Executable (output directory) 381
 - executing a program up to a breakpoint 46
 - execution history, tracing 132
 - execution time, reducing 157
 - \$EXE_DIR\$ (argument variable) 306
 - Exit (File menu) 77
 - exit, of user application 125
 - expressions. *See* C-SPY expressions
 - External Editor (IDE Options dialog box) 320
 - external editor, using 106
 - external input (in JTAG Watchpoints dialog box) 254
 - Extra Options
 - for assembler 408
 - for compiler 399
 - for C-SPY 431
 - for linker 425
- ## F
- factory settings
 - linker 428
 - restoring default settings 93
 - features
 - assembler 16
 - compiler 14
 - editor 5
 - source code control 4
 - file extensions. *See* filename extensions
 - File menu 291
 - file types
 - device description 20
 - specifying in Embedded Workbench 115
 - device selection 20
 - documentation 20
 - drivers 20
 - flash loader applications 20
 - header 21
 - include 21
 - library 21
 - linker configuration files 20
 - macro 115, 430
 - project templates 20
 - readme 20, 22
 - syntax coloring configuration 20
 - filename extensions 22
 - cfg, syntax highlighting 324
 - ddf, selecting device description file 115
 - eww, the workspace file 76
 - mac
 - the macro file 150
 - using macro file 115
 - other than default 24

sfr, register definitions for C-SPY	118
Filename Extensions dialog box (Tools menu)	336
Filename Extensions Overrides dialog box (Tools menu)	337
files	
adding to a project	31
checking in and out	89
compiling, example	35
editing	99
navigating among	85
readme.htm	24
\$FILE_DIR\$ (argument variable)	306
\$FILE_FNAME\$ (argument variable)	306
\$FILE_PATH\$ (argument variable)	306
Fill dialog box (Memory window)	351
Fill pattern (linker option)	424
Fill unused code memory (linker option)	424
Filter Files (Register filter option)	332
Find dialog box (Edit menu)	297
Find in Files dialog box (Edit menu)	299
Find in Files window	
context menu	289
Find in Files window (View menu)	288
Find in Trace dialog box	174
Find in Trace window	174
Find Next (button)	265
Find Previous (button)	265
Find (button)	265
first activation time (interrupt property)	191
definition of	186
Fixed width font (IDE option)	314
Flash download (C-SPY Download option)	216
flash loader applications	20
Flash Loader Overview dialog box	257
flash memory	
loading externally built applications to	117
--flash_loader (C-SPY command line option)	445
floating windows	77
fmt (filename extension)	23

font	
Editor	323
Fixed width	314
Proportional width	314
for (macro statement)	461
Forced Interrupt window (Simulator menu)	192
formats	
assembler list file	54
compiler list file	36
--fpu (C-SPY command line option)	436
FPU (General option)	380
Freescal MAC71x1 flash loader	259
function calls, displaying in C-SPY	63
function level profiling	157
Function Trace (C-SPY window)	172
function trace, definition of	170
functions	
C-SPY running to when starting	114, 430
shortcut to in editor windows	105, 275

G

__gdbserver_exec_command (C-SPY system macro)	472
--gdbserv_exec_command (C-SPY command line option)	446
general options	379
Core	379
Device	379
specifying, example	32
Endian mode	380
FPU	380
Library Configuration	382
Library Options	384
MISRA C	385
Output	381
Processor variant	379
Target	379
Generate browse information (IDE Project options)	327
Generate checksum (linker option)	424
Generate debug info (assembler option)	403

Generate debug information (compiler option)	393
Generate interwork code (compiler option)	390
Generate linker map file (linker option)	420
Generate log (linker option)	420
--generate_sim (C-SPY command line option)	446
GNU binary utilities, documentation	20
Go to Bookmark (button)	265
Go to function (editor button)	105, 275
Go To (button)	265
Go (button)	345
Go (Debug menu)	123
Group members (Register filter option)	332
Groups (Register filter option)	332
groups, definition of	83

H

h (filename extension)	23
Hardware reset (C-SPY Macraigor option)	232
header files	21
quick access to	105
Help menu	340
helpfiles (filename extension)	23
highlighting, in C-SPY	124
hold time (interrupt property)	191
definition of	186
htm (filename extension)	23
html (filename extension)	23
HW Trace (C-SPY J-Link option)	227
__hwReset (C-SPY system macro)	472
__hwResetWithStrategy (C-SPY system macro)	473

