

# **MSP430 IAR Embedded Workbench™ IDE**

## User Guide

for Texas Instruments'  
**MSP430 Microcontroller Family**

## **COPYRIGHT NOTICE**

© Copyright 1996–2004 IAR Systems. All rights reserved.

No part of this document may be reproduced without the prior written consent of IAR Systems. 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, IAR Embedded Workbench, IAR XLINK Linker, IAR XAR Library Builder, IAR XLIB Librarian, IAR MakeApp, and IAR PreQual are trademarks owned by IAR Systems. C-SPY is a trademark registered in Sweden by IAR Systems. IAR visualSTATE is a registered trademark owned by IAR Systems.

Texas Instruments is a registered trademark of Texas Instruments Incorporated.

Microsoft and Windows are registered trademarks of Microsoft Corporation. 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**

Third edition: March 2004

Part number: U430-3

This guide describes version 3.x of the IAR Embedded Workbench for Texas Instruments' MSP430 microcontroller family.

# Brief contents

Tables .....	xix
Figures .....	xxiii
Preface .....	xxix
<b>Part 1. Product overview .....</b>	<b>1</b>
Product introduction .....	3
Installed files .....	15
<b>Part 2. Tutorials .....</b>	<b>23</b>
Creating an application project .....	25
Debugging using the IAR C-SPY™ Debugger .....	37
Mixing C and assembler modules .....	49
Using C++ .....	53
Simulating an interrupt .....	59
Working with library modules .....	69
<b>Part 3. Project management and building .....</b>	<b>73</b>
The development environment .....	75
Managing projects .....	81
Building .....	89
Editing .....	95
<b>Part 4. Debugging .....</b>	<b>101</b>
The IAR C-SPY Debugger .....	103

Executing your application .....	111
Working with variables and expressions .....	117
Using breakpoints .....	123
Monitoring memory and registers .....	129
Using C-SPY macros .....	135
Analyzing your application .....	143
<b>Part 5. IAR C-SPY Simulator</b> .....	149
Simulator introduction .....	151
Simulator-specific characteristics .....	153
Simulating interrupts .....	159
<b>Part 6. IAR C-SPY FET Debugger</b> .....	171
Introduction to the IAR C-SPY FET Debugger .....	173
C-SPY FET driver-specific characteristics .....	179
C-SPY FET Debugger options and menus .....	189
Design considerations for in-circuit programming .....	209
<b>Part 7. Reference information</b> .....	215
IAR Embedded Workbench IDE reference .....	217
C-SPY Debugger IDE reference .....	271
General options .....	301
Compiler options .....	309
Assembler options .....	319
Custom build options .....	327

<b>XLINK options .....</b>	329
<b>Library builder options .....</b>	341
<b>Debugger options .....</b>	343
<b>C-SPY macros reference .....</b>	347
<b>Glossary .....</b>	367
<b>Index .....</b>	379



# Contents

Tables .....	ix
Figures .....	xxiii
Preface .....	xxix
<b>Who should read this guide</b> .....	xxix
<b>How to use this guide</b> .....	xxix
<b>What this guide contains</b> .....	xxx
<b>Other documentation</b> .....	xxxiii
<b>Document conventions</b> .....	xxxiii
<b>Part I. Product overview</b> .....	1
Product introduction .....	3
<b>The IAR Embedded Workbench IDE</b> .....	3
An extensible and modular environment .....	4
Features .....	4
Documentation .....	5
<b>IAR C-SPY Debugger</b> .....	5
General C-SPY Debugger features .....	6
RTOS awareness .....	8
Documentation .....	8
<b>IAR C-SPY Debugger systems</b> .....	8
IAR C-SPY Simulator .....	8
IAR C-SPY FET Debugger .....	9
<b>IAR C/C++ Compiler</b> .....	9
Features .....	10
Runtime environment .....	10
Documentation .....	11
<b>IAR Assembler</b> .....	11
Features .....	11
Documentation .....	11

<b>IAR XLINK Linker</b>	11
Features	12
Documentation	12
<b>IAR XAR Library Builder</b>	12
Features	13
Documentation	13
<b>Installed files</b>	15
<b>Directory structure</b>	15
Root directory	15
The 430 directory	16
The common directory	17
<b>File types</b>	18
<b>Documentation</b>	20
The user and reference guides	20
Online help	21
IAR on the web	21
<b>Part 2. Tutorials</b>	23
<b>Creating an application project</b>	25
<b>Setting up a new project</b>	25
Creating a workspace window	26
Creating the new project	27
Adding files to the project	28
Setting project options	29
<b>Compiling and linking the application</b>	32
Compiling the source files	32
Viewing the list file	33
Linking the application	35
Viewing the map file	36
<b>Debugging using the IAR C-SPY™ Debugger</b>	37
<b>Debugging the application</b>	37
Starting the debugger	37

Organizing the windows .....	37
Inspecting source statements .....	38
Inspecting variables .....	40
Setting and monitoring breakpoints .....	42
Debugging in disassembly mode .....	43
Monitoring registers .....	44
Monitoring memory .....	44
Viewing terminal I/O .....	46
Reaching program exit .....	46
<b>Mixing C and assembler modules .....</b>	<b>49</b>
<b>Examining the calling convention .....</b>	<b>49</b>
<b>Adding an assembler module to the project .....</b>	<b>50</b>
Setting up the project .....	50
<b>Using C++ .....</b>	<b>53</b>
<b>Creating a C++ application .....</b>	<b>53</b>
Compiling and linking the C++ application .....	53
Setting a breakpoint and executing to it .....	54
Printing the Fibonacci numbers .....	56
<b>Simulating an interrupt .....</b>	<b>59</b>
<b>Adding an interrupt handler .....</b>	<b>59</b>
The application—a brief description .....	59
Writing an interrupt handler .....	60
Setting up the project .....	60
<b>Setting up the simulation environment .....</b>	<b>60</b>
Defining a C-SPY setup macro file .....	61
Specifying C-SPY options .....	62
Building the project .....	63
Starting the simulator .....	63
Specifying a simulated interrupt .....	63
Setting an immediate breakpoint .....	64
<b>Simulating the interrupt .....</b>	<b>66</b>
Executing the application .....	66

<b>Using macros for interrupts and breakpoints .....</b>	67
<b>Working with library modules .....</b>	69
<b>Using libraries .....</b>	69
Creating a new project .....	70
Creating a library project .....	70
Using the library in the project .....	71
<b>Part 3. Project management and building .....</b>	73
<b>The development environment .....</b>	75
<b>The IAR Embedded Workbench IDE .....</b>	75
Running the IAR Embedded Workbench .....	76
Exiting .....	77
<b>Customizing the environment .....</b>	77
Organizing the windows on the screen .....	77
Customizing the IDE .....	78
Communicating with external tools .....	79
<b>Managing projects .....</b>	81
<b>The project model .....</b>	81
How projects are organized .....	81
Creating and managing workspaces .....	84
<b>Navigating project files .....</b>	85
Viewing the workspace .....	86
Displaying source browse information .....	87
<b>Building .....</b>	89
<b>Building your application .....</b>	89
Setting options .....	89
Building a project .....	91
Building multiple configurations in a batch .....	91
Correcting errors found during build .....	91
Building from the command line .....	92

<b>Extending the tool chain .....</b>	92
Tools that can be added to the tool chain .....	93
Adding an external tool .....	93
<b>Editing .....</b>	95
<b>Using the IAR Embedded Workbench editor .....</b>	95
Editing a file .....	95
Searching .....	98
<b>Customizing the editor environment .....</b>	98
Using an external editor .....	99
<b>Part 4. Debugging .....</b>	101
<b>The IAR C-SPY Debugger .....</b>	103
<b>Debugger concepts .....</b>	103
IAR C-SPY Debugger and target systems .....	103
Debugger .....	104
Target system .....	104
User application .....	104
IAR C-SPY Debugger systems .....	105
ROM-monitor program .....	105
Third-party debuggers .....	105
<b>The C-SPY environment .....</b>	105
An integrated environment .....	105
<b>Setting up the IAR C-SPY Debugger .....</b>	106
Choosing a debug driver .....	106
Executing from reset .....	107
Using a setup macro file .....	107
Selecting a device description file .....	107
Loading plugin modules .....	108
<b>Starting the IAR C-SPY Debugger .....</b>	108
Redirecting debugger output to a file .....	109
<b>Adapting C-SPY to target hardware .....</b>	109
Device description file .....	109

<b>Executing your application</b> .....	111
<b>Source and disassembly mode debugging</b> .....	111
<b>Executing</b> .....	112
Step .....	112
Go .....	114
Run to Cursor .....	114
Highlighting .....	114
Using breakpoints to stop .....	114
Using the Break button to stop .....	115
Stop at program exit .....	115
<b>Call stack information</b> .....	115
<b>Terminal input and output</b> .....	116
<b>Working with variables and expressions</b> .....	117
<b>C-SPY expressions</b> .....	117
C symbols .....	117
Assembler symbols .....	118
Macro functions .....	118
Macro variables .....	118
<b>Limitations on variable information</b> .....	119
Effects of optimizations .....	119
<b>Viewing variables and expressions</b> .....	120
Working with the windows .....	120
<b>Using breakpoints</b> .....	123
<b>The breakpoint system</b> .....	123
<b>Defining breakpoints</b> .....	124
Toggling a simple code breakpoint .....	124
Setting a breakpoint in the Memory window .....	124
Defining breakpoints using the dialog box .....	125
Defining breakpoints using system macros .....	126
<b>Monitoring memory and registers</b> .....	129
<b>Memory addressing</b> .....	129
<b>Using the Memory window</b> .....	130

<b>Working with registers</b>	132
Register groups	132
<b>Using the Stack window</b>	134
<b>Using C-SPY macros</b>	135
<b>The macro system</b>	135
The macro language	136
The macro file	136
Setup macro functions	137
<b>Using C-SPY macros</b>	137
Using the Macro Configuration dialog box	138
Registering and executing using setup macros and setup files	139
Executing macros using Quick Watch	140
Executing a macro by connecting it to a breakpoint	141
<b>Analyzing your application</b>	143
<b>Function-level profiling</b>	143
Using the profiler	143
<b>Code coverage</b>	146
Using Code Coverage	146
<b>Part 5. IAR C-SPY Simulator</b>	149
<b>Simulator introduction</b>	151
<b>The IAR C-SPY Simulator</b>	151
Features	151
Selecting the simulator driver	151
<b>Simulator-specific characteristics</b>	153
<b>Simulator-specific menus</b>	153
Simulator menu	153
<b>Memory configuration</b>	154
Memory map	154
<b>Using breakpoints</b>	156
Size attribute on code and data breakpoints	156

Immediate breakpoints .....	156
<b>Hardware multiplier</b> .....	158
Simulating interrupts .....	159
<b>The C-SPY interrupt simulation system</b> .....	159
Interrupt characteristics .....	160
<b>Using the interrupt simulation system</b> .....	161
Adapting the interrupt system for MSP430 .....	161
Interrupts dialog box .....	162
Interrupt Setup dialog box .....	163
Forced interrupt window .....	164
C-SPY system macros for interrupt .....	165
Interrupt Log window .....	166
<b>Simulating a simple interrupt</b> .....	166
<b>Description of interrupt system macros</b> .....	168
<b>Part 6. IAR C-SPY FET Debugger</b> .....	171
Introduction to the IAR C-SPY FET Debugger .....	173
<b>The FET C-SPY Debugger</b> .....	173
Supported devices .....	174
Differences between the C-SPY FET and the simulator drivers .....	174
<b>Hardware installation</b> .....	175
Hardware Installation, MSP-FET430X110 .....	175
Hardware Installation, MSP-FET430Pxx0 .....	175
<b>Getting started</b> .....	176
Running a demo application .....	176
C-SPY FET driver-specific characteristics .....	179
<b>Using breakpoints</b> .....	179
Hardware and virtual breakpoints .....	179
System breakpoints .....	180
Customizing the use of breakpoints .....	181
<b>Stepping</b> .....	182
Programming flash .....	182

Single-stepping with active interrupts .....	182
<b>Using state storage</b> .....	182
<b>Using the sequencer</b> .....	184
<b>Memory configuration</b> .....	186
<b>C-SPY FET communication</b> .....	186
Releasing JTAG .....	186
Parallel port designators .....	186
Troubleshooting .....	187
<b>C-SPY FET Debugger options and menus</b> .....	189
<b>Setup options</b> .....	189
<b>FET Debugger setup</b> .....	190
Verify download .....	190
Marginal read check .....	190
Download control .....	191
Parallel Port .....	191
Use virtual breakpoints .....	191
System breakpoints on .....	191
<b>Range breakpoints</b> .....	192
Setting a range breakpoint in the Breakpoints dialog box .....	192
Using a system macro to set a range breakpoint .....	194
<b>Conditional breakpoints</b> .....	196
Setting a conditional breakpoint in the Breakpoints dialog box .....	196
Using a macro to set a conditional breakpoint .....	199
<b>Emulator menu</b> .....	201
State Storage Control window .....	203
State Storage Window .....	205
Sequencer Control window .....	206
<b>Design considerations for in-circuit programming</b> .....	209
<b>Bootstrap loader</b> .....	209
<b>Device signals</b> .....	209
<b>External power</b> .....	210
<b>Signal connections for in-system programming</b> .....	210
MSP-FET430X110 .....	210

MSP-FET430Pxx0 ('P120, 'P140, 'P410, 'P440) .....	212
<b>Part 7. Reference information</b> .....	215
<b>IAR Embedded Workbench IDE reference</b> .....	217
<b>Windows</b> .....	217
IAR Embedded Workbench IDE window .....	218
Workspace window .....	220
Editor window .....	222
Source Browser window .....	226
Build window .....	227
Find in Files window .....	228
Tool Output window .....	229
Debug Log window .....	230
<b>Menus</b> .....	230
File menu .....	231
Edit menu .....	232
View menu .....	242
Project menu .....	243
Tools menu .....	251
Window menu .....	267
Help menu .....	268
<b>C-SPY Debugger IDE reference</b> .....	271
<b>C-SPY windows</b> .....	271
Editing in C-SPY windows .....	271
IAR C-SPY Debugger window .....	272
Disassembly window .....	273
Memory window .....	276
Register window .....	279
Watch window .....	280
Locals window .....	281
Auto window .....	282
Live Watch window .....	282

Call Stack window .....	283
Terminal I/O window .....	285
Code Coverage window .....	286
Profiling window .....	287
Trace window .....	289
Stack window .....	292
<b>C-SPY menus .....</b>	293
Debug menu .....	294
Simulator menu .....	299
Emulator menu .....	299
<b>General options .....</b>	301
<b>Compiler options .....</b>	309
<b>Assembler options .....</b>	319
<b>Custom build options .....</b>	327
<b>XLINK options .....</b>	329
<b>Library builder options .....</b>	341
<b>Debugger options .....</b>	343
<b>C-SPY macros reference .....</b>	347
<b>The macro language .....</b>	347
Macro functions .....	347
Predefined system macro functions .....	347
Macro variables .....	348
Macro statements .....	348
<b>Setup macro functions summary .....</b>	350
<b>C-SPY system macros summary .....</b>	351
<b>Description of C-SPY system macros .....</b>	352
<b>Glossary .....</b>	367
<b>Index .....</b>	379



# Tables

1: Typographic conventions used in this guide .....	xxxiii
2: File types .....	18
3: General settings for project1 .....	30
4: Compiler options for project1 .....	31
5: XLINK options for project1 .....	35
6: Compiler options for project2 .....	50
7: Project options for C++ tutorial .....	54
8: Interrupts dialog box .....	63
9: Breakpoints dialog box .....	65
10: XLINK options for project 5 .....	70
11: XLINK options for library project .....	70
12: Command shells .....	80
13: C-SPY assembler symbols expressions .....	118
14: Handling name conflicts between hardware registers and assembler labels .....	118
15: Project options for enabling profiling .....	143
16: Project options for enabling code coverage .....	146
17: Description of Simulator menu commands .....	153
18: __setSimBreak return values .....	158
19: Characteristics of a forced interrupt .....	165
20: Description of the Interrupt Log window .....	166
21: Timer interrupt settings .....	167
22: __cancelInterrupt return values .....	168
23: __disableInterrupts return values .....	169
24: __enableInterrupts return values .....	169
25: Simulator and FET differences .....	174
26: Project options for FET C example .....	176
27: Project options for FET assembler example .....	177
28: Available hardware breakpoints .....	179
29: Sequencer settings - example .....	185
30: State Storage Control settings - example .....	185
31: Range breakpoint start value types .....	193

32: Range breakpoint types .....	194
33: Range breakpoint access types .....	194
34: __setRangeBreak return values .....	195
35: Conditional break at location types .....	197
36: Conditional breakpoint types .....	198
37: Conditional breakpoint condition operators .....	198
38: Conditional breakpoint access types .....	198
39: Conditional breakpoint condition types .....	199
40: __setConditionalBreak return values .....	200
41: Emulator menu commands .....	201
42: Columns in State Storage window .....	205
43: IAR Embedded Workbench IDE menu bar .....	218
44: Workspace window context menu commands .....	221
45: Editor keyboard commands for insertion point navigation .....	224
46: Editor keyboard commands for scrolling .....	225
47: Editor keyboard commands for selecting text .....	225
48: Columns in Source Browser window .....	226
49: File menu commands .....	231
50: Edit menu commands .....	233
51: Find dialog box options .....	234
52: Replace dialog box options .....	234
53: Find in Files dialog box options .....	235
54: Break At Location types .....	240
55: Memory Access types .....	240
56: Breakpoint conditions .....	241
57: View menu commands .....	242
58: Project menu commands .....	243
59: Argument variables .....	244
60: Configurations for project dialog box options .....	245
61: New Configuration dialog box options .....	246
62: Description of Create New Project dialog box .....	247
63: Project option categories .....	248
64: Description of the Batch Build dialog box .....	249
65: Description of the Edit Batch Build dialog box .....	250

66:	Tools menu commands .....	251
67:	Key Bindings page options .....	253
68:	External Editor options .....	254
69:	Messages page options .....	256
70:	Editor page options .....	256
71:	Editor Colors and Fonts page options .....	258
72:	Project page options .....	259
73:	Debugger page options .....	260
74:	Register Filter options .....	262
75:	Register Filter options .....	262
76:	Configure Tools dialog box options .....	264
77:	Command shells .....	265
78:	Window menu commands .....	267
79:	Help menu commands .....	268
80:	Editing in C-SPY windows .....	271
81:	C-SPY menu .....	273
82:	Disassembly window operations .....	274
83:	Disassembly context menu commands .....	275
84:	Memory window operations .....	276
85:	Memory window operations .....	277
86:	Fill dialog box options .....	278
87:	Memory fill operations .....	278
88:	Profiling window columns .....	289
89:	Trace toolbar commands .....	291
90:	Trace context commands .....	291
91:	Stack window operations .....	292
92:	Stack window settings .....	293
93:	Debug menu commands .....	294
94:	Log file options .....	298
95:	Libraries .....	304
96:	Compiler list file options .....	315
97:	Assembler list file options .....	322
98:	XLINK range check options .....	335
99:	XLINK list file options .....	336

100: XLINK list file format options .....	337
101: XLINK checksum algorithms .....	339
102: C-SPY driver options .....	344
103: Examples of C-SPY macro variables .....	348
104: C-SPY setup macros .....	350
105: Summary of system macros .....	351
106: __driverType return values .....	354
107: __openFile return values .....	354
108: __setCodeBreak return values .....	359
109: __setDataBreak return values .....	360

# Figures

1: Directory structure .....	15
2: New dialog box .....	26
3: New Workspace dialog box .....	26
4: Create New Project dialog box .....	27
5: Workspace window .....	28
6: Adding files to project1 .....	29
7: Setting general options .....	30
8: Setting compiler options .....	31
9: Compilation message .....	32
10: Workspace window after compilation .....	33
11: Setting the option Scan for Changed Files .....	34
12: XLINK options dialog box for project1 .....	35
13: The C-SPY Debugger main window .....	38
14: Stepping in C-SPY .....	39
15: Using Step Into in C-SPY .....	40
16: Inspecting variables in the Auto window .....	41
17: Watching variables in the Watch window .....	41
18: Setting breakpoints .....	42
19: Debugging in disassembly mode .....	43
20: Register window .....	44
21: Monitoring memory .....	45
22: Displaying memory contents as 16-bit units .....	45
23: Output from the I/O operations .....	46
24: Reaching program exit in C-SPY .....	46
25: Assembler settings for creating a list file .....	51
26: Setting a breakpoint in CPPtutor.cpp .....	54
27: Setting the breakpoint condition .....	55
28: Inspecting the function calls .....	56
29: Printing Fibonacci sequences .....	57
30: Specifying setup macro file .....	62
31: Inspecting the interrupt settings .....	64

32: Displaying breakpoint information .....	65
33: Printing the Fibonacci values in the Terminal I/O window .....	66
34: IAR Embedded Workbench IDE window .....	76
35: Configure Tools dialog box .....	79
36: Customized Tools menu .....	80
37: Examples of workspaces and projects .....	82
38: Displaying a project in the workspace window .....	86
39: Workspace window—an overview .....	87
40: General options .....	90
41: Editor window .....	95
42: Editor window status bar .....	98
43: Specifying external command line editor .....	99
44: External editor DDE settings .....	100
45: IAR C-SPY Debugger and target systems .....	104
46: Quick Watch dialog box .....	121
47: Trace window (Expression page) .....	122
48: Trace window (Output page) .....	122
49: Breakpoint on a function call .....	124
50: Breakpoints dialog box .....	125
51: Zones in C-SPY .....	129
52: Memory window .....	130
53: Memory Fill dialog box .....	131
54: Register window .....	132
55: Register Filter page .....	133
56: Measuring the stack .....	134
57: Macro Configuration dialog box .....	139
58: Quick Watch dialog box .....	141
59: Profiling window .....	144
60: Graphs in Profiling window .....	145
61: Function details window .....	145
62: Code Coverage window .....	147
63: Simulator menu .....	153
64: Breakpoint Usage dialog box .....	154
65: Memory Map dialog box .....	155

66: Immediate breakpoints page .....	157
67: Simulated interrupt configuration .....	160
68: Interrupts dialog box .....	162
69: Interrupt Setup dialog box .....	163
70: Forced Interrupt window .....	164
71: Interrupt Log window .....	166
72: Communication overview .....	174
73: The Used Breakpoints window .....	181
74: Flash Emulation Tool options .....	190
75: Range breakpoints dialog box .....	192
76: Conditional breakpoints dialog box .....	196
77: Emulator menu .....	201
78: State Storage Control window .....	203
79: State Storage window .....	205
80: Sequencer Control window .....	207
81: JTAG signal connection (MSP-FET430X110) .....	211
82: JTAG signal connection (MSP-FET430Pxx0) .....	213
83: IAR Embedded Workbench IDE window .....	218
84: IAR Embedded Workbench IDE toolbar .....	219
85: IAR Embedded Workbench IDE window status bar .....	220
86: Project window .....	220
87: Workspace window context menu .....	221
88: Editor window .....	222
89: Editor window context menu .....	223
90: Source Browser window .....	226
91: Build window (message window) .....	227
92: Build window context menu .....	227
93: Find in Files window (message window) .....	228
94: Find in Files window context menu .....	228
95: Tool Output window (message window) .....	229
96: Find in Files window context menu .....	229
97: Debug Log window (message window) .....	230
98: Debug Log window context menu .....	230
99: File menu .....	231

100: Edit menu .....	232
101: Find in Files dialog box .....	235
102: Breakpoints dialog box .....	237
103: Code breakpoints page .....	238
104: Data breakpoints page .....	239
105: Enter Location dialog box .....	239
106: View menu .....	242
107: Project menu .....	243
108: Configurations for project dialog box .....	245
109: New Configurations dialog box .....	246
110: Create New Project dialog box .....	247
111: Batch Build dialog box .....	249
112: Edit Batch Build dialog box .....	250
113: Tools menu .....	251
114: Common Fonts page .....	252
115: Key Bindings page .....	253
116: External Editor page with command line settings .....	254
117: Messages page .....	255
118: Editor page .....	256
119: Editor Colors and Fonts page .....	258
120: Projects page .....	259
121: Debugger page .....	260
122: Register Filter page .....	261
123: Terminal I/O page .....	262
124: Configure Tools dialog box .....	263
125: Customized Tools menu .....	264
126: Filename Extensions dialog box .....	265
127: Filename Extension Overrides dialog box .....	266
128: Edit Filename Extensions dialog box .....	266
129: Window menu .....	267
130: Embedded Workbench Startup dialog box .....	269
131: Embedded Workbench with C-SPY Simulator started .....	272
132: C-SPY debug toolbar .....	273
133: C-SPY Disassembly window .....	274

134: Disassembly window context menu .....	275
135: Memory window .....	276
136: Memory window context menu .....	277
137: Fill dialog box .....	278
138: Register window .....	279
139: Watch window .....	280
140: Watch window context menu .....	280
141: Locals window .....	281
142: C-SPY Locals window context menu .....	281
143: Auto window .....	282
144: Live Watch window .....	282
145: C-SPY Live Watch window context menu .....	283
146: Call Stack window .....	283
147: Call Stack window context menu .....	284
148: Terminal I/O window .....	285
149: Ctrl codes menu .....	285
150: Change Input Mode dialog box .....	285
151: Code Coverage window .....	286
152: Code coverage context menu .....	286
153: Profiling window .....	288
154: Profiling context menu .....	288
155: Trace window (Expression page) .....	290
156: Trace window (Output page) .....	290
157: Trace window context menu .....	291
158: Stack window .....	292
159: Stack Settings dialog box .....	293
160: Debug menu .....	294
161: Quick Watch dialog box .....	295
162: Autostep settings dialog box .....	296
163: Macro Configuration dialog box .....	297
164: Log File dialog box .....	298
165: Terminal I/O Log File dialog box .....	299
166: Target options .....	301
167: Output options .....	302

168: Library Configuration options .....	304
169: Library Options page .....	306
170: Stack/Heap page .....	307
171: Compiler language options .....	309
172: Compiler code options .....	312
173: Compiler output options .....	314
174: Compiler list file options .....	315
175: Compiler preprocessor options .....	316
176: Compiler diagnostics options .....	317
177: Assembler language options .....	319
178: Choosing macro quote characters .....	320
179: Assembler output options .....	320
180: Assembler list file options .....	322
181: Assembler preprocessor options .....	323
182: Assembler diagnostics options .....	325
183: Custom tool options .....	327
184: XLINK output file options .....	329
185: XLINK extra output file options .....	332
186: XLINK defined symbols options .....	333
187: XLINK diagnostics options .....	334
188: XLINK list file options .....	336
189: XLINK include files options .....	337
190: XLINK processing options .....	339
191: XAR Library Builder output options .....	341
192: Generic C-SPY options .....	343
193: C-SPY plugin options .....	345

# Preface

Welcome to the MSP430 IAR Embedded Workbench™ IDE User Guide. The purpose of this guide is to help you fully utilize the features in the MSP430 IAR Embedded Workbench with its integrated Windows development tools for the MSP430 microcontroller. The IAR Embedded Workbench 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 IAR Embedded Workbench. 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 MSP430 microcontroller (refer to the chip manufacturer's documentation)
- The operating system of your host machine.

Refer to the *MSP430 IAR C/C++ Compiler Reference Guide*, *MSP430 IAR Assembler Reference Guide*, and *IAR Linker and Library Tools Reference Guide* for more information about the other development tools incorporated in the IAR Embedded Workbench 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 IAR Embedded Workbench can offer.

If you already have had some experience using the 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 the C-SPY Debugger can be found in *Part 4. Debugging*, page 101.

If you are an experienced user and need this guide only for reference information, see the reference chapters in *Part 7. Reference information*.

Finally, we recommend the *Glossary* 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 XLINK Linker™, IAR XAR Library Builder™, and IAR C-SPY™ Debugger—for the MSP430 microcontroller.
- *Installed files* describes the directory structure and the type 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 C++ is used for creating 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.
- *Working with library modules* 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 import 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 the facilities for customizing it.
- *Executing your application* describes how you initialize the IAR C-SPY Debugger, 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 C-SPY macros* 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. IAR C-SPY Simulator**

- *Simulator introduction* gives a brief introduction to the simulator.

- *Simulator-specific characteristics* 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. IAR C-SPY FET Debugger**

- *Introduction to the IAR C-SPY FET Debugger* introduces you to the C-SPY Flash Emulation Tool Debugger. The chapter briefly shows the difference in functionality provided by the different debugger systems.
- *C-SPY FET driver-specific characteristics* describes the additional functionality provided by the FET driver.
- *C-SPY FET Debugger options and menus* describes the additional options and menus specific to the C-SPY FET Debugger.
- *Design considerations for in-circuit programming* describes the design considerations related to the bootstrap loader, device signals, and external power if you want to use C-SPY with your own hardware.

### **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 Debugger IDE reference* provides detailed reference information about the graphical user interface of the IAR C-SPY Debugger.
- *General options* specifies the runtime, output, and library object file options.
- *Compiler options* specifies compiler options for language, code, output, list file, preprocessor, and diagnostics.
- *Assembler options* describes the assembler options for language, output, list file, preprocessor, and diagnostics.
- *Custom build options* describes the options available for custom tool configuration.
- *XLINK options* describes the XLINK options for output, defining symbols, diagnostics, list generation, setting up the include paths, input, and processing.
- *Library builder options* describes the XAR options available in the Embedded Workbench.
- *Debugger options* gives reference information about generic C-SPY options.
- *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.

### **Glossary**

The glossary contains definitions of programming terms.

---

## Other documentation

The complete set of IAR development tools for the MSP430 microcontroller are described in a series of guides. For information about:

- Programming for the MSP430 IAR C/C++ Compiler, refer to the *MSP430 IAR C/C++ Compiler Reference Guide*
- Programming for the MSP430 IAR Assembler, refer to the *MSP430 IAR Assembler Reference Guide*
- Using the IAR XLINK Linker™ and the IAR XAR Library Builder™, refer to the *IAR Linker and Library Tools Reference Guide*
- Using the IAR CLIB Library, refer to the *IAR C Library Functions Reference Guide*, available from the MSP430 IAR Embedded Workbench™ IDE online help system.
- Using the IAR DLIB Library, refer to the MSP430 IAR Embedded Workbench™ IDE online help system.

All of these guides are delivered in PDF or HTML format on the installation media. Some of them are also delivered as printed books.

Recommended websites:

- The Texas Instruments website, [www.ti.com](http://www.ti.com), contains information and news about the MSP430 microcontrollers.
- The IAR website, [www.iar.com](http://www.iar.com), holds application notes and other product information.
- Finally, the Embedded C++ Technical Committee website, [www.caravan.net/ec2plus](http://www.caravan.net/ec2plus), contains information about the Embedded C++ standard.

---

## Document conventions

This book uses the following typographic conventions:

Style	Used for
computer	Text that you type or that appears on the screen.
parameter	A label representing the actual value you should type as part of a command.
[option]	An optional part of a command.
{a   b   c}	Alternatives in a command.
<b>bold</b>	Names of menus, menu commands, buttons, and dialog boxes that appear on the screen.

Table 1: Typographic conventions used in this guide

Style	Used for
reference	A cross-reference within this guide or to another guide.
	Identifies instructions specific to the IAR Embedded Workbench interface.
	Identifies instructions specific to the command line interface.
	Identifies helpful tips and programming hints.

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

# Part I. Product overview

This part of the MSP430 IAR Embedded Workbench™ IDE User Guide includes the following chapters:

- Product introduction
- Installed files.





# Product introduction

The IAR Embedded Workbench™ is a very powerful Integrated Development Environment (IDE), 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 IAR Embedded Workbench IDE and provides a general overview of all the tools that are integrated in this product.

---

## The IAR Embedded Workbench IDE

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

- The highly optimizing IAR MSP430 C/C++ compiler
- The IAR MSP430 assembler
- The versatile IAR XLINK Linker™
- The IAR XAR Library Builder™
- A powerful editor
- A project manager
- A command line build utility
- IAR C-SPY™ debugger, a state-of-the-art high-level language debugger.

The 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. The IAR Embedded Workbench promotes a useful working methodology, and thus a significant reduction of the development time can be achieved by using the IAR 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 website [www.iar.com](http://www.iar.com) for information about recent product releases.

## AN EXTENSIBLE AND MODULAR ENVIRONMENT

Although the IAR Embedded Workbench IDE provides all the features required for a successful project, we also recognize the need to integrate other tools. Therefore the IAR Embedded Workbench can be easily adapted to work with your editor of choice. The IAR XLINK Linker can produce a large number of output formats, allowing for debugging on most third-party emulators. Support for RTOS-aware debugging 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 IAR Embedded Workbench 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 IAR Embedded Workbench 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 utility recompiles, reassembles, and links files only when necessary
- 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.

### Windows 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 or 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
- Multi-byte character support
- Parenthesis matching
- Automatic indentation
- Bookmarks
- Unlimited undo and redo for each window.

## DOCUMENTATION

The MSP430 IAR Embedded Workbench IDE is documented in the *MSP430 IAR Embedded Workbench™ IDE User Guide* (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 compilers and assemblers, and it is completely integrated in the IAR Embedded Workbench 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.

The IAR C-SPY Debugger consists both of a general part which provides a basic set of C-SPY 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.

At the time of writing this guide, the IAR C-SPY Debugger for the MSP430 microcontroller is available with drivers for the following target systems:

- Simulator
- FET Debugger

Contact your software distributor or IAR representative for information about available C-SPY drivers. You can also find information on the IAR website, [www.iar.com](http://www.iar.com).

For further details about the concepts that are related to the IAR C-SPY Debugger, see *Debugger concepts*, page 103. In the following sections you can find general descriptions of the different drivers.

## 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. The IAR C-SPY Debugger offers the general features described in this section.

### Source and disassembly level debugging

The IAR C-SPY Debugger 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 or 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 were 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 the IAR C-SPY Debugger, 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 MSP430 IAR C/C++ Compiler generates extensive call stack information. This allows C-SPY 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**

The IAR C-SPY Debugger 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 register (CPU and peripherals) and memory windows
- Dedicated Stack window
- Support for code coverage and function level profiling
- Optional terminal I/O emulation
- UBROF, Intel-extended, and Motorola input formats supported.

## RTOS AWARENESS

The IAR C-SPY Debugger has functionality for supporting Real-time OS awareness debugging.

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

## DOCUMENTATION

The IAR C-SPY Debugger is documented in the *MSP430 IAR Embedded Workbench™ IDE User Guide* (this guide). Generic debugger features are described in *Part 4. Debugging*, whereas features specific to each debugger driver are described in *Part 5. IAR C-SPY Simulator*, and *Part 6. IAR C-SPY FET Debugger*. There are also help and hypertext versions of the documentation available online.

---

# IAR C-SPY Debugger systems

## 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 the C-SPY Debugger the simulator driver also provides:

- Instruction-level simulation
- Memory configuration and validation
- Interrupt simulation

- Peripheral simulation, using the C-SPY macro system in conjunction with immediate breakpoints.

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

### IAR C-SPY FET DEBUGGER

The IAR C-SPY Flash Emulation Tool Debugger is a JTAG debugger that supports all Texas Instruments' boards. It provides automatic flash download and takes advantage of on-chip debug facilities.

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

### Features

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

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

On devices with the Enhanced Emulation Module (EEM), you have access also to:

- State storage
- Sequencer
- Clock control

**Note:** Code coverage, function level profiling, and live watch are not supported by the C-SPY FET Debugger. Trace and data breakpoints are available if the device has support for it.

For additional information about the IAR C-SPY FET Debugger, refer to *Part 6. IAR C-SPY FET Debugger* in this guide.

---

## IAR C/C++ Compiler

The MSP430 IAR C/C++ Compiler is a state-of-the-art compiler that offers the standard features of the C or Embedded C++ languages, plus many extensions designed to take advantage of the MSP430-specific facilities.

The compiler is integrated with other IAR Systems software for the MSP430 microcontroller.

## FEATURES

The MSP430 IAR C/C++ Compiler provides the following features:

### Code generation

- Generic and MSP430-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
- The object code can be linked together with assembler routines
- Generation of extensive debug information.

### Language facilities

- Support for the C/EC++ programming languages
- Support for 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 MSP430 IAR Embedded Workbench provides two sets of runtime libraries:

- The IAR DLIB Library, which supports ISO/ANSI C, Standard Embedded C++, and Extended Embedded C++. This library also supports floating-point numbers in IEEE 754 format, multibyte characters, and locales.
- The IAR CLIB Library is a light-weight library, which is not fully compliant with ISO/ANSI C. Neither does it fully support floating-point numbers in IEEE 754 format or C++.

There are several mechanisms available for customizing the runtime environment and the runtime libraries. For both sets of runtime libraries, library source code is included.

## DOCUMENTATION

The MSP430 IAR C/C++ Compiler is documented in the *MSP430 IAR C/C++ Compiler Reference Guide*.

---

## IAR Assembler

The MSP430 IAR Assembler is integrated with other IAR Systems software for the MSP430 microcontroller. 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 MSP430 IAR Assembler uses the same mnemonics and operand syntax as the Texas Instruments MSP430 Assembler, which simplifies the migration of existing code. For detailed information, see the *MSP430 IAR Assembler Reference Guide*.

## FEATURES

The MSP430 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
- Up to 65536 relocatable segments per module
- 255 significant characters in symbol names.

## DOCUMENTATION

The MSP430 IAR Assembler is documented in the *MSP430 IAR Assembler Reference Guide*.

---

## IAR XLINK Linker

The IAR XLINK Linker links one or more relocatable object files produced by the MSP430 IAR Assembler or MSP430 IAR C/C++ Compiler to produce machine code for the MSP430 microcontroller. It is equally well suited for linking small, single-file, absolute assembler applications as for linking large, relocatable, multi-module, C or C++, or mixed C or C++ and assembler applications.

It can generate one out of more than 30 industry-standard loader formats, in addition to the IAR Systems proprietary debug format used by the IAR C-SPY Debugger—UBROF (Universal Binary Relocatable Object Format). An application can be made up of any number of UBROF relocatable files, in any combination of assembler and C or C++ applications.

The final output produced by the IAR XLINK Linker is an absolute, target-executable object file that can be downloaded to the MSP430 microcontroller or to a hardware emulator. Optionally, the output file might or might not contain debug information depending on the output format you choose.

The IAR XLINK Linker supports user libraries, and will load only those modules that are actually needed by the application you are linking. Before linking, the IAR XLINK Linker performs a full C-level type checking across all modules as well as a full dependency resolution of all symbols in all input files, independent of input order. It also checks for consistent compiler settings for all modules and makes sure that the correct version and variant of the C/C++ runtime library is used.