I

i (filename extension)	23
IAR Assembler Reference Guide	25
IAR debugger driver plugin (Third-party Driver option)	237
IAR Development Guide	25
IAR ROM-monitor interface, specifying	219

IAR ROM-monitor (C-SPY options)	219
IAR Systems web site	26
iarbuild, building from the command line	95
IarIdePm.exe	76
icf (filename extension)	23
icons, in this guide	xlii
IDE	3–4
if else (macro statement)	461
if (macro statement)	461
Ignore standard include directories (assembler option)	406
Ignore standard include directories (compiler option)	395
ILINK options, typographic convention	xlii
ILINK <i>See</i> linker	
illegal memory accesses, checking for	176
immediate breakpoints	182
inc (filename extension)	23
Include compiler call frame information (compiler option)	394
Include debug information in output (linker option)	419
include files	21
assembler, specifying path	406
compiler, specifying path	395
Include header (assembler option)	404
Include listing (assembler option)	404
Include source (compiler option)	394
Incremental Search dialog box (Edit menu)	300
inc, arm (subdirectory)	21
Indent Size (editor option)	317
indentation, in editor	102
information center	341
information, product	24
inherited settings, overriding	92
ini (filename extension)	23
input	
redirecting to Terminal I/O window	364
special characters in Terminal I/O window	364
Input Mode dialog box	365
Input (linker option)	418
insertion point, shortcut key for moving	101
installation path, default	19

installed files	19
documentation	20, 22
executable	21
include	21
library	21
Integrated Development Environment (IDE).	3–4
Intel-extended, C-SPY input format	113
Interface (C-SPY J-Link option).	224
Internet, IAR Systems web site.	26
Interrupt Log window (Simulator menu).	194
Interrupt Setup dialog box (Simulator menu)	189
interrupt system, using device description file	189
interrups	
adapting C-SPY system for target hardware	189
options	191
simulated, definition of	185
timer, example	195
using system macros	192
interwork code, generating	390
ISO/ANSI C, compiler adhering to	389
italic style, in this guide	xlii
ITM Stimulus Ports (C-SPY J-Link option).	228
i79 (filename extension)	23

J

__jlinkExecCommand (C-SPY system macro)	473
--jlink_device_select (C-SPY command line option)	446
--jlink_exec_commmmand (C-SPY command line option) . . .	447
--jlink_initial_speed (C-SPY command line option). . . .	447
--jlink_interface (C-SPY command line option)	448
--jlink_ir_length (C-SPY command line option).	448
--jlink_reset_strategy (C-SPY command line option) . . .	448
--jlink_speed (C-SPY command line option)	449
JTAG interfaces	
EPI JEENI	216
J-Link.	221, 229
JTAG scan chain with	232
JTAG scan chain (C-SPY J-Link options).	225

JTAG speed (C-SPY J-Link option)	223, 229
JTAG speed (C-SPY Macraigor option)	231
JTAG watchpoints.	251
JTAG Watchpoints (dialog box)	252
__jtagCommand (C-SPY system macro)	474
__jtagCP15IsPresent (C-SPY system macro)	474
__jtagCP15ReadReg (C-SPY system macro)	474
__jtagCP15WriteReg (C-SPY system macro).	475
__jtagData (C-SPY system macro)	475
__jtagRawRead (C-SPY system macro)	476
__jtagRawSync (C-SPY system macro)	476
__jtagRawWrite (C-SPY system macro).	477
__jtagResetTRST (C-SPY system macro)	478
J-Link communication problem	223
J-Link driver, features	10
J-Link JTAG interface.	221, 229
J-Link watchpoints	252
J-Link/J-Trace Connection (C-SPY options).	224
J-Link/J-Trace Reset (C-SPY option)	221
J-Link/J-Trace Setup (C-SPY options)	218, 221

K

Keep symbol (linker option)	418
Key bindings (IDE Options dialog box)	315
key summary, editor	278

L

Label (c) (Configure auto indent option).	320
labels (assembler), viewing.	132
Language conformance (compiler option)	389
language extensions	
disabling in compiler	389
language facilities, in compiler	15
Language (assembler options).	401
Language (compiler options)	388
Language (IDE Options dialog box).	316
Language (Language option)	316