## FEATURES

- Full inter-module type checking
- Simple override of library modules
- Flexible segment 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.

## DOCUMENTATION

The IAR XLINK Linker is documented in the *IAR Linker and Library Tools Reference Guide*.

---

## IAR XAR Library Builder

A library is a single file that contains a number of relocatable object modules, each of which can be loaded independently from other modules in the file as it is needed. The IAR XAR Library Builder assists you to build libraries easily.

A library file is no different from any other relocatable object file produced by the assembler or compiler, except that it includes a number of modules of the LIBRARY type. All C or C++ applications make use of libraries, and the MSP430 IAR C/C++ Compiler is supplied with a number of standard library files.

## FEATURES

The IAR XAR Library Builder provides the following features:

- Modules can be combined into a library file
- Interactive or batch mode operation.

## DOCUMENTATION

The IAR XAR Library Builder is documented in the *IAR Linker and Library Tools Reference Guide*, a PDF document available from the IAR Embedded Workbench **Help** menu.



# 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 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 4.n\` directory where `x` is the drive where Microsoft Windows is installed and `4.n` is the version number of the IAR Embedded Workbench IDE.

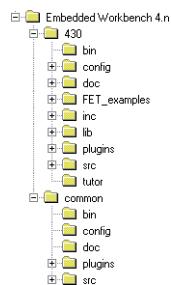


Figure 1: Directory structure

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

## THE 430 DIRECTORY

The 430 directory contains all product-specific subdirectories.

### The 430\bin directory

The 430\bin subdirectory contains executable files for MSP430-specific components, such as the MSP430 IAR C/C++ Compiler, the MSP430 IAR Assembler, and the MSP430 IAR C-SPY drivers.

### The 430\config directory

The 430\config subdirectory contains files used for configuring the development environment and projects, for example:

- Linker command files (\*.xcl)
- Special function register description files (\*.sfr)
- The C-SPY device description files (\*.ddf)
- Syntax coloring configuration files (\*.cfg)
- Project templates for both application and library projects (\*.ewp) and their corresponding library configuration files.

### The 430\doc directory

The 430\doc subdirectory contains release notes with recent additional information about the MSP430 tools. We recommend that you read all of these files. The directory also contains online versions (PDF format) of this user guide, and of the MSP430 reference guides.

### The 430\FET\_examples directory

The 430\FET\_examples subdirectory contains example projects for use with the C-SPY FET Debugger.

### The 430\inc directory

The 430\inc subdirectory holds include files, such as the header files for the standard C or Embedded 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 430\lib directory

The 430\lib subdirectory holds prebuilt libraries and the corresponding library configuration files, used by the compiler.

## The 430\plugins directory

The 430\plugins subdirectory contains executable files and description files for components that can be loaded as plugin modules.

## The 430\src directory

The 430\src subdirectory holds source files for some configurable library functions, and application code examples. This directory also holds the library source code.

## The 430\tutor directory

The 430\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 IAR XLINK Linker™, the IAR XAR Library Builder™, the editor and the graphical user interface components. The executable file for the IAR Embedded Workbench is also located here.

## The common\config directory

The common\config subdirectory contains files used by the IAR Embedded Workbench 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.

## The common\src directory

The common\src subdirectory contains source files for components common to all IAR Embedded Workbench products, such as a sample reader of the IAR XLINK Linker output format SIMPLE.

---

## File types

The MSP430 versions of the IAR Systems development tools use the following default filename extensions to identify the IAR-specific file types:

Ext.	Type of file	Output from	Input to
a43	Target application	XLINK	EPROM, C-SPY, etc.
asm	Assembler source code	Text editor	Assembler
c	C source code	Text editor	Compiler
cfg	Syntax coloring configuration	Text editor	IAR Embedded Workbench
cpp	C++ source code	Text editor	Compiler
d43	Target application with debug information	XLINK	C-SPY and other symbolic debuggers
dbg	Target application with debug information	XLINK	C-SPY and other symbolic debuggers
dbgdt	Debugger desktop settings	C-SPY	C-SPY
ddf	Device description file	Text editor	C-SPY
dep	Dependency information	IAR Embedded Workbench	IAR Embedded Workbench
dni	Debugger initialization file	C-SPY	C-SPY
ewd	Project settings for C-SPY	IAR Embedded Workbench	IAR Embedded Workbench
ewp	IAR Embedded Workbench project (current version)	IAR Embedded Workbench	IAR Embedded Workbench
eww	Workspace file	IAR Embedded Workbench	IAR Embedded Workbench
fmt	Formatting information for the Locals and Watch windows	IAR Embedded Workbench	IAR Embedded Workbench
h	C or C++ or assembler header source	Text editor	Compiler or assembler #include

Table 2: File types

Ext.	Type of file	Output from	Input to
i	Preprocessed source	Compiler	Compiler
inc	Assembler header source	Text editor	Assembler #include
lst	List output	Compiler and assembler	—
mac	C-SPY macro definition	Text editor	C-SPY
map	List output	XLINK	—
pbd	Source browse information	IAR Embedded Workbench	IAR Embedded Workbench
pbi	Source browse information	IAR Embedded Workbench	IAR Embedded Workbench
prj	IAR Embedded Workbench project (old project format)	IAR Embedded Workbench	IAR Embedded Workbench
r43	Object module	Compiler and assembler	XLINK and XAR
s43	MSP430 assembler source code	Text editor	MSP430 IAR Assembler
sfr	Special function register definitions	Text editor	C-SPY
wsdt	Workspace desktop settings	IAR Embedded Workbench	IAR Embedded Workbench
xcl	Extended command line	Text editor	Assembler, compiler, XLINK
xlb	Extended librarian batch command	Text editor	XLIB Librarian

Table 2: File types (Continued)

You can override the default filename extension by including an explicit extension when specifying a filename.

Files with the extensions `ini` and `dni` are created dynamically when you run the IAR Embedded Workbench tools. These files, which contain information about your project configuration and other settings, are located in a `settings` directory under your project directory.



**Note:** If you run the tools from the command line, the XLINK listings (map files) will by default have the extension `lst`, which might overwrite the list file generated by the compiler. Therefore, we recommend that you name XLINK map files explicitly, for example `project1.map`.

---

## Documentation

This section briefly describes the information that is available in the MSP430 user and reference guides, in the online help, and on the Internet.

You can access the MSP430 online documentation from the **Help** menu in the IAR Embedded Workbench. Help is also available via the F1 key in the IAR Embedded Workbench IDE.

We recommend that you read the `readme.htm` file for recent information that might not be included in the user guides. It is located in the `430\doc` directory.

### THE USER AND REFERENCE GUIDES

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

#### **MSP430 IAR Embedded Workbench™ IDE User Guide**

This guide.

#### **MSP430 IAR C/C++ Compiler Reference Guide**

This guide provides reference information about the MSP430 IAR C/C++ Compiler. 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.

#### **MSP430 IAR Assembler Reference Guide**

This guide provides reference information about the MSP430 IAR Assembler, including details of the assembler source format, and reference information about the assembler operators, directives, mnemonics, and diagnostics.

#### **IAR Linker and Library Tools Reference Guide**

This online PDF guide provides reference information about the IAR linker and library tools:

- The IAR XLINK Linker reference sections provide information about XLINK options, output formats, environment variables, and diagnostics.

- The IAR XAR Library Builder reference sections provide information about XAR options and output.
- The IAR XLIB Librarian reference sections provide information about XLIB commands, environment variables, and diagnostics.

### **DLIB Library Reference information**

This online documentation in HTML format provides reference information about the ISO/ANSI C, the Embedded C++, and the Extended Embedded C++ parts of the IAR DLIB Library. It is available from the MSP430 IAR Embedded Workbench™ IDE online help system.

### **CLIB Library Reference Guide**

This online guide in PDF format contains reference information about the IAR CLIB Library. It is available from the MSP430 IAR Embedded Workbench™ IDE online help system.

### **ONLINE HELP**

The context-sensitive online help contains reference information about the menus and dialog boxes in the IAR Embedded Workbench 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.

**Note:** If you select a function name in the editor window and press F1 while using the CLIB library, you will get reference information for the DLIB library.

### **IAR ON THE WEB**

The latest news from IAR Systems can be found at the website [www.iar.com](http://www.iar.com), available from the **Help** menu in the Embedded Workbench IDE. Visit it for information about:

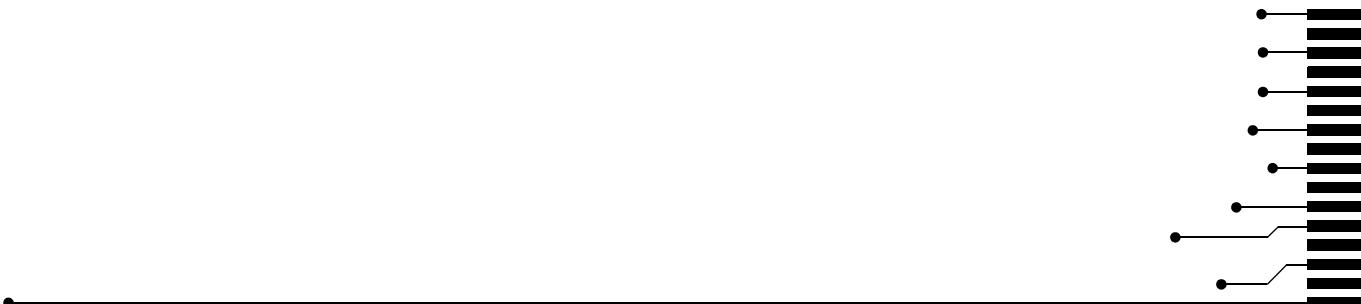
- Product announcements
- Updates and news about current versions
- Special offerings
- Evaluation copies of the IAR products
- Technical Support, including technical notes
- Application notes
- Links to chip manufacturers and other interesting sites
- Distributors; the names and addresses of distributors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



# Part 2. Tutorials

This part of the MSP430 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
- Working with library modules.





# Creating an application project

This chapter introduces you to the IAR Embedded Workbench™ integrated development environment. The tutorial demonstrates a typical development cycle and shows how you use the compiler and the linker to create a small application for the MSP430 microcontroller. 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, *Debugging using the IAR C-SPY™ Debugger*.

---

## Setting up a new project

The IAR Embedded Workbench lets you 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 `430\tutor` 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 IAR Embedded Workbench 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.

- 1 Choose **File>New** and select **Workspace** in the **New** dialog box.

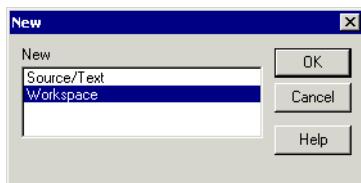


Figure 2: New dialog box

The **Help** button provides access to information about the IAR Embedded Workbench IDE. You can at any time press the F1 key to access the online help.

Click **OK**. An empty workspace window is displayed.

- 2 To save the workspace, choose **File>Save Workspace**. 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 3: 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.

Now you are ready to create a project and add it to the workspace.

## CREATING THE NEW PROJECT

- I 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.

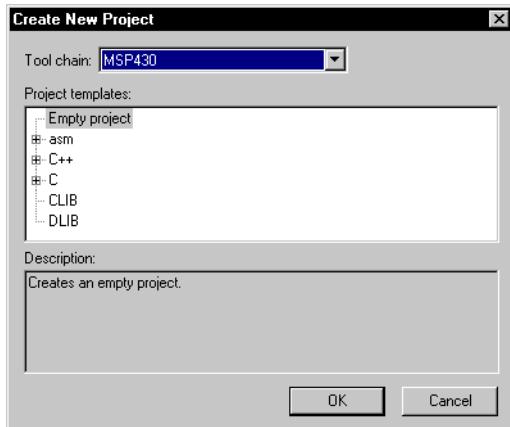


Figure 4: Create New Project dialog box

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

- 2 Make sure the **Tool chain** is set to **MSP430**, and click **OK**.
- 3 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.

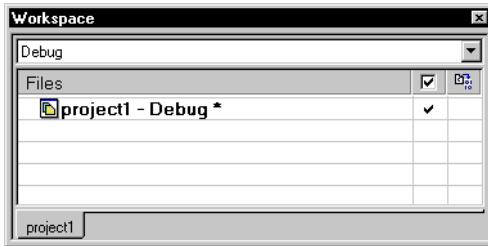


Figure 5: Workspace window

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.

A project file—with the filename extension `ewp`—has now been created in the `projects` directory. This file contains information about your project-specific settings, such as build options.

## 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*.

- I In the workspace window, select the destination you want to add a source file to; 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.

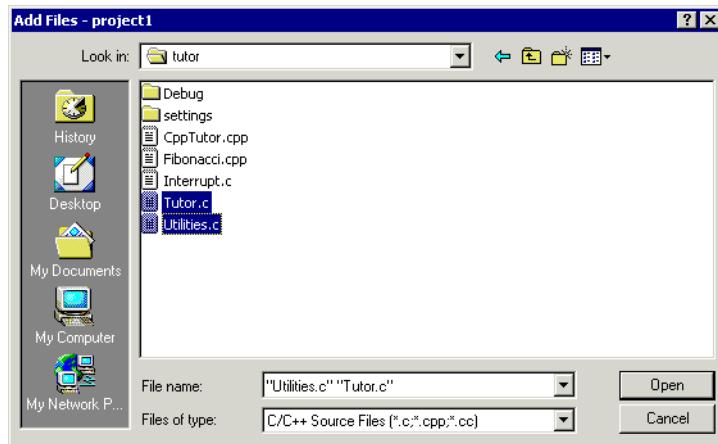


Figure 6: Adding files to project1

## 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.

- I 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.

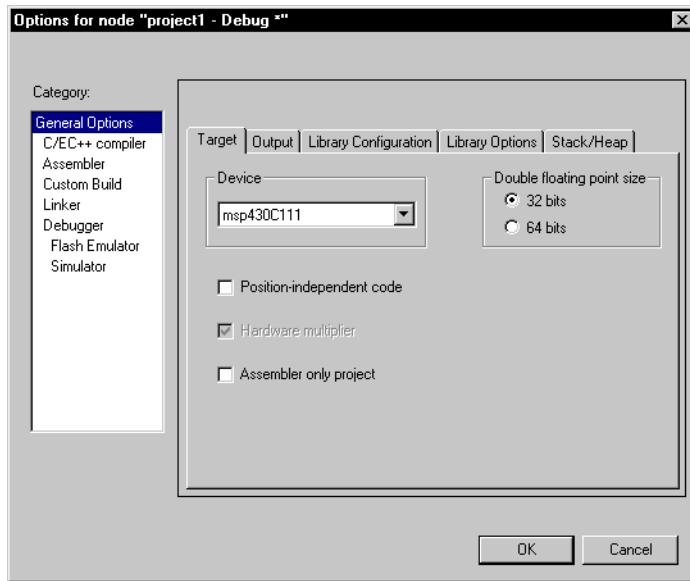


Figure 7: Setting general options

Verify the following settings:

Page	Setting
Target	Device: msp430F149
Output	Executable
Library	IAR CLIB (C library)

Table 3: 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.

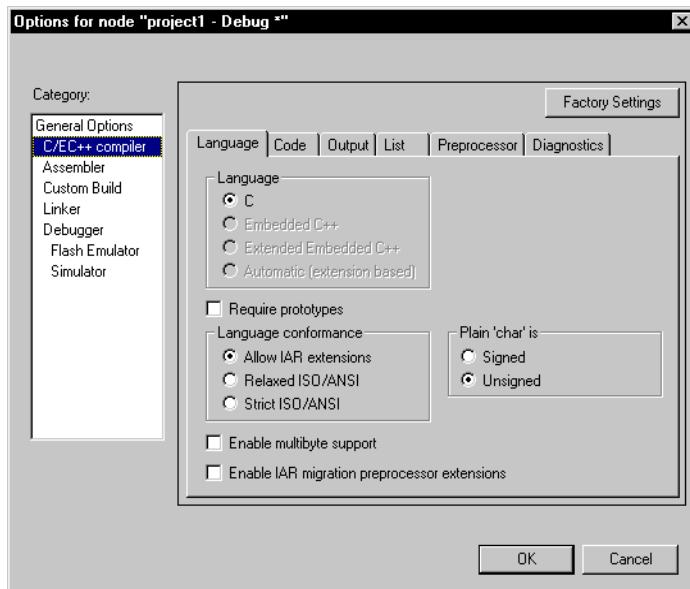


Figure 8: Setting compiler options

- 3** Select the following option settings:

Page	Setting
Code	Optimizations, Size: None (Best debug support)
Output	Generate debug info
List	Output list file Assembler mnemonics

Table 4: 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 `Utilities.c` file, 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.

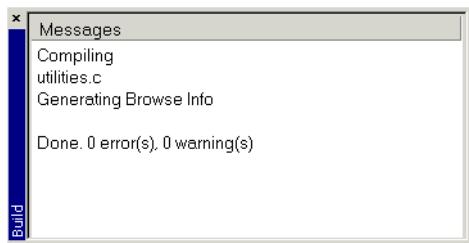


Figure 9: Compilation message

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

The IAR Embedded Workbench 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 `r43` and will be used as input to the IAR XLINK Linker.
- The `Exe` directory is the destination directory for the executable file. It has the extension `d43` 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, the 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.

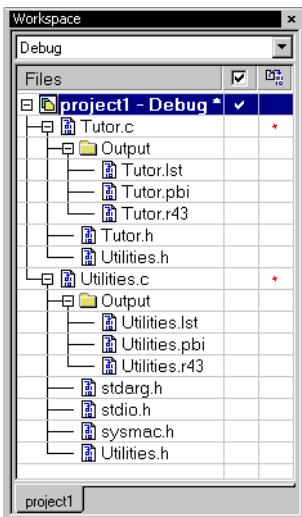


Figure 10: Workspace window after compilation

## 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 segments
  - 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.

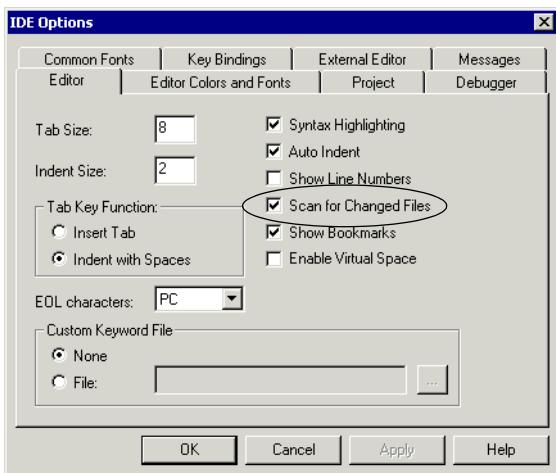


Figure 11: Setting the option *Scan for Changed Files*

- 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 **Code** tab and select the **Override inherited settings** option. Choose **High** from the **Optimizations** 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 notice, 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 **Utilities.c** file.

## LINKING THE APPLICATION

Now you should set up the options for the IAR XLINK Linker™.

- 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 XLINK option pages.

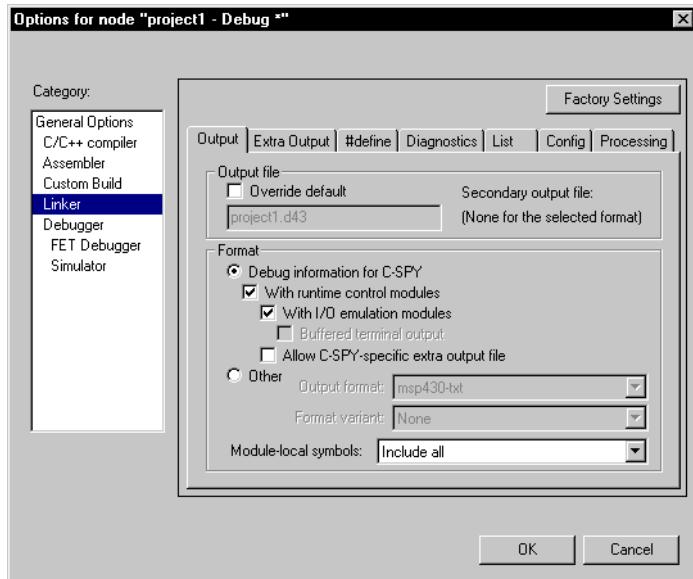


Figure 12: XLINK options dialog box for project1

- Make sure that the following options are selected on the appropriate pages:

Page	Options
Output	Debug information for C-SPY <input checked="" type="checkbox"/> With runtime control modules <input checked="" type="checkbox"/> With I/O emulation modules
List	Generate linker listing <input checked="" type="checkbox"/> Segment map <input checked="" type="checkbox"/> Symbols: Module map

Table 5: XLINK options for project1

Choose the output format that suits your purpose. You might want to load it to a debugger—which means that you need output with debug information. Alternatively, you might want to load the output to a PROM programmer—in which case you need output without debug information, such as Intel-hex or Motorola S-records. In this tutorial you will use the output option **Debug information for C-SPY** with the option **With I/O emulation modules** selected, which means that some low-level routines will be linked that direct `stdin` and `stdout` to the Terminal I/O window in the C-SPY Debugger.

If you want to examine the linker command 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. In the linker command file, the XLINK command line options for segment control are used. These options are described in the *IAR Linker and Library Tools Reference Guide*.

For more information about linker command files, see the *MSP430 IAR C/C++ Compiler Reference Guide*.

- 3 Click **OK** to save the XLINK options.

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

- 4 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.d43` with debug information and a map file `project1.map`.

## VIEWING THE MAP FILE

Examine the `project1.map` file to see how the segment definitions and code were placed in memory. These are the main points of interest in a map file:

- The header includes the options used for linking.
- The CROSS REFERENCE section shows the address of the program entry.
- The RUNTIME MODEL section shows the runtime model attributes that are used.
- The MODULE MAP shows the files that are linked. For each file, information about the modules that were loaded as part of your application, including segments and global symbols declared within each segment, is displayed.
- The SEGMENTS IN ADDRESS ORDER section lists all the segments that constitute your application.

The `project1.d43` application is now ready to be run in the IAR C-SPY Debugger.

# Debugging using the IAR C-SPY™ Debugger

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

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

---

## Debugging the application

The `project1.d43` application, created in the previous chapter, is now ready to be run in the IAR C-SPY Debugger 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 the IAR C-SPY Debugger you must set a few C-SPY 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>Debug**. Alternatively, click the **Debugger** button in the toolbar. The IAR C-SPY Debugger starts with the `project1.d43` application loaded. In addition to the windows already opened in the Embedded Workbench, a set of C-SPY-specific windows are now available.



### ORGANIZING THE WINDOWS

In the IAR Embedded Workbench, 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 IAR 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.

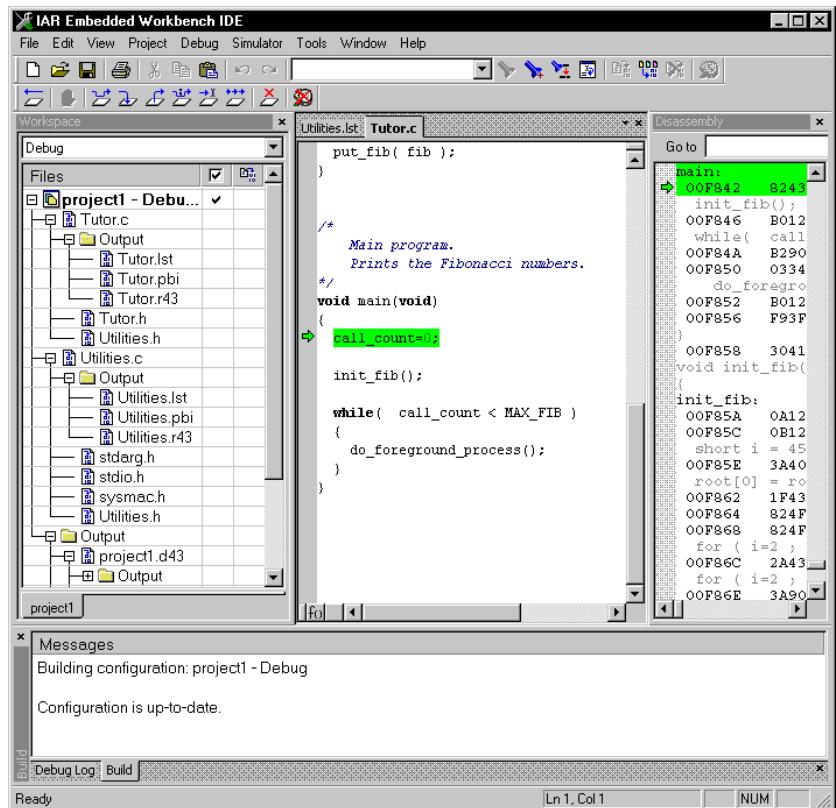


Figure 13: The C-SPY Debugger main window

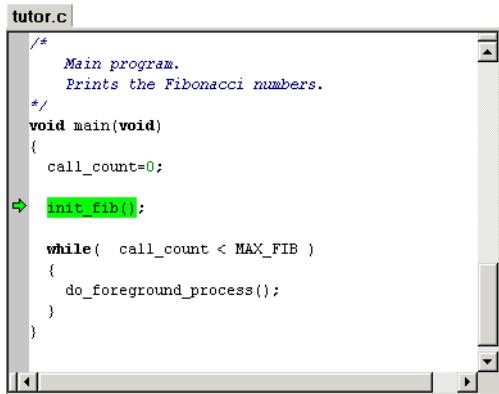
## INSPECTING SOURCE STATEMENTS

- 1 To inspect the source statements, double-click the **Tutor.c** file in the workspace window.
- 2 With the **Tutor.c** file 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.



```
tutor.c
/*
   Main program.
   Prints the Fibonacci numbers.
*/
void main(void)
{
    call_count=0;

    ➤ init_fib();

    while( call_count < MAX_FIB )
    {
        do_foreground_process();
    }
}
```

Figure 14: 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 or instruction 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 112.

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.

```

tutor.c utilities.c
unsigned int root[MAX_FIB];

/*
   Initialize MAX_FIB Fibonacci numbers.
*/
void init_fib( void )
{
    short i = 45;
    root[0] = root[1] = 1;

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

    /*
       Return the Fibonacci number 'nr'.
    */
}

```

Figure 15: Using Step Into in C-SPY

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



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

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

## INSPECTING VARIABLES

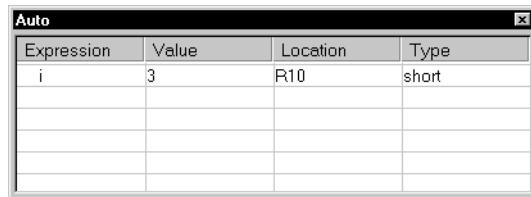
C-SPY allows you to watch variables or expressions in the source code, so that you can keep track of their values as you execute your application. You can look at a variable in a number of ways; for example by pointing at it in the source window with the mouse pointer, or by opening one of the Locals,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

- | Choose **View>Auto** to open the Auto window.

The Auto window will show the current value of recently modified expressions.



A screenshot of the 'Auto' window. The window has a title bar 'Auto'. Inside, there is a table with four columns: 'Expression', 'Value', 'Location', and 'Type'. A single row is present, showing 'i' in the Expression column, '3' in the Value column, 'R10' in the Location column, and 'short' in the Type column. Below the table are several empty rows for additional entries.

Expression	Value	Location	Type
i	3	R10	short

Figure 16: Inspecting variables in the Auto window

- Keep stepping to see how the value of `i` changes.

### Setting a watchpoint

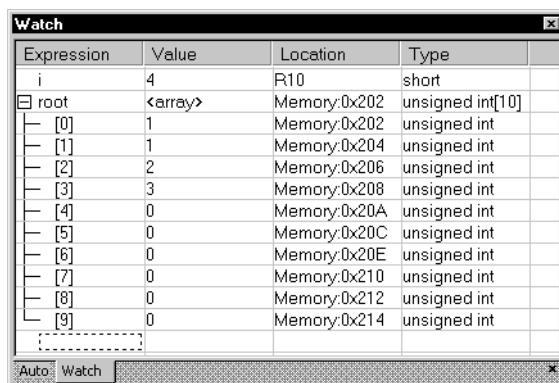
Next you will use the Watch window to inspect variables.

- 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*.
- 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.

- 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.



A screenshot of the 'Watch' window. The window has a title bar 'Watch'. Inside, there is a table with four columns: 'Expression', 'Value', 'Location', and 'Type'. Below the table is a tree view of the 'root' array. The 'root' node is expanded, showing elements [0] through [9]. Each element is a dotted rectangle, indicating it can be selected for further inspection. The table below shows the values for each element: [0] is 1, [1] is 1, [2] is 2, [3] is 3, [4] is 0, [5] is 0, [6] is 0, [7] is 0, [8] is 0, and [9] is 0. The 'Location' column shows memory addresses like 0x202, 0x204, etc., and the 'Type' column shows 'unsigned int[10]'. At the bottom of the window, there is a tab bar with 'Auto' and 'Watch' tabs, and the 'Watch' tab is currently selected.

Expression	Value	Location	Type
i	4	R10	short
root	<array>	Memory:0x202	unsigned int[10]
[0]	1	Memory:0x202	unsigned int
[1]	1	Memory:0x204	unsigned int
[2]	2	Memory:0x206	unsigned int
[3]	3	Memory:0x208	unsigned int
[4]	0	Memory:0x20A	unsigned int
[5]	0	Memory:0x20C	unsigned int
[6]	0	Memory:0x20E	unsigned int
[7]	0	Memory:0x210	unsigned int
[8]	0	Memory:0x212	unsigned int
[9]	0	Memory:0x214	unsigned int

Figure 17: 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

The IAR C-SPY Debugger contains a powerful breakpoint system with many features. For detailed information about the different breakpoints, see *The breakpoint system*, page 123.

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.

- I Set a breakpoint on the statement `get_fib(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 an X in the margin to show that there is a breakpoint there.

```
/*
 * Initialize MAX_FIB Fibonacci numbers.
 */
void init_fib( void )
{
    short i = 45;
    root[0] = root[1] = 1;

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

Figure 18: Setting breakpoints

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.

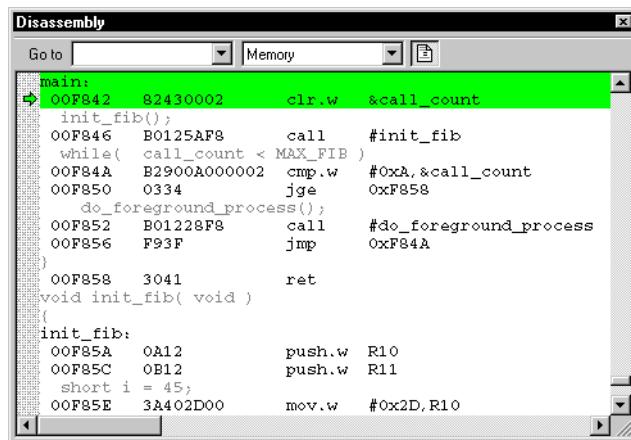
The application will execute up to the breakpoint you set. The Watch window will display the value of the `root` expression and the Debug Log window will contain information about the breakpoint.

- 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 or 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.



```

Disassembly
Go to Memory
main:
00F842 82430002    clr.w  &call_count
init_fib();
00F846 B0125AF8    call    #init_fib
while( call_count < MAX_FIB )
00F84A B2900A000002  cmp.w  #0xA,&call_count
00F850 0334         jge    0xF858
do_foreground_process();
00F852 B01228F8    call    #do_foreground_process
00F856 F93F         jmp    0xF84A
)
00F858 3041         ret
void init_fib( void )
{
init_fib:
00F85A 0A12         push.w R10
00F85C 0B12         push.w R11
short i = 45;
00F85E 3A402D00    mov.w  #0x2D,R10

```

Figure 19: Debugging in disassembly mode

Try the different step commands also in the Disassembly window.

## MONITORING REGISTERS

The Register window lets you monitor and modify the contents of the processor registers.

- 1 Choose **View>Register** to open the Register window.

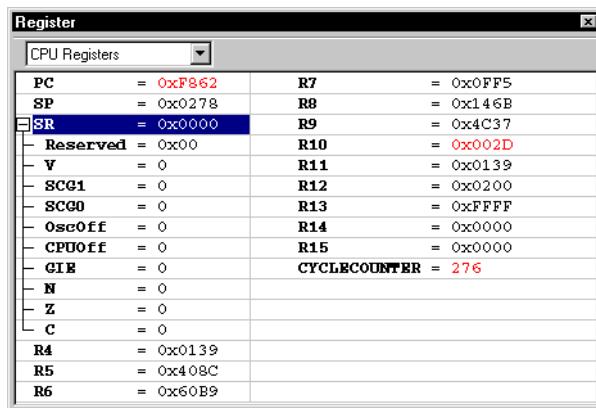


Figure 20: Register window

- 2 Step Over to execute the next instructions, and watch how the values change in the Register window.
- 3 Close the Register window.

## 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.

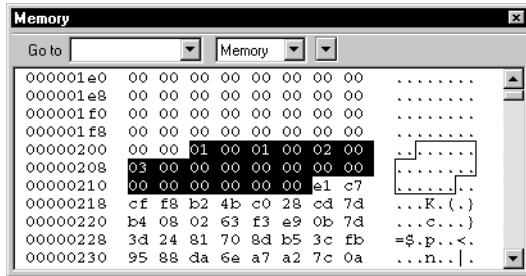


Figure 21: Monitoring memory

- 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.

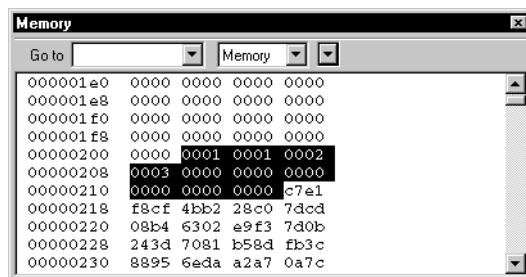


Figure 22: Displaying memory contents as 16-bit units

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 output option **With I/O emulation modules**. 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 35.

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

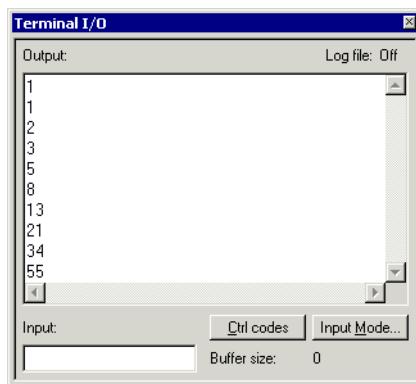


Figure 23: Output from the I/O operations

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 24: 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*.



# 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 or 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™ 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, choose **Project>Options**, and select the **General Options** category. On the **Target** page, choose **msp430F149** from the **Device** drop-down menu.
- 4** 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.

- 5** In the **C/C++ Compiler** category, select **Override inherited settings** and verify the following settings:

Page	Option
Code	Size: None (Best debug support)
List	Output assembler file Include source

*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.

- 6** Click **OK** and return to the workspace window.
- 7** Compile the file `Utilities.c`. You can find the output file `Utilities.s43` in the subdirectory `projects\debug\list`.
- 8** To examine the calling convention and to see how the C or C++ code is represented in assembler language, open the file `Utilities.s43`.

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 *MSP430 IAR C/C++ Compiler Reference Guide*.

## 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.

### SETTING UP THE PROJECT

- I** Modify `project2` by removing the file `Utilites.c` and adding the file `Utilities.s43` instead.
- Note:** To view assembler files in the **Add files** dialog box, choose **Project>Add Files** and select **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 **List file**.

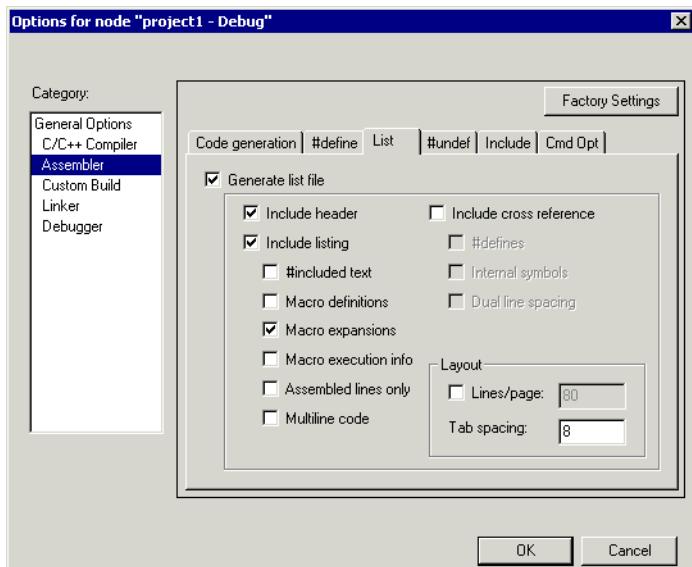


Figure 25: Assembler settings for creating a list file

Click **OK**.

- 3** Select the `Utilities.s43` file in the workspace window and choose **Project>Compile** to compile it.

Assuming that the source file was assembled successfully, the `Utilities.r43` file will be created, containing the linkable object code.

### Viewing the assembler list file

- 4** Open the list file by double-clicking the file `Utilities.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, and a checksum (CRC).

**Note:** The CRC number depends on the date of assembly.

For further details of the list file format, see *MSP430 IAR Assembler Reference Guide*.

- 5** Select **Project>Make** to relink `project2`.

- 6 Start C-SPY to run the `project2.d43` program and see that it behaves like in the previous tutorial.

# 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™ described in the previous tutorial chapters.

Note that, depending on what IAR product package you have installed, support for C++ may or may not be included. This tutorial assumes that there is support for it.

---

## Creating a C++ application

This tutorial will demonstrate how to use the MSP430 IAR Embedded Workbench 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	Device: msp430F149
	Library Configuration	Library: Normal DLIB
C/C++ Compiler	Language	Embedded C++

Table 7: Project options for C++ tutorial

All you need to do to switch to the Embedded C++ programming language is to select the options **Normal DLIB** and **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 the IAR C-SPY Debugger.

## 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 line

```
fibonacci fib1;
```

```
CppTutor.cpp Fibonacci.cpp
#include <iostream>
#include "Fibonacci.h"

int main(void)
{
    // Create two fibonacci objects.
    fibonacci fib1;
    fibonacci fib2(7); // fib2 starts at fibonacci number 7.

    // Extract two series of fibonacci numbers.
    for (int i = 1; i < 30; ++i)
    {
        cout << fib1.next();

        // If "i" is even, we print out the next fibonacci number of
        // the sequence represented by fib2.
        if (i % 2 == 0)
        {
            cout << " " << fib2.next();
        }
    }
}
```

Figure 26: 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** three more times and **Step Into** once 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 edit the breakpoint, choose **Edit>Breakpoints**. In the **Breakpoints** dialog box, select the breakpoint and set the value in the **Skip count** text box to 4 and click **Apply**.

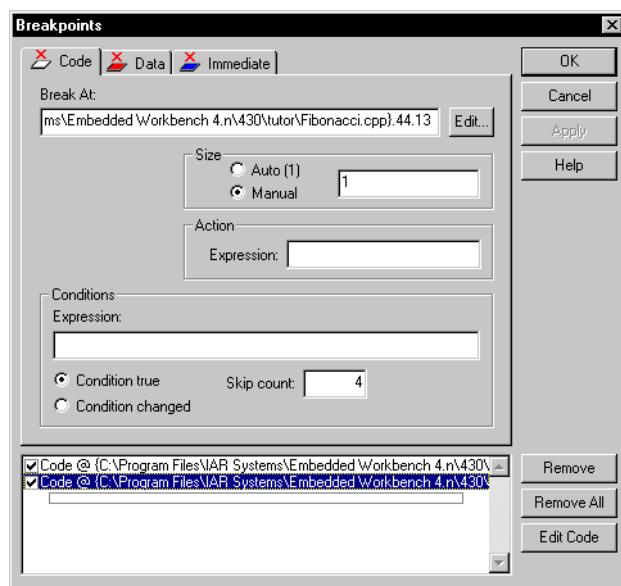


Figure 27: Setting the breakpoint condition

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.

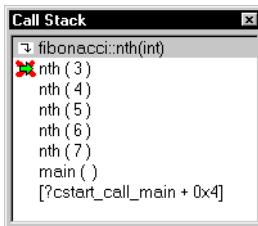


Figure 28: Inspecting the function calls

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

- I 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.

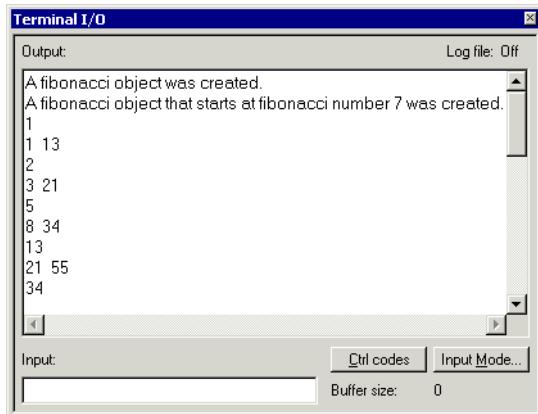


Figure 29: Printing Fibonacci sequences



# 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 (USART0).

This tutorial will show how the MSP430 IAR C/C++ Compiler keyword `__interrupt` and the `#pragma vector` directive 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™ 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 (USART0), `U0RXBUF`. 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 `430\tutor` directory):

```
// define the interrupt handler
#pragma vector=USART0RX_VECTOR
__interrupt void uartReceiveHandler( void )
```

The `#pragma vector` directive is used for specifying the interrupt vector address—in this case the interrupt vector for the USART0 receive interrupt—and the keyword `__interrupt` is used to direct the compiler to use the calling convention needed for interrupt functions.

For detailed information about the extended keywords and pragma directives used in this tutorial, see the *MSP430 IAR C/C++ Compiler Reference Guide*.

## 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 **msp430F149** from the **Device** drop-down menu.

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 USART0, values will be read from the file `InputData.txt`, which contains the Fibonacci series. You will set an *immediate read breakpoint* on the USART0 receive register, `U0RXBUF`, 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 `U0RXBUF` and the breakpoint will be triggered, the `Access()` macro function will be executed and the Fibonacci values will be fed into the USART0 receive register.

The immediate read breakpoint will trigger the break *before* the processor reads the U0RXBUF 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 C-SPY 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 166.

## 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 setup macro file `SetupSimple.mac`, available in the `430\tutor` directory. It is structured as follows.

First the macro function `execUserSetup()` is defined. As it is automatically executed during C-SPY setup, 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 `InputData.txt` file, which contains the Fibonacci series to be fed into USART0, will be opened:

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

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

```
Access()
{
    __message "Access() called\n";
    __var _fibValue;
    if( 0 == __readFile( _fileHandle, &_fibValue ) )
    {
        U0RXBUF = _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 C-SPY macros* and *C-SPY macros reference*.

Next you will specify the macro file and set the other C-SPY options needed.

## SPECIFYING C-SPY OPTIONS

- 1** Choose **Project>Options**. In the **Debugger** category, click the **Setup** tab.
- 2** Use the **Use setup 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 Table 59, *Argument variables*, page 244, for details.

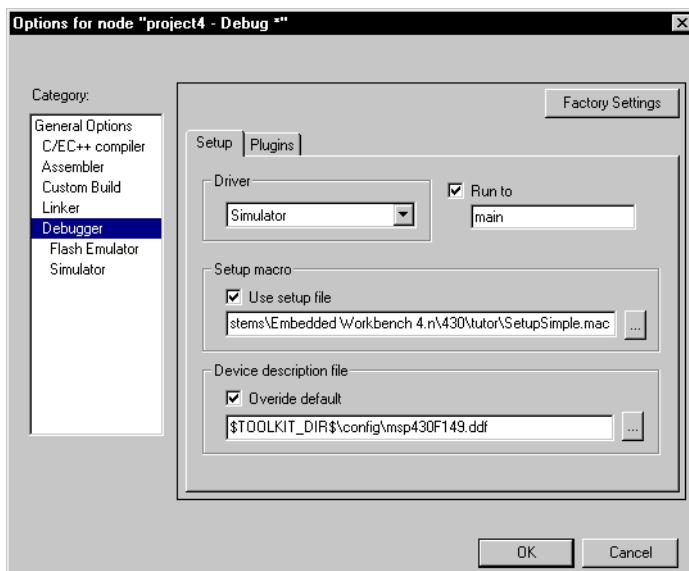


Figure 30: Specifying setup macro file

- 3 Set the **Use device description file** option to `msp430F149.ddf`. The information in this file is 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

- I 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



- I Start the IAR C-SPY Debugger 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 Debug Log window. Notice 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.

- I Choose **Simulator>Interrupts** and click **Add** to display the **Interrupt Setup** dialog box.
- 2 Make the following settings for your interrupt:

Setting	Value	Description
Interrupt	USART0RX_VECTOR	Specifies which interrupt to use; the name is defined in the <code>*.ddf</code> file.
First activation	4000	Specifies the activation moment for the first 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
Variance %	0	Time variance, not used here.
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. Specify a lower percentage to simulate a more random interrupt behavior.

Table 8: Interrupts dialog box (Continued)

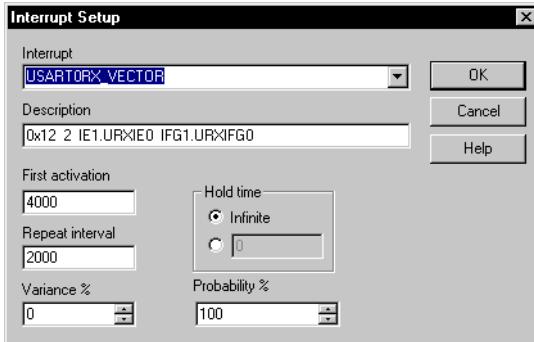


Figure 31: Inspecting the interrupt settings

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.

- 3 When you have specified the settings, click **OK**, and then close the **Interrupts** dialog box.

To automate the procedure of defining the interrupt you can instead use the system macro `__orderInterrupt` in a C-SPY setup file. At the end of this chapter we will show how this macro can be used to automate the process.

## 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 USART0 is simulated by setting an immediate read breakpoint on the U0RXBUF address and connecting the defined `Access()` macro to it. The macro will simulate the input to the USART0. These are the steps involved:

- I Choose **Edit>Breakpoints** to display the **Breakpoints** dialog box, and click the **Immediate** tab.

- 2** Add the following parameters for your breakpoint and click **Apply**.

Setting	Value	Description
Break at	U0RXBUF	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

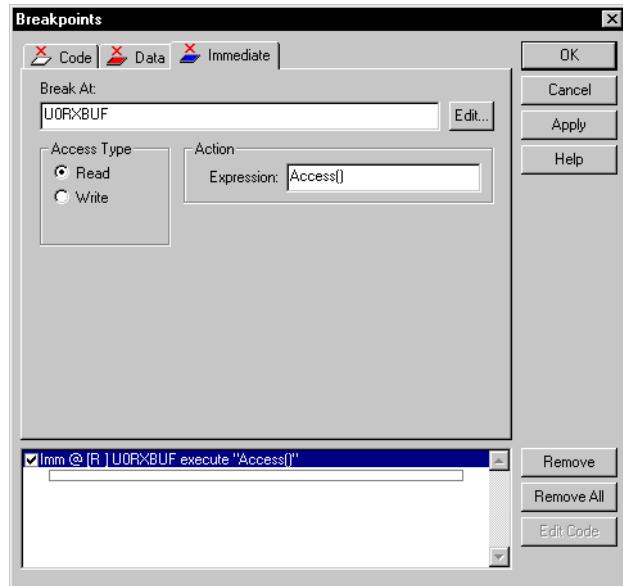


Figure 32: Displaying breakpoint information

During execution, when C-SPY detects a read access from the U0RXBUF address, C-SPY will temporarily suspend the simulation and execute the `Access()` macro. The macro will read a value from the `InputData.txt` file and write it to U0RXBUF. C-SPY will then resume the simulation by reading the receive buffer value in U0RXBUF.

- 3** Click **OK** to close the **Breakpoints** dialog box.

To automate the breakpoint setting, you can instead use the system macro `__setSimBreak` in a C-SPY setup file. At the end of this chapter we will also show how this macro can be used to automate the process.

## 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, which waits for input.

2 In the `Interrupt.c` source window, locate the function `uartReceiveHandler`.

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

The application should stop in the interrupt function.

5 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.

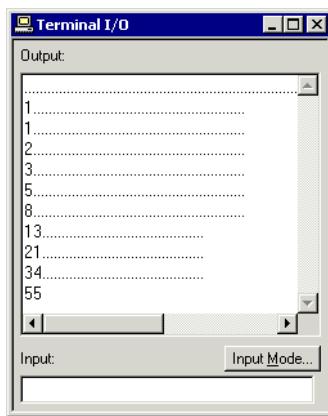


Figure 33: Printing the Fibonacci values in the Terminal I/O window

## 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 `SetupAdvanced.mac` file is extended with system macro calls for setting the breakpoint and specifying the interrupt:

```
SimulationSetup()
{
    ...
    _interruptID = __orderInterrupt( "USART0RX_VECTOR", 4000,
                                    2000, 0, 1, 0, 100 );

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

    _breakID = __setSimBreak( "U0RXBUF", "R", "Access()" );
}
```

By replacing the `SetupSimple.mac` file, used in the previous tutorial, with the `SetupAdvanced.mac` file, 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 `SetupAdvanced.mac` file you should remove the previously defined breakpoint and interrupt.



# Working with library modules

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

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

---

## Using libraries

If you are working on a large project you will soon accumulate a collection of useful routines that are used by several of your programs. To avoid having to assemble a routine each time the routine is needed, you can store such routines as object files, that is, assembled but not linked.

A collection of routines in a single object file 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.

You use the IAR XAR Library Builder to build libraries. For more information about this tool, see the *IAR Linker and Library Tools Reference Guide*.

### The Main.s43 program

The Main.s43 program uses a routine called `r_shift` to right-shift the contents of the register R4 the number of times of the value stored in register R5. The result is returned in R4. The EXTERN directive declares `r_shift` as an external symbol, to be resolved at link time.

A copy of the program is provided in the 430\tutor directory.

### The library routines

The two library routines will form a separately assembled library. It consists of the `r_shift` routine called by `main`, and a corresponding `l_shift` routine, both of which operate on the contents of the registers A and B and return the result in A. The file containing these library routines is called `Shifts.s43`, and a copy is provided with the product.

The routines are defined as library modules by the `MODULE` directive, which instructs the IAR XLINK Linker™ to include the modules only if they are referenced by another module.

The `PUBLIC` directive makes the `r_shift` and `l_shift` entry addresses public to other modules.

For detailed information about the `MODULE` and `PUBLIC` directives, see the *MSP430 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.s43` to the new project.
- 3** To set options, choose **Project>Options**. Then select **Linker** in the **Category** list and set the following option:

Page	Option
Include	Library Ignore CSTARTUP in library

Table 10: XLINK options for project 5

This tutorial uses the default options in the **General Options** and **Assembler** categories.

- 4** To assemble and link the `Main.s43` file, choose **Project>Make**.



Alternatively, click the **Make** 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 `shift_library`.
- 2** Add the file `Shifts.s43` to the project.
- 3** To set options, choose **Project>Options**. From the **General Options** category, make sure the following option is selected:

Page	Option
Output	Output file Library

Table 11: XLINK options for library project

Note that **Library Builder** appears in the list of categories. It is not necessary to set any XAR-specific options for this tutorial.

Click **OK**.

**4 Choose Project>Make.**

The library output file `library.r43` has now been created.

## USING THE LIBRARY IN THE PROJECT

You can now add your library containing the shift routine to `project5`.

- 1** In the workspace window, click the `project5` tab. Choose **Project>Add Files** and add the file `library.r43` 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 *LAR Linker and Library Tools Reference Guide*.



# **Part 3. Project management and building**

This part of the MSP430 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. The chapter also demonstrates how you can customize the environment to suit your requirements.

---

## The IAR Embedded Workbench IDE

The IAR Embedded Workbench IDE is the framework where all necessary tools are seamlessly integrated: a C/C++ compiler, an assembler, the IAR XLINK Linker™, the IAR XAR Library Builder™, an editor, a project manager with Make utility, and C-SPY™, a high-level language debugger.

The compiler, assembler, linker, and library builder 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.

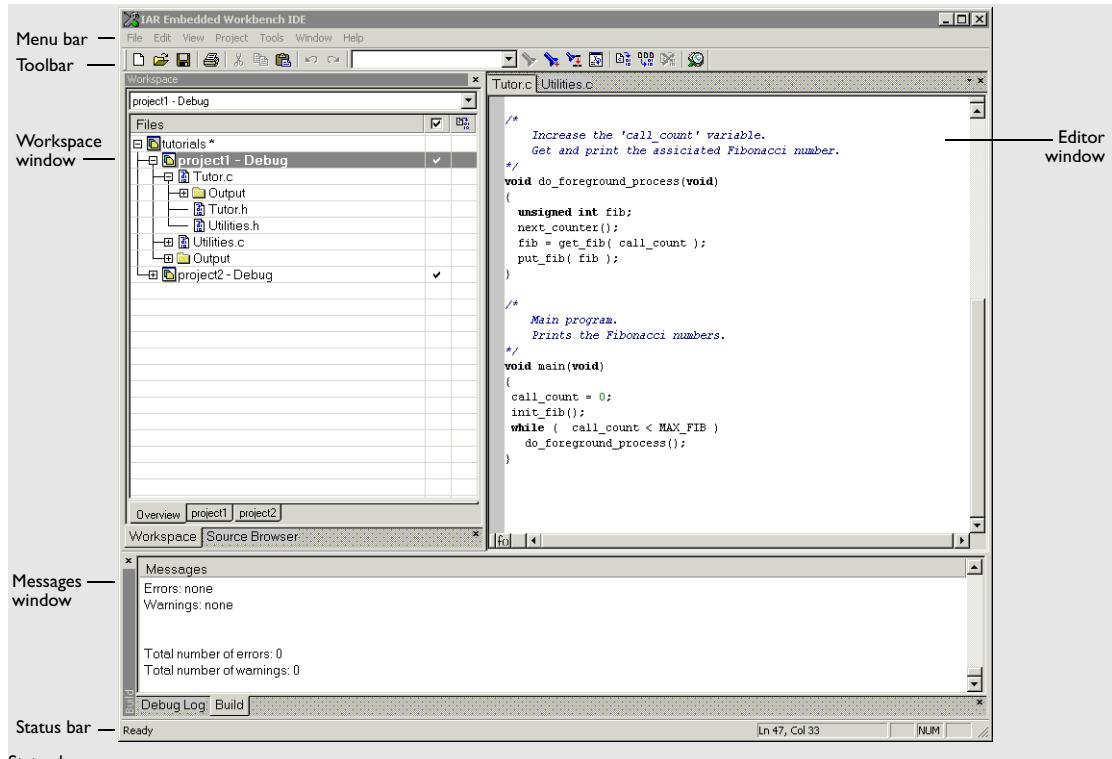


Figure 34: IAR Embedded Workbench IDE window

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

Reference information for each IDE component can be found in *Part 7. Reference information*. Information about how to use the different components can be found in parts 3 to 6.

## RUNNING THE IAR EMBEDDED WORKBENCH

Click the Start button on the taskbar and choose **Programs>IAR Systems>IAR Embedded Workbench for MSP430 V3>IAR Embedded Workbench**.

The `IarIdePm.exe` file 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 IAR Embedded Workbench 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 Embedded Workbench that uses that file type.

### EXITING

To exit the IAR Embedded Workbench, 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 IAR Embedded Workbench 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 IAR Embedded Workbench, 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 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 95.

### Organizing windows

To place a window as a *separate* window, drag it next to another open window. Hold down the Ctrl key to prevent it from docking.

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 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, the IAR Embedded Workbench lets you 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 251. You can also find further information related to customizing the editor in the section *Customizing the editor environment*, page 98. For further information about customizations related to C-SPY, see *Part 4. Debugging*.

## COMMUNICATING WITH 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 IAR Embedded Workbench. 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.

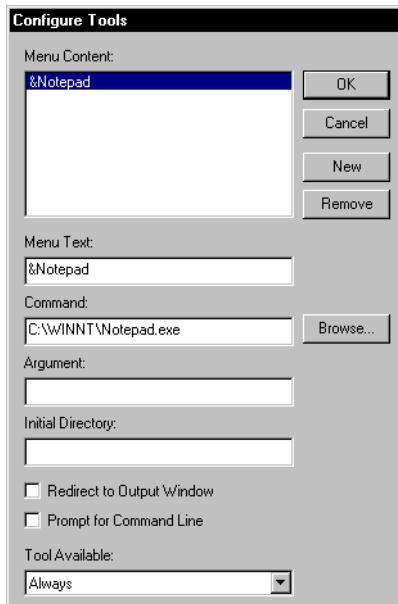


Figure 35: Configure Tools dialog box

For reference information about this dialog box, see *Configure Tools dialog box*, page 263.

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

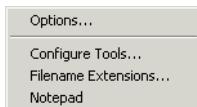


Figure 36: Customized Tools menu

### 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 98/Me	command.com
Windows NT/2000/XP	cmd.exe (recommended) or command.com

Table 12: 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 IAR Embedded Workbench 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. 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 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 IAR Embedded Workbench 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 IAR Embedded Workbench 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 IAR Embedded Workbench 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, the IAR Embedded Workbench lets you 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.

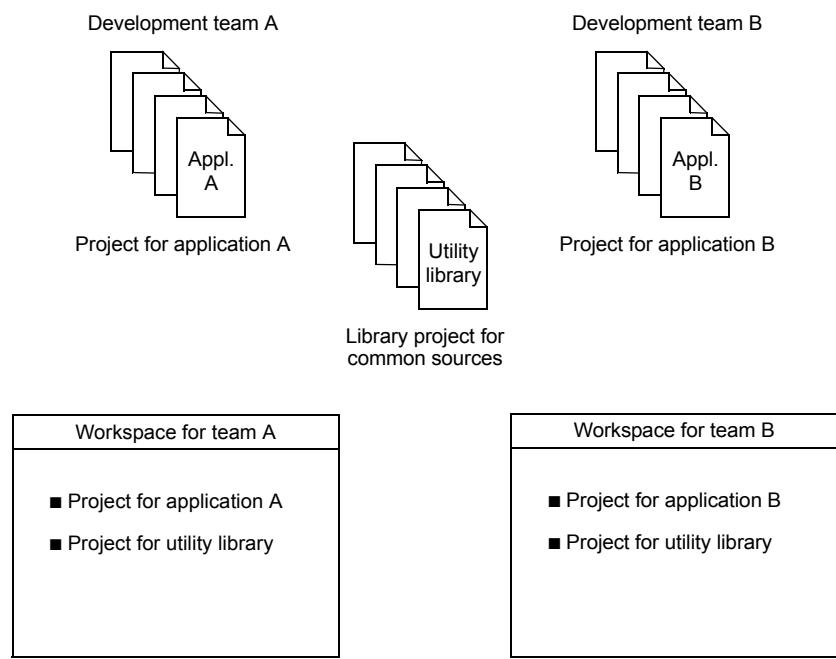


Figure 37: Examples of workspaces and projects

For an example where a library project has been combined with an application project, see the chapter *Working with library modules*.

## 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, for instance, the options used for optimization, debug information, and output format.

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 actions 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.

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

For a detailed example, see *Creating an application project*, page 25.

### Source file paths

The IAR Embedded Workbench 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 IAR Embedded Workbench IDE will use a path relative to the project file when accessing the source file.

---

## 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 Brower window displays an hierarchical view in alphabetical order of all globally defined symbols, such as variables, functions, and type definitions.

## VIEWING THE WORKSPACE

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

- Choose which project you want to view by clicking its tab at the bottom of the workspace window, for instance **project1**.

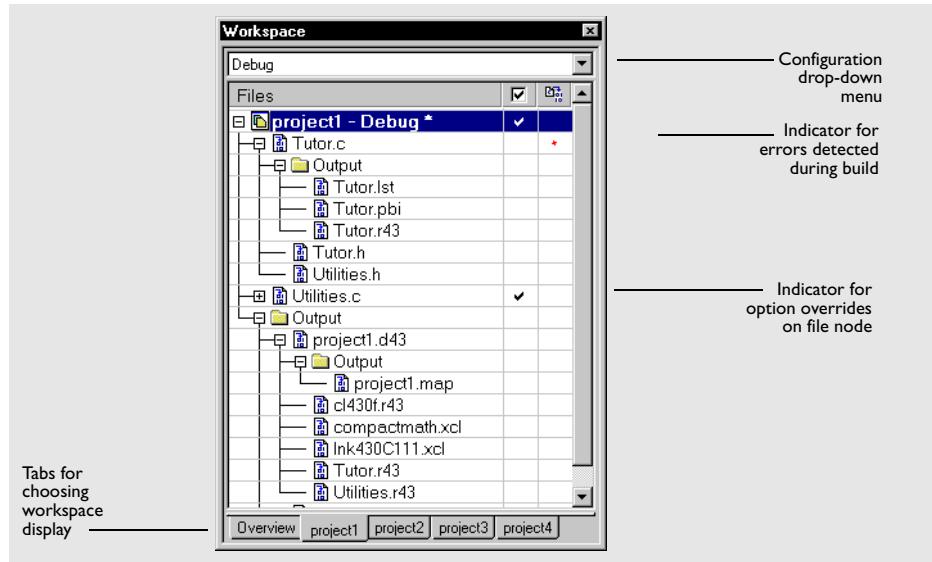


Figure 38: Displaying a project in the workspace window

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.

- To display the project with a different build configuration, choose that build configuration from the drop-down list at the top of the workspace window, for instance **project1 – Release**.

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.

- 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.

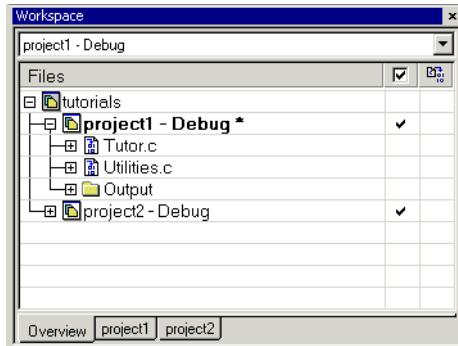


Figure 39: Workspace window—an overview

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

## DISPLAYING SOURCE BROWSE INFORMATION

To make the Source Browser window contain updated information, 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 226.

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.



# 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 fine-tune the build process, you can use the **Batch Build** command. This gives you the possibility to perform several builds in one operation.

In addition to using the IAR Embedded Workbench IDE for building projects, you can also use the IAR Command Line Build Utility (`iarbuild.exe`) to do this.

For examples of building application and library projects, see *Part 2. Tutorials* in this guide. For further information about building library projects, see the *MSP430 IAR C/C++ Compiler Reference Guide*.

### 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.

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. On the project level, you can set global and default settings for each configuration. Global settings affect the entire build configuration. Examples of such settings are runtime settings (for example, **Device** and **Output file type**), linker settings, and debug settings.

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.

### 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.

**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.

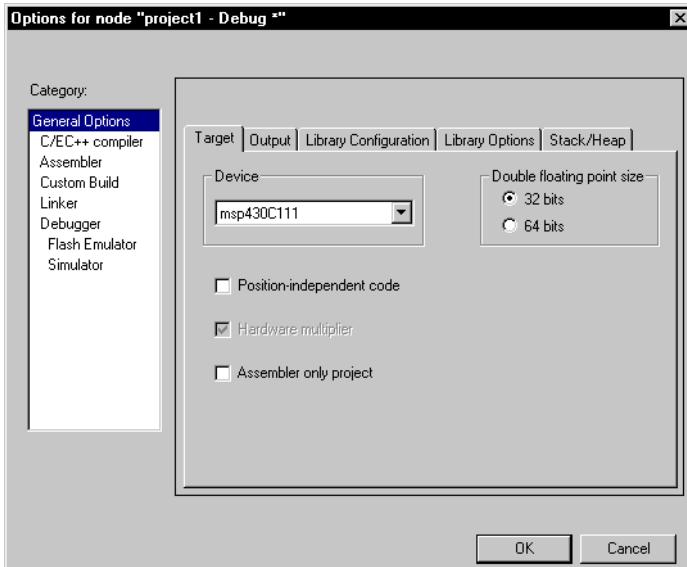


Figure 40: General options

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 exchanged with **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**.

For information about each option and how to set options, see the chapters *General options*, *Compiler options*, *Assembler options*, *XLINK 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 debugger 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, the IAR Embedded Workbench lets you add support for additional filename extensions. For reference information, see *Filename Extensions dialog box*, page 265.

## 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 IAR Embedded Workbench while your project is being built.

For further reference information, see *Source Browser window*, page 226.

## 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 249.

## CORRECTING ERRORS FOUND DURING BUILD

The compiler, assembler, and debugger are fully integrated in 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 window.

To specify the level of output to the Build 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, see *Messages page*, page 255.

For reference information about the Build window, see *Build window*, page 227.

## 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, for instance `project1.ewp`, as in this example:

```
iarbuild project1.ewp -build Debug
```

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

---

## Extending the tool chain

The 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). 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 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 `r43` files. See the chapter *Custom build options* 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. See Table 59, *Argument variables*, page 244.

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.

- 1 Add the file you want to work with to your project, for example `foo.lex`.
- 2 Select this file in the workspace window and choose **Project>Options**. Select **Custom Build** from the list of categories.
- 3 In the **Filename extensions** field, type the filename extension `.lex`. Remember to specify the leading period (`.`).
- 4 In the **Command line** field, type the command line for executing the external tool, for example:

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

During the build process, this command line will be expanded to:

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

Note the usage of *argument variables*. For further details of these variables, see Table 59, *Argument variables*, page 244.

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 how to use the IAR Embedded Workbench editor, 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 basic features like unlimited undo/redo, automatic completion, and drag and drop. In addition, it provides functions specific to software development, like coloring of keywords (C or C++), assembler, and user-defined), block indent, and function navigation in source files. It also recognizes C or C++ language elements, like matching brackets, as well as DLIB library functions for which there is instant access to reference information.

### EDITING A FILE

The editor window is where you write, view, and modify 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 *tab groups*. 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.

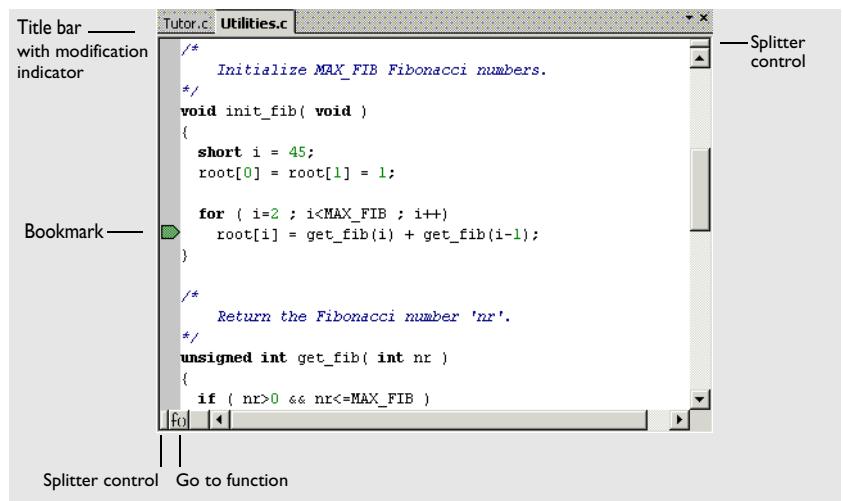


Figure 41: Editor window

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 read-only file has been modified after it was last saved, an asterisk appears after the window title, for example Utilities.c \*.

The commands on the **Window** menu allow you to split the editor window into panes. For reference information about each command on the menu, see *Window menu*, page 267. For reference information about the editor window, see *Editor window*, page 222.

### **Accessing reference information for DLIB library functions**

When you need to know the syntax for any C or 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.

**Note:** If you select a function name in the editor window and press F1 while using the CLIB library, you will get reference information for the corresponding function in the DLIB library. In some cases, this information may not be correct for CLIB functions. Information about CLIB functions can be found in the *IAR C Library Functions Reference Guide*, available in PDF format from the MSP430 IAR Embedded Workbench™ IDE online help system.

### **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 232.

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 224.

To change the default shortcut key bindings, choose **Tools>Options**, and click the **Key Bindings** tab. For further details, see *Key Bindings page*, page 253.

### **Adding bookmarks**

Use the **Edit>Toggle Bookmark** command to add and remove bookmarks, for instance to parts of the text for which you have a particular interest. To switch between the marked locations, choose **Edit>Go to Bookmark**.

## **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 bar 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**

The IAR Embedded Workbench editor automatically recognizes the syntax of:

- Assembler, C, and C++ keywords
- C and C++ comments
- Assembler comments
- Preprocessor directives
- Strings.

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

To change these styles, choose **Tools>Options**, and click the **Editor Colors and Fonts** tab in the **IDE Options** dialog box. For additional information, see *Editor Colors and Fonts page*, page 258.

## **Customizing indentation**

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.

To customize the indentation, choose **Tools>Options**, and click the **Editor** tab. For additional information, see *Editor page*, page 256.

## **Matching brackets**

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

## 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.

## 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 42: Editor window status bar

## 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.

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**, and **Find in Files** functions, choose the corresponding command from the **Edit** menu. For reference information about each search function, see *Edit menu*, page 232.

## Customizing the editor environment

The IAR Embedded Workbench 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 251.

## USING AN EXTERNAL EDITOR

The **External Editor** page—available by choosing **Tools>Options**—lets you specify an external editor of your choice.

- 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).

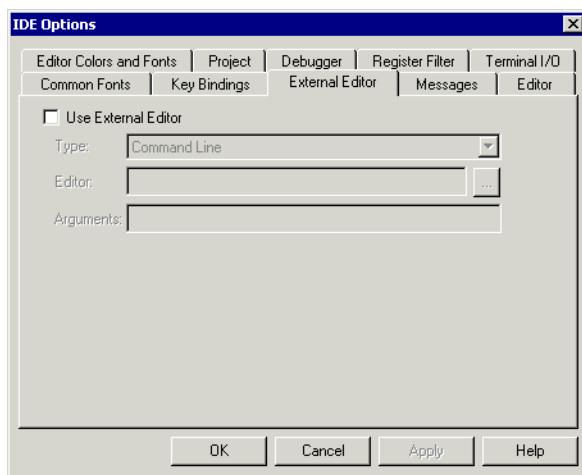


Figure 43: Specifying external command line editor

- 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®:

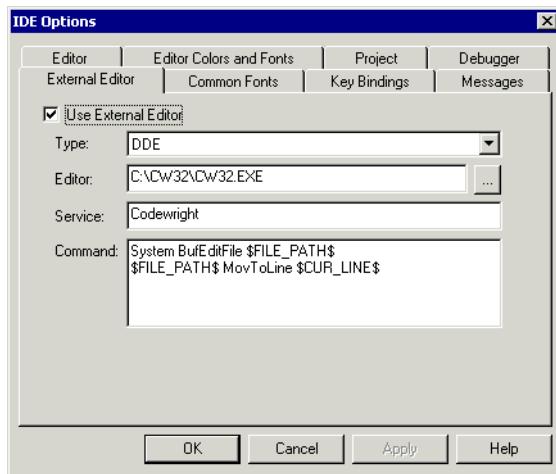


Figure 44: External editor DDE settings

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 Table 59, *Argument variables*, page 244.

# Part 4. Debugging

This part of the MSP430 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 C-SPY macros
- 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 the IAR C-SPY Debugger in particular. Then the debugger 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 the IAR C-SPY Debugger in particular. This section does not contain specific conceptual information related to the functionality of the IAR C-SPY Debugger. Instead, such information can be found in each chapter of this part of the guide. The IAR user documentation uses the following terms when referring to these concepts.

### IAR C-SPY DEBUGGER AND TARGET SYSTEMS

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

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

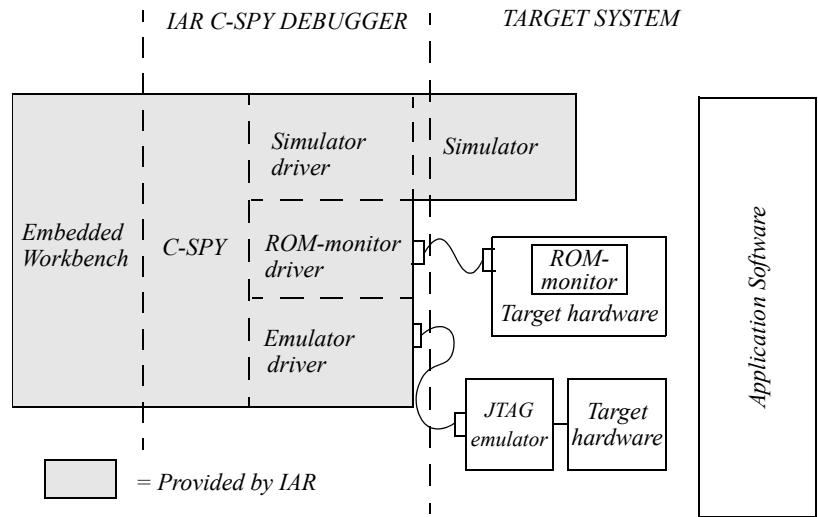


Figure 45: IAR C-SPY Debugger and target systems

**Note:** For information about which C-SPY drivers are available for the MSP430 Embedded Workbench IDE, see *IAR C-SPY Debugger systems*, page 8.

## 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 the IAR C-SPY Debugger.

## IAR C-SPY 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 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 IAR Embedded Workbench.

For a list of drivers available for the MSP430 Embedded Workbench IDE, see *IAR C-SPY Debugger*, page 5. In that chapter you can also find an overview of the general features of the C-SPY Debugger. For an overview of the functionality provided by each driver, see *IAR C-SPY Debugger systems*, page 8.

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

---

## The C-SPY environment

### AN INTEGRATED ENVIRONMENT

The IAR C-SPY Debugger is a high-level-language debugger for embedded applications. It is designed for use with the IAR MSP430 C/C++ Compiler and IAR MSP430 Assembler, and is completely integrated in the IAR Embedded Workbench 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. Breakpoints are highlighted in the editor windows 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.

In addition to the features available in the IAR Embedded Workbench, the debugger 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 Debugger IDE reference*, page 271.

For information about a specific C-SPY driver, see the part of the book corresponding to the driver.

---

## Setting up the IAR C-SPY Debugger

Before you start the IAR C-SPY Debugger 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 page*, page 260.

For information about how to configure the debugger to reflect the target hardware, see *Adapting C-SPY to target hardware*, page 109.

### 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 6. IAR C-SPY FET Debugger*.

**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 136. 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

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 `430\config` directory and they have the filename extension `ddf`.

By default, a suitable device description file is always selected. To load a different device description file, you must, before you start the C-SPY debugger, choose **Project>Options** and select the **Debugger** category. On the **Setup** page, 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 109.

## 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 web site, for information about available modules.

For information about how to load plugin modules, see *Plugins*, page 345.

### The IAR C-SPY RTOS awareness plugin modules

You can load plugin modules for real-time operating systems for use with the IAR C-SPY Debugger. 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 information about other RTOS awareness plugin modules, refer to the manufacturer of the plugin module.

---

## Starting the IAR C-SPY Debugger

When you have setup the debugger, you can start it.



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

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 Embedded Workbench

It is also possible to load C-SPY with a project that was built outside the Embedded Workbench, for example projects built on the command line. To be able to set C-SPY options for the externally built project, you must create a project within the Embedded Workbench.

To load an externally built application, you must first create a project for it in your workspace. Choose **Project>Create New Project**, and specify a project name. To add the application 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 `d43`). 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.

## 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 by choosing **Debug>Logging>Set Log File**—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 298.

---

## Adapting C-SPY to target hardware

### 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:

- Memory information for device-specific memory zones
- Definitions of memory-mapped peripheral units, device-specific CPU registers, and groups of these
- Definitions for interrupt simulation in the simulator.

You can find device description files for each MSP430 device in the `430\config` directory.

For information about how to load a device description file, see *Selecting a device description file*, page 107.

## Memory zones

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

If your hardware does not have the same memory layout as any of the predefined device description files, you can define customized zones by adding them to the file. For further details about customizing the file, see *Modifying a device description file*, page 110.

For information about memory zones, see *Memory addressing*, page 129.

## Registers

For each device there is a hard-wired 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 MSP430 microcontrollers.

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 MSP430 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 132.

## Interrupts

Device description files also contain a section that defines all device-specific interrupts, which makes it possible to simulate these interrupts in the C-SPY Simulator. You can read more about how to do this in *Simulating interrupts*, page 159.

## 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.

# 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

The IAR C-SPY Debugger 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 or 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 the chapter *Debugging the application*, page 37.

---

## Executing

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

- Step Into
- Step Over
- Next Statement
- Step Out

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

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

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

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

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

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

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

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

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

When inside the function, you have the choice of stepping out of it before reaching the function exit, by using the **Step Out** command. This will take you directly to the statement immediately after the function call:

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

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

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

## GO

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

## RUN TO CURSOR

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

## HIGHLIGHTING

At each stop, C-SPY highlights the corresponding C or C++ source with a green color.