- layout, of Embedded Workbench 77
- library
 - creating a project for 70
 - runtime. 15
- library builder options
 - Output 427
- Library Configuration (general options) 382
- library files 21
- library functions
 - configurable. 21
 - reference information. 100
- Library low-level interface
- implementation (general option) 383
- library modules
 - example 69
 - using 69
- Library Options (general options) 384
- Library (general option) 382
- Library (linker options) 417
- lib, arm (subdirectory) 21
- lightbulb icon, in this guide. xlii
- #line directives, generating
 - in compiler. 396
- Lines/page (assembler option) 405
- linker
 - command line version 75
 - documentation 25
 - overview 16
- Linker configuration file editor (dialog box) 416
- Linker configuration file (linker option) 415
- linker configuration files 20
- linker diagnostics, suppressing 422
- linker options 415
 - factory settings. 428
 - Output 409
- linker symbols, defining 421
- linking, example 38
- list files
 - assembler 54
 - compiler runtime information, including. 394
 - conditional information, specifying 404
 - cross-references, generating 405
 - header, including. 404
 - lines per page, specifying 405
 - tab spacing, specifying 405
- compiler
 - assembler mnemonics, including 394
 - example. 36
 - generating 394
 - source code, including 394
 - option for specifying destination 382
- List (assembler options) 404
- List (compiler options) 394
- List (linker option) 420
- \$LIST_DIR\$ (argument variable) 306
- Little endian (General option) 380
- Live Watch window 359
 - context menu 360
- Live watch (IDE option) 329
- LMI FTDI driver, features 11
- LMI FTDI Setup (C-SPY options) 229
- lmiftdi_speed (C-SPY command line option). 450
- Locals window 358
 - context menu 359
- log (iarbuild command line option) 95
- Log communication (Angel C-SPY option) 217
- Log communication (C-SPY JTAG option) ... 218, 225, 232
- Log communication (C-SPY Third-Party Driver option) . 237
- Log communication (LMI FTDI C-SPY option). 229
- Log communication (OKI ROM-monitor C-SPY option). 220
- Log File dialog box (Debug menu) 377
- log file, generate from linker. 420
- Log RDI communication (C-SPY RDI option). 235
- log (filename extension) 23
- loop statements, in C-SPY macros 461
- lst (filename extension). 23

M

mac (filename extension)	23
the macro file	150
using a macro file	115
Macraigor driver, features	12
Macraigor (C-SPY options)	230
--macro (C-SPY command line option)	453–454
Macro Configuration dialog box (Debug menu)	375
macro files, specifying	115, 430
Macro quote characters (assembler option)	402
macro statements	461
macros	
executing	152
system	459
using	149
--mac_handler_address (C-SPY command line option)	450
--mac_interface (C-SPY command line option)	450
--mac_jtag_device (C-SPY command line option)	451
--mac_multiple_targets (C-SPY command line option)	451
--mac_reset_pulls_reset (C-SPY command line option)	452
--mac_set_temp_reg_buffer (C-SPY command line option)	452
--mac_speed (C-SPY command line option)	453
--mac_xscale_ir7 (C-SPY command line option)	453
main function, C-SPY running to when starting	114, 430
main.s (assembler tutorial file)	69
-make (iarbuild command line option)	95
Make before debugging (IDE Project options)	326
managing projects	4
map files	
example	39
viewing	39
map file, generate from linker	420
--mapu (C-SPY command line option)	454
mask (in JTAG Watchpoints dialog box)	253
Max number of errors (assembler option)	407
Max.s (assembler tutorial file)	69

memory	
filling unused	424
monitoring	145
example	47
remapping in C-SPY	119
memory access checking	176, 178
Memory Access Configuration window (Simulator menu)	168
Memory Access Setup dialog box (Simulator menu)	176
memory accesses, illegal	176
memory map	168, 176
Memory Restore dialog box	353
Memory Save dialog box	352
Memory window	348
context menu	350
using	145
memory zones	143
menu bar	264
C-SPY-specific	344
menu (filename extension)	23
Menu (Key bindings option)	315
menus	291
specific to C-SPY	373
Messages window, amount of output	324
Messages (IDE Options dialog box)	324
migration, from earlier IAR compilers	xli
Min.s (assembler tutorial file)	69
MISRA C	
compiler options	398
documentation	25
general options	385
Motorola, C-SPY input format	113
Multi-file compilation (compiler option)	387

N

naming conventions	xliii
Navigate Backward (button)	265
NDEBUG, preprocessor symbol	83
New Configuration dialog box (Project menu)	308