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

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

## 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 solution you are using you might also have access to additional types of breakpoints. For instance, if you are using C-SPY Simulator there is a special kind of breakpoint to facilitate simulation of simple hardware devices. See the chapter *Simulator-specific characteristics* 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, and the application will exit properly. Before you start C-SPY, choose **Project>Options**, and select the **Linker** category. On the **Output** page, select the output format **Debug information for C-SPY**.

---

## Call stack information

The MSP430 IAR C/C++ 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 (---).

In the editor and Disassembly windows, a green highlight color indicates the topmost, or current, call frame; a yellow highlight color 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 *MSP430 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 link your application with the option **Debug information for C-SPY:With I/O emulation modules**. C-SPY will then direct `stdin`, `stdout`, and `stderr` to this window.

For reference information, see *Terminal I/O window*, page 285.

### 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 File dialog box—available by choosing **Debug>Logging>Set Terminal I/O Log File**—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 299.

# 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 or 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.

## ASSEMBLER SYMBOLS

Assembler symbols can be assembler labels or register names. That is, general purpose registers, such as R4–R15, and special purpose registers, such as the program counter and the status register. If a device description file is used, all memory-mapped peripheral units, such as I/O ports, can also be used as assembler symbols in the same way as the CPU registers. See *Device description file*, page 109.

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

Example	What it does
#pc++	Increments the value of the program counter.
myptr = #label7	Sets myptr to the integral address of label7 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
#pc	Refers to the program counter.
#`pc`	Refers to the assembler label pc.

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 132.

## 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 136.

## 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 347.

## 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 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—automatically displays the value of variables with static location, such as global variables.
- The **Quick Watch** dialog box.
- The Trace window.

For reference information about the different windows, see *C-SPY windows*, page 271.

For reference information about the **Quick Watch** dialog box, see *Quick Watch dialog box*, page 295.

## 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** dialog box 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 dialog box

The **Quick Watch** dialog box—available from the **Debug** menu—lets you watch the value of a variable or expression and evaluate expressions.

The **Quick Watch** dialog box is different from the Watch window in the following ways:

- The **Quick Watch** dialog box 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 dialog box.
- The Watch window lets you monitor an expression during a longer period of time, whereas **Quick Watch** provides you with a one-shot view.
- In contrast to the Watch window, the **Quick Watch** dialog box 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.

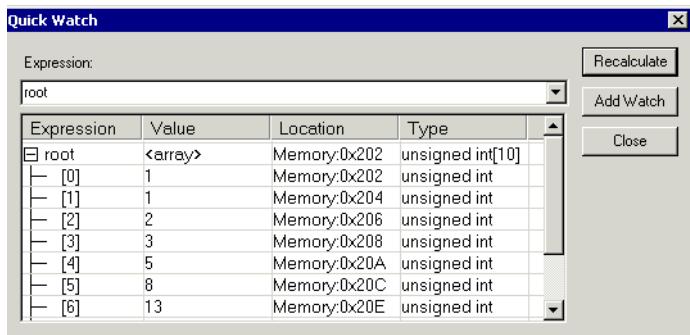


Figure 46: Quick Watch dialog box

### Using the Trace window

The Trace window—available from the **View** menu—lets you trace the values of C-SPY expressions. You can trace the program flow until 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 which have irregular symptoms, and occur sporadically. Trace information can also be useful as test documentation.

Define the expressions you want to trace on the **Expression** page.

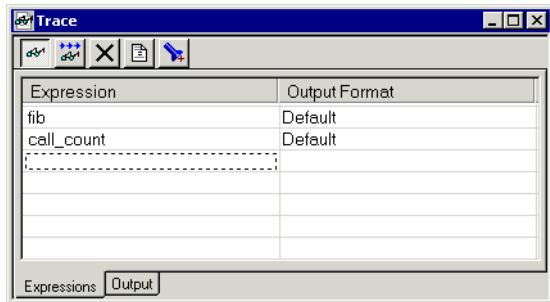


Figure 47: Trace window (Expression page)

#### Logging trace information

Click the **Output** tab to view the **Output** page. When the Trace window is open, trace information is logged for each expression every time execution in C-SPY stops. For instance, if you step on C/C++ source level, trace information is logged for each step. If you single step in the Disassembly window, trace information is logged for each executed assembler instruction.



To automate the generation of trace information, click the **Go with trace** button. The application is executed, and during the execution no C-SPY windows are refreshed. Trace information is logged continuously for each executed PC location, which is useful if you want to produce trace information for a longer period of execution time.

The screenshot shows the 'Trace' window with the 'Output' tab selected. It displays a table of trace data:

PC	fib	call_count
0x114A	—	0
0x1142	1	1
0x1142	1	2
0x1142	2	3
0x1142	3	4
0x1142	5	5
0x1142	8	6

Figure 48: Trace window (Output page)

To be able to browse the trace information conveniently in a file, choose **File>Save As**. A standard **Save As** dialog box appears.

For reference information, see *Trace window*, page 289.

# Using breakpoints

This chapter describes the breakpoint system and different ways to define 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. You can also set a *data* breakpoint, to investigate how and when the data changes. However, the latter requires that data breakpoints are supported by the target system.

For a more advanced simulation, you can stop 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 two interfaces for defining breakpoints, either interactively in a dialog box or by using system macros.

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

For reference information about code and data breakpoints, see *Breakpoints dialog box*, page 237.

In addition to the breakpoint categories listed above, and depending on which debugger driver you are using, additional breakpoint categories might be available. For instance, if you are using the simulator driver you can also set *immediate* breakpoints. For details about any additional breakpoint categories, see the driver-specific *Part 5. IAR C-SPY Simulator* and *Part 6. IAR C-SPY FET Debugger*.

## Defining breakpoints

There are different ways of setting breakpoints: using the **Toggle Breakpoint** command, using the Memory window, using the **Breakpoints** dialog box, or using predefined system macros. The different methods allow different levels of complexity and automation.

Toggling a code breakpoint is an unsophisticated but quick method for setting breakpoints.

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.

Macros, on the other hand, are useful when you already have 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 automatically set 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.

### TOGGLING A SIMPLE CODE BREAKPOINT

The easiest way to toggle a breakpoint is to place the insertion point in the C source statement, or assembler instruction, where you want the breakpoint and click the **Toggle Breakpoint** button on the **Debug** toolbar. Alternatively, you can toggle a breakpoint using the command **Edit>Toggle Breakpoint**. The breakpoint is marked in the C-SPY editor window with a highlight color:

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

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

Figure 49: Breakpoint on a function call

### SETTING A BREAKPOINT IN THE MEMORY WINDOW

For information about how to set breakpoints using the Memory window, see *Setting a breakpoint in the Memory window*, page 131.

## DEFINING BREAKPOINTS USING THE DIALOG BOX

The **Edit>Breakpoints** command displays the following dialog box which shows all currently defined breakpoints. Use it to edit existing breakpoints or to define new breakpoints.

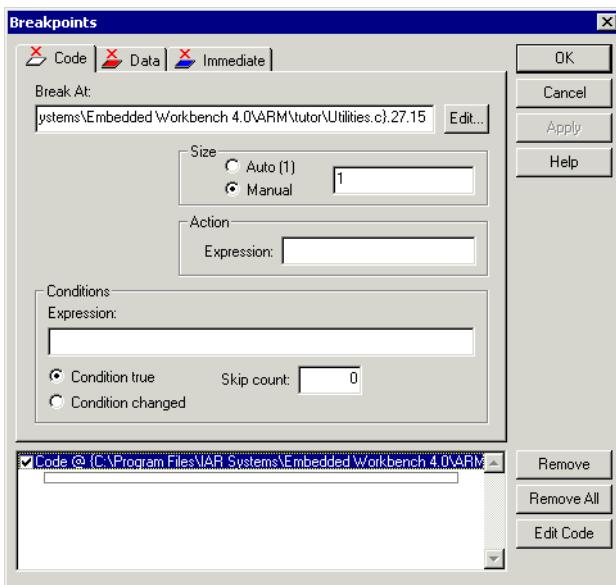


Figure 50: Breakpoints dialog box

To define a new breakpoint, choose the type of breakpoint you want to define by clicking the **Code**, **Data**, or **Immediate** tab. Specify the breakpoint settings according to your needs and click **Apply**. The breakpoint will appear in the list of breakpoints at the bottom of the dialog box.

For detailed information about the breakpoint settings, see *Breakpoints dialog box*, page 237.

### 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.

## Periodically monitoring data



If you are using a device that does not support the break-on-data capability, you can associate a condition with a code breakpoint. If the expression is FALSE, program execution resumes. If the expression is TRUE, C-SPY halts the program execution and displays the machine state. The breakpoint expression can be arbitrarily complex.

## 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.

If you use system macros for setting breakpoints it is still possible to view and modify them in the **Breakpoints** dialog box. 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.

This is a list of all available breakpoint macros:

```
__setCodeBreak  
__setConditionalBreak  
__setDataBreak  
__setRangeBreak  
__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 139.



# Monitoring memory and registers

This chapter describes how to use the features available in the IAR C-SPY™ Debugger for examining memory and registers:

- The Memory window
- The Register window
- Predefined and user-defined register groups
- The Stack window.

---

## Memory addressing

In C-SPY, the term *zone* is used for a named memory area. A memory address, or *location*, is a combination of a zone and a numerical offset into that zone. The MSP430 architecture has only one zone, *Memory*, which covers the whole MSP430 memory range. If you load a device description file, additional zones that adhere better to the specific device memory layout are defined.

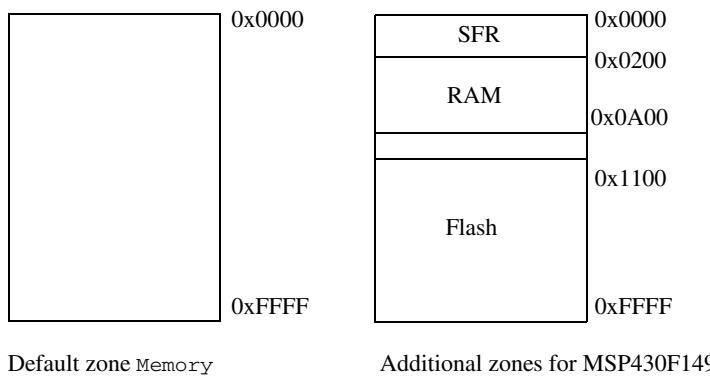


Figure 51: Zones in C-SPY

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.

Memory zones are defined in the device description files. For further information, see *Memory zones*, page 110.

## Using the 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 monitor different memory or register areas.

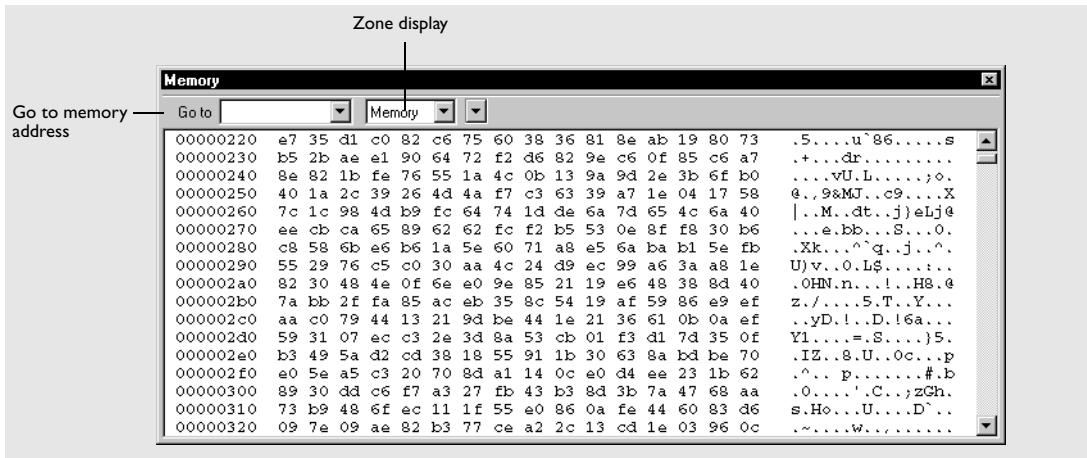


Figure 52: Memory window

The window consists of three columns. The left-most part displays the addresses currently being viewed. The middle part of the window displays the memory contents in the format you have chosen. Finally, the right-most part displays the memory contents in ASCII format. You can edit the contents of the Memory window, both in the hexadeciml part and the ASCII part of the window.

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

### Memory window operations

At the top of the window there are commands for navigation and configuration. These commands are also available on the context menu that appears when you right-click in the Memory window. In addition, commands for editing, opening the **Fill** dialog box, and setting breakpoints are available.

For reference information about each command, see *Memory window*, page 276.

## Memory Fill

The **Fill** dialog box allows you to fill a specified area of memory with a value.

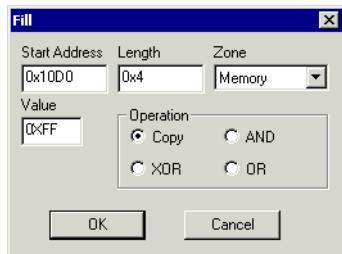


Figure 53: Memory Fill dialog box

For reference information about the dialog box, see *Fill dialog box*, page 278.

## 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** dialog box, which is available from the **Edit** 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.

**Note:** Setting different types of breakpoints in the Memory window is only supported if the driver you use supports these types of breakpoints.

## Working with registers

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.

Register			
CPU Registers			
PC	= 0xF862	R7	= 0x0FF5
SP	= 0x0278	R8	= 0x146B
SR	= 0x0000	R9	= 0x4C37
- Reserved	= 0x00	R10	= 0x002D
- Y	= 0	R11	= 0x0139
- SCG1	= 0	R12	= 0x0200
- SCG0	= 0	R13	= 0xFFFF
- OscOff	= 0	R14	= 0x0000
- CPUOff	= 0	R15	= 0x0000
- GIE	= 0	CYCLEDOUNTER	= 276
- N	= 0		
- Z	= 0		
- C	= 0		
R4	= 0x0139		
R5	= 0x408C		
R6	= 0x60B9		

Figure 54: 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 are expandable, which means that the register contains interesting 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 list 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 MSP430 debugger: **CPU Registers**.

In addition to the **CPU Registers** there are additional register groups predefined in the device description files—available in the `430/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 107.

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 design your own register groups that better suit the use of registers in your application.

To define new register groups, choose **Tools>Options** and click the **Register Filter** tab. This page is only available when the IAR C-SPY Debugger is running.

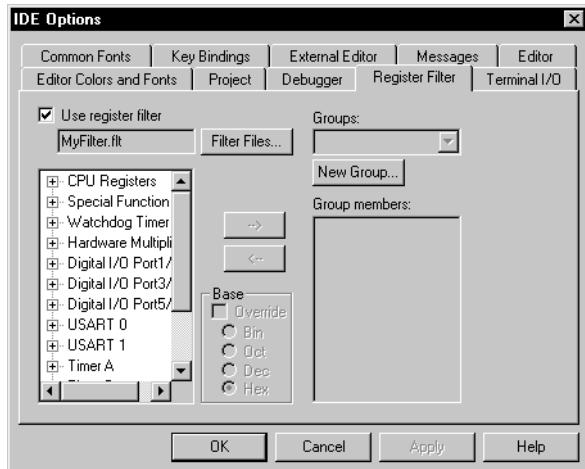


Figure 55: Register Filter page

For reference information about this dialog box, see *Register Filter page*, page 261.

## Using the Stack window

The Stack window—available from the **View** menu—provides an easy access to the memory area where the MSP430 microcontroller stack is located.

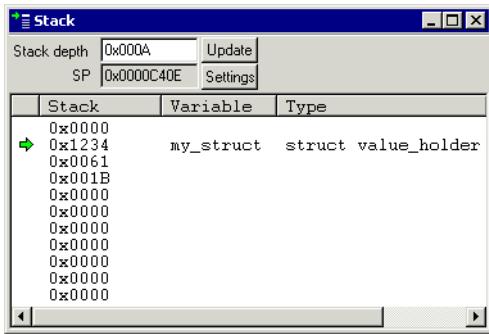


Figure 56: Measuring the stack

During application execution the green arrow continuously indicates the top of the stack.

Having a dedicated Stack window can be useful in many contexts, for instance for investigating the stack usage when assembler modules are called from C modules and vice versa, whether correct elements are located on the stack, and whether the stack is restored properly.

For reference information, see *Stack window*, page 292.

# Using C-SPY macros

The IAR C-SPY™ Debugger 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.
- 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 347.

### 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 IAR Embedded Workbench. 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 `430\tutor` directory.

For information about how to load a setup macro file, see *Registering and executing using setup macros and setup files*, page 139.

## 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 350.

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.

---

## 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 358.
- The **Quick Watch** dialog box 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.

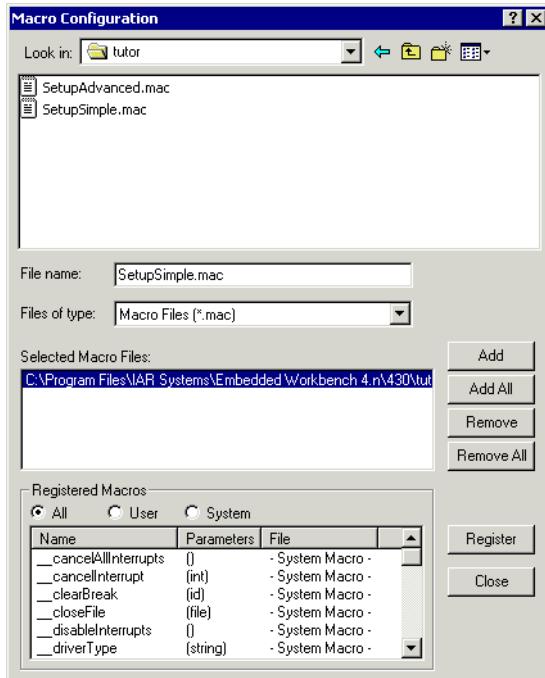


Figure 57: Macro Configuration dialog box

For reference information about this dialog box, see *Macro Configuration dialog box*, page 296.

## 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 the C-SPY Debugger.

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()
{
    __orderInterrupt("USART0RX_VECTOR", 4000, 2000, 0, 1, 0, 100);
}
```

This macro function generates a repeating interrupt that is first activated after 4000 cycles and then repeated approximately every 2000th cycle. Because the macro 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

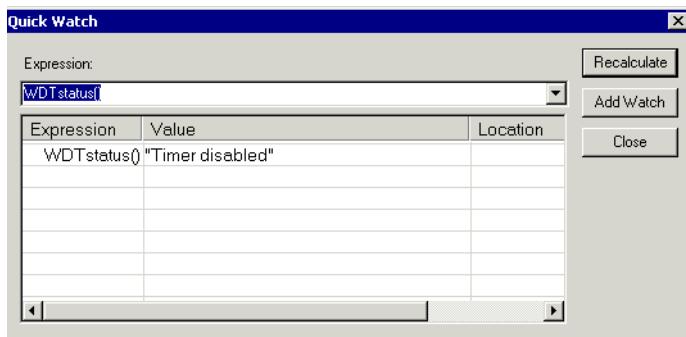
The **Quick Watch** dialog box—available from the **Debug** menu—lets you watch the value of any variables or expressions and evaluate them. For macros, the **Quick Watch** dialog box is especially useful because it is a method which lets you dynamically choose when to execute a macro function.

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

```
WDTstatus()
{
    if (#IE1 & 0x01 != 0) // Checks the status of WDTIE
        return "Timer enabled"; // C-SPY macro string used
    else
        return "Timer disabled"; // C-SPY macro string used
}
```

- 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 58: Quick Watch dialog box*

The macro will automatically be displayed in the **Quick Watch** dialog box.

Click **Close** to close the **Quick Watch** dialog box.

## **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:

- | 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 *Printing messages*, page 349.

For a complete example where a serial port input buffer is simulated using the method of connecting a macro to a breakpoint, see *Simulating an interrupt*, page 59.

# 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 is only supported by the IAR 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 memory which uses the most efficient addressing mode. For detailed information about efficient memory usage, see the *MSP430 IAR C/C++ Compiler Reference Guide*.

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.

### 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	Format Debug information for C-SPY

Table 15: Project options for enabling profiling



- | 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.

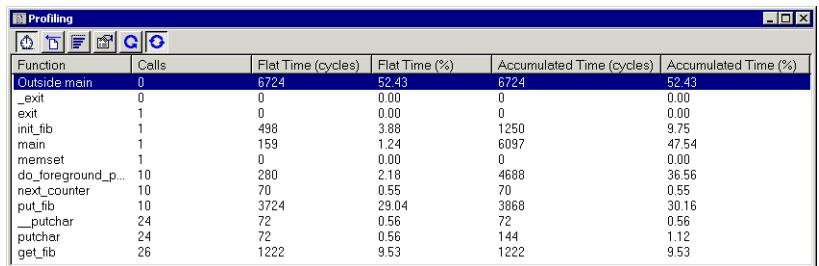


Figure 59: Profiling window

Profiling information is displayed in the window.

### Viewing the figures

Clicking on a column header sorts the complete 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 or C++ library code.



Clicking the **Graph** button toggles the percentage columns to be displayed either as numbers or as bar charts.

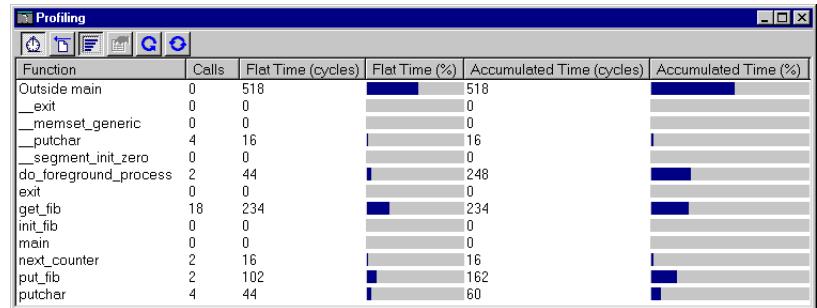


Figure 60: Graphs in Profiling window



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:

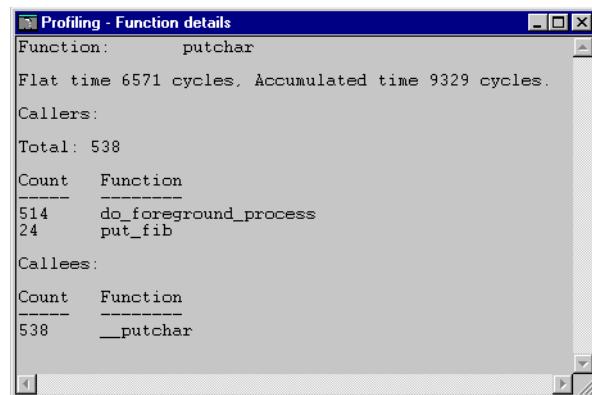


Figure 61: Function details window

## 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.

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	Format Debug information for C-SPY

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:

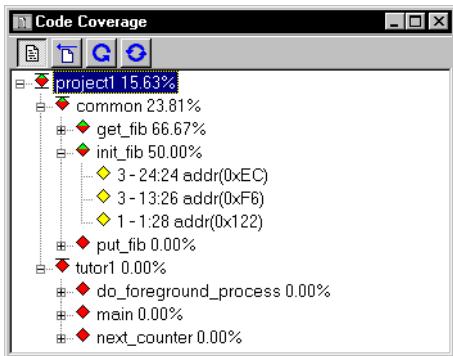


Figure 62: Code Coverage window

### 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. IAR C-SPY Simulator**

This part of the MSP430 IAR Embedded Workbench™ IDE User Guide contains the following chapters:

- Simulator introduction
- Simulator-specific characteristics
- Simulating interrupts.





# Simulator introduction

This chapter gives a brief introduction to the C-SPY Simulator.

It is assumed that you already have some working knowledge of the IAR Embedded Workbench and the IAR C-SPY Debugger. For a quick introduction, see *Part 2. Tutorials*.

---

## The IAR C-SPY Simulator

The IAR 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 IAR C-SPY Simulator also provides:

- Instruction accurate simulated execution
- Memory configuration and validation
- Interrupt simulation
- Immediate breakpoint with resume functionality
- Peripheral simulation (using the C-SPY macro system)
- Hardware multiplier simulation.

### SELECTING THE SIMULATOR DRIVER

Before starting the IAR C-SPY Debugger you must choose the simulator driver. In the IAR Embedded Workbench, choose **Project>Options** and click the **Setup** tab in the **Debugger** category. Choose **Simulator** from the **Driver** drop-down list.

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



# Simulator-specific characteristics

In addition to the features available in the C-SPY Debugger, the C-SPY Simulator provides some additional features. This chapter describes the additional menus and features provided by the Simulator driver.

You will get reference information, as well as information about how to use driver-specific characteristics, such as memory configuration and breakpoints.

---

## Simulator-specific menus

When you use the simulator driver, the **Simulator** menu is created in the menu bar.

### SIMULATOR MENU



Figure 63: Simulator menu

Menu item	Description
Forced Interrupts	Displays a window from which you trigger an interrupt.
Interrupts	Displays a dialog box to allow you to configure C-SPY interrupt simulation
Interrupt Log	Displays a window which shows the status of all defined interrupts.
Memory map	Displays a dialog box to allow you to configure different memory types by specifying a memory map with different access types.
Breakpoint Usage	Displays the Breakpoint Usage dialog box which lists all active breakpoints.

Table 17: Description of Simulator menu commands

For details about using the **Interrupts** dialog box, see the chapter *Simulating interrupts*.

### Breakpoint Usage dialog box

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

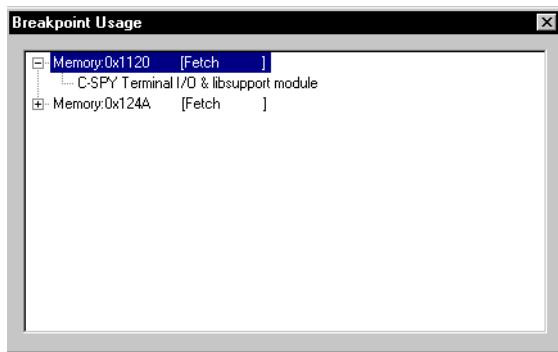


Figure 64: Breakpoint Usage dialog box

In addition to listing all active breakpoints that you have defined, it also lists the internal breakpoints that the debugger is using. The list of used internal breakpoints is useful when running on hardware that has a limited number of hardware breakpoints available. This gives you a possibility to check that the number of active breakpoints is within the number of breakpoints supported by the hardware. The listing does not include the single step breakpoint that is active only during C or C++ single step operations.

---

## Memory configuration

This section describes the facilities for simulating the memory configuration of the target hardware.

### MEMORY MAP

C-SPY allows the simulation of different memory access types by means of a memory map.

If a memory access occurs that violates the access type specified for a specific memory area, C-SPY will regard this as an illegal access. Execution will be halted and an error message will be displayed. The purpose of this feature is to help you to identify any memory access violations.

A memory map consists of specified memory areas where each memory area has an access type attached to it. The areas can either be the zones predefined in the device description file, or you can define your own memory areas. The access type can be **Read and Write**, **Read Only**, or **Write Only**. If the simulator tries to access memory outside of the mapped areas, write to read-only memory, or read from write-only memory, the simulation will stop.

The mapping system will strive to limit the number of ranges for efficiency reasons. This means that it will concatenate ranges whenever possible. If you define two ranges `0x0000–0x1FFF` and `0x2000–0x2FFF` of the same access type and in the same address space, the system will treat them as one range `0x0–0x2FFF`.

It is not possible to map two different access types to the same memory area.

Choose **Simulator>Memory Map** to open the **Memory Access Configuration** dialog box.

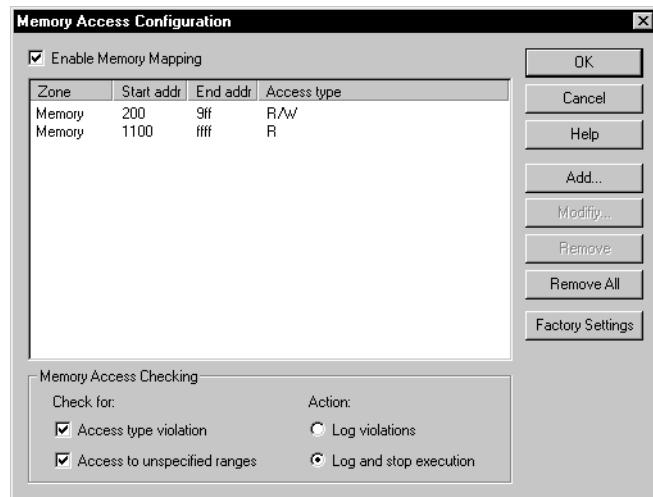


Figure 65: Memory Map dialog box

To define a new range, you can either:

- Click **Factory Settings** to load settings from a device description file that you specified before starting C-SPY
- Manually enter the characteristics of the memory you want to monitor.

If you manually have defined your own memory areas and then load factory settings, your own defined areas will be cleared before the preset definitions in the device description file are loaded.

To edit a defined memory area, select it in the list and click **Modify** to display or edit its properties, or **Remove** to delete it.

The settings you specify in the **Memory Access Configuration** dialog box are saved between debug sessions.

### **Memory Map configurations**

Use the **Enable memory mapping** check box to toggle memory mapping on and off. When memory mapping is disabled the definition remains, but no memory map checks will be performed.

For each memory area you can define the following properties:

Zone	Select the appropriate memory zone from the drop-down list.
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	Choose the access type from the drop-down list. The available types are <b>Read and Write</b> , <b>Read Only</b> , and <b>Write Only</b> .

## **Using breakpoints**

When 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. It is also possible to set immediate breakpoints.

### **SIZE ATTRIBUTE ON CODE AND DATA BREAKPOINTS**

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

To set a breakpoint on a range, you can either use the **Breakpoints** dialog box, or you can select an area in the Memory window. For information about how to specify a range using the dialog box, see *Breakpoints dialog box*, page 237.

### **IMMEDIATE BREAKPOINTS**

In addition to generic breakpoints, the C-SPY Simulator lets you set *immediate* breakpoints, which will temporarily halt, but not stop, the instruction execution. 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.

You specify this type of breakpoint and its characteristics on the **Immediate** page of the **Breakpoints** dialog box, available by choosing **Edit>Breakpoints**.

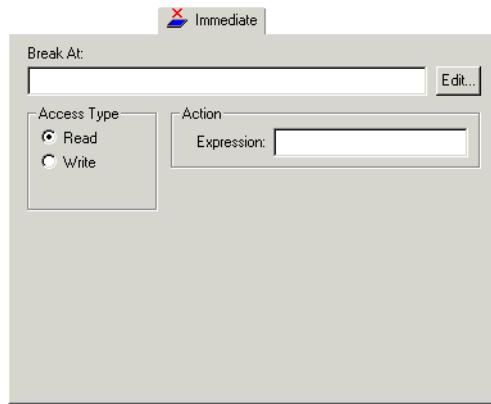


Figure 66: Immediate breakpoints page

For details of the breakpoint settings, see *Breakpoints dialog box*, page 237.

You can also set an immediate breakpoint by using the system macro

```
--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 <code>{filename}.line.col</code> (for example <code>{D:\\src\\prog.c}.12.9</code> ), although this is not very useful for simulation breakpoints. An <i>absolute location</i> on the form <code>zone:hexaddress</code> or simply <code>hexaddress</code> (for example <code>Memory: 0xE01E</code> ). An <i>expression</i> whose value designates a location (for example <code>main</code> ).
<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 18: `__setSimBreak` return values

See *Simulating an interrupt*, page 59, for an example where an immediate breakpoint is used.

---

## Hardware multiplier

The C-SPY Simulator can simulate the MSP430 hardware multiplier peripheral unit. To enable this simulation, choose **Project>Options** before starting C-SPY. Click the **Target** tab in the **General Options** category and select the option **Hardware multiplier**. The option is only enabled when you have chosen a device containing the hardware multiplier from the **Device** drop-down list.

# 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 *MSP430 IAR C/C++ Compiler Reference Guide*.

---

## The C-SPY interrupt simulation system

The IAR 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 MSP430 microcontroller
- Single-occasion or periodical interrupts based on the cycle counter
- Configuration of hold time, probability, and timing variation
- 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.

## 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*.

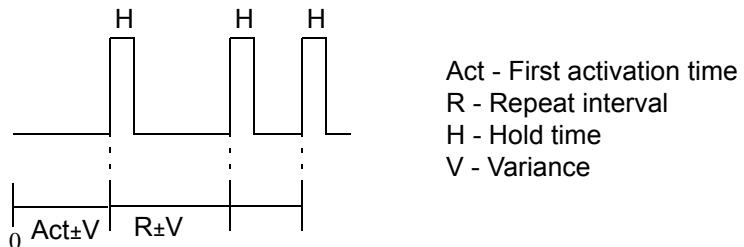


Figure 67: Simulated interrupt configuration

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.

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 **Interrupts** dialog box, or by using a system macro. Defined interrupts will be preserved until you remove them. All interrupts you define using the **Interrupts** dialog box are preserved between debug sessions.

## Using the interrupt simulation system

To use the interrupt simulation system, you should be familiar with the following elements:

- The method for adapting the interrupt system for MSP430
- The Forced Interrupt window
- The **Interrupts** and **Interrupt Setup** dialog boxes
- The C-SPY system macros for interrupts
- The Interrupt Log window

### ADAPTING THE INTERRUPT SYSTEM FOR MSP430

The MSP430 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. This information is provided in the device description files. You can find preconfigured `.ddf` files in the `430/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 109.

## INTERRUPTS DIALOG BOX

The **Interrupts** dialog box—available from the **Simulator** menu—lists all defined interrupts and provides you with a graphical interface where you interactively can fine-tune the interrupt simulation parameters. You can add the parameters and quickly test that the interrupt is generated according to your needs.

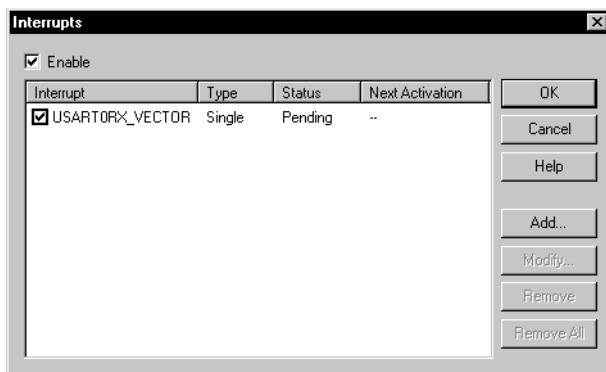


Figure 68: Interrupts dialog box

The option **Enable** 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 <b>Forced</b> , <b>Single</b> , or <b>Repeat</b> .
Status	Shows the status of the interrupt. The status can be <b>Pending</b> , <b>Executed</b> , or <b>Expired</b> .
Next Activation	Shows the next activation time in cycles.

Click **Add** or **Modify** to open the **Interrupt Setup** dialog box.

## INTERRUPT SETUP DIALOG BOX

Use the **Interrupt Setup** dialog box—available from the **Interrupts** dialog box—to add and modify interrupts.

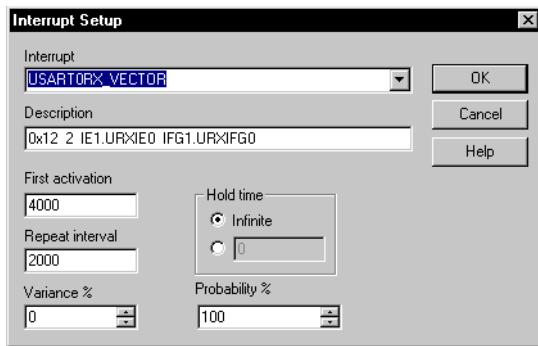


Figure 69: Interrupt Setup dialog box

### Interrupt Setup options

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 __orderInterrupt the Description field 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 <b>Infinite</b> , 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.

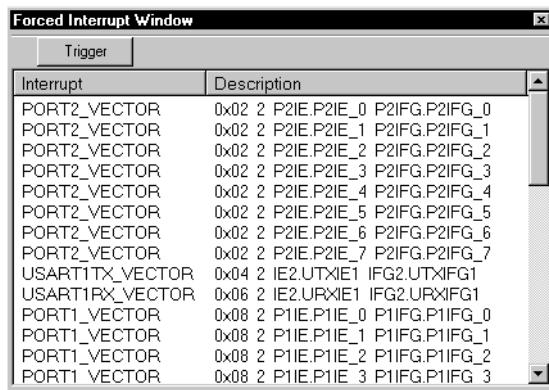


Figure 70: Forced Interrupt window

To force an interrupt, the interrupt simulation system must be enabled. To enable the interrupt simulation system, see *Interrupts dialog box*, page 162.

The Forced Interrupt window lists all available interrupts and their definitions. The description field is editable and the information 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.

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 19: Characteristics of a forced interrupt

## C-SPY SYSTEM MACROS FOR INTERRUPT

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()
```

The parameters of each macro corresponds to the equivalent entries of the **Interrupts** dialog box.

For detailed information of each macro, see *Description of interrupt system macros*, page 168.

### 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 139.

## INTERRUPT LOG WINDOW

The **Interrupt Log** window—available from the **Simulator** menu—contains runtime information about the interrupt system. The information is useful for debugging the interrupt handling in the target system.

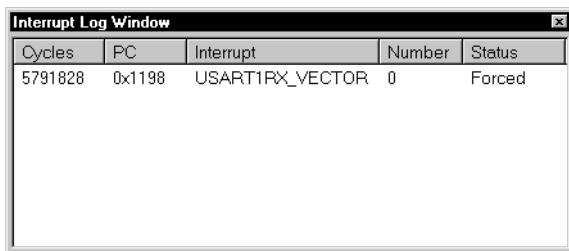


Figure 71: Interrupt Log window

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, Executed, and Expired. Triggered: The interrupt has passed its activation time Forced: The same as Triggered, but the interrupt has been forced from the Forced Interrupt window. Expired: When the interrupt hold time has expired without the interrupt being executed.

Table 20: Description of the Interrupt Log window

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

## Simulating a simple interrupt

In this example you will simulate a timer interrupt. However, the procedure can also be used for other types of interrupts.