New Document (button)	265
New Group (Register filter option)	332
Next Statement (button)	345
Nohau LPC2800 flash loader	259
NXP LPC flash loader	260
NXP LPC2888 flash loader.	259

O

o (filename extension).	23
object files, specifying output directory	382
\$OBJ_DIR\$ (argument variable)	306
OCD interface device (C-SPY JTAG option)	234
OCD interface device (C-SPY Macraigor option).	231
Olimex LPCH 288x flash loader.	260
online documentation	
available from Help menu	340
common, in directory.	22
target-specific, in directory	20
online help	25
Open existing workspace (Startup option)	340
Open Workspace (File menu)	292
__openFile (C-SPY system macro).	478
Opening Brace (a) (Configure auto indent option)	320
optimization levels	391
Optimizations page (compiler options).	391
Optimizations (compiler option).	391
optimizations, effects on variables	129
options	
assembler	401
compiler.	387
converter	409
Custom Build.	413
custom build	411
C-SPY	429
C-SPY command line option	438
editor	317
general	379
general, specifying.	32

hardware debugger systems	213
library builder	427
linker	415
setup files for editor.	322
Options dialog box (Project menu)	309
using	92
__orderInterrupt (C-SPY system macro).	479
out (filename extension)	22
output	
assembler	
including debug information.	403
compiler	
including debug information.	393
preprocessor, generating	396
converter	
specifying filename.	409
converting from ELF	409
from C-SPY, redirecting to a file	117
linker, specifying filename.	419
specifying filename for linker output.	419
Output assembler file (compiler option)	394
Output file (converter option)	409
Output file (linker option).	419
Output list file (compiler option)	394
Output (assembler option).	403
Output (compiler options).	392
Output (general options).	381
Output (library builder options)	427
Output (linker options)	409, 419
Override default program entry (linker option)	417

P

-p (C-SPY command line option)	454
parameters, typographic convention	xlii
parentheses and brackets, matching (in editor)	103
part number, of user guide	ii
Paste (button)	265

paths	
assembler include files	406
compiler include files	395
relative, in Embedded Workbench	85, 278
source files	278
pbd (filename extension)	23
pbi (filename extension)	23
peripheral units, definitions	115
pew (filename extension)	23
Phytec LPC3180 flash loader	260
Pipeline Trace window (Simulator menu)	167
Plain 'char' is (compiler option)	389
Play a sound after build operations (IDE Project options)	327
plugin modules (C-SPY)	8
loading	116
Plugins (C-SPY options)	431
plugins, arm (subdirectory)	21
plugins, common (subdirectory)	22
__popSimulatorInterruptExecutingStack (C-SPY system macro)	480
Port (Angel C-SPY option)	217
Port (C-SPY Macraigor option)	231
Port (OKI ROM-monitor C-SPY option)	220
powerpac, arm (subdirectory)	21
Preinclude file (compiler option)	396
preprocessor directives, text style in editor	101
Preprocessor output to file (compiler option)	396
Preprocessor (assembler option)	405
preprocessor (compiler options)	395
prerequisites, programming experience	xxvii
Press shortcut key (Key bindings option)	315
Primary (Key bindings option)	315
Printf formatter (general option)	384
prj (filename extension)	23
probability (interrupt property)	191
definition of	186
Processor mode (compiler option)	390
Processor variant (General option)	379
-proc_stack_xxx (C-SPY command line option)	455
product overview	
assembler	16
compiler	14
C-SPY Debugger	5
directory structure	19
documentation	24
file types	22
IAR Embedded Workbench IDE	3
linker	16
profiling information	157
Profiling window	367
using	158
program execution, in C-SPY	121
programming experience	xxvii
Project Make, options	326
Project menu	304
project model	81
Project page (IDE Options dialog box)	326
projects	
adding files to	84, 304
example	31
build configuration, creating	84
building	93
in batches	93
compiling, example	35
creating	30, 84
example	70
definition of	81
excluding groups and files	84
files	
checking in and out	89
moving	84
for debugging externally built applications	117
groups, creating	84
managing	4, 81
organization	81
removing items	84
setting options	91
source code control	88

testing	94
version control systems	88
workspace, creating	84
\$PROJ_DIR\$ (argument variable)	307
\$PROJ_FNAME\$ (argument variable)	307
\$PROJ_PATH\$ (argument variable)	307
Promable output format (linker option)	409
Proportional width font (IDE option)	314
PUBLIC (assembler directive)	69