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

```
#include <io430x41x.h>
int ticks = 0;
void main (void)
{
    //Timer setup code
    WDTCTL = WDTPW + WDTHOLD;      //Stop WDT
    IE2 |= BTIE;                  //Enable BT interrupt
    BTCTL = BTSSEL+BTIP2+BTIP1+BTIP0;
    _EINT();                      //Enable interrupts

    while (ticks < 100);           //Endless loop
    printf("Done\n");
}

// Timer interrupt service routine
#pragma vector = BASICTIMER_VECTOR
__interrupt void basic_timer(void)
{
    ticks += 1;
}
```

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** C-SPY needs information about the interrupt to be able to simulate it. This information is provided in the device description files. The correct file will automatically be used when you select a device from the **Device** drop-down menu; choose **Project>Options**, click the **Setup** tab in the **Debugger** category, and select a device.
- 3** Build your project and start the simulator.
- 4** Choose **Simulator>Interrupts** to open the **Interrupts** dialog box. Make sure interrupt simulation is enabled. For the BasicTimer example, click **Add** and add the following settings:

Option	Settings
Interrupt	BASICTIMER_VECTOR
First activation	4000
Repeat interval	2000

Table 21: Timer interrupt settings

Option	Settings
Variance %	0
Hold time	0
Probability %	100

*Table 21: Timer interrupt settings (Continued)*Click **OK**.

- 5 Execute your application. If you have enabled the interrupt properly in your application source code, C-SPY will:
- Generate an interrupt when the cycle counter has passed 4000
  - Continuously repeat the interrupt after approximately 2000 cycles.

## Description of interrupt system macros

This section gives reference information about each of the C-SPY system macros for interrupts.

---

`__cancelAllInterrupts` `__cancelAllInterrupts()`

### Return value

`int 0`

### Description

Cancels all ordered interrupts.

---

`__cancelInterrupt` `__cancelInterrupt(interrupt_id)`

### Parameter

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

### Return value

Result	Value
Successful	<code>int 0</code>
Unsuccessful	Non-zero error number

*Table 22: \_\_cancelInterrupt return values*

---

<b>Description</b>	
Cancels the specified interrupt.	
<code>__disableInterrupts</code>	<code>__disableInterrupts()</code>
<b>Return value</b>	
<b>Result</b>	<b>Value</b>
Successful	int 0
Unsuccessful	Non-zero error number

*Table 23: \_\_disableInterrupts return values*

---

<b>Description</b>	
Disables the generation of interrupts.	
<code>__enableInterrupts</code>	<code>__enableInterrupts()</code>
<b>Return value</b>	
<b>Result</b>	<b>Value</b>
Successful	int 0
Unsuccessful	Non-zero error number

*Table 24: \_\_enableInterrupts return values*

---

<b>Description</b>	
Enables the generation of interrupts.	
<code>__orderInterrupt</code>	<code>__orderInterrupt(specification, first_activation, repeat_interval, variance, infinite_hold_time, hold_time, probability)</code>
<b>Parameters</b>	
<i>specification</i>	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.
<i>first_activation</i>	The first activation time in cycles (integer)

<i>repeat_interval</i>	The periodicity in cycles (integer)
<i>variance</i>	The timing variation range in percent (integer between 0 and 100)
<i>infinite_hold_time</i>	1 if infinite, otherwise 0.
<i>hold_time</i>	The hold time (integer). This value is only used if <i>infinite_hold_time</i> is set to 0.
<i>probability</i>	The probability in percent (integer between 0 and 100)

### Return value

The macro returns an interrupt identifier (unsigned long).

If the syntax of *specification* is incorrect, it returns -1.

### Description

Generates an interrupt.

### Example

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

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

# Part 6. IAR C-SPY FET Debugger

This part of the MSP430 IAR Embedded Workbench™ IDE User Guide contains the following chapters:

- Introduction to the IAR C-SPY FET Debugger
- C-SPY FET driver-specific characteristics
- C-SPY FET Debugger options and menus
- Design considerations for in-circuit programming.





# Introduction to the IAR C-SPY FET Debugger

This chapter introduces you to the IAR C-SPY Flash Emulation Tool Debugger (C-SPY FET Debugger), as well as to how it differs from the C-SPY Simulator. This chapter describes how you install the hardware and then run the demo applications. The chapter also briefly describes the communication between the C-SPY FET driver and the target system, and gives some troubleshooting hints.

The chapters specific to the C-SPY FET Debugger assumes that you already have some working knowledge of the FET Debugger, as well as some working knowledge of the IAR C-SPY Debugger. For a quick introduction, see *Part 2. Tutorials*.

Note that additional features may have been added to the software after the *MSP430 IAR Embedded Workbench™ IDE User Guide* was printed. The readme file cs430f.htm contains the latest information.

---

## The FET C-SPY Debugger

The MSP430 microcontroller has built-in, on-chip debug support. To make the C-SPY FET Debugger work, a communication driver must be installed on the host PC. This driver is automatically installed during the installation of the IAR Embedded Workbench. Because the hardware debugger kernel is built into the microcontroller, no ordinary ROM-monitor program or extra specific hardware is needed to make the debugging work. It is also possible to use the debugger on your own hardware design.

The C-SPY FET Debugger provides general C-SPY Debugger features, and features specific to the C-SPY FET driver. For detailed information about the general debugger features, see *Part 4. Debugging* in this guide.

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

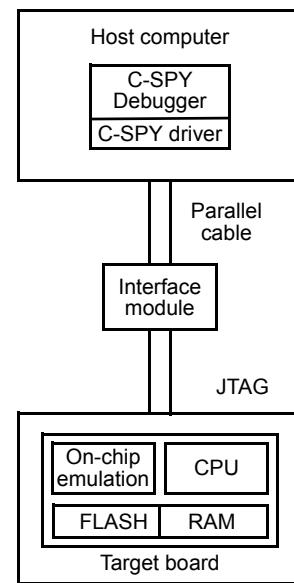


Figure 72: Communication overview

For more details about the communication, see *C-SPY FET communication*, page 186.

## SUPPORTED DEVICES

For a complete list of supported devices, see the `cs430.htm` readme file.

## DIFFERENCES BETWEEN THE C-SPY FET AND THE SIMULATOR DRIVERS

The following table summarizes the key differences between the FET and simulator drivers:

Feature	Simulator	FET
OP-fetch	x	x
Data breakpoints <sup>1</sup>	x	x
Execution in real time		x

Table 25: Simulator and FET differences

Feature	Simulator	FET
Simulated interrupts	x	
Real interrupts		x
Cycle counter <sup>2</sup>	x	x
Code coverage	x	
Profiling	x	x <sup>3</sup>
Enhanced Emulation Module support		x

*Table 25: Simulator and FET differences (Continued)*

1. Data breakpoints are supported for the devices with the Enhanced Emulation module. For further details, see *Using breakpoints*, page 179.
2. Cycle counter is supported during single step, you can then view the value of the cycle counter in the Register window.
3. The FET Debugger must single step during profiling.

---

## Hardware installation

### HARDWARE INSTALLATION, MSP-FET430X110

- 1 Connect the 25-conductor cable originating from the FET to the parallel port of your PC.
- 2 Ensure that the MSP430 device is securely seated in the socket, and that its pin 1 (indicated with a circular indentation on the top surface) aligns with the 1 mark on the PCB.
- 3 Ensure that jumpers J1 (near the non-socketed IC on the FET) and J5 (near the LED) are in place.

### HARDWARE INSTALLATION, MSP-FET430PXX0

('P120, 'P140, 'P410, 'P440)

- 1 Use the 25-conductor cable to connect the FET Interface module to the parallel port of your PC.
- 2 Use the 14-conductor cable to connect the FET Interface module to the Target Socket module.
- 3 Ensure that the MSP430 device is securely seated in the socket, and that its pin 1 (indicated with a circular indentation on the top surface) aligns with the 1 mark on the PCB.
- 4 Ensure that the two jumpers (LED and Vcc) near the 2x7 pin male connector are in place.

## Getting started

This section demonstrates two demo applications—one in assembler language and one in C—that flash the LED. The applications are built and downloaded to the FET Debugger, and then executed.

There is one demo workspace file supplied with the C-SPY FET Debugger `fet_projects.eww`. This workspace contains two projects per FET variant—one in C and one in assembler. The files are provided in the directory `430\FET_examples`.

The majority of the examples use the various resources of the MSP430 to time the flashing of the LED.

**Note:** The examples often assume the presence of a 32kHz crystal, and not all FET Debuggers are supplied with a 32kHz crystal.

### RUNNING A DEMO APPLICATION

The following examples assume that you are using an MSP430F149 device. See the HTML document which `FET project suits my device.htm`.

#### C Example

- 1 In the Embedded Workbench, choose **File>Open Workspace** to open the workspace file `fet_projects.eww`.
- 2 To display the C project, click the appropriate project tab at the bottom of the workspace window, for instance `fet140_1_C`.  
If you want to run the application for a different FET Debugger, click the appropriate project tab.
- 3 Select the **Debug** build configuration from the drop-down list at the top of the workspace window.
- 4 Choose **Project>Options**. In addition to the factory settings, verify the following settings:

Category	Page	Option/Setting
General Options	Target	Device: msp430F149
C/C++ Compiler	Output	Generate debug info
Debugger	Setup	Driver: FET Debugger
FET Debugger	Setup	<b>Deselect</b> Suppress download

Table 26: Project options for FET C example

For more information about the C-SPY FET Debugger options and how to configure C-SPY to interact with the target board, see *C-SPY FET Debugger options and menus*, page 189.

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

- 5** Choose **Project>Make** to compile and link the source code.
- 6** Start C-SPY by clicking the **Debug** button or by choosing **Project>Debug**. C-SPY will erase the flash memory of the device, and then download the application to the target system.
- 7** In C-SPY, choose **Debug>Go** or click the **Go** button to start the application. The LED should flash.
- 8** Click the **Stop** button to stop the execution.

### Assembler example

- 1** In the Embedded Workbench, choose **File>Open Workspace** to open the workspace file `fet_projects.eww`.
- 2** To display the assembler project, click the appropriate project tab at the bottom of the workspace window, for instance `fet140_1_asm`.  
If you want to run the application for a different FET Debugger, click the appropriate project tab.
- 3** Select the **Debug** build configuration from the drop-down list at the top of the workspace window.
- 4** Choose **Project>Options**. In addition to the factory settings, verify the following settings:

Category	Page	Option/Setting
General Options	Target	Device: msp430F149
C/C++ Compiler	Output	Generate debug info
Debugger	Setup	Driver: FET Debugger
FET Debugger	Setup	<b>Deselect</b> Suppress download

Table 27: Project options for FET assembler example

For more information about the C-SPY FET Debugger options and how to configure C-SPY to interact with the target board, see *C-SPY FET Debugger options and menus*, page 189.

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

- 5** Choose **Project>Make** to assemble and link the source code.

- 6 Start C-SPY by clicking the **Debug** button or by choosing **Project>Debug**. C-SPY will erase the flash memory of the device, and then download the application object file to the target system.
- 7 In C-SPY, choose **Debug>Go** or click the **Go** button to start the application. The LED should flash.
- 8 Click the **Stop** button to stop the execution.

# C-SPY FET driver-specific characteristics

This chapter describes the additional features provided by the C-SPY FET Debugger driver. There are certain restrictions and prerequisites you need to know about, to debug your code successfully.

You will also find descriptions of memory configuration and FET Debugger communication with some common troubleshooting tips.

---

## Using breakpoints

With the C-SPY FET Debugger you can set *code* breakpoints. If you are using a device that supports the *Enhanced Emulation module* you also have access to an extended breakpoint system with support for:

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

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

### HARDWARE AND VIRTUAL BREAKPOINTS

To set breakpoints, the C-SPY FET Debugger uses the hardware breakpoints available on the device. When all hardware breakpoints are used, C-SPY can use *virtual breakpoints*, which means that you can set an unlimited amount of breakpoints.

The number of available hardware breakpoints for each device is:

Device	Breakpoints (N)
MSP430F11x1	2
MSP430F11x2	2
MSP430F12x	2

Table 28: Available hardware breakpoints

Device	Breakpoints (N)
MSP430F12x2	2
MSP430F13x	3
MSP430F14x	3
MSP430F15x	8
MSP430F16x	8
MSP430F41x	2
MSP430F42x	2
MSP430F43x	3
MSP430F44x	8

*Table 28: Available hardware breakpoints (Continued)*

If there are N or fewer breakpoints active, C-SPY will always operate at full speed. If there are more than N breakpoints active, and virtual breakpoints are enabled, C-SPY will be forced to single step between the breakpoints. This means that execution will not be at full speed.

## SYSTEM BREAKPOINTS

Sometimes C-SPY must set breakpoints for internal use. These breakpoints are called *system breakpoints*. In the CLIB runtime environment, C-SPY will set a system breakpoint when:

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

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

C-SPY will also set a temporary system breakpoint when:

- the command **Edit>Run to Cursor** is used
- the option **Run to** is selected

The system breakpoints will use hardware breakpoints when available. When the number of available hardware breakpoints is exceeded, virtual breakpoints will be used instead.

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

## CUSTOMIZING THE USE OF BREAKPOINTS

It is possible to prevent the debugger from executing in single-step mode. You do this by disabling the use of virtual breakpoints and—in the CLIB runtime environment—by fine-tuning the use of system breakpoints. This will increase the performance of the debugger, but you will only have access to the available number of hardware breakpoints. For further information about the necessary options, see *FET Debugger setup*, page 190.

To monitor all currently used breakpoints, choose **Emulator>Show used breakpoints** to open the **Used Breakpoints** window.

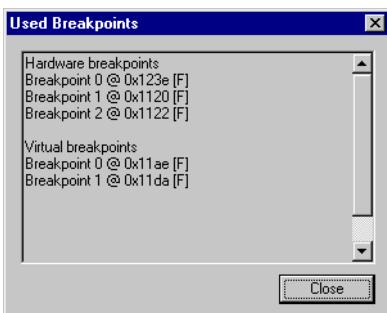


Figure 73: The Used Breakpoints window

All used hardware and virtual breakpoints are listed, as well as all currently defined conditional breakpoints.

### Periodically monitoring data



If you are using a device that does not support the Enhanced Emulation Module, the break-on-data capability of the MSP430 is not utilized. In that case, breakpoints can only be set to occur during an instruction fetch. However, C-SPY provides a non-realtime data breakpoint mechanism, which lets you periodically monitor data without using data breakpoints. For a description of the data breakpoint mechanism, see the chapter *Using breakpoints*.

### Using breakpoints when programming flash memory



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

---

## Stepping

Be aware that stepping might cause some unexpected side-effects.

### PROGRAMMING FLASH

Multiple internal machine cycles are required to clear and program the flash memory. When single-stepping over instructions that manipulate the flash, control is given back to C-SPY before these operations are complete. Consequently, C-SPY will update its memory window with erroneous information. A workaround to this behavior is to follow the flash access instruction with a NOP instruction, and then step past the NOP before reviewing the effects of the flash access instruction.

### SINGLE-STEPPING WITH ACTIVE INTERRUPTS

When you single-step with active and enabled interrupts, it can seem as if only the interrupt service routine (ISR) is active. That is, the non-ISR code never appears to execute, and the single-step operation always stops on the first line of the ISR. However, this behavior is correct because the device will always process an active and enabled interrupt.

There is a workaround for this behavior. While within the ISR, disable the `GIE` bit on the stack so that interrupts will be disabled after exiting the ISR. This will permit the non-ISR code to be debugged (but without interrupts). Interrupts can later be re-enabled by setting `GIE` in the status register in the Register window.

On devices with Clock Control, it may be possible to suspend a clock between single steps and delay an interrupt request.

---

## Using state storage

The state storage module is a limited variant of a traditional trace module. It can store eight states and can be used for monitoring program states or program flow, without interfering with the execution. The state storage module is only available if you are using a device that supports the Enhanced Emulation Module.

To use the state storage module, you must:

- | Define one or multiple range breakpoints or conditional breakpoints that you want to trigger the state storage module. In the Edit Breakpoints dialog box, make sure to select the action **State Storage Trigger**. This means that the breakpoint is defined as a state storage trigger. (State storage can also be triggered from the Sequencer Control window.)

Note that depending on the behavior you want when the state storage module is triggered, it is useful to consider the combination of the **Action** options and the options available in the State Storage Control window. See the examples following immediately after these steps.

- 2 Choose **Emulator>State Storage Control** to open the State Storage Control window.
- 3 Select the option **Enable state storage**. Set the options **Buffer wrap around**, **Trigger action**, and **Storage action** according to your preferences.

In the list **State Storage Triggers**, all breakpoints defined as state storage triggers are listed.

For further details about the options, see *State Storage Control window*, page 203.

- 4 Click **Apply**.
- 5 Choose **Emulator>State Storage window** to open the State Storage window.
- 6 Choose **Debug>Go** to execute your application. Before you can view the state storage information, you must stop the execution. You can do this, for instance, by using the **Break** command.

For information about the window contents, see *State Storage Window*, page 205.

## Example

Assume the following setup:

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

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

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

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

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

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

## Using the sequencer

The sequencer module lets you break the execution or trigger the state storage module by using a more complex method than a standard breakpoint. This is useful if you want to stop the execution under certain conditions, for instance a specific program flow. You can combine this with letting the state storage module continuously store information. At the time when the execution stops, you will have useful state information logged in the State storage window.

Consider this example:

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

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

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

To achieve this, you can:

- 1 Set a breakpoint on the statement `my_putchar('@');`.
- 2 Set another breakpoint on the suspected code within the function `my_putchar()`.
- 3 Define these breakpoints as transition triggers. Choose **Emulator>Sequencer Control** to open the Sequencer Control window. Select the option **Enable sequencer**.

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

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

Table 29: Sequencer settings - example

The transition trigger 1 depends on the transition trigger 0. This means that the execution will stop only when the function `my_putchar()` is called by the function call `my_putchar('@');`

Click **OK**.

- 5** Now you should set up the state storage module. Choose **Emulator>State Storage Control** to open the State Storage Control window. Make sure the following options are selected:

Option	Setting
Enable state storage	Selected
Buffer wrap around	Selected
Storage action	Instruction fetch
Trigger action	None

Table 30: State Storage Control settings - example

Click **OK**.

- 6** Start the program execution. The state storage module will continuously store trace information. Execution stops when the function `my_putchar()` has been called by the function call `my_putchar('@');`
- 7** Choose **Emulator>State Storage Window** to open the **State Storage** window. You can now examine the stored trace information. For further details, see *State Storage Window*, page 205.
- 8** When the sequencer is in state 3, C-SPY's breakpoint mechanism—which is used for all breakpoints, not only transition triggers—can be locked. Therefore, you should always end the session with one of these steps:
- Disabling the sequencer module. This will restore all breakpoint actions.
  - Resetting the state machine by clicking the **Reset States** button. The sequencer will still be active and trigger on the defined setup during the program execution.

## Memory configuration

No special considerations regarding memory configuration are needed for the FET C-SPY Debugger. You can download your application into RAM, or non-volatile memory.

Your application will be linked according to the directives specified in the linker command file, and your application will automatically be downloaded to the appropriate memory addresses.

You can find detailed information about the linker command file and the usage of the different segments in the *MSP430 IAR C/C++ Compiler Reference Guide*. Detailed information about the segment control directive `-z` can be found in the *IAR Linker and Library Tools Reference Guide*.

## C-SPY FET communication

C-SPY uses the JTAG pins of the device to debug the device. On some MSP430 devices, these JTAG pins are shared with the device port pins. Normally, C-SPY maintains the pins in JTAG mode so that the device can be debugged. During this time the port functionality of the shared pins is not available.

### RELEASING JTAG

When you choose **Emulator>Release JTAG on Go**, the JTAG drivers are set to tri-state and the device will be released from JTAG control (the TEST pin is set to GND) when GO is activated. Any active on-chip breakpoints are retained and the shared JTAG port pins revert to their port functions.

At this time, C-SPY has no access to the device and cannot determine if an active breakpoint has been triggered. C-SPY must be manually told to stop the device, at which time the state of the device will be determined (that is, has a breakpoint been reached?).

If you choose **Emulator>Release JTAG on Go**, the JTAG pins will be released if, and only if, there are N or fewer active breakpoints.

When making current measurements of the device, ensure that the JTAG control signals are released (**Emulator>Release JTAG on Go**), otherwise the device will be powered by the signals on the JTAG pins and the measurements will be erroneous.

### PARALLEL PORT DESIGNATORS

The parallel port designators (LPTx) have the following physical addresses: LPT1: 0x378, LPT2: 0x278, LPT3: 0x3BC. The configuration of the parallel port (ECP, Compatible, Bidirectional, Normal) is not significant; ECP is recommended.

## TROUBLESHOOTING

If establishing communication between the C-SPY FET driver and the target system fails, possible solutions to this problem include:

- Restart your host computer.
- Ensure that R6 on the MSP-FET430X110 and the FET Interface module has a value of 82 ohms. Early units were built using a 330-ohm resistor for R6. The FET Interface module can be opened by inserting a thin blade between the case halves, and then carefully twisting the blade to pry the case halves apart.
- Ensure that the correct parallel port has been specified in the options category **FET Debugger** available from the **Project>Options** menu. Check the PC BIOS for the parallel port address (0x378, 0x278, 0x3BC), and the parallel port configuration (ECP, Compatible, Bidirectional, or Normal).
- Ensure that no other software application has reserved/taken control of the parallel port (for instance, printer drivers, ZIP drive drivers, etc.). Such software can prevent the C-SPY FET driver from accessing the parallel port, and therefore also from communicating with the device.
- Revisions 1.0, 1.1, and 1.2 of the FET Interface module require a hardware modification; a 0.1 $\mu$ F capacitor needs to be installed between U1 pin 1 (signal VCC\_MSP) and ground. A convenient (electrically equivalent) installation point for this capacitor is between pins 4 and 5 of U1.

**Note:** The hardware modification may already have been performed during manufacturing, or your tool might contain an updated version of the FET Interface module.

- Revisions 0.1 and 1.0 of the MSP-TS430PM64 Target Socket module require a hardware modification; the PCB trace connecting pin 6 of the JTAG connector to pin 9 of the MSP430 (signal XOUT) needs to be severed.

**Note:** The hardware modification may have already been performed during manufacturing, or your tool might contain an updated version of the Target Socket module.

Also note that if the modified Target Socket module is used with the PRGS, Version 1.10 or later of the PRGS software is required.

For revisions 1.0, 1.1, and 1.2 of the FET Interface module, install a 0.1 $\mu$ F capacitor between the indicated points (pins 4 and 5 of U1).

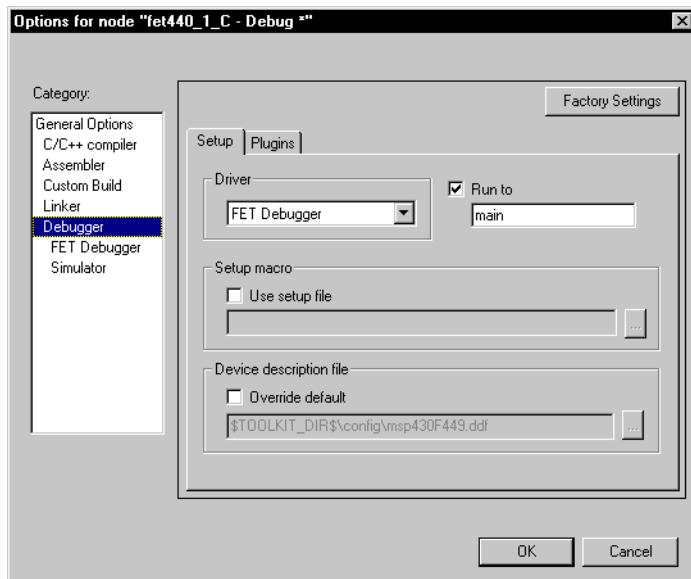


# C-SPY FET Debugger options and menus

This section describes the additional options and menus specific to the C-SPY FET Debugger.

## Setup options

On the **Setup** page, select the appropriate driver. In this case, select the **FET Debugger** driver:



To restore all settings to the default factory settings, click the **Factory Settings** button.

For information about the options **Setup macro** and **Device description file**, see *Setup*, page 343.

## FET Debugger setup

The **FET Debugger** category contains all the options specific to the C-SPY FET Debugger.

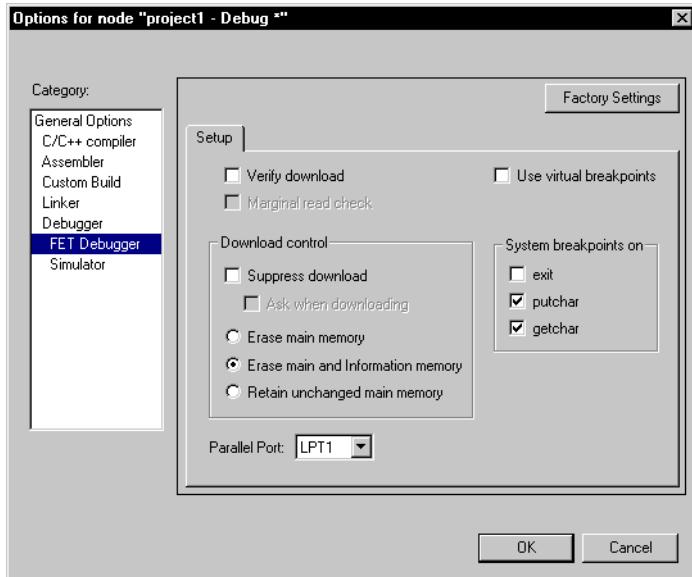


Figure 74: Flash Emulation Tool options

### VERIFY DOWNLOAD

The option **Verify download** verifies that the program data has been correctly transferred from the driver to the device. This verification does increase the programming sequence time.

### MARGINAL READ CHECK

Marginal read check is a feature, available on some devices, that detects weak or aged flash cells by setting the flash controller to dedicated test states, rereading the flash region, and comparing the results to the original flash contents. The results are written to the Debug Log window.

## DOWNLOAD CONTROL

If you already have your application in flash memory, select **Suppress download**. Select **Ask when downloading** to make C-SPY prompt for each download whether download should be suppressed or not.

The option **Erase main memory** erases only the main flash memory before download. The information memory is not erased. Whereas, the option **Erase main and Information memory** erases both flash memories before download.

When you select the option **Retain unchanged main memory** the information and main flash memories are read into a buffer. Only the flash segments needed are erased. If data that are to be written into a segment match the data on the target, the data on the target is left as is. No download is performed.

The new data effectively replaces the old data, and unaffected old data is retained.

## PARALLEL PORT

The drop-down list **Parallel Port** lets you select which parallel port to use. The available settings are **LPT1**, **LPT2**, and **LPT3**.

## USE VIRTUAL BREAKPOINTS

The option **Use virtual breakpoints** allows C-SPY to use virtual breakpoints when all available hardware breakpoints have been used. When virtual breakpoints are used, C-SPY is forced into single-step mode.

To prevent C-SPY from entering single-step mode, disable this option. In this case C-SPY will not use virtual breakpoints, even though all hardware breakpoints are already used. For further information about how to use breakpoints, see *Using breakpoints*, page 179.

## SYSTEM BREAKPOINTS ON

The option **System breakpoints on** can be used for fine-tuning the use of system breakpoints in the CLIB runtime environment. If the C-SPY Terminal I/O window is not required or if you do not need a breakpoint on the `exit` label, you can save hardware breakpoints by not reserving system breakpoints. Select or deselect the options **exit**, **putchar**, and **getchar** respectively, if you want, or not want, C-SPY to use system breakpoints for these. For further information about breakpoints, see *Using breakpoints*, page 179.

In the DLIB runtime environment, C-SPY will always set a system breakpoint on the `__DebugBreak` label. You cannot disable this behavior.

## Range breakpoints

*Range breakpoints* can be set in the **Breakpoints** dialog box, available from the **Edit** menu. They can be set on a data or an address range, and the action can be specified to occur on an access inside or outside the specified range.

These breakpoints are only available if you are using a device that supports the Enhanced Emulation Module.

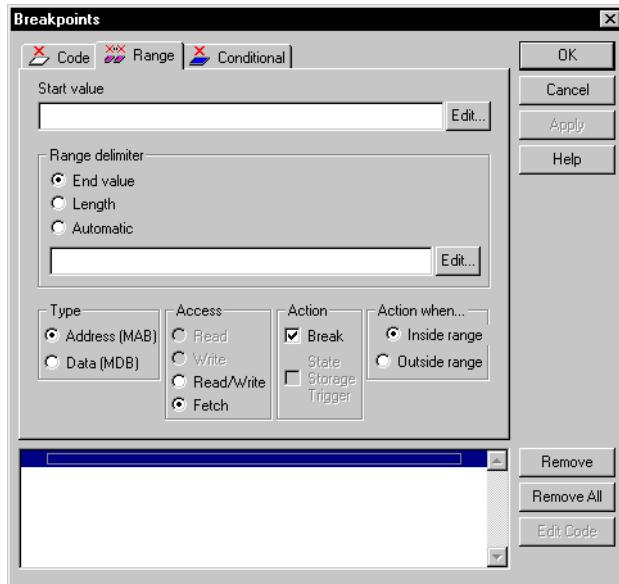


Figure 75: Range breakpoints dialog box

### SETTING A RANGE BREAKPOINT IN THE BREAKPOINTS DIALOG BOX

Defining a range breakpoint consists of specifying the options:

- Start value
- Range delimiter
- Type
- Access
- Action
- Action when.

The breakpoints you define appear in the list at the bottom of the dialog box.

## Start value

Set the start value location for the range breakpoint using the **Edit** button. These are the locations you can choose between and their possible settings:

Location	Description/Examples
Expression	Any expression that evaluates to a valid address, such as a variable name. For example, my_var refers to the location of the variable my_var, and arr[3] refers to the third element of the array arr.
Absolute Address	An absolute location on the form zone:hexaddress or simply hexaddress. Zone specifies in which memory the address belongs. For example Memory:0x42 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 program using the syntax {file path}.row.column 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. For example, {C:\IAR Systems\xxx\Utilities.c}.22.3 sets a breakpoint on the third character position on line 22 in the source file Utilities.c.

Table 31: Range breakpoint start value types

## Range delimiter

This option sets the end location of the range. It can be one of the value types used for the **Start value**, the **Length** of the range in hexadecimal notation, or **Automatic**. Automatic means that the range 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 range of the breakpoint will be 12 bytes.

## Type

To choose which breakpoint type to use, select one of the following options:

Breakpoint type	Description
<b>Address (MAB)</b>	Sets a breakpoint on a specified address, or anything that can be evaluated to such. The breakpoint is triggered when the specified location is accessed. If you have set the breakpoint on a specific machine instruction, the breakpoint will be triggered and the execution will stop, exactly before the instruction will be executed.
<b>Data (MDB)</b>	Sets a breakpoint on a specified value. It is the value on the data bus that triggers the breakpoint.

Table 32: Range breakpoint types

## Access type

You can specify the type of access that triggers the selected breakpoint. Select one of the following options:

Access	Description
Read	Read from location.
Write	Write to location.
Read/Write	Read from or write to location.
Fetch	At instruction fetch.

Table 33: Range breakpoint access types

## Action

There are two action options—**Break** and **State Storage Trigger**.

If you select the option **Break**, the execution will stop when the breakpoint is triggered.

If you select the option **State Storage Trigger**, the breakpoint is defined as a state storage trigger. To define the behavior of the state storage module further, use the options in the State Storage Control window.

## Action when

Specifies whether the action should occur at an access inside or outside of the specified range.

## USING A SYSTEM MACRO TO SET A RANGE BREAKPOINT

You can also use the system macro to set a range breakpoint.

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

## Parameters

All parameters are strings.

<i>start_loc</i>	The start location. This can be either: A <i>source location</i> on the form " <i>{filename}.line.col</i> " (for example " <i>D:\src\prog.c</i> .12.9")
	An <i>absolute location</i> on the form " <i>zone:hexaddress</i> " or simply " <i>hexaddress</i> " (for example "Memory:0x42")
	An <i>expression</i> whose value designates a location (for example " <i>my_global_variable</i> ").
<i>end_loc</i>	The end location. This can be either the same as for <i>start_loc</i> above or the length of the range.
<i>end_cond</i>	The type of end condition, either "Location", "Length", or "Automatic".
<i>type</i>	The breakpoint type; either "Address" or "Data".
<i>access</i>	The memory access type: "Read", "Write", "ReadWrite", or "Fetch".
<i>action</i>	The action type: "Break", "Trigger", or "BreakTrigger".
<i>action_when</i>	Specifies if the action should happen at an access inside or outside of the specified range, either "Inside" or "Outside".

## Return value

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

Table 34: \_\_setRangeBreak return values

This macro is only applicable to the FET Debugger version of C-SPY.

## Example

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

```
...
__clearBreak(brk);
```

For additional information, see *Defining breakpoints*, page 124.

## Conditional breakpoints

*Conditional breakpoints* can be set in the **Breakpoints** dialog box, which is available from the **Edit** menu. These breakpoints are only available if you are using a device that supports the Enhanced Emulation Module.

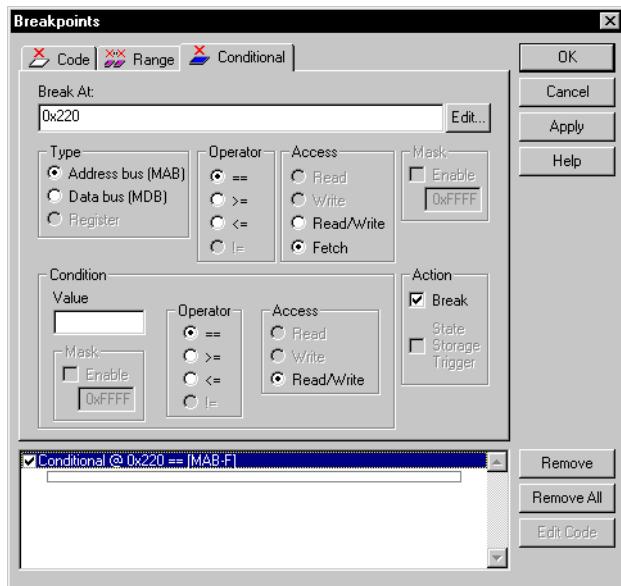


Figure 76: Conditional breakpoints dialog box

### SETTING A CONDITIONAL BREAKPOINT IN THE BREAKPOINTS DIALOG BOX

Defining a conditional breakpoint consists of specifying the options:

- Break At
- Type
- Operator
- Access
- Mask

- Condition
- Action.

The breakpoints you define appear in the list at the bottom of the dialog box.

### **Break At location**

Set the break location using the **Edit** button. These are the locations you can choose between and their possible settings:

Location	Description/Examples
Expression	Any expression that evaluates to a valid address, such as a function or variable name. Setting the location on a function is used for code breakpoints, and setting the location on a variable name is used for data breakpoints. 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> . <code>Zone</code> specifies in which memory the address belongs. For example <code>Memory:0x42</code> If you enter a combination of a <code>Zone</code> and address that is not valid, C-SPY will indicate the mismatch.
Source Location	A location in the C source program using the syntax <code>{file path}.row.column</code> <code>File</code> specifies the filename and full path. <code>Row</code> specifies the row in which you want the breakpoint. <code>Column</code> specifies the column in which you want the breakpoint. Note that the Source Location type is only meaningful for code breakpoints. For example, <code>{C:\IAR Systems\yyy\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 35: Conditional break at location types

## Type

To choose which breakpoint type to use, select one of the following options:

Breakpoint type	Description
<b>Address bus (MAB)</b>	Sets a breakpoint on a specified address, or anything that can be evaluated to such. The breakpoint is triggered when the specified location is accessed. If you have set the breakpoint on a specific machine instruction, the breakpoint will be triggered and the execution will stop, exactly before the instruction will be executed.
<b>Data bus (MDB)</b>	Sets a breakpoint on a specified value. It is the value on the data bus that triggers the breakpoint.
<b>Register</b>	Sets a breakpoint on a register. In the <b>Register Value</b> text box, type the value that should trigger the breakpoint. Specify the register, or anything that can be evaluated to such, in the <b>Break At</b> text box.

Table 36: Conditional breakpoint types

## Operator

You can specify one of the following condition operators for when the breakpoint should be triggered:

Condition	Description
<code>==</code>	Equal to.
<code>&gt;=</code>	Greater than or equal to.
<code>&lt;=</code>	Less than or equal to.
<code>!=</code>	Not equal to.

Table 37: Conditional breakpoint condition operators

## Access

You can specify the type of access that triggers the selected breakpoint. Select one of the following options:

Access	Description
Read	Read from location.
Write	Write to location.
Read/Write	Read from or write to location.
Fetch	At instruction fetch.

Table 38: Conditional breakpoint access types

## Mask

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

## Condition

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

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

Table 39: Conditional breakpoint condition types

## Action

There are two action options—**Break** and **State Storage Trigger**.

If you select the option **Break**, the execution will stop when the breakpoint is triggered.

If you select the option **State Storage Trigger**, the breakpoint is defined as a state storage trigger. To define the behavior of the state storage module further, use the options in the State Storage Control window.

## USING A MACRO TO SET A CONDITIONAL BREAKPOINT

You can also use the system macro to set a conditional breakpoint.

```
--setConditionalBreak(location, type, operator,
                     access, action, mask, cond_value,
                     cond_operator, cond_access, cond_mask)
```

## Parameters

All parameters are strings.

<i>location</i>	The breakpoint location. This can be either: A <i>source location</i> on the form " <i>{filename}.line.col</i> " (for example " <i>D:\\src\\prog.c</i> .12.9")
	An <i>absolute location</i> on the form " <i>zone:hexaddress</i> " or simply " <i>hexaddress</i> " (for example "Memory:0x42")
	An <i>expression</i> whose value designates a location (for example "my_global_variable").
	A register (for example "R10")
<i>type</i>	The breakpoint type; either "Address", "Data", or "Register".
<i>operator</i>	The breakpoint operator, either "==" , ">=" , "<=" , or "!=".
<i>access</i>	The memory access type: "Read", "Write", "ReadWrite", or "Fetch".
<i>action</i>	The action type: "Break", "Trigger", or "BreakTrigger".
<i>mask</i>	A 16-bit value that the breakpoint address or value will be masked with.
<i>cond_value</i>	An extra conditional data value.
<i>cond_operator</i>	The condition operator, either "==" , ">=" , "<=" , or "!=".
<i>cond_access</i>	The access type of the condition: "Read" or "Write".
<i>cond_mask</i>	The mask value of the condition.

## Return value

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

Table 40: `__setConditionalBreak` return values

This macro is only applicable to the FET Debugger version of C-SPY.

### Example

```
--var brk;
brk = __setConditionalBreak("R10", "Register", ">=", "Write",
                           "Trigger", "0x0000", "0x4000",
                           "<=", "Write", "0x00FF");
...
__clearBreak(brk);
```

For additional information, see *Defining breakpoints*, page 124.

## Emulator menu

Using the C-SPY FET driver creates a new menu on the menu bar—the **Emulator** menu.

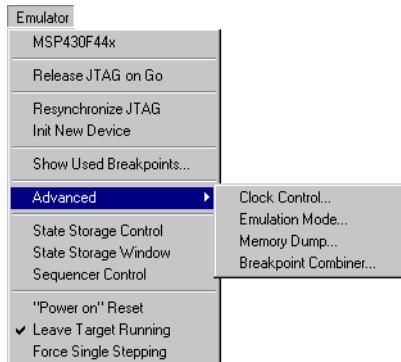


Figure 77: Emulator menu

Menu Command	Description
Connected device	The name of the device used for debugging.
Release JTAG on Go	Sets the JTAG drivers in tri-state so that the device is released from JTAG control—TEST pin is set to GND—when <b>GO</b> is activated.
Resynchronize JTAG	Regains control of the device. It is not possible to Resynchronize JTAG while the device is operating.

Table 41: Emulator menu commands

Menu Command	Description
Init New Device	Initializes the device according to the specified options on the <b>Flash Emulation Tool</b> page. The current program file is downloaded to the device memory, and the device is then reset. This command can be used to program multiple devices with the same program from within the same C-SPY session.
	It is not possible to choose <b>Init New Device</b> while the device is operating, thus the command will be dimmed.
Show Used Breakpoints	Lists all used hardware and virtual breakpoints, as well as all currently defined conditional breakpoints.
Advanced>Clock Control	Depending on the hardware support, clock control comes in one of two variants, General Clock Control or Extended Clock Control. Extended Clock Control gives you module level control over the clocks.
Advanced>Emulation Mode	Specifies the device to be emulated. The device must be reset (or reinitialized by using the menu command <b>Init New Device</b> ) following a change to the emulation mode.
Advanced>Memory Dump	Writes the specified device memory contents to a specified file. A dialog box is displayed where you can specify a filename, a memory starting address, and a length. The addressed memory is then written in a text format to the named file. Options permit you to select word or byte text format, and address information and register contents can also be appended to the file.  The Dump Memory length specifier is restricted to four hexadecimal digits (0–FFFF). This limits the number of bytes that can be written from 0 to 65535. Consequently, it is not possible to write memory from 0 to 0xFFFF inclusive as this would require a length specifier of 65536 (or 0x10000).
Advanced>Breakpoint Combiner	Combines two already defined breakpoints. Select a breakpoint in the <b>Breakpoint combiner</b> dialog box, then right-click to display a list to select the breakpoint to combine it with.  Only available if you are using a device that supports the Enhanced Emulation Module. The settings are not saved when the debug session is closed.
State Storage Control	Opens the State Storage Control window, which lets you define the use of the state storage module. This is only possible if you are using a device that contains support for the Enhanced Emulation Module.
State Storage Window	Opens the State Storage window which contains state storage information according to your definitions.

Table 41: Emulator menu commands (Continued)

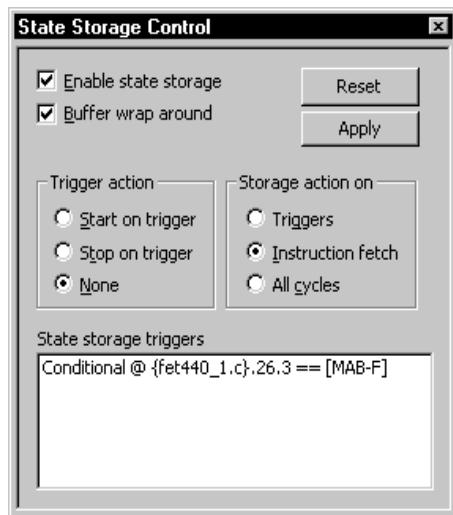
Menu Command	Description
Sequencer Control	Opens the Sequencer Control window, which lets you define a state machine.
“Power on” Reset	The device is reset by cycling power to the device.
Leave Target Running	Leaves the application running on the target hardware after the debug session is closed.
Force Single Stepping	

*Table 41: Emulator menu commands (Continued)*

**Note:** Not all **Emulator>Advanced** submenus are available on all MSP430 devices.

## STATE STORAGE CONTROL WINDOW

Use the State Storage Control window—available from the **Emulator** menu—to define how to use the state storage module available on devices that support the Enhanced Emulation Module.

*Figure 78: State Storage Control window*

### Enable state storage

This option enables the state storage module.

## Buffer wrap around

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

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

## Reset

Resets the state storage module.

## Trigger action

This option acts upon the breakpoints that are defined as state storage triggers. The option defines what action should take place when these breakpoints have been triggered. You can choose between the following options:

Start on trigger	State storage starts when the breakpoint is triggered.
Stop on trigger	State storage starts immediately when execution starts. State storage stops when the breakpoint is triggered.
None	State storage starts immediately when execution starts. State storage does not stop when the breakpoint is triggered. However, if execution stops, state storage also stops but it will resume when execution resumes.

## Storage action on

This option defines when the state information should be collected. You can choose between the following options:

Triggers	Stores state information at the time when the state storage trigger is triggered. That is, when the breakpoint defined as a state storage trigger is triggered.
Instruction fetch	Stores state information at all instruction fetches.
All cycles	Stores state information for all cycles.

## State storage triggers

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

## STATE STORAGE WINDOW

The State Storage window—available from the **Emulator** menu—displays state storage information for eight states. Invalid data is displayed in red color.

The screenshot shows a Windows-style dialog box titled "State Storage Window". At the top, there are three checkboxes: "Update" (checked), "Automatic update" (checked), "Automatic restart" (unchecked), and "Append data" (unchecked). The main area is a table with columns: Address bus..., Instr., Mnemonic, Data bus ..., Control Signals..., and Control Signals... (repeated). The data rows are as follows:

Address bus...	Instr.	Mnemonic	Data bus ...	Control Signals...	Control Signals...
0x1100	31400A	mov.w #0xA00,SP	0x4031	0x03	Break Trig. = 0; ...
0x1104	B0121211	call #main	0x12B0	0x03	Break Trig. = 0; ...
0x0000	----	????	0x0000	0x00	Break Trig. = 0; ...
0x0000	----	????	0x0000	0x00	Break Trig. = 0; ...
0x0000	----	????	0x0000	0x00	Break Trig. = 0; ...
0x0000	----	????	0x0000	0x00	Break Trig. = 0; ...
0x0000	----	????	0x0000	0x00	Break Trig. = 0; ...
0x0000	----	????	0x0000	0x00	Break Trig. = 0; ...

Figure 79: State Storage window

### Update

Click the update button to refresh the data in the State Storage window, alternatively to append new data.

### Automatic update

Select this option to automatically update the data in the state storage window each time new data is available in the state storage buffer.

### Automatic restart

Select this option to reset the state storage module for consecutive data readouts after each readout.

### Append data

Select this option to append collected data from the state storage buffer to the data that is already present in the State Storage window. The new data is added below the data that is already present.

The window contains the following columns:

Column	Description
Address bus	Displays the stored value of the address bus.
Instruction	Displays the instruction.
Mnemonic	Displays the mnemonic.

Table 42: Columns in State Storage window

Column	Description
Data bus	Displays the stored content of the data bus.
Control signals (byte)	Displays the stored value of the control signals during storage. Bit 1: Instruction fetch Bit 2: Byte/Word Bit 3: Interrupt request Bit 4: CPU off Bit 5: The value of the Power Up Clear (PUC) signal Bit 6:ZERO HALT (which one depends on the used device) Bit 7: Break trigger
Control signals (bits)	Displays each bit in the stored value of the control signals during storage.

Table 42: Columns in State Storage window (Continued)

## SEQUENCER CONTROL WINDOW

The Sequencer Control window—available from the **Emulator** menu—lets you break the execution or trigger the state storage module by using a more complex method than a standard breakpoint. This is useful if you, for instance, want to stop the execution or start the state storage module under certain conditions, for instance, a specific program flow. The sequencer is only available if you are using a device that supports the Enhanced Emulation Module.

The sequencer works as a state machine. In a simple setup, you can define three transition triggers, where the last one triggers an action.

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

For each state you can define up to two different transitions (a-b) to other states. For each transition you define a transition trigger and which the next state should be. For state 3 you must also define an action: stop the execution or start the state storage module.

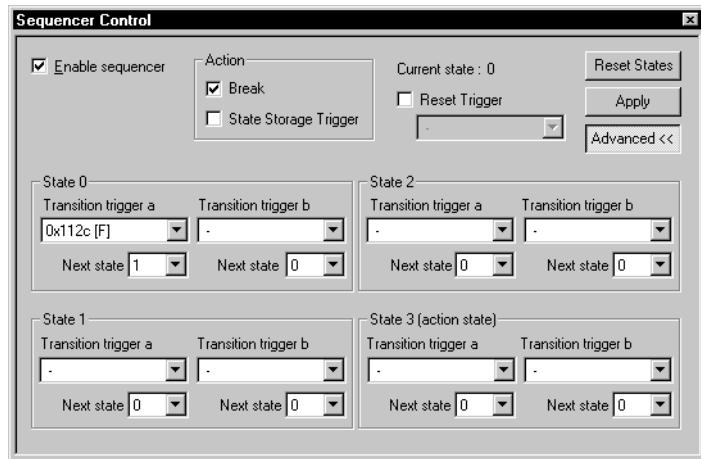


Figure 80: Sequencer Control window

To enable the sequencer, select the option **Enable Sequencer**. From the eight available hardware breakpoints (0-7) of the device, the breakpoint number 7 will be reserved for state 3.

The **Transition trigger** drop-down lists let you define one breakpoint each, where the breakpoint should act as a transition trigger.

To define an advanced setup, click the **Advanced** button. This will let you define 4 states (0-3) with two transition triggers each (a and b). For each transition trigger, you can define which state should be the next state after the transition.

Use the following options:

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

**Next state** Defines which state should be the next state after the transition. Select one state, out of four, from the drop-down list.

Finally, you must define an action. This option defines the result of the final transition trigger. If you select the option **Break**, the execution will stop. If you select the option **State Storage Trigger**, the state storage module will be triggered.

The **Reset States** button will set the state machine to state 0. **Current state** shows the current state of the state machine.

# Design considerations for in-circuit programming

This chapter describes the design considerations related to the bootstrap loader, device signals, and external power for in-circuit programming. This chapter also describes how you can adapt your own target hardware to be run with C-SPY.

---

## Bootstrap loader

The JTAG pins provide access to the flash memory of the MSP430Fxxx devices. On some devices, these pins must be shared with the device port pins, and this sharing of pins can complicate a design (or it might simply not be possible to do so). As an alternative to using the JTAG pins, MSP430Fxxx devices contain a program—a *bootstrap loader*—that permits the flash memory to be easily erased and programmed, using a reduced set of signals.

---

## Device signals

The following device signals should be made accessible so that the FET and PRGS (serial programming adapter) tools can be utilized:

RST/NMI, TMS, TCK, TDI, TDO, GND, VCC, and TEST (if present).

**Note:** Connections to XIN and XOUT are not required, and should not be made. PRGS software Version 1.10 or later must be used.

The BSL tool requires the following device signals: RST/NMI, TCK, GND, VCC, P1.1, P2.2, and TEST (if present).

## External power

The PC parallel port is capable of supplying a limited amount of current. Because of the ultra low power requirement of the MSP430, a stand-alone FET Debugger can run on the available current. However, if additional circuitry is added to the tool, this current might not be enough. In this case, external power can be supplied to the tool via the connections provided on the MSP-FET430X110 and the Target Socket modules. Refer to Figure 81, *JTAG signal connection (MSP-FET430X110)* and Figure 82, *JTAG signal connection (MSP-FET430Pxx0)*, respectively, to locate the external power connections.

When an MSP-FET430X110 device is powered from an external supply, an on-board device regulates the external voltage to the level required by the MSP430.

When a Target Socket module is powered from an external supply, the external supply powers the device on the Target Socket module and any user circuitry connected to the Target Socket module, and the FET Interface module continues to be powered from the PC via the parallel port. If the externally supplied voltage differs from that of the FET Interface module, the Target Socket module must be modified so that the externally supplied voltage is routed to the FET Interface module (so that it can adjust its output voltage levels accordingly). For details of the Target Socket module schematic, see the documentation supplied by the chip manufacturer.

## Signal connections for in-system programming

With the proper connections, you can use the C-SPY Debugger and the MSP-FET430X110, as well as the MSP-FET430Pxx0 ('P120, 'P140, 'P410, 'P440), to program and debug code on your own target board. In addition, the connections will support the MSP430 Serial Programming Adapter (PRGS), thus providing an easy way to program prototype boards, if desired.

**Note:** The IAR XLINK Linker can be configured to output objects in `msp430-txt` format for use with the PRGS tool. Choose **Project>Options** and click the **Output** tab in the **Linker** category. Select the option **Other** and then choose **msp430-txt** from the **Output format** drop-down list. The Intel and Motorola formats can also be used.

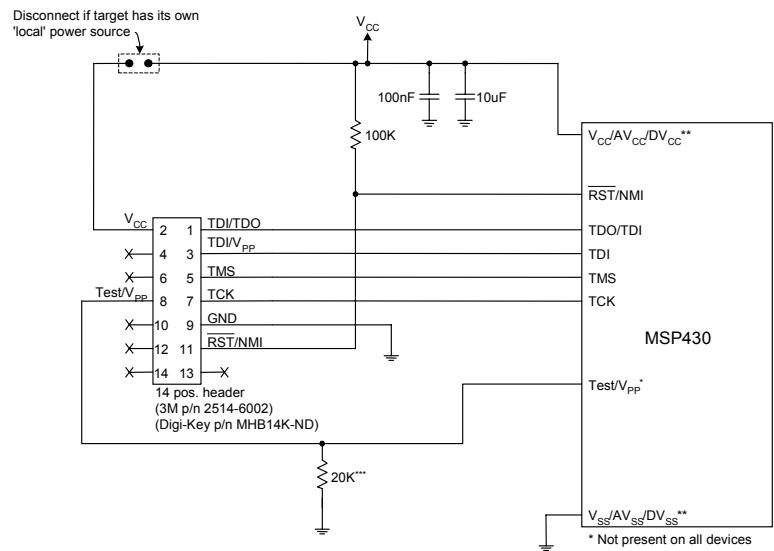
### MSP-FET430X110

Figure 81, *JTAG signal connection (MSP-FET430X110)*, shows the connections between the FET device and the target device required to support in-system programming and debugging using C-SPY. If your target board has its own “local” power supply, such as a battery, do not connect Vcc to pin 2 of the JTAG header. Otherwise, contention might occur between the FET device and your local power supply.

The figure shows a 14-pin header (available from Digi-Key, p/n MHB14K-ND), being used for the connections on your target board. It is recommended that you build a wiring harness from the FET device with a connector which mates to the 14-pin header, and mount the 14-pin header on your target board. This will allow you to unplug your target board from the FET device as well as use the Serial Programming Adapter to program prototype boards, if desired.

The signals required are routed on the FET device to header locations for easy accessibility. Refer to the hardware documentation for more details.

After you make the connections from the FET device to your target board, remove the MSP430 device from the socket on the FET device so that it does not conflict with the MSP430 device in your target board. Now simply use C-SPY as you would normally to program and debug.



\*\*\* Pulldown not required on all devices.  
Check device datasheet pin description.

Figure 81: JTAG signal connection (MSP-FET430X110)

**Note:** No JTAG connection is required to the XOUT pin of the MSP430 device as shown on some schematics.

### MSP-FET430PXX0 ('P120, 'P140, 'P410, 'P440)

Figure 82, *JTAG signal connection (MSP-FET430Pxx0)* shows the connections between the FET Interface module and the target device required to support in-system programming and debugging using C-SPY. The figure shows a 14-pin header (available from Digi-Key, p/n MHB14K-ND) connected to the MSP430. With this header mounted on your target board, the FET Interface module can be plugged directly into your target. Then simply use C-SPY as you would normally to program and debug.

The connections for the FET Interface module and the Serial Programming Adapter (PRGS) are identical with the exception of VCC. Both the FET Interface module and PRGS can supply VCC to your target board (via pin 2). In addition, the FET Interface module has a VCC-sense feature that, if used, requires an alternate connection (pin 4 instead of pin 2). The FET Interface module VCC-sense feature senses the local VCC (present on the target board, i.e. a battery or other “local” power supply) and adjusts its output signals accordingly. The PRGS does not support this feature, but does provide the user the ability to adjust its JTAG signal levels to the VCC level on your target board through the GUI.

If the target board is to be powered by a local VCC, then the connection to pin 4 on the JTAG should be made, and not the connection to pin 2. This utilizes the VCC-sense feature of the FET Interface module and prevents any contention that might occur if the local on-board VCC were connected to the VCC supplied from the FET Interface module or the PRGS. If the VCC-sense feature is not necessary (that is, the target board is to be powered from the FET Interface module or the PRGS) the VCC connection is made to pin 2 on the JTAG header and no connection is made to pin 4.

The figure shows a jumper block in use. The jumper block supports both scenarios of supplying VCC to the target board. If this flexibility is not required, the desired VCC connections can be hard-wired eliminating the jumper block.

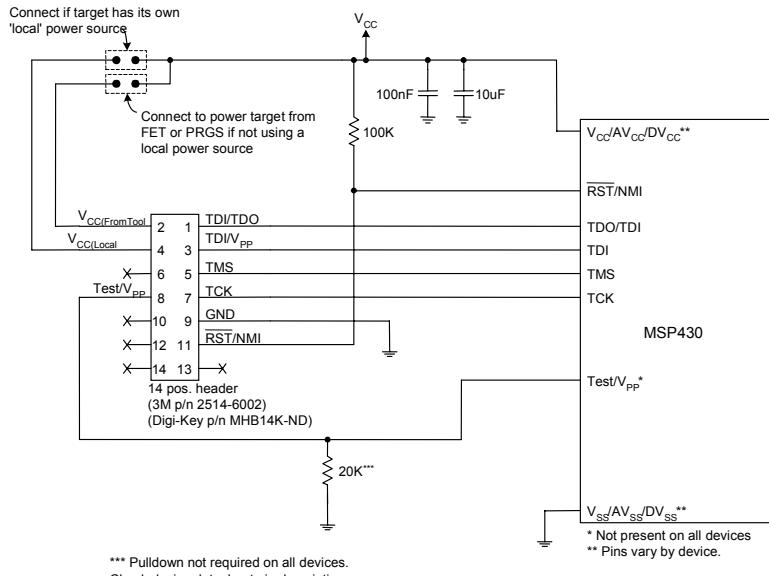


Figure 82: JTAG signal connection (MSP-FET430Pxx0)

**Note:** No JTAG connection is required to the XOUT pin of the MSP430 as shown on some schematics.



# Part 7. Reference information

This part of the MSP430 IAR Embedded Workbench™ IDE User Guide contains the following chapters:

- IAR Embedded Workbench IDE reference
- C-SPY Debugger IDE reference
- General options
- Compiler options
- Assembler options
- Custom build options
- XLINK options
- Library builder options
- Debugger options
- 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 IAR Embedded Workbench IDE. Information about how to best use the Embedded Workbench for your purposes can be found in parts 3 to 6 in this book.

The IAR Embedded Workbench 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
- Build window (message window)
- Find in Files window (message window)
- Tool Output window (message window)
- Debug Log window (message window).

In addition, a set of C-SPY-specific windows becomes available when you start the IAR C-SPY Debugger. Reference information about these windows can be found in the *C-SPY Debugger IDE reference* chapter in this book.

## IAR EMBEDDED WORKBENCH IDE WINDOW

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

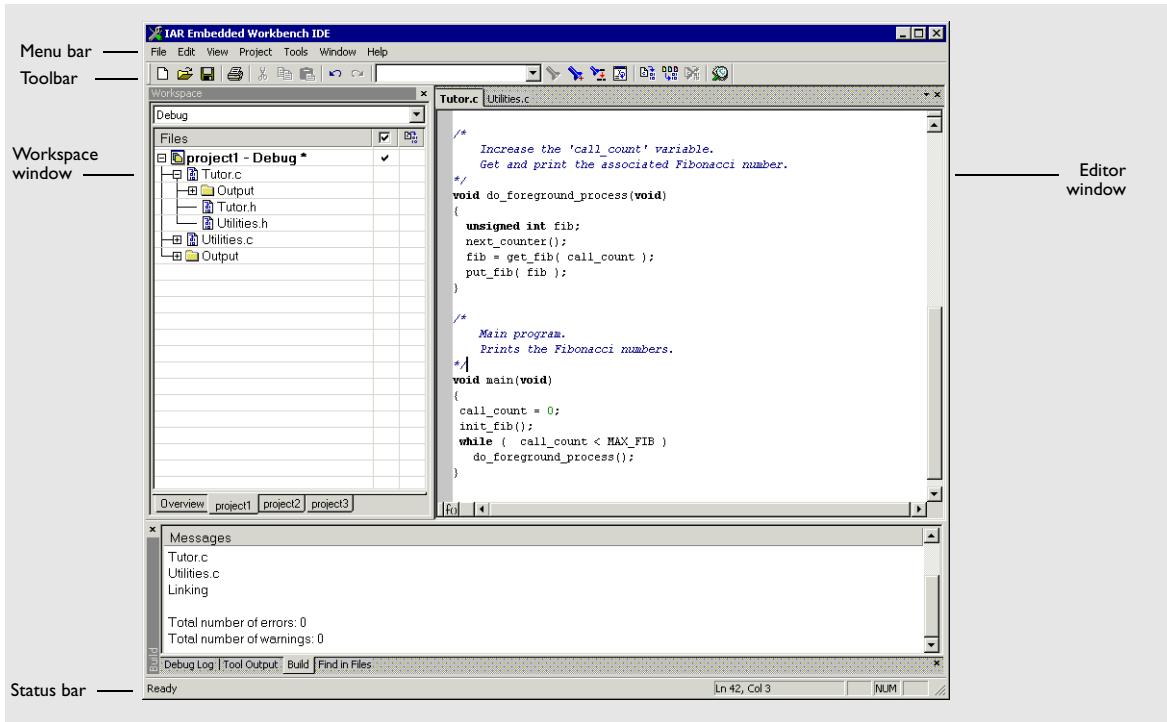


Figure 83: IAR Embedded Workbench IDE window

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

### Menu bar

Gives access to the IAR Embedded Workbench IDE menus.

Menu	Description
File	The File menu provides commands for opening source and project files, saving and printing, and exiting from the IAR Embedded Workbench IDE.

Table 43: IAR Embedded Workbench IDE menu bar

Menu	Description
Edit	The Edit menu provides commands for editing and searching in Editor windows and for editing breakpoints in C-SPY.
View	With the commands on the View menu you can open windows and decide which toolbars should be displayed.
Project	The Project menu provides commands for adding files to a project, creating groups, and running the IAR tools on the current project.
Tools	The Tools menu is a user-configurable menu to which you can add tools for use with the IAR Embedded Workbench IDE.
Window	With the commands on the Window menu you can manipulate the IAR Embedded Workbench IDE windows and change their arrangement on the screen.
Help	The commands on the Help menu provide help about the IAR Embedded Workbench IDE.

Table 43: IAR Embedded Workbench IDE menu bar (Continued)

For reference information for each menu, see *Menus*, page 230.

## Toolbar

The IAR Embedded Workbench IDE toolbar—available from the **View** menu—provides buttons for the most useful commands on the IAR Embedded Workbench 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:

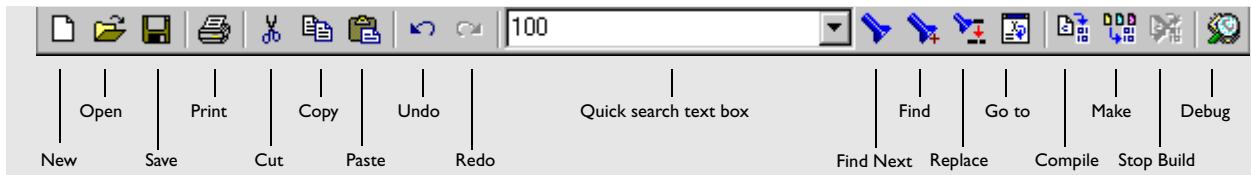


Figure 84: IAR Embedded Workbench IDE toolbar

## Status bar

The Status bar at the bottom of the window—available from the **View** menu—displays the status of the IAR Embedded Workbench IDE, and the state of the modifier keys.

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.

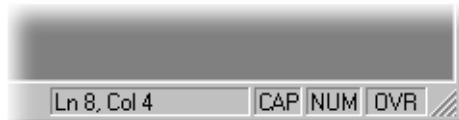


Figure 85: IAR Embedded Workbench IDE window status bar

## WORKSPACE WINDOW

The workspace window, available from the **View** menu, shows the name of the current workspace and a tree representation of the projects, groups and files included in the workspace.

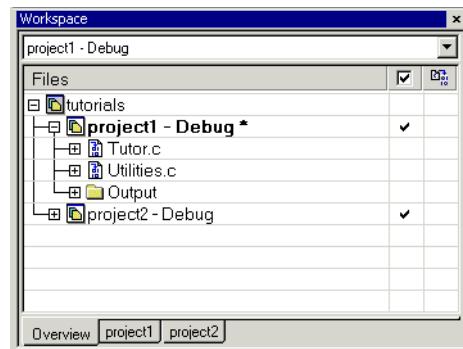


Figure 86: Project window

In the drop-down list at the top of the window you can choose a build configuration to display in the window for a specific project.

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* of this guide.

Clicking the right mouse button in the workspace window displays a context menu which gives you convenient access to several commands.

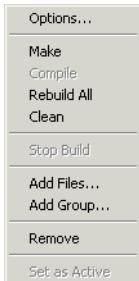


Figure 87: Workspace window context menu

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 Files	Opens a dialog box where you can add files to the project.
Add Group	Opens a dialog box where you can add new groups to the project.
Remove	Removes the selected items.
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 <b>Make</b> command is executed.

Table 44: Workspace window context menu commands

## EDITOR WINDOW

Source 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.

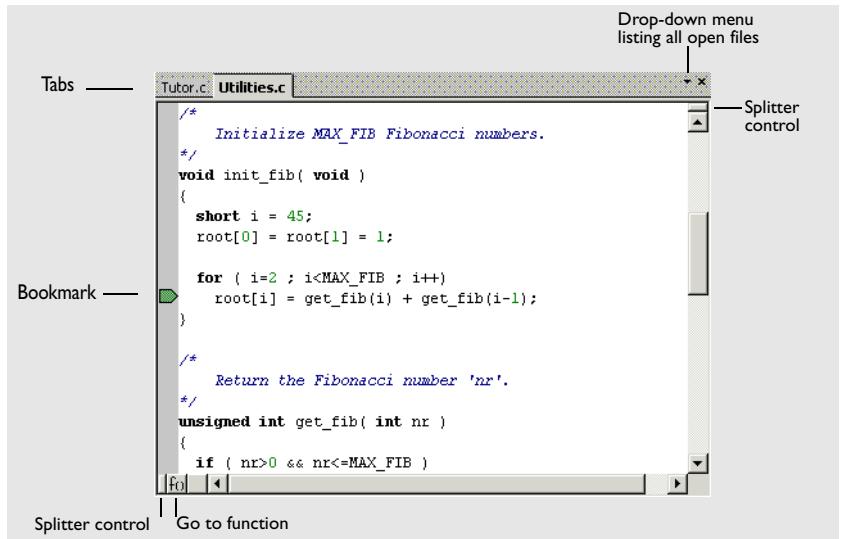


Figure 88: Editor window

The name of the open source 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 95.

### 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 context menu

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

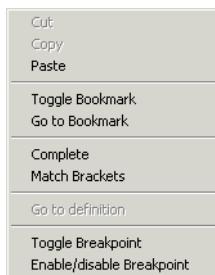


Figure 89: Editor window context menu

## Commands

Cut, copy, paste	Standard window commands.
Toggle Bookmark	Toggles a bookmark at the line where the insertion point is located in the active editor window.
Go to bookmark	Moves the insertion point to the next bookmark that has been defined with the Toggle Bookmark command.
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 hierachic pair of brackets, or beeps if there is no higher bracket hierarchy.
Go to definition of	Shows the declaration of the symbol where the insertion point is placed.
Toggle Breakpoint	Toggles a breakpoint at the statement or instruction containing or close to the cursor in the source window.
Enable/disable Breakpoint	Toggles a breakpoint between being disabled, but not actually removed—making it available for future use—and being enabled again.

## Source file paths

The IAR Embedded Workbench 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 IAR Embedded Workbench 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

Table 45: Editor keyboard commands for insertion point navigation

To move the insertion point	Press
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 45: Editor keyboard commands for insertion point navigation (Continued)*

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 46: 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 47: Editor keyboard commands for selecting text*

## SOURCE BROWSER WINDOW

The Source Browser window—available from the **View** menu—displays an hierarchical view in alphabetical order of all symbols defined in the active build configuration. The information can be sorted by any column.

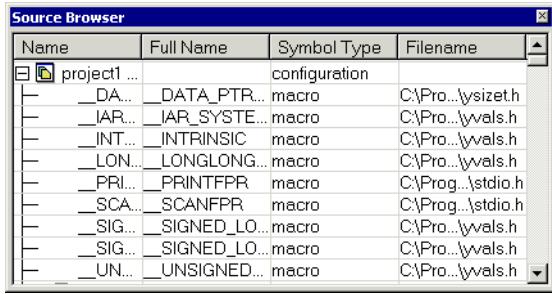


Figure 90: Source Browser window

The window consists of four columns:

Column	Description
Name	Displays the names of global symbols and functions defined in the project.
Full Name	Displays the unique name of each element, for instance <code>classname::membername</code> .
Symbol Type	Displays the symbol type for each element.
Filename	Specifies the path to the file in which the element is defined.

Table 48: Columns in Source Browser window

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

## 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 217.

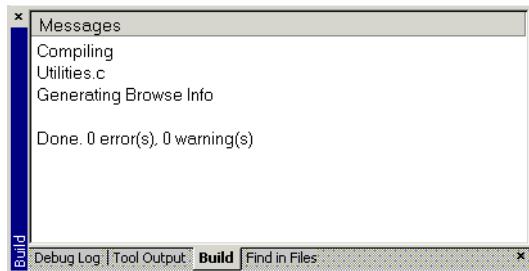


Figure 91: Build window (message window)

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.

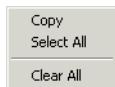


Figure 92: Build window context menu

## FIND IN FILES WINDOW

The Find in Files window—available by choosing **View>Messages**—displays the output from the **Edit>Find in Files** command. When opened, this window is by default grouped together with the other message windows, see *Windows*, page 217.

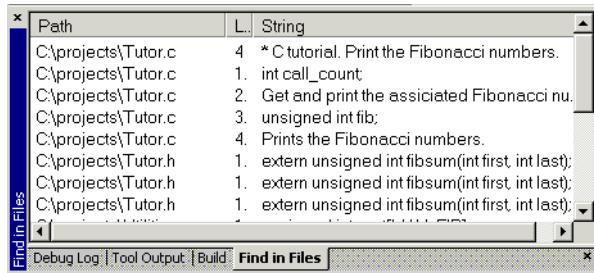


Figure 93: Find in Files window (message window)

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.

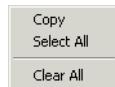


Figure 94: Find in Files window context menu

## 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 263. When opened, this window is by default grouped together with the other message windows, see *Windows*, page 217.

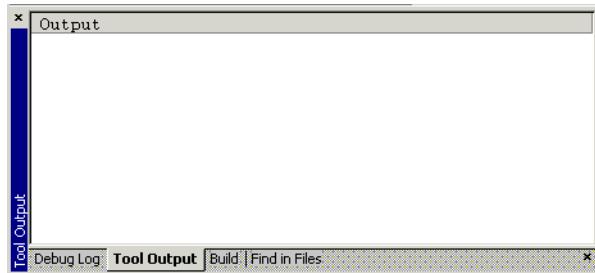


Figure 95: Tool Output window (message window)

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

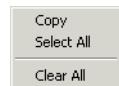


Figure 96: Find in Files window context menu

## 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 the C-SPY Debugger is running. When opened, this window is by default grouped together with the other message windows, see *Windows*, page 217.

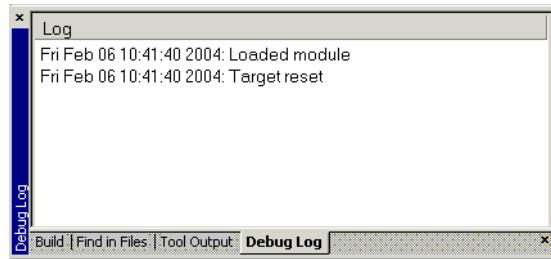


Figure 97: Debug Log window (message window)

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

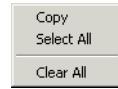


Figure 98: Debug Log window context menu

## Menus

The following menus are available in the IAR Embedded Workbench:

- 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 IAR C-SPY Debugger. Reference information about these menus can be found in the chapter *C-SPY Debugger IDE reference*, page 271.

## FILE MENU

The **File** menu provides commands for opening workspaces and source files, saving and printing, and exiting from the IAR Embedded Workbench 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.

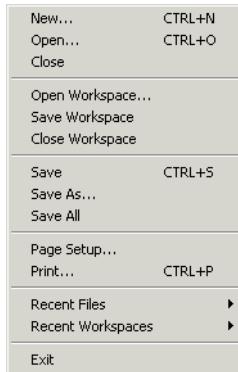


Figure 99: File menu

Menu command	Shortcut	Description
New	CTRL+N	Displays a dialog box where you can create a new workspace, or a new text file.
Open	CTRL+O	Displays a dialog box where you can select a text or workspace file to open. Before a new workspace is opened you will be prompted for saving and closing any currently open workspaces.
Close		Closes the active window. You will be given the opportunity to save any files that have been modified before closing.
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.

Table 49: File menu commands

Menu command	Shortcut	Description
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 IAR Embedded Workbench IDE. You will be asked whether to save any changes to text windows before closing them. Changes to the project are saved automatically.

Table 49: 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...	Ctrl+F
Find Next	F3
Replace...	Ctrl+H
Find in Files...	
Go To...	Ctrl+G
Next Error/Tag	F4
Previous Error/Tag	Shift+F4
Toggle Bookmark	Ctrl+F2
Go to Bookmark	F2
Complete	Ctrl+Space
Match Brackets	Ctrl+B
Toggle Breakpoint	F9
Enable/Disable Breakpoint	Ctrl+F9
Breakpoints...	

Figure 100: 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 <b>Undo</b> in the current editor window. You can undo and redo an unlimited number of edits independently in each editor window.
Cut	CTRL+X	Provides the standard Windows functions for text editing in editor windows and text boxes.
Copy	CTRL+C	
Paste	CTRL+V	
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.
Find	CTRL+F	Displays a dialog box where you can search for text within the current editor window.
Find Next	F3	Finds the next occurrence of the specified string.
Replace	CTRL+H	Displays a dialog box where you can search for a specified string and replace each occurrence with another string.
Find in Files		Displays a dialog box where you can search for a specified string in multiple text files.
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.
Next Error/Tag	F4	If there is a list of error messages or the results from a <b>Find in Files</b> 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 <b>Find in Files</b> search in the Messages window, this command will display the previous item from that list in the editor window.
Toggle Bookmark	CTRL+F2	Toggles a bookmark at the line where the insertion point is located in the active editor window.
Go to Bookmark	F2	Moves the insertion point to the next bookmark that has been defined with the Toggle Bookmark command.
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.
Match Brackets		Selects all text between the brackets immediately surrounding the insertion point, increases the selection to the next hierarchical pair of brackets, or beeps if there is no higher bracket hierarchy.

Table 50: Edit menu commands

Menu command	Shortcut	Description
Toggle Breakpoint F9		Toggles on or off a breakpoint at the statement or instruction containing or 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.
Breakpoints		Opens a dialog box, where you can create, edit, and remove breakpoints.

Table 50: Edit menu commands (Continued)

### Find dialog box

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

Option	Description
Find What	Selects the text to search for.
Match Whole Word Only	To find the specified text only if it occurs as a separate word. Otherwise specifying int will also find print, sprintf etc.
Match Case	To find only occurrences that exactly match the case of the specified text. Otherwise specifying int will also find INT and Int.
Incremental Search	To gradually fine-tune or expand the search by changing the search string continuously.
Up or Down	To specify the direction of the search.
Find Next	To find the next occurrence of the selected text.

Table 51: 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	To find the specified text only if it occurs as a separate word. Otherwise searching for int will also find print, sprintf etc.
Match Case	To find only occurrences that exactly match the case of the specified text. Otherwise searching for int will also find INT and Int.

Table 52: Replace dialog box options

Option	Description
Find Next	To find the next occurrence of the text you have specified.
Replace	To replace the searched text with the specified text.
Replace All	To replace all occurrences of the searched text in the current editor window.

Table 52: Replace dialog box options (Continued)

### Find in Files dialog box

The **Find in Files** dialog box is available from the **Edit** menu.

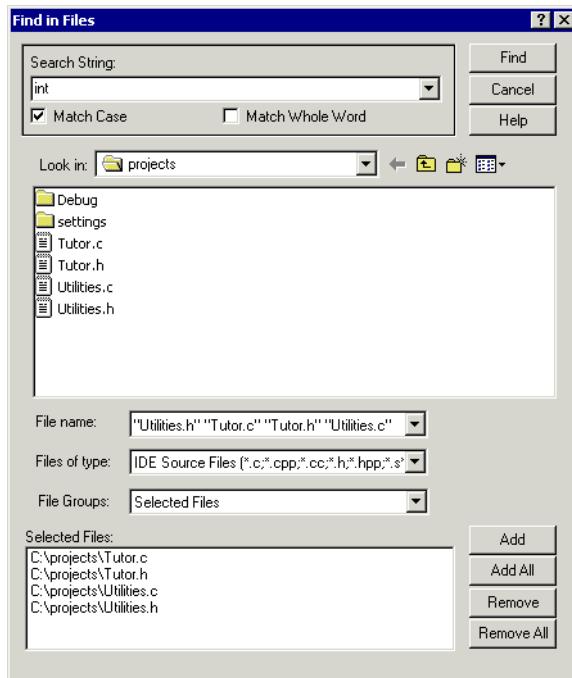


Figure 101: Find in Files dialog box

Option	Description
Search String	Selects the string to search for.
Match Case	To find only occurrences that exactly match the case of the specified text. Otherwise searching for <code>int</code> will also find <code>INT</code> and <code>Int</code> .