Q

Quick Search text box	265
Quick Watch window	360
executing C-SPY macros	154
using	131

R

Raw binary image (linker option)	418
RDI driver, features	11
RDI (C-SPY options)	233
--rdi_allow_hardware_reset	
(C-SPY command line option)	456
--rdi_driver_dll (C-SPY command line option)	456
--rdi_heartbeat (C-SPY command line option)	436
--rdi_step_max_one (C-SPY command line option)	457
--rdi_use_etm (C-SPY command line option)	456
__readFile (C-SPY system macro)	480
__readFileByte (C-SPY system macro)	481
reading guidelines	xxxvii
readme files	20, 22
readme.htm	24
__readMemoryByte (C-SPY system macro)	481
__readMemory16 (C-SPY system macro)	482
__readMemory32 (C-SPY system macro)	482
__readMemory8 (C-SPY system macro)	481
Recent workspace (Startup option)	341
Redo (button)	265

Reed-Solomon	260
reference information, typographic convention	xlii
Register Filter (IDE Options dialog box)	332
register groups	147
application-specific, defining	148
predefined, enabling	147
Register window	356
using	147
registered trademarks	ii
__registerMacroFile (C-SPY system macro)	482
registers, in device description file	118
relative paths	85, 278
Relaxed ISO/ANSI (compiler option)	389
release notes	22
Reload last workspace at startup (IDE Project options)	327
remapping memory	119
remarks	
compiler diagnostics	397
linker diagnostics	422
Remove trailing blanks (editor option)	319
repeat interval (interrupt property)	191
definition of	186
Replace dialog box (Edit menu)	298
Replace (button)	265
Require prototypes (compiler option)	389
Reset All (Key bindings option)	316
Reset (button)	345
Reset (Debug menu), example	49
Reset (J-Link/J-Trace option)	221
__resetFile (C-SPY system macro)	483
Restore software breakpoints at (J-Link/J-Trace option)	244
__restoreSoftwareBreakpoint (C-SPY system macro)	483
restoring default factory settings	93
return (macro statement)	462
ROM-monitor driver, features	13
ROM-monitor protocols	219
Angel	216
ROM-monitor, definition of	113
root directory	19

RTOS awareness debugging	9
RTOS awareness (C-SPY plugin module)	116
RTOS plugins	432
Run to Cursor (button)	345
Run to Cursor, description	124
Run to (C-SPY option)	114, 430
runtime libraries	15
documentation	25

S

s (filename extension)	23
Save All (button)	265
Save All (File menu)	293
Save As (File menu)	293
Save editor windows before building (IDE Project options)	326
Save workspace and projects before building (IDE Project options)	326
Save Workspace (File menu)	292
Save (button)	265
Save (File menu)	293
Scan for Changed Files (editor option)	319
using	37
Scanf formatter (general option)	384
SCC. <i>See</i> source code control systems	
scrolling, shortcut key for	101
searching in editor windows	105
Select SCC Provider dialog box (Project menu)	271
Select Statics dialog box (Statics window)	362
selecting text, shortcut key for	101
Semihosted, SWI (option)	364
using	126
--semihosting (C-SPY command line option)	457
Send heartbeat (Angel C-SPY option)	217
Serial port settings (Angel C-SPY option)	217
Serial port settings (OKI ROM-monitor C-SPY option)	220
Service (External editor option)	321
Set Log file dialog box (Debug menu)	375
__setCodeBreak (C-SPY system macro)	484
__setDataBreak (C-SPY system macro)	485
__setSimBreak (C-SPY system macro)	486
settings (directory)	24
Setup macros (C-SPY option)	430
setup macros, in C-SPY. <i>See</i> C-SPY macros	
Setup (C-SPY options)	429
SFR, in header files	21
shortcut keys	101
Show Bookmarks (editor option)	319
Show Line Number (editor option)	318
Show right margin (editor option)	318
Show timestamp (Trace Setup option)	239
--silent (C-SPY command line option)	458
simulating interrupts, enabling/disabling	189
Simulator menu	166
simulator, features	10
size optimization	391
Size (Breakpoints dialog)	181, 248–249, 284
sizeof	128
__sleep (C-SPY system macro)	487
Source Browser window	280
context menu	281
using	87
source code	
including in compiler list file	394
templates	103
Source code color in Disassembly window (IDE option)	329
Source Code Control context menu	269
source code control systems	88
Source Code Control (IDE Options dialog box)	327
source code control, features	4
source file paths	85, 278
source files	
adding to a project	31
editing	99
managing in projects	83
__sourcePosition (C-SPY system macro)	487

- special function registers (SFR)
 - description files 118
 - header files 21
 - using as assembler symbols 128
- speed optimization 391
- src, arm (subdirectory) 21
- Stack window 368
 - using 145
- Stack (IDE Options dialog box) 330
- stack.mac 149
- Stall processor on FIFO full (Trace Setup option). 239
- Start address (linker option) 424
- Statics window 360
 - context menu 361
- status bar 266
- stdin and stdout
 - redirecting to C-SPY window 126
 - redirecting to file 126
- Step Into 345
 - example of 43, 122
- Step into functions (IDE option). 329
- Step Out 345
 - example of 123
- Step Over 345
 - example of 123
- stepping 122
 - example 42
- STL container expansion (IDE option) 329
- Stop build operation on (IDE Project options) 326
- Stop Debugging (button). 345
- __strFind (C-SPY system macro) 487
- Strict ISO/ANSI (compiler option) 389
- strings, text style in editor. 101
- __subString (C-SPY system macro) 488
- support, technical 26
- Suppress download (C-SPY Download option) 215
- Suppress download (C-SPY option) 165
- Suppress these diagnostics (compiler option) 397
- Suppress these diagnostics (linker option) 422

- SWD interface 224
 - information in Trace window 241
- SWO clock (C-SPY J-Link option). 227
- SWO communication channel
 - enabling 224
- SWO communication, for timestamps in trace 227
- Symbolic Memory window. 354
 - context menu 355
 - toolbar 354
- symbols
 - See also* user symbols
 - defining in assembler 406
 - defining in compiler 396
 - defining in linker 421
 - using in C-SPY expressions 127
- Symbols window 372
 - context menu 372
- syntax coloring
 - configuration files 20
 - in editor 101
- Syntax Coloring (Editor colors and fonts option) 323
- Syntax Highlighting (editor option) 318
- syntax highlighting, in editor window. 102
- system macros. 459

T

- Tab Key Function (editor option) 317
- Tab Size (editor option) 317
- Tab spacing (assembler option). 405
- target options
 - Big endian 380
- target system, definition of 112
- Target (general options) 379
- \$TARGET_BNAME\$ (argument variable). 307
- \$TARGET_BPATH\$ (argument variable). 307
- \$TARGET_DIR\$ (argument variable) 307
- \$TARGET_FNAME\$ (argument variable) 307
- \$TARGET_PATH\$ (argument variable) 307

TCPIP address or hostname (C-SPY GDB Server option)	218
TCP/IP (Angel C-SPY option)	217
TCP/IP (C-SPY Macraigor option)	231
technical support	26
Template dialog box (Edit menu)	301
Terminal IO Log File	126
Terminal I/O Log File dialog box (Debug menu)	378
Terminal I/O window	126, 364
example of using	49
Terminal I/O (IDE Options dialog box)	333
terminal output, buffered	384
terminology	xlii
testing, of code	94
Texas Instruments TMS470 flash loader	260
Third Party Driver (C-SPY options)	236
Thumb code, mixing with ARM code	390
Thumb (compiler option)	390
Toggle Bookmark (button)	265
Toggle Breakpoint (button)	265
toggle breakpoint, example	45, 65
__toLower (C-SPY system macro)	488
tool chain	
extending	95
specifying	30
Tool Output window	289
context menu	290
toolbar	
debug	345
IDE	265
Trace	171, 242
\$TOOLKIT_DIR\$ (argument variable)	307
tools icon, in this guide	xlii
Tools menu	313
tools, user-configured	334
__toString (C-SPY system macro)	489
touch, open-source command line utility	94
__toUpper (C-SPY system macro)	489
Trace buffer size (Trace Setup option)	239
Trace Expressions window	173