Table 53: Find in Files dialog box options

Option	Description
Match Whole Word	To find the specified text only if it occurs as a separate word. Otherwise searching for int will also find print, sprintf etc.
Look in	Standard file navigation.
File name	Selects the file to search.
Files of type	File type filter.
File Groups	Filter for which files to search. If the option Selected Files is chosen, you should manually select the files to search by adding them to the Selected Files list.
Selected Files	List of selected files to search.
Add	Adds the selected file to the Selected Files list.
Add All	Adds multiple files to the Selected Files list.
Remove	Removes a selected file from the Selected Files list.
Remove All	Removes multiple files from the Selected Files list.
Find	Proceed with the search.

*Table 53: Find in Files dialog box options (Continued)*

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 double-clicking the messages. 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.

## Breakpoints dialog box

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

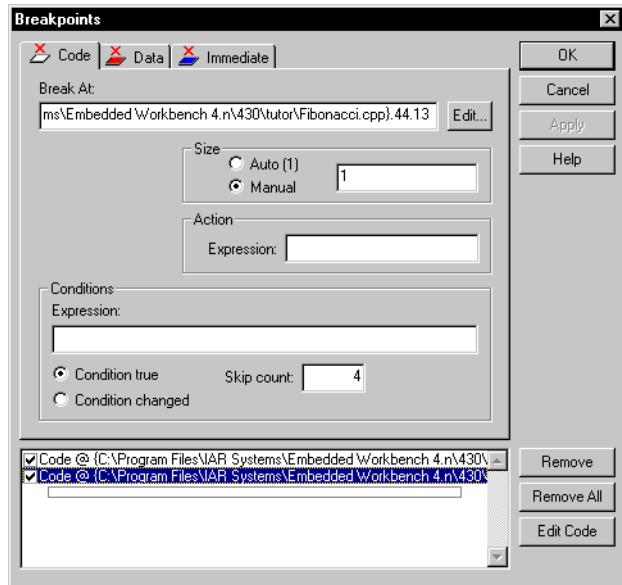


Figure 102: Breakpoints dialog box

**Note:** Depending on what C-SPY driver you are using, other types of breakpoints and breakpoint settings might be available than described here. For driver-specific breakpoint information, see *Part 5. IAR C-SPY Simulator* and *Part 6. IAR C-SPY FET Debugger*.

To define a breakpoint you should first click a breakpoint tab and then specify a set of options:

- Break At
- Access type (only for data and immediate breakpoints)
- Size
- Action (optional)
- Conditions (only for code and data breakpoints).

The breakpoints you define appear in the list at the bottom of the dialog box. All breakpoints you define using the **Breakpoints** dialog box are preserved between debug sessions.

For more information about the breakpoint system and how to set breakpoints, see the *Using breakpoints* chapter in *Part 4. Debugging*.

### **Code breakpoints**

For code breakpoints, operation code 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, exactly before the instruction is executed.

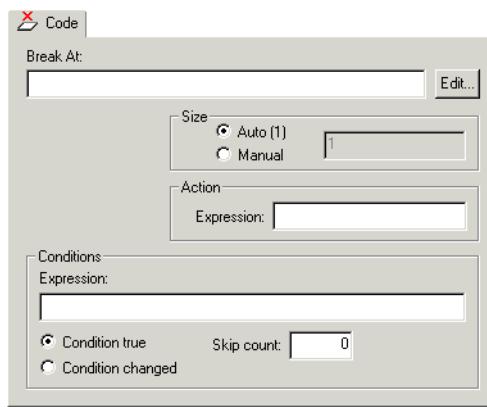


Figure 103: Code breakpoints page

### Data breakpoints

For data breakpoints data is accessed at the specified location. The execution will usually stop directly after the instruction that has accessed the data is executed.

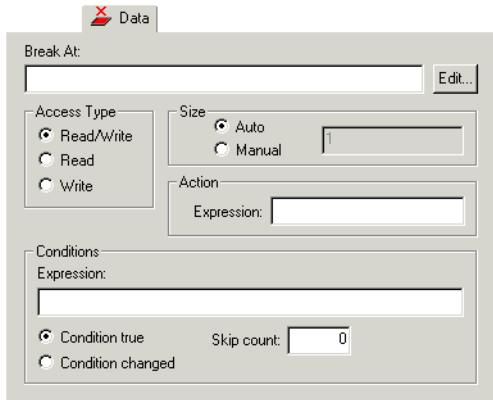


Figure 104: Data breakpoints page

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.

#### Break At Location

Specify the location for the breakpoint in the **Break At** text box. Alternatively, use the **Edit** browse button to open the **Enter Location** dialog box.



Figure 105: Enter Location dialog box

You can choose between these locations and their possible settings:

Location	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> . <code>filepath</code> specifies the filename and full path. <code>row</code> specifies the row in which you want the breakpoint. <code>column</code> specifies the column in which you want the breakpoint. Note that the Source Location type is usually only meaningful for code breakpoints. For example, <code>{C:\IAR Systems\xxx\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 54: Break At Location types

### Access Type

You can specify the type of memory access that triggers data or immediate breakpoints.

Memory Access type	Description
<b>Read/Write</b>	Read or write from location (not available for immediate breakpoints).
<b>Read</b>	Read from location.
<b>Write</b>	Write to location.

Table 55: 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 *Using breakpoints*, page 156.)

### Size

By default, you can specify a size in bytes—in practice, an *address range*—for code and data breakpoints. Each read and write access to the specified memory range will trigger the breakpoint. 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 (in bytes) of the breakpoint in the **Size** text box.

### Action

You can connect an action to a breakpoint. You specify an optional expression, for instance a macro function, which is evaluated when the breakpoint is triggered and the condition is true.

### Conditions

You can specify simple and complex conditions to code and data breakpoints.

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 56: Breakpoint conditions

## VIEW MENU

With the commands on the **View** menu you can choose what to display in the IAR Embedded Workbench window. During a debug session you can also open debugger-specific windows from the **View** menu.

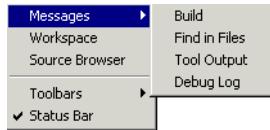


Figure 106: View menu

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.
Toolbars	The options <b>Main</b> and <b>Debug</b> toggle the two toolbars on and off.
Status bar	Toggles the status bar on and off.
Debugger windows	During a debugging session, the different debugging windows are also available from the <b>View</b> menu: Disassembly window Memory window Register window Watch window Locals window Auto window Live Watch window Call Stack window Terminal I/O window Code Coverage window Profiling window Trace window Stack window
	For descriptions of these windows, see <i>C-SPY windows</i> , page 271.

Table 57: View menu commands

## 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.



Figure 107: Project menu

Menu Command	Description
Add Files	Displays a dialog box 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.
Edit Configurations	Displays the <b>Configurations for project</b> 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	Displays the <b>Options for node</b> 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.

Table 58: Project menu commands

Menu Command	Description
	Brings the current build configuration up to date by compiling, assembling, and linking only the files that have changed since the last build.
	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	Rebuilds and relinks all files in the current target.
Clean	Removes any intermediate files.
Batch Build	Displays a dialog box where you can configure named batch build configurations, and build a named batch.
Stop Build	Stops the current build operation.
	Starts the IAR C-SPY Debugger 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. Depending on your IAR product installation, you can choose which debugger drive to use by selecting the appropriate C-SPY driver on the C-SPY Setup page available by using the <b>Project&gt;Options</b> command.

Table 58: 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
\$EW_DIR\$	Top directory of the IAR Embedded Workbench, for example c:\program files\iar systems\embedded workbench 4.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)

Table 59: Argument variables

Variable	Description
\$LIST_DIR\$	Directory for list output
\$OBJ_DIR\$	Directory for object output
\$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 4.n\430

Table 59: 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.

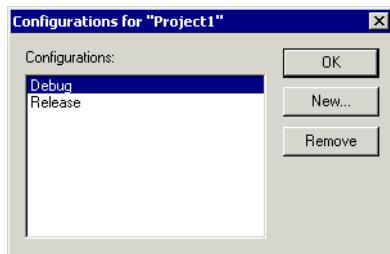


Figure 108: Configurations for project dialog box

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 <b>Configurations</b> list.

Table 60: 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.

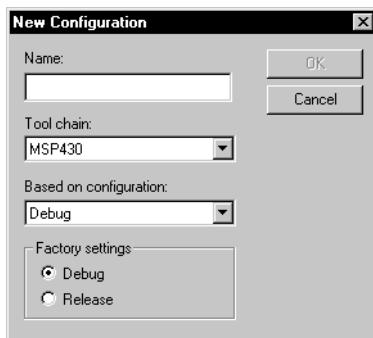


Figure 109: New Configurations dialog box

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. 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 settings that you want to be applied on your new build configuration.

Table 61: 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.

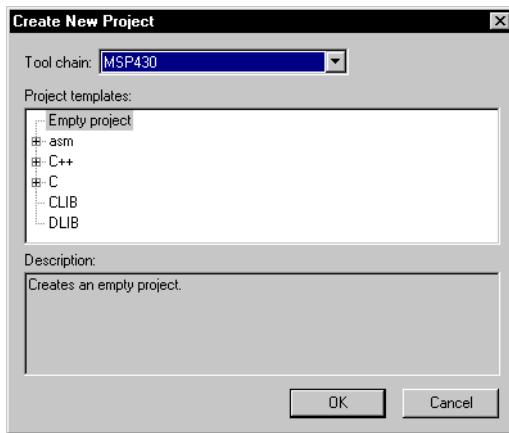


Figure 110: Create New Project dialog box

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 62: 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 for which build tool to set options for. 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	MSP430 IAR C/C++ Compiler options
Assembler	MSP430 IAR Assembler options
Custom Build	Options for extending the tool chain
Linker	IAR XLINK Linker™ options. This category is available for application projects.
Library Builder	IAR XAR Library Builder™ options. This category is available for library projects.
Debugger	IAR C-SPY™ Debugger options
FET Debugger	FET-specific options
Simulator	Simulator-specific options

*Table 63: Project option categories*

Selecting a category displays one or more pages of options for that component of the IAR Embedded Workbench IDE.

For detailed information about each option, see the option reference section. For information about the options related to the different C-SPY debugger systems, see the C-SPY debugger part.

## Batch Build dialog box

The **Batch Build** dialog box—available by choosing **Project>Batch build**—lists all defined batches of build configurations.

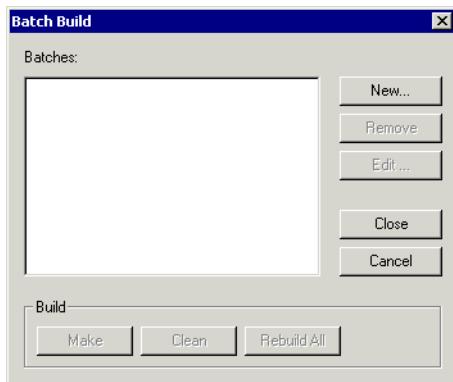


Figure 111: Batch Build dialog box

The dialog box contains the following:

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

Table 64: 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.

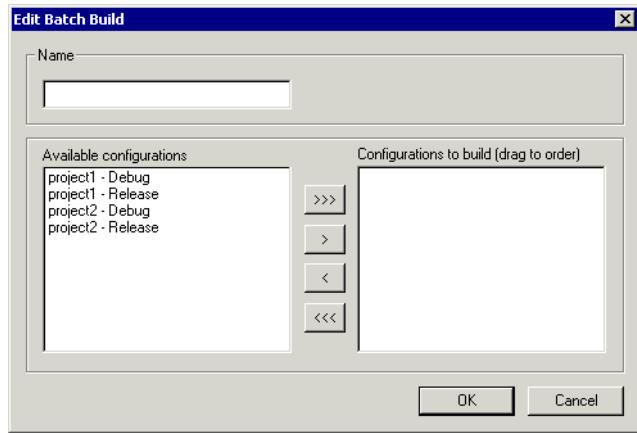


Figure 112: Edit Batch Build dialog box

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 65: Description of the Edit Batch Build dialog box

You can move appropriate build configurations from the list of available build configurations to the list of configurations to build with the arrow buttons.

## 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 the 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 263.

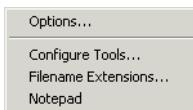


Figure 113: Tools menu

### Tools menu commands

Menu command	Description
Options	Displays a dialog box where you can customize the IAR Embedded Workbench IDE. Select the feature you want to customize by clicking the appropriate tab. Which pages are available in this dialog box depends on your IAR Embedded Workbench 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.
Notepad	User-configured. This is an example of a user-configured addition to the Tools menu.

Table 66: Tools menu commands

## Common fonts page

The **Common Fonts** page—available by choosing **Tools>Options**—displays the fonts used for all project windows except the editor windows.

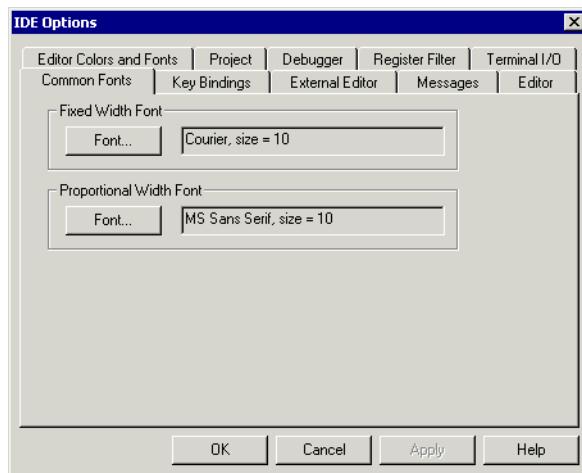


Figure 114: Common Fonts page

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 *Customizing the editor environment*, page 98.

## Key Bindings page

The **Key Bindings** page—available by choosing **Tools>Options**—displays the shortcut keys used for each of the menu options, which you can change, if you wish.

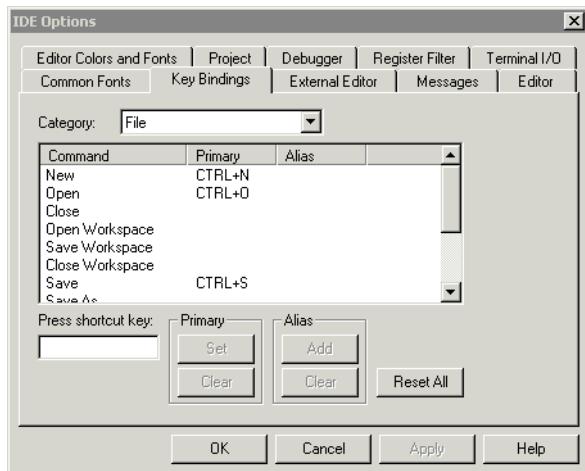


Figure 115: Key Bindings page

### Options

Option	Description
Category	Drop-down menu to choose the menu you want to edit. Any currently defined shortcut keys are shown in the scroll list below.
Press shortcut key	Type the key combination you want to use as shortcut key.
Primary	The shortcut key will be displayed next to the command on the menu. Click Set to set the combination, or Clear to delete the shortcut.
Alias	The shortcut key will work but not be displayed on the menu. Click either Add to make the key take effect, or Clear to delete the shortcut.
Reset All	Reverts all command shortcut keys to the factory settings.

Table 67: Key Bindings page options

It is not possible to set or add the shortcut if it is already used by another command.

To delete a shortcut key definition, select the corresponding menu command in the scroll list and click **Clear** under **Primary** or **Alias**. To revert all command shortcuts to the factory settings, click **Reset All**. Click **OK** to make the new shortcut key bindings take effect.

## External Editor page

On the **External Editor** page—available by choosing **Tools>Options**—you can specify an external editor.

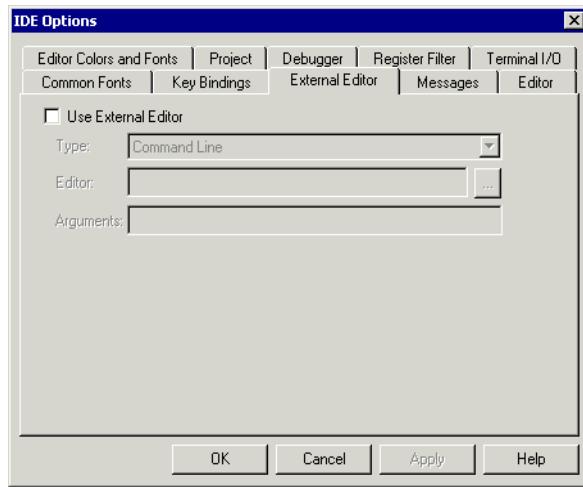


Figure 116: External Editor page with command line settings

## Options

Option	Description
Use External Editor	Enables the use of an external editor.
Type	Selects the method for interfacing with the external editor. The type can be either Command Line or DDE (Windows Dynamic Data Exchange).
Editor	Type the filename and path of your external editor. A browse button is available for your convenience.
Arguments	Type any arguments to pass to the editor. Only applicable if you have selected <b>Type</b> as Command Line.
Service	Type the DDE service name used by the editor. Only applicable if you have selected <b>Type</b> as DDE.

Table 68: External Editor options

Option	Description
Command	Type a sequence of command strings to send to the editor. The command strings should be typed as: <i>DDE-Topic CommandString</i> <i>DDE-Topic CommandString</i> Only applicable if you have selected <b>Type as DDE</b> .

Table 68: External Editor options (Continued)

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.

**Note:** Variables can be used in arguments. See Table 59, *Argument variables*, page 244, for information about available argument variables.

## Messages page

On the **Messages** page—available by choosing **Tools>Options**—you can choose the amount of output in the Messages window.

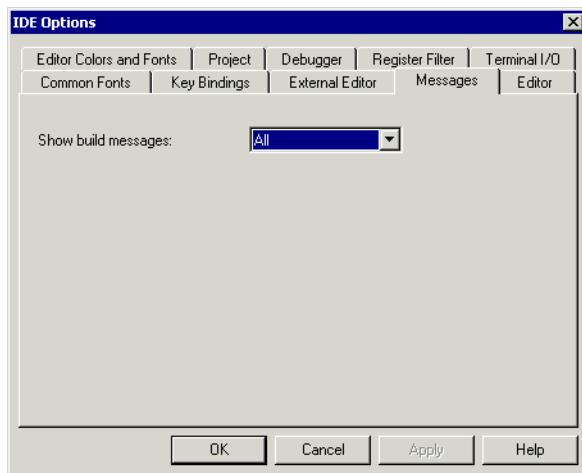


Figure 117: Messages page

## Options

Option	Description
Show build messages	Specifies the amount of output in the Messages window. All: Show all messages. Include compiler and linker information. Messages: Show messages, warnings, and errors. Warnings: Show warnings and errors. Errors: Show errors only.

Table 69: Messages page options

## Editor page

On the **Editor** page—available by choosing **Tools>Options**—you can change the editor options.

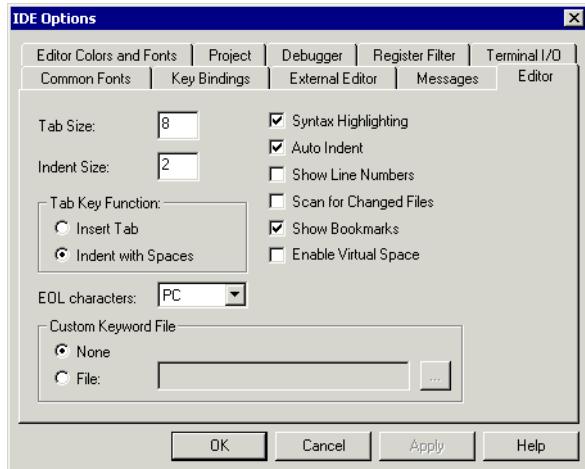


Figure 118: Editor page

## Options

Option	Description
Tab Size	Specifies the number of character spaces corresponding to each tab.
Indent Size	Specifies the number of character spaces to be used for indentation.
Tab Key Function	Specifies how the tab key is used. Either as Insert Tab or as Indent with Spaces.

Table 70: Editor page options

Option	Description
EOL character	Selects line break character.
	<b>PC</b> (default) uses Windows and DOS end of line character.
	<b>Unix</b> uses UNIX end of line characters.
	<b>Preserve</b> uses the same end of line character as the file had when it was read from the disc drive. The PC format is used by default, and if the read file did not have any breaks, or if there is a mixture of break characters used in the file.
Syntax Highlighting	Displays the syntax of C or C++ applications in different text styles.
Auto Indent	Ensures that when you insert a line, the new line will automatically have the same indentation as the previous line.
Show Line Numbers	Displays line numbers in the Editor window.
Scan for Changed Files	Checks if files have been modified by some other tool and automatically reloads them. If a file has been modified in the IAR Embedded Workbench IDE, you will be prompted first.
Show Bookmarks	Displays compiler errors, lines found with the <b>Find in Files</b> command, and user bookmarks.
Enable Virtual Space	Allows the insertion point to move outside the text area.
Custom Keyword File	Selects a text file containing keywords for which you want the editor to use a special syntax highlighting.
Use Default Size	Sets the default size of editor windows.

Table 70: Editor page options (Continued)

For more information about the IAR Embedded Workbench IDE Editor and how it can be used, see *Editing*, page 95.

## Editor Colors and Fonts page

The **Editor Colors and Fonts** page—available by choosing **Tools>Options**—allows you to specify the colors and fonts used for text in the Editor windows.

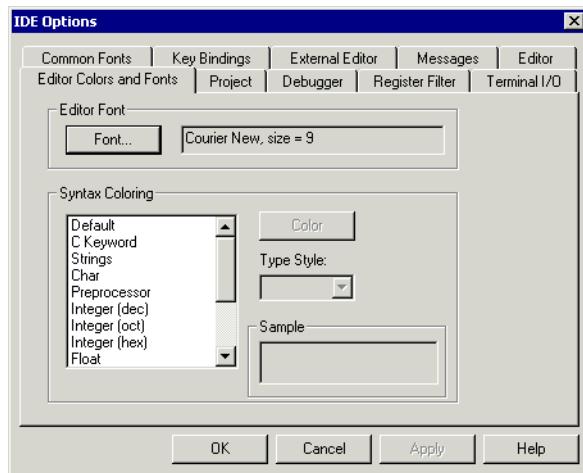


Figure 119: Editor Colors and Fonts page

## Options

Option	Description
Font	Opens a dialog box to choose font and its size.
Syntax Coloring	Lists the possible items for which you can specify font and style of syntax. The elements you can customize are: C or C++, compiler keywords, assembler keywords, and user-defined keywords.
Color	Chooses a color from a list of colors.
Type Style	Chooses a type style from a drop-down list.
Sample	Displays the current setting.

Table 71: Editor Colors and Fonts page options

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.

## Project page

On the **Project** page—available by choosing **Tools>Options**—you can set options for Make and Build. The following table describes the options and their available settings.

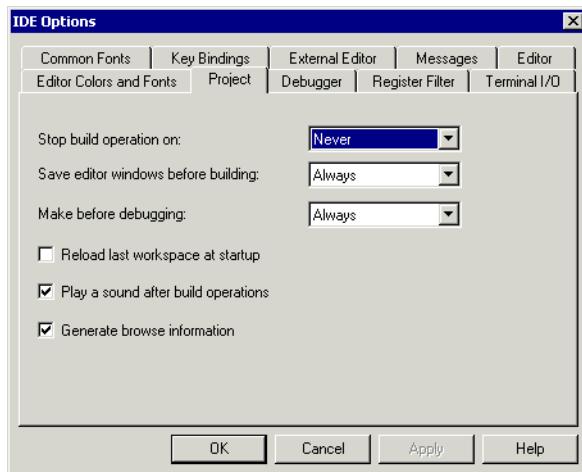


Figure 120: Projects page

## Options

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 build	Always: Always save before Make or Build. Ask: Prompt before saving. Never: Do not save.
Make before debugging	Always: Always make before debugging. Ask: Always prompt before Making. Never: Do not make.
Reload last workspace at startup	Select this option if you want the last active workspace to load automatically the next time you start the IAR Embedded Workbench.
Play a sound after build operations	Plays a sound when the build operations are finished.

Table 72: Project page options

Option	Description
Generate browse information	Enables generation of source browser information.

Table 72: Project page options (Continued)

## Debugger page

On the **Debugger** page—available by choosing **Tools>Options**—you can set options for configuring the debugger environment.

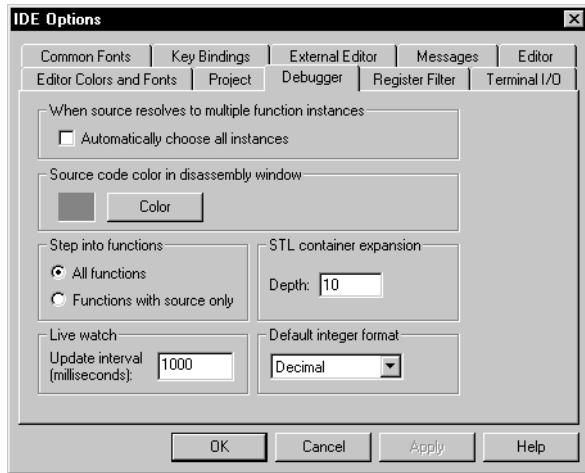


Figure 121: Debugger page

## Options

Option	Description
When source resolves to multiple function instances: Automatically choose all 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. This option lets C-SPY act on all instances without first asking.
Source code color in Disassembly window	Specifies the color of the source code in the Disassembly window.

Table 73: Debugger page options

Option	Description
Step into functions	This option controls the behavior of the Step Into command. If you choose the <b>Functions with source only</b> option, 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 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 value decides how often the C-SPY Live Watch window is updated during execution.
Default integer format	Sets the default integer format in the Watch, Locals, and related windows.

Table 73: Debugger page options (Continued)

## Register Filter page

On the **Register Filter** page—available by choosing **Tools>Options** when the IAR C-SPY Debugger is running—you can choose to display registers in the Register window in groups you have created yourself. See *Register groups*, page 132, for more information about how to create register groups.

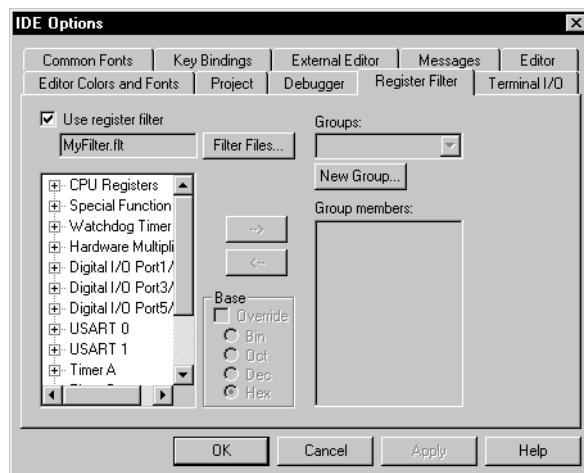


Figure 122: Register Filter page

## Options

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 74: Register Filter options

## Terminal I/O page

On the **Terminal I/O** page—available by choosing **Tools>Options** when the IAR C-SPY Debugger is running—you can configure the C-SPY terminal I/O functionality.

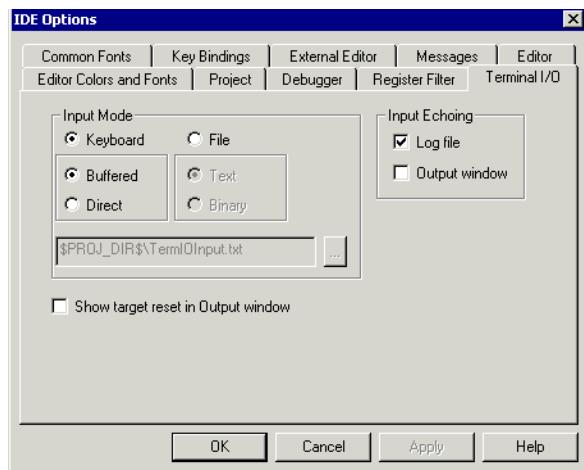


Figure 123: Terminal I/O page

## Options

Option	Description
Input Mode: Keyboard	Buffered: All input characters are buffered. Direct: Input characters are not buffered.

Table 75: Register Filter options

Option	Description
Input Mode: File	Input characters are read from a file, either a text file or a binary file. A browse button is available for locating the file.
Show target reset in Output window	When target resets, a message is displayed in the C-SPY Terminal I/O window.
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 <b>Enable log file</b> that is available by choosing <b>Debug&gt;Logging</b> .

Table 75: Register Filter options (Continued)

### 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.

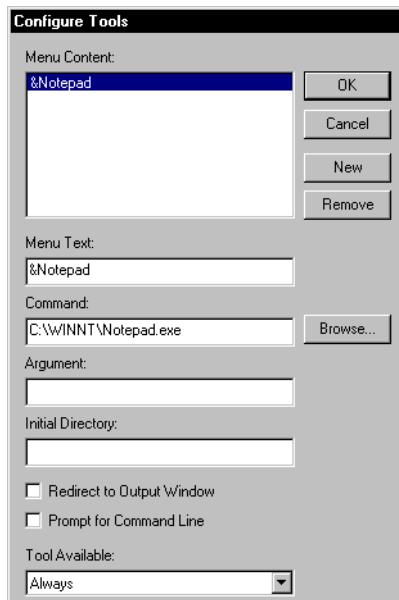


Figure 124: Configure Tools dialog box

## Options

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 appear as the mnemonic key for this command. The text you type in this field will be reflected in the <b>Menu Content</b> field.
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 <b>Tool Output</b> page in the Messages window. Tools that are launched with this option cannot receive any user input, for instance input from the keyboard. Tools that require user input or make special assumptions regarding the console that they execute in, will not work at all if launched with this option.
Prompt for Command Line	Displays a prompt for the command line argument when the command is chosen from the <b>Tools</b> menu.
Tool Available	Specifies in which context the tool should be available: Always, only when debugging, or only when not debugging.

Table 76: Configure Tools dialog box options

**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.

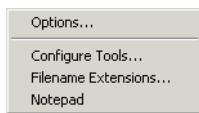


Figure 125: Customized Tools menu

## 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 98/Me	command.com
Windows NT/2000/XP	cmd.exe (recommended) or command.com

Table 77: 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.

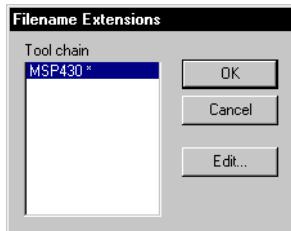


Figure 126: Filename Extensions dialog box

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.

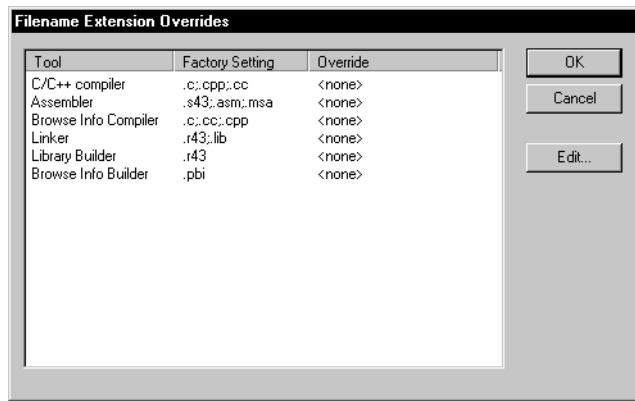


Figure 127: *Filename Extension Overrides* dialog box

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.

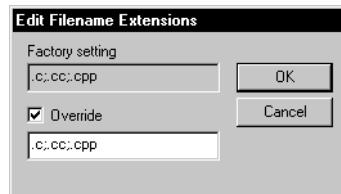


Figure 128: *Edit Filename Extensions* dialog box

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.

## WINDOW MENU

Use the commands on the **Window** menu to manipulate the IAR Embedded Workbench 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.

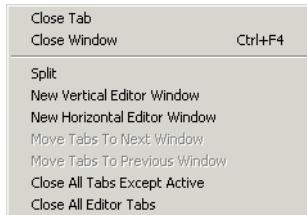


Figure 129: Window menu

### Window menu commands

Menu Commands	Description
Close Tab	Closes the active tab.
Close Window	Ctrl+F4 Closes the active 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 editor window to the next editor window.
Move Tabs To Previous Window	Moves all tabs in current editor window to the previous editor window.
Close All Tabs Except Active	Closes all the editor window tabs except the active tab.
Close All Editor Tabs	Closes all tabs currently available in editor windows.

Table 78: Window menu commands

## HELP MENU

The **Help** menu provides help about the IAR Embedded Workbench IDE and displays the version numbers of the user interface and of the MSP430 IAR Embedded Workbench IDE.

Menu Command	Description
Content	Opens the contents page of the IAR Embedded Workbench IDE online help.
Index	Opens the index page of the IAR Embedded Workbench IDE online help.
Search	Opens the search page of the IAR Embedded Workbench IDE online help.
Release notes	Provides access to late-breaking information about the IAR Embedded Workbench.
MSP430 Assembler Reference Guide	Provides access to an online version of the <i>MSP430 IAR Assembler Reference Guide</i> , available in PDF format.
MSP430 Embedded Workbench User Guide	Provides access to an online version of this user guide, available in PDF format.
MSP430 C/C++ Compiler Reference Guide	Provides access to an online version of the <i>MSP430 IAR C/C++ Compiler Reference Guide</i> , available in PDF format.
MSP430 Migration guide	Provides access to an online version of the <i>MSP430 IAR Embedded Workbench Migration Guide</i> , available in PDF format.
Linker and Library Tools Reference Guide	Provides access to the online version of the <i>IAR Linker and Library Tools Reference Guide</i> , available in PDF format.
IAR on the Web	Allows you to browse the home page, the news page, and the technical notes search page of the IAR website, and to contact IAR Technical Support.
Startup Screen	Displays the Embedded Workbench Startup dialog box.
About	Clicking the <b>Product info</b> button in the dialog box that opens displays detailed information about the installed IAR products. Copy this information (using the <b>Ctrl+C</b> keyboard shortcut) and include it in your message if you contact IAR Technical Support via electronic mail.

Table 79: Help menu commands

## Embedded Workbench Startup dialog box

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

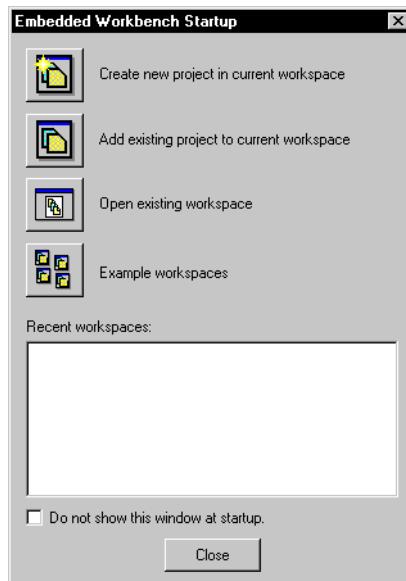


Figure 130: Embedded Workbench Startup dialog box



# C-SPY Debugger IDE reference

This chapter contains detailed reference information about the windows, menus, menu commands, and the corresponding components that are specific for the IAR C-SPY™ Debugger.

---

## C-SPY windows

The following windows specific to C-SPY are available in the IAR C-SPY Debugger:

- IAR C-SPY Debugger window
- Disassembly window
- Memory window
- Register window
- Watch window
- Locals window
- Auto window
- Live Watch window
- Call Stack window
- Terminal I/O window
- Code Coverage window
- Profiling window
- Trace window
- Stack window.

### EDITING IN C-SPY WINDOWS

You can edit the contents of the Memory, Register, Auto, Watch, Locals, and Live Watch windows, and the **Quick Watch** dialog box.

Use the following keyboard keys to edit the contents of the Register and Watch windows:

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

*Table 80: Editing in C-SPY windows*

## IAR C-SPY DEBUGGER WINDOW

The following figure shows the main IAR Embedded Workbench IDE window when you are debugging, using the IAR C-SPY Simulator. The window might look different depending on which components you are using.

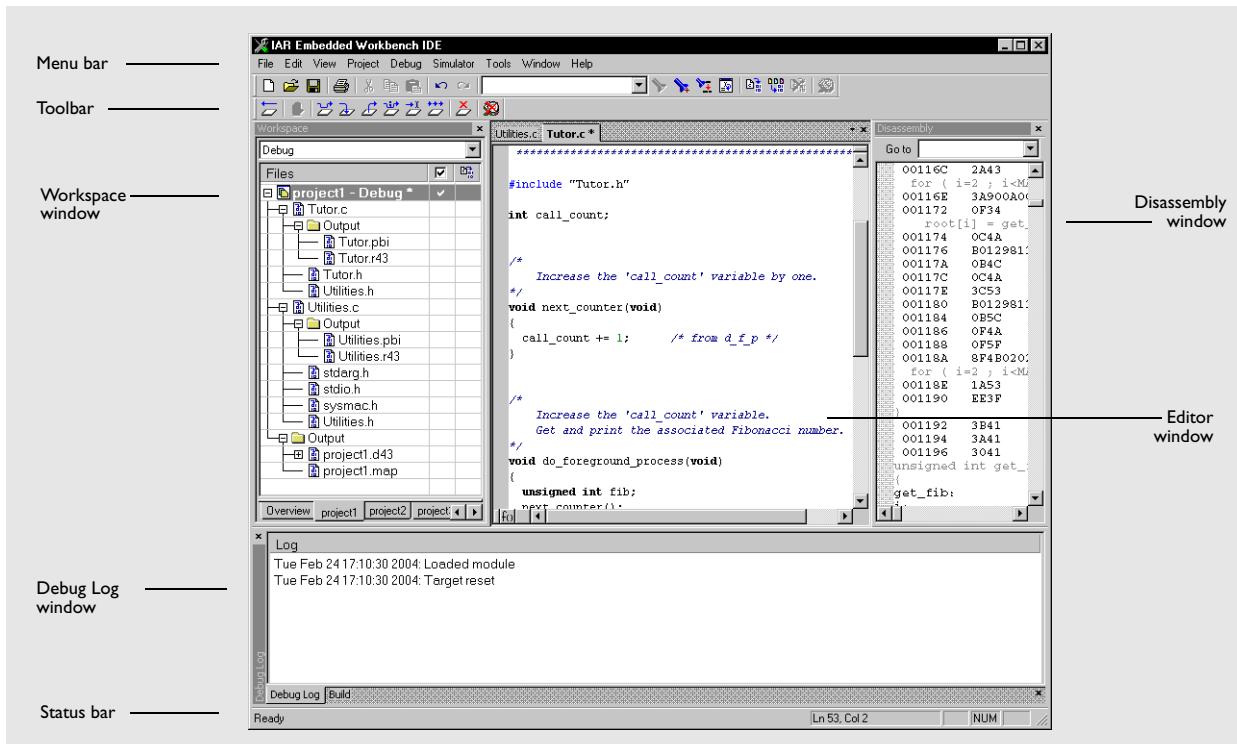


Figure 131: Embedded Workbench with C-SPY Simulator started

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 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.
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.
Emulator	The Emulator menu provides access to commands specific to the C-SPY FET Debugger. Only available when the C-SPY FET Debugger is used.