Trace port mode (Trace Setup option)	238
Trace port width (Trace Setup option)	238
Trace Save dialog box	226, 240
Trace Setup dialog box	238
Trace toolbar	242
Trace window	170
toolbar	171
Trace (RDI window)	240
trace, definition of	131
trademarks	ii
transformations, enabled in compiler	392
Treat all warnings as errors (compiler option)	398
Treat all warnings as errors (linker option)	423
Treat these as errors (compiler option)	398
Treat these as errors (linker option)	423
Treat these as remarks (compiler option)	397
Treat these as remarks (linker option)	422
Treat these as warnings (compiler option)	398
Treat these as warnings (linker option)	423
tutor, arm (subdirectory)	21
Type (External editor option)	321
type-checking	15
typographic conventions	xlii

U

Undo (button)	265
Use Code Templates (editor option)	322
Use Custom Keyword File (editor option)	322
Use External Editor (External editor option)	321
Use register filter (Register filter option)	332
user application, definition of	112
User symbols are case sensitive (assembler option)	401

V

variables	
effects of optimizations	129
information, limitation on	129

using in arguments	335
using in C-SPY expressions	127
watching in C-SPY	130
example	44
variance (interrupt property)	191
definition of	186
Vector Catch dialog box (JTAG menu)	251
Verify download (C-SPY Download option)	215
version control systems	88
version number, of Embedded Workbench	340
View menu	302
visualSTATE, plugin module for	8
vsp (filename extension)	24

W

Warn when exceeding stack threshold (Stack option)	331
Warn when stack pointer is out of bounds (Stack option) .	331
warnings	
compiler	398
linker	423
warnings icon, in this guide	xlii
Watch window	356
context menu	357
using	130
watchpoints	
JTAG	251
J-Link watchpoints	252
setting	44
Watchpoints (J-Link menu)	252
web sites, recommended	xli
web site, IAR Systems	26
When source resolves to multiple function instances	328
while (macro statement)	461
Window menu	339
windows	263
organizing on the screen	77
specific to C-SPY	343

Workspace window	266
context menu	268, 282
drag-and-drop of files	84
example	30
workspaces	
creating	29, 84
using	83
__writeFile (C-SPY system macro)	490
__writeFileByte (C-SPY system macro)	490
__writeMemoryByte (C-SPY system macro)	491
__writeMemory16 (C-SPY system macro)	491
__writeMemory32 (C-SPY system macro)	491
__writeMemory8 (C-SPY system macro)	491
wsdt (filename extension)	24
www.iar.com	26

X

xcl (filename extension)	24
------------------------------------	----

Z

zone, in C-SPY	143
--------------------------	-----

Symbols

#define statement, in compiler	396
#define (linker option)	421
#line directives, generating in compiler	396
\$CONFIG_NAME\$ (argument variable)	306
\$CUR_DIR\$ (argument variable)	306
\$CUR_LINE\$ (argument variable)	306
\$EW_DIR\$ (argument variable)	306
\$EXE_DIR\$ (argument variable)	306
\$FILE_DIR\$ (argument variable)	306
\$FILE_FNAME\$ (argument variable)	306
\$FILE_PATH\$ (argument variable)	306
\$LIST_DIR\$ (argument variable)	306
\$OBJ_DIR\$ (argument variable)	306