Table 81: C-SPY menu

Additional menus might be available, depending on which debugger drivers that have been installed.

## 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:

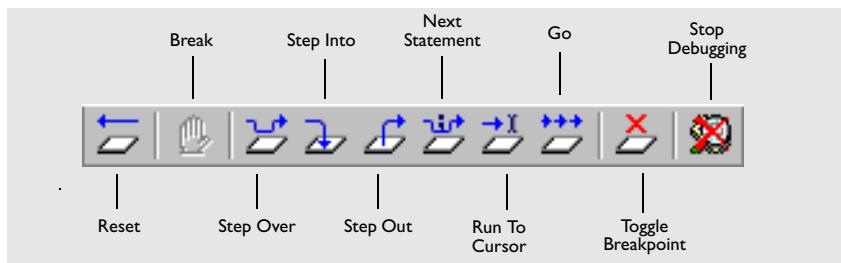


Figure 132: C-SPY debug toolbar

## DISASSEMBLY WINDOW

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

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. Breakpoints are indicated in red. Code that has been executed—code coverage—is indicated with a green diamond.

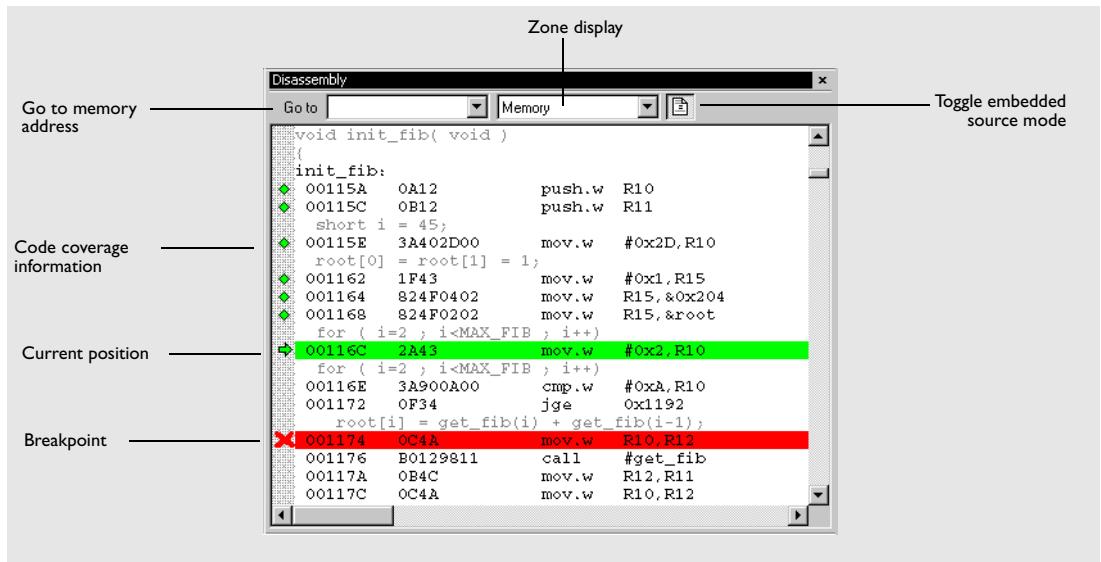


Figure 133: C-SPY Disassembly window

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.

### Disassembly window operations

At the top of the window you can find a set of useful text boxes, drop-down lists and command buttons:

Operation	Description
Go to	The memory location you want to view.
Zone display	Lists the available memory or register zones to display. Read more about Zones in section <a href="#">Memory zones</a> , page 110.

Table 82: Disassembly window operations

Operation	Description
Disassembly mode	Toggles between showing only disassembly or disassembly together with the corresponding source code. Source code requires that the corresponding source file has been compiled with debug information.

Table 82: Disassembly window operations (Continued)

### Disassembly context menu

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

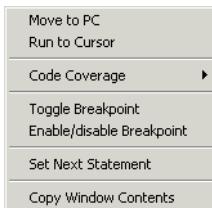


Figure 134: Disassembly window context menu

Operation	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	<b>Enable</b> toggles code coverage on and off.
Show	<b>Show</b> toggles between displaying and hiding code coverage. Executed code is indicated by a green diamond.
Clear	<b>Clear</b> clears all code coverage information.
Toggle Breakpoint	Toggles a code breakpoint. Assembler instructions at which code breakpoints have been set are highlighted in red.
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.

Table 83: Disassembly context menu commands

## 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.

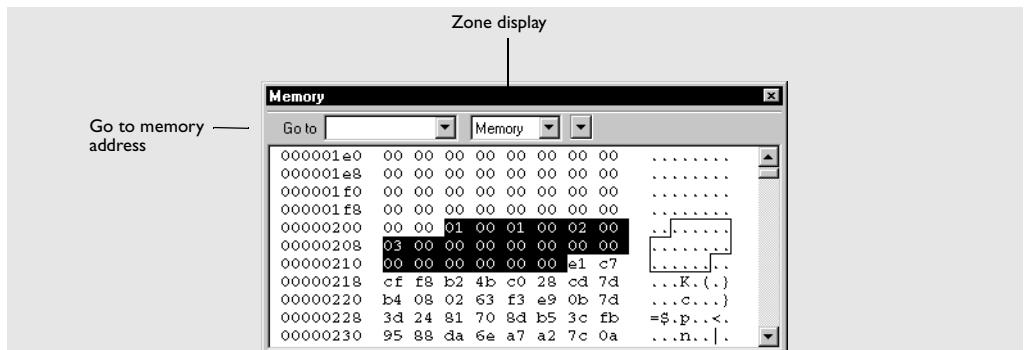


Figure 135: Memory window

### Memory window operations

At the top of the window you can find commands for navigation:

Operation	Description
Go to	The address of the memory location you want to view.
Zone display	Lists the available memory or register zones to display. Read more about Zones in the section <i>Memory zones</i> , page 110.

Table 84: Memory window operations

## Memory window context menu

The context menu available in the Memory window provides above commands, edit commands, and a command for opening the **Fill** dialog box.



Figure 136: Memory window context menu

Operation	Description
Copy, Paste	Standard editing commands.
Zone	Lists the available memory or register zones to display. Read more about Zones in <i>Memory zones</i> , page 110.
x1, x2, x4 Units	Switches between displaying the memory contents in units of 8, 16, or 32 bits
Little Endian Big Endian	Switches between displaying the contents in big-endian or little-endian order. An asterisk (*) indicates the default byte order.
Data Coverage	<b>Enable</b> toggles data coverage on and off. <b>Show</b> toggles between showing and hiding data coverage. <b>Clear</b> clears all data coverage information.
Memory Fill	Opens the Fill dialog box, where you can fill a specified area with a value.
Memory Upload	Displays the Memory Upload dialog box, where you can save a selected memory area to a file in Intel Hex format.
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 85: Memory window operations

## Data coverage display

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.

## Fill dialog box

In the **Fill** dialog box—available from the context menu available in the Window memory—you can fill a specified area of memory with a value.

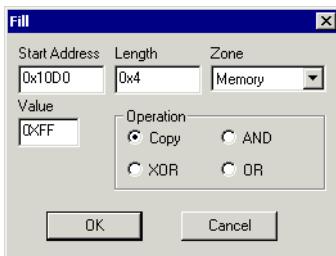


Figure 137: Fill dialog box

### 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 86: Fill dialog box options

These are the available memory fill operations:

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

Table 87: Memory fill operations

Operation	Description
XOR	An XOR operation will be performed between the <b>Value</b> and the existing contents of memory before writing the result to memory.
OR	An OR operation will be performed between the <b>Value</b> and the existing contents of memory before writing the result to memory.

Table 87: Memory fill operations (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.

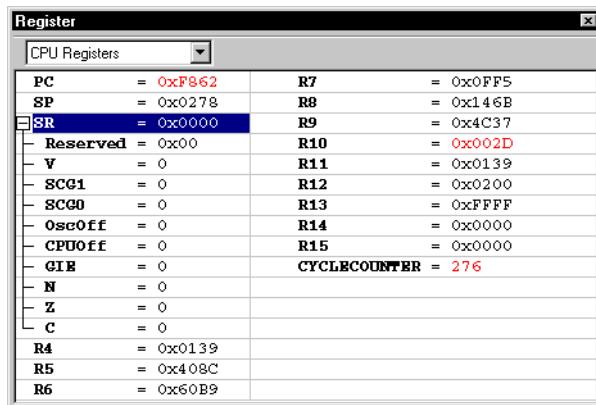
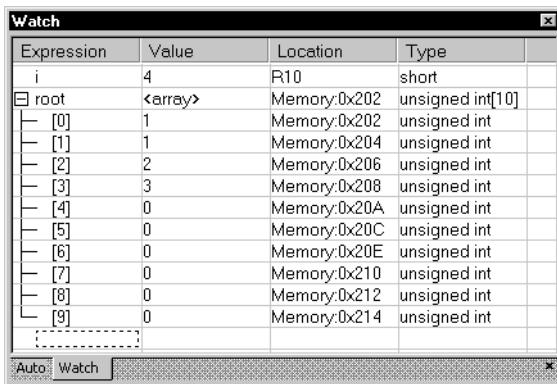


Figure 138: Register window

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 133.

## 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.



The screenshot shows the C-SPY Watch window with the title bar "Watch". The window contains a table with four columns: Expression, Value, Location, and Type. The table shows the memory dump of a variable named "i". The first row shows "i" with a value of 4 at location R10, type short. Below it, under the "root" node, is an array <array>. This array has elements [0] through [9]. Each element has a value of 1, located at Memory:0x202, type unsigned int[10]. The table continues with elements [1] through [9], each with a value of 1, located at Memory:0x202, type unsigned int. Element [0] also has sub-elements [0] through [9], each with a value of 1, located at Memory:0x204, type unsigned int. Element [1] has sub-elements [0] through [9], each with a value of 1, located at Memory:0x206, type unsigned int. Element [2] has sub-elements [0] through [9], each with a value of 2, located at Memory:0x208, type unsigned int. Element [3] has sub-elements [0] through [9], each with a value of 3, located at Memory:0x20A, type unsigned int. Element [4] has sub-elements [0] through [9], each with a value of 0, located at Memory:0x20C, type unsigned int. Element [5] has sub-elements [0] through [9], each with a value of 0, located at Memory:0x20E, type unsigned int. Element [6] has sub-elements [0] through [9], each with a value of 0, located at Memory:0x210, type unsigned int. Element [7] has sub-elements [0] through [9], each with a value of 0, located at Memory:0x212, type unsigned int. Element [8] has sub-elements [0] through [9], each with a value of 0, located at Memory:0x214, type unsigned int. Element [9] has sub-elements [0] through [9], each with a value of 0, located at Memory:0x216, type unsigned int. At the bottom of the window, there are tabs for "Auto" and "Watch", with "Auto" being selected.

Figure 139: Watch window

Every time execution in C-SPY stops, a value that has changed since last stop is highlighted. In fact, every time memory changes, the values in the Watch window are recomputed, including updating the red highlights.

**Note:** To watch assembler variables, you must specify the memory type as in this example:

```
(__data16 unsigned int *)my_var
```

### Watch window context menu

The context menu available in the Watch window provides commands for changing the display format of expressions, as well as commands for adding and removing expressions.

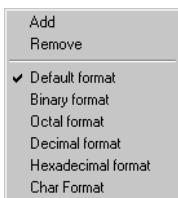


Figure 140: Watch window context menu

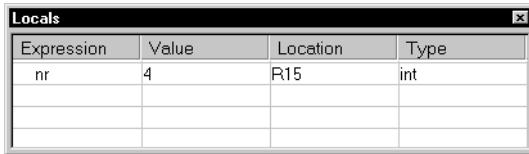
The display format setting affects different types of expressions in different ways:

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

The default integer format can be changed on the Debugger page of the **IDE Options** dialog box, see *Debugger page*, page 260. Your selection of display format is saved between debug sessions.

## LOCALS WINDOW

The Locals window—available from the **View** menu—automatically displays the local variables and function parameters.



Expression	Value	Location	Type
nr	4	R15	int

Figure 141: Locals window

### Locals window context menu

The context menu available in the Locals window provides commands for changing the display format of expressions.

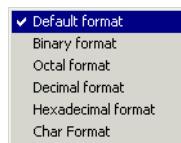
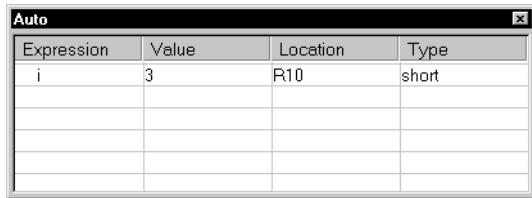


Figure 142: C-SPY Locals window context menu

The default integer format can be changed on the Debugger page of the **IDE Options** dialog box, see *Debugger page*, page 260. Your selection of display format is saved between debug sessions.

## 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	3	R10	short

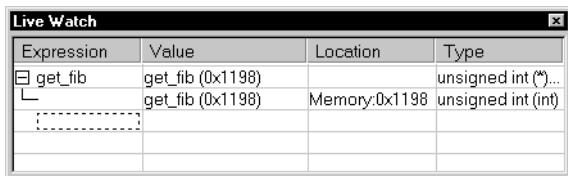
Figure 143: Auto window

## Auto window context menu

The context menu available in the Auto window provides commands for changing the display format of expressions. For reference information, see *Locals window context menu*, page 281.

## LIVE WATCH WINDOW

The Live Watch window—available from the **View** menu—automatically displays the value of variables with a static location, such as global variables.



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

Figure 144: Live Watch window

All expressions are evaluated repeatedly during execution, in effect, sampling the values of the expressions.

## Live Watch window context menu

The context menu available in the Live Watch window provides commands for changing the display format of expressions. The default integer format can be changed on the Debugger page of the **IDE Options** dialog box, see *Debugger page*, page 260. Your selection of display format is saved between debug sessions.

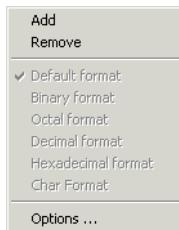


Figure 145: C-SPY Live Watch window context menu

The **Options** command 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.

## 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.

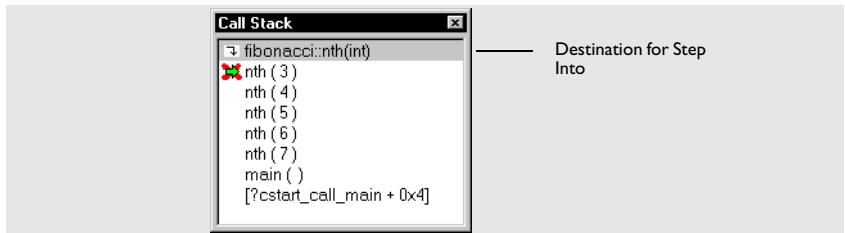


Figure 146: Call Stack window

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 in the Call Stack window provides some useful commands when you right-click.

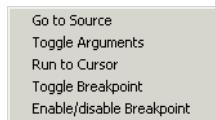


Figure 147: Call Stack window context menu

#### Commands

Go to Source	Displays the selected functions in the Disassembly or editor windows.
Toggle Arguments	Shows or hides function arguments.
Run to Cursor	Executes to the function selected in the call stack.
Toggle Breakpoint	Toggles a code breakpoint. This breakpoint is not saved between debug sessions.
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 link the application with the option **With I/O emulation modules**. C-SPY will then direct `stdin`, `stdout` and `stderr` to this window. If the Terminal I/O window is closed, C-SPY will open it automatically when input is required, but not for output.

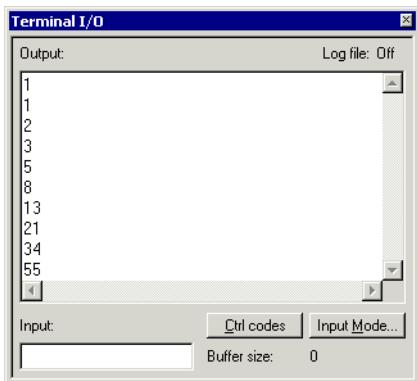


Figure 148: Terminal I/O window

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



Figure 149: Ctrl codes menu

Clicking the **Input Mode** button opens the **Change Input Mode** dialog box where you choose whether to input data from the keyboard or from a text file.

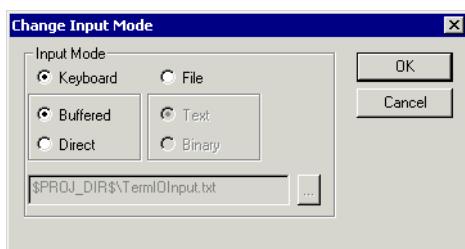


Figure 150: Change Input Mode dialog box

For reference information about the options available in the dialog box, see *Terminal I/O page*, page 262.

## CODE COVERAGE WINDOW

**Note:** Code coverage is only supported by the C-SPY Simulator.

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.

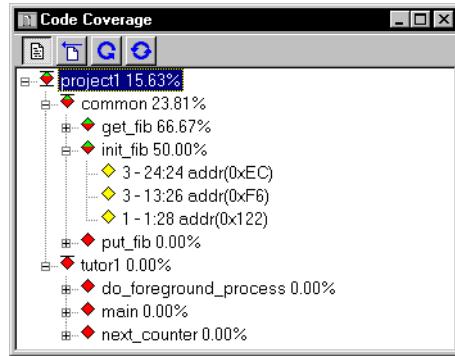


Figure 151: Code Coverage window

### 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 152: 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.

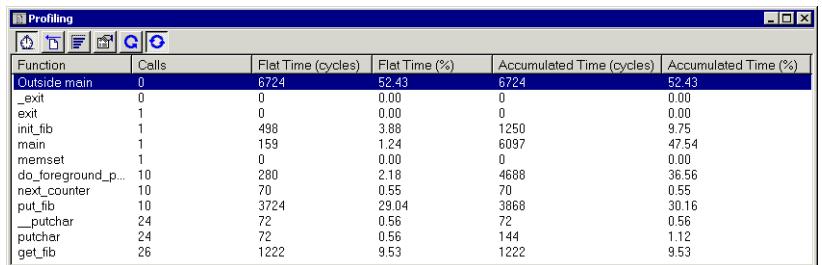
For step point lines, the information displayed is the column number range and the row number of the step point in the source window, followed by the address of the step point.

<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	Cells	Flat Time (cycles)	Flat Time (%)	Accumulated Time (cycles)	Accumulated Time (%)
Outside main	0	6724	52.43	6724	52.43
_exit	0	0	0.00	0	0.00
exit	1	0	0.00	0	0.00
init_fib	1	498	3.88	1250	9.75
main	1	159	1.24	6097	47.54
memset	1	0	0.00	0	0.00
do_foreground_p...	10	280	2.18	4688	36.56
next_counter	10	70	0.55	70	0.55
put_fib	10	3724	29.04	3868	30.16
__putchar	24	72	0.56	72	0.56
putchar	24	72	0.56	144	1.12
get_fib	26	1222	9.53	1222	9.53

Figure 153: Profiling window

## 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:

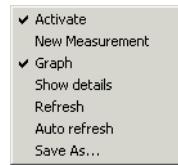


Figure 154: Profiling context menu

You can find the following commands on the menu:

- |   |                        |  |
|---|------------------------|--|
|  | <b>Activate</b>        | Toggles profiling on and off during execution.   |
|  | <b>New measurement</b> | Starts a new measurement. By clicking the button, the values displayed are reset to zero.  |
|  | <b>Graph</b>           | Displays the percentage information for Flat Time and Accumulated Time as graphs (bar charts) or numbers.  |
|  | <b>Show details</b>    | 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. |
|  | <b>Refresh</b>         | 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 88: 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.

### TRACE WINDOW

In the Trace window—available from the **View** menu—you can trace the values of C-SPY expressions. There are two pages in the Trace window, the **Expression** page and the **Output** page.

## Expression page

Define the expressions you want to trace on the **Expression** page.

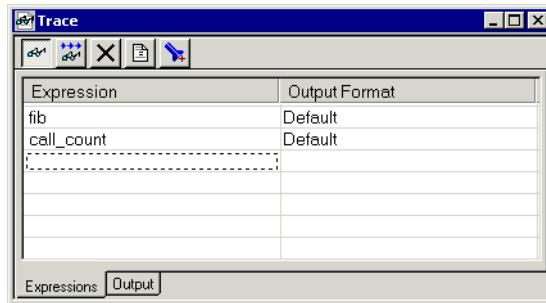


Figure 155: Trace window (Expression page)

The expressions you define will appear in the left-most column **Expression**. The **Output Format** column shows which display format is used for each expression.

## Output page

The **Output** page displays the trace buffer which contains the generated trace information. C-SPY generates trace information based on the location of the program counter.

PC	fib	call_count
0x114A		0
0x1142	1	1
0x1142	1	2
0x1142	2	3
0x1142	3	4
0x1142	5	5
0x1142	8	6

Figure 156: Trace window (Output page)

The column **PC** displays the current location of the program counter at which the expression was evaluated. Each expression you have defined appears in a separate column. Each entry in this column displays, for the current location of the program counter, the value of the expression.

You can save the trace information to a file and there browse the information conveniently.

## Trace toolbar

Command	Description
Activate/Deactivate	Activates and deactivates the trace function.
Go with Trace	The application is executed and the values of expressions are logged continuously for each PC location. During execution, no C-SPY windows are refreshed.
Clear	Clears the trace buffer.
Toggle Mixed-Mode	Displays C and assembler source on the Output page together with values of expressions.
	Toggles between showing only PC locations and the calculated values, or disassembly together with corresponding source code.
Find	Opens the Find dialog box where you can search for PC locations within the trace information.

Table 89: Trace toolbar commands

## Trace window context menu

The context menu—available on the Expressions page in the Trace window—gives you access to the following commands:

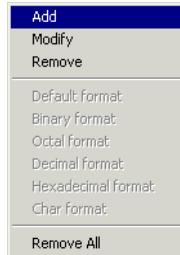


Figure 157: Trace window context menu

Command	Description
Add	Adds an expression.
Modify	Modifies an expression.
Remove	Removes the selected expression.
Display format	Can be one of Default, Binary, Octal, Decimal, Hexadecimal, and Char format.
Remove All	Removes all expressions.

Table 90: Trace context commands

Your expression definitions and selection of display format are saved between debug sessions.

## STACK WINDOW

The Stack window—available from the **View** menu—displays the MSP430 microcontroller stack.

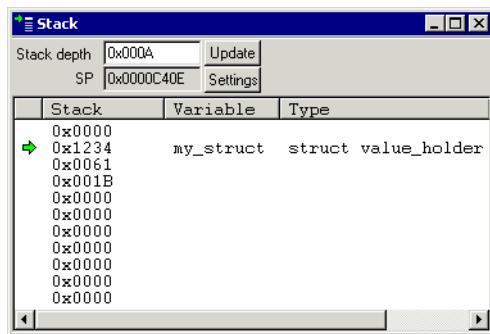


Figure 158: Stack window

The **Stack** column displays the value of the stack pointer. The **Symbol** column displays any C or C++ symbols located on the stack, and the **Type** column displays the value of each symbol. The stack pointer values can be edited.

Operation	Description
Stack depth	Displays the stack address range, always at least one element before the stack pointer.
Update	To make changes take effect or to force a refresh of the window contents.
SP	Displays the current stack pointer address. SP is the stack pointer name, which can be any known register or symbol name; depends on the selected stack pointer name.
Settings	Opens the Stack Settings dialog box

Table 91: Stack window operations

A green arrow indicates the top of the stack contents

## Stack Settings dialog box

Click the **Settings** button in the Stack window to display the **Stack Settings** dialog box.

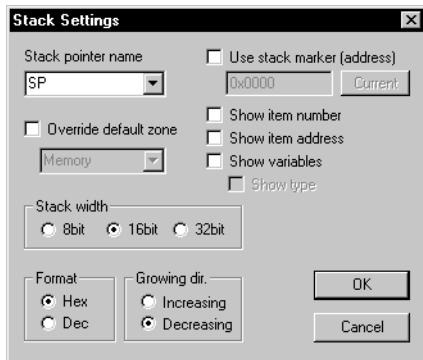


Figure 159: Stack Settings dialog box

These are the available settings:

Setting	Description
Stack pointer name	The symbol name of the stack pointer. Possible stack pointer names are any available register.
Override default zone	Overrides the default zone in case the stack is not located there. This option requires that other zones than the default are defined.
Stack width	Width of stack displayed in the Stack window.
Format	Selects the display format of the stack contents in the Stack window.
Growing direction	Selects the direction in which the stack grows.

Table 92: Stack window settings

---

## C-SPY menus

The following C-SPY specific menus are available in the IAR C-SPY Debugger:

- Debug menu
- Simulator menu
- Emulator menu.

## DEBUG MENU

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



Figure 160: Debug menu

Menu Command	Description
	Executes from the current statement or instruction until a breakpoint or program exit is reached.
	Stops the application execution.
	Resets the target processor.
	Stops the debugging session and returns you to the project manager.
	Executes the next statement or instruction, without entering C or C++ functions or assembler subroutines.
	Executes the next statement or instruction, entering C or C++ functions or assembler subroutines.
	Executes from the current statement up to the statement after the call to the current function.
	If stepping into and out of functions is unnecessarily slow, use this command to step directly to the next statement.
	Executes from the current statement or instruction up to a selected statement or instruction.

Table 93: Debug menu commands

Menu Command	Description
Autostep	Displays the <b>Autostep settings</b> dialog box which lets you customize and perform autostepping.
Quick Watch	Displays a dialog box to allow you to watch the value of a variable or expression and to execute macros.
Refresh	Refreshes the contents of the Memory, Register, Watch, and Locals windows.
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.
Macros	Displays the <b>Macro Configuration</b> dialog box to allow you to list, register, and edit your macro files and functions.
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 93: Debug menu commands (Continued)

### Quick Watch dialog box

In the Quick Watch dialog box—available from the **Debug** menu—you can watch the value of a variable or expression and evaluate expressions.

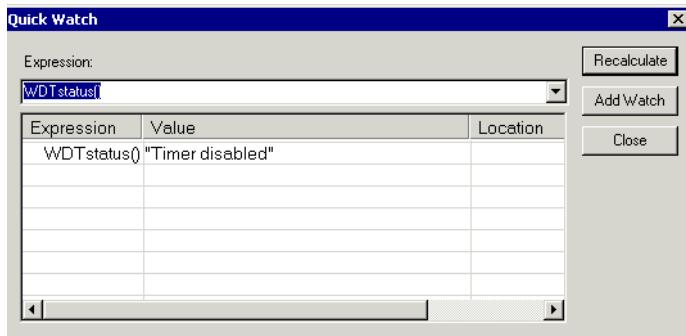


Figure 161: Quick Watch dialog box

Type the expression you want to examine in the **Expressions** text box. Click **Recalculate** to calculate the value of the expression. For examples about how to use the **Quick Watch** dialog box, see *Using the Quick Watch dialog box*, page 120 and *Executing macros using Quick Watch*, page 140.

### Autostep settings dialog box

In the **Autostep settings** dialog box—available by choosing **Debug>Autostep**—you can customize autostepping.

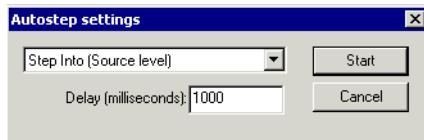


Figure 162: Autostep settings dialog box

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.

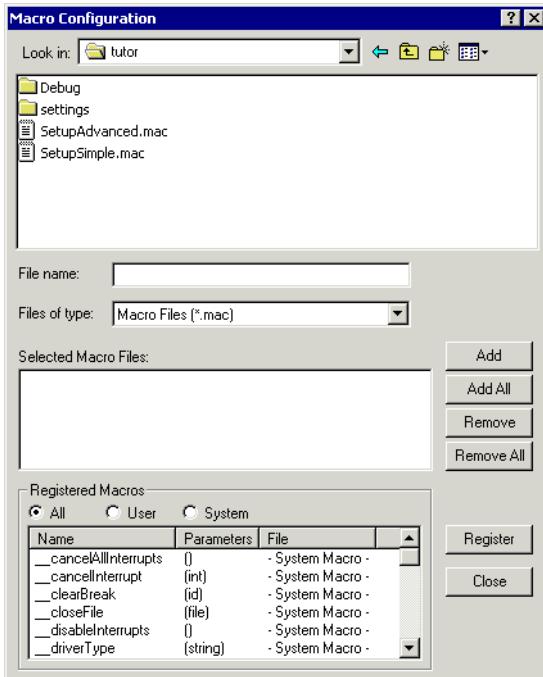


Figure 163: Macro Configuration dialog box

### **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.

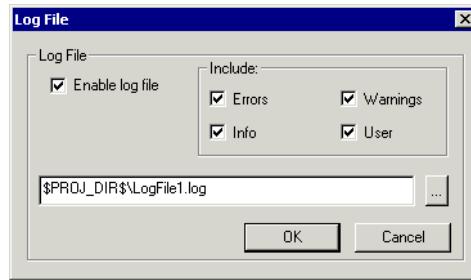


Figure 164: Log File dialog box

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 94: 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 165: 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**.

## SIMULATOR MENU

For details of the Simulator menu, see *Simulator-specific menus*, page 153.

## EMULATOR MENU

For details of the Emulator menu, see *Emulator menu*, page 201.



# General options

This chapter describes the general options in the IAR Embedded Workbench™.

For information about how options can be set, see *Setting options*, page 89.

---

## Target

The **Target** options specify the options **Device**, **Position-independent code**, **Hardware multiplier**, **Assembler only project**, and **Double floating point size** for the MSP430 IAR C/C++ Compiler and Assembler.

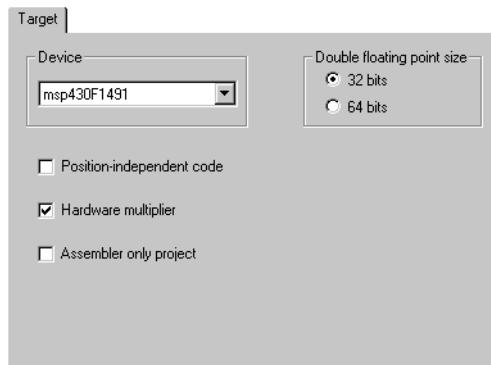


Figure 166: Target options

### DEVICE

Use the drop-down list to select the device for which you will build your application. The choice of device controls which linker command file and device description file that will be used.

### POSITION-INDEPENDENT CODE

Select normal or position-independent code generation. Note that position-independent code will lead to a rather large overhead in code size. For more details about position-independent code, see *MSP430 IAR C/C++ Compiler Reference Guide*.

## HARDWARE MULTIPLIER

Generates code for the MSP430 hardware multiplier peripheral unit. The option is only enabled when you have chosen a device containing the hardware multiplier from the **Device** drop-down list.

## ASSEMBLER ONLY PROJECT

Use this option if your project only contains assembler source files. The option will make the necessary settings required for an assembler only project, for instance, disabling the use of a C or C++ runtime library and the cstartup system.

## DOUBLE FLOATING POINT SIZE

The compiler represents floating-point values by 32- and 64-bit numbers in standard IEEE format. By default, both the `float` data type and the `double` data type are represented by the 32-bit floating-point format. If you select the **64 bits** option, the data type `double` is instead represented by the 64-bit floating-point format.

For more details about floating point sizes, see *MSP430 IAR C/C++ Compiler Reference Guide*.

---

## 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.

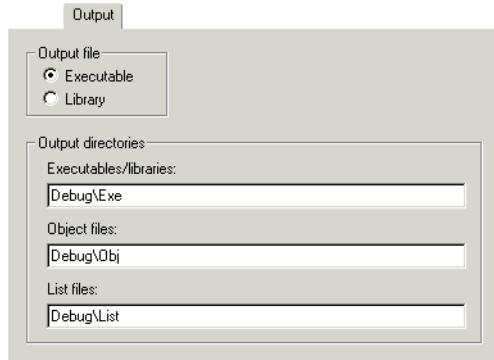


Figure 167: Output options

## OUTPUT FILE

You can choose the type of output file with the **Executable** and **Library** options.

If you select the **Executable** option, XLINK options will be available in the **Options** dialog box. As a result of the build process, the XLINK Linker will create an application (an executable output file). Before you create the output you should set the XLINK options.

If you select the **Library** option, XAR options will be available in the **Options** dialog box, and **Linker** will disappear from the list of categories. As a result of the build process, the XAR Library Builder will create a library output file. Before you create the library you can set the XAR options.

## OUTPUT DIRECTORIES

Use these options to specify paths to destination directories. Note that incomplete paths are relative to your project directory.

### Executables libraries

Use this option to override the default directory for executable 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 runtime library to use.

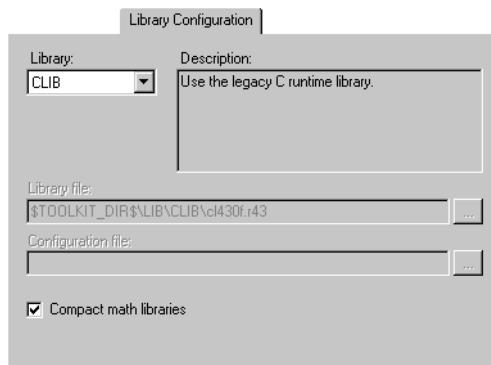


Figure 168: Library Configuration options

For information about the runtime library, library configurations, the runtime environment they provide, and the possible customizations, see *MSP430 IAR C/C++ Compiler Reference Guide*.

## LIBRARY

In the **Library** drop-down list you choose between the following runtime libraries:

Library	Description
None	Do not link the application with a runtime library.
Normal DLIB	Use the normal configuration of the C/C++ runtime library. No locale interface, C locale only, no file descriptor support, no multibytes in printf and scanf, and no hex floats in strtod.
Full DLIB	Use the full configuration of the C/C++ runtime library. Full locale interface, C locale only, file descriptor support, multibytes in printf and scanf, and hex floats in strtod.
Custom DLIB	Use a customized C/C++ runtime library. In this case you must specify a library file and a library configuration file.
CLIB (default)	Use the legacy C runtime library.
Custom CLIB	Use a customized legacy C runtime library. In this case you must specify a library file.

Table 95: Libraries

Note that the DLIB library comes in two variants of configurations—Normal and Full.

For C++ projects, you must use one of the DLIB library variants.

The library object file and library configuration file that actually will be used is displayed in the **Library** file and **Configuration** file text boxes, respectively.

### LIBRARY FILE

The **Library** file text box displays the library object file that will be used. A library object file is automatically chosen depending on your settings of the following options:

- Library type
- Size of double floating-point values
- Library configuration (DLIB only)
- Position-independent code.

If you have chosen **Custom DLIB** or **Custom CLIB** in the **Library** drop-down list, you must specify your own library object file.

### CONFIGURATION FILE

The **Configuration** file text box displays the DLIB library configuration file that will be used. A library configuration file is chosen automatically depending on the project settings. If you have chosen **Custom DLIB** in the **Library** drop-down list, you must specify your own library configuration file.

**Note:** A library configuration file is only required for the DLIB library.

### COMPACT MATH LIBRARIES

When this option is selected, the compiler can use a floating-point library which is smaller than the ordinary CLIB library, but lacking in a number of features. See the *MSP430 IAR C/C++ Compiler Reference Guide* for more information.

## Library Options

With the options on the **Library Options** page you can choose `printf` and `scanf` formatters.

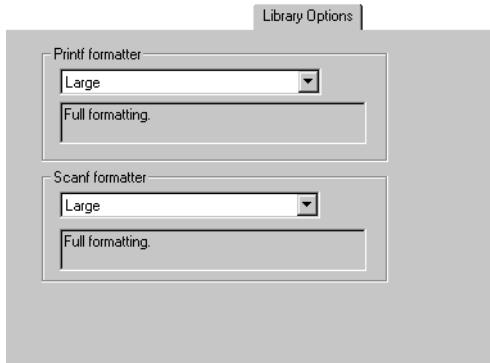


Figure 169: Library Options page

See the *MSP430 IAR C/C++ Compiler Reference Guide* 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 IAR DLIB Library are: Full, Large, Small, and Tiny.
- Printf formatters in the IAR CLIB Library are: Large, Medium, and Small.

### 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:

- Sccanf formatters in the IAR DLIB Library are: Full, Large, and Small.
- Sccanf formatters in the IAR CLIB Library are: Large, and Medium.

## Stack/Heap

With the options on the **Stack/Heap** page you can customize the heap and stack sizes.

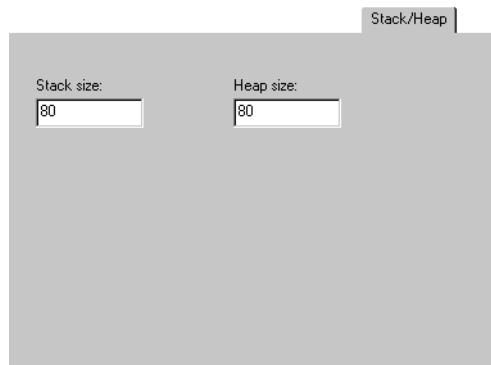


Figure 170: Stack/Heap page

### STACK SIZE

Enter the required stack size in bytes in the **Stack size** text box.

### HEAP SIZE

Enter the required heap size in bytes in the **Heap size** text box.



# Compiler options

This chapter describes the compiler options available in the IAR Embedded Workbench™.

For information about how to set options, see *Setting options*, page 89.

---

## Language

The **Language** options enable the use of extensions to the C or Embedded C++ language.

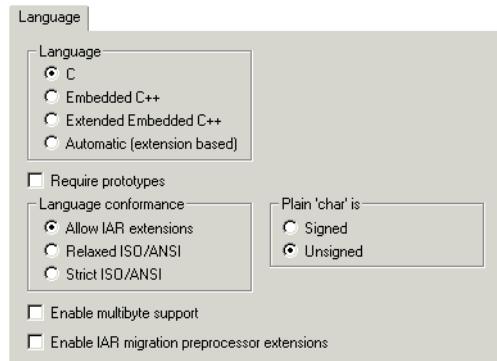


Figure 171: Compiler language options

## LANGUAGE

With the **Language** options you can specify the language support you need.

For information about Embedded C++ and Extended Embedded C++, see the *MSP430 IAR C/C++ Compiler Reference Guide*.

## C

By default, the MSP430 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 C++. This means that features specific to C++, such as classes and overloading, can be utilized.

Embedded C++ requires that a DLIB library (C/EC++ library) is used.

## **Extended Embedded C++**

In Extended Embedded C++ mode, you can take advantage of features like namespaces or the standard template library in your source code.

Extended Embedded C++ requires