\$PROJ_DIR\$ (argument variable)	307	--jlink_speed (C-SPY command line option)	449
\$PROJ_FNAME\$ (argument variable)	307	--lmiftdi_speed (C-SPY command line option)	450
\$PROJ_PATH\$ (argument variable)	307	--macro (C-SPY command line option)	453–454
\$TARGET_BNAME\$ (argument variable)	307	--mac_handler_address (C-SPY command line option)	450
\$TARGET_BPATH\$ (argument variable)	307	--mac_interface (C-SPY command line option)	450
\$TARGET_DIR\$ (argument variable)	307	--mac_jtag_device (C-SPY command line option)	451
\$TARGET_FNAME\$ (argument variable)	307	--mac_multiple_targets (C-SPY command line option)	451
\$TARGET_PATH\$ (argument variable)	307	--mac_reset_pulls_reset (C-SPY command line option)	452
\$TOOLKIT_DIR\$ (argument variable)	307	--mac_set_temp_reg_buffer (C-SPY command line option)	452
% stack usage threshold (Stack option)	330	--mac_speed (C-SPY command line option)	453
-B (C-SPY command line option)	438	--mac_xscale_ir7 (C-SPY command line option)	453
-d (C-SPY command line option)	439	--mapu (C-SPY command line option)	454
-p (C-SPY command line option)	454	--proc_stack_xxx (C-SPY command line option)	455
--backend (C-SPY command line option)	438–439	--rdi_allow_hardware_reset (C-SPY command line option)	456
--BE32 (C-SPY command line option)	435	--rdi_driver_dll (C-SPY command line option)	456
--BE8 (C-SPY command line option)	435	--rdi_heartbeat (C-SPY command line option)	436
--cpu (C-SPY command line option)	435	--rdi_step_max_one (C-SPY command line option)	457
--device (C-SPY command line option)	440	--rdi_use_etm (C-SPY command line option)	456
--drv_attach_to_program (C-SPY command line option)	435	--semihosting (C-SPY command line option)	457
--drv_catch_exceptions (C-SPY command line option)	440	--silent (C-SPY command line option)	458
--drv_communication (C-SPY command line option)	441	__cancelAllInterrupts (C-SPY system macro)	467
--drv_communication_log (C-SPY command line option)	443	__cancelInterrupt (C-SPY system macro)	467
--drv_default_breakpoint (C-SPY command line option)	443	__clearBreak (C-SPY system macro)	468
--drv_reset_to_cpu_start (C-SPY command line option)	444	__closeFile (C-SPY system macro)	468
--drv_restore_breakpoints (C-SPY command line option)	444	__disableInterrupts (C-SPY system macro)	468
--drv_suppress_download (C-SPY command line option)	436	__driverType (C-SPY system macro)	469
--drv_vector_table_base (C-SPY command line option)	445	__emulatorSpeed (C-SPY system macro)	469
--endian (C-SPY command line option)	436	__emulatorStatusCheckOnRead (C-SPY system macro)	470
--flash_loader (C-SPY command line option)	445	__enableInterrupts (C-SPY system macro)	471
--fpu (C-SPY command line option)	436	__evaluate (C-SPY system macro)	471
--gdbserv_exec_command (C-SPY command line option)	446	__fmessage (C-SPY macro statement)	462
--generate_sim (C-SPY command line option)	446	__gdbserver_exec_command (C-SPY system macro)	472
--jlink_device_select (C-SPY command line option)	446	__hwReset (C-SPY system macro)	472
--jlink_exec_command (C-SPY command line option)	447	__hwResetWithStrategy (C-SPY system macro)	473
--jlink_initial_speed (C-SPY command line option)	447	__jlinkExecCommand (C-SPY system macro)	473
--jlink_interface (C-SPY command line option)	448	__jtagCommand (C-SPY system macro)	474
--jlink_ir_length (C-SPY command line option)	448	__jtagCP15IsPresent (C-SPY system macro)	474
--jlink_reset_strategy (C-SPY command line option)	448		

__jtagCP15ReadReg (C-SPY system macro)	474
__jtagCP15WriteReg (C-SPY system macro)	475
__jtagData (C-SPY system macro)	475
__jtagRawRead (C-SPY system macro)	476
__jtagRawSync (C-SPY system macro)	476
__jtagRawWrite (C-SPY system macro)	477
__jtagResetTRST (C-SPY system macro)	478
__message (C-SPY macro statement)	462
__openFile (C-SPY system macro)	478
__orderInterrupt (C-SPY system macro)	479
__popSimulatorInterruptExecutingStack (C-SPY system macro)	480
__readFile (C-SPY system macro)	480
__readFileByte (C-SPY system macro)	481
__readMemoryByte (C-SPY system macro)	481
__readMemory8 (C-SPY system macro)	481
__readMemory16 (C-SPY system macro)	482
__readMemory32 (C-SPY system macro)	482
__registerMacroFile (C-SPY system macro)	482
__resetFile (C-SPY system macro)	483
__restoreSoftwareBreakpoint (C-SPY system macro)	483
__setCodeBreak (C-SPY system macro)	484
__setDataBreak (C-SPY system macro)	485
__setSimBreak (C-SPY system macro)	486
__sleep (C-SPY system macro)	487
__smessage (C-SPY macro statement)	462
__sourcePosition (C-SPY system macro)	487
__strFind (C-SPY system macro)	487
__subString (C-SPY system macro)	488
__toLower (C-SPY system macro)	488
__toString (C-SPY system macro)	489
__toUpper (C-SPY system macro)	489
__writeFile (C-SPY system macro)	490
__writeFileByte (C-SPY system macro)	490
__writeMemoryByte (C-SPY system macro)	491
__writeMemory8 (C-SPY system macro)	491
__writeMemory16 (C-SPY system macro)	491
__writeMemory32 (C-SPY system macro)	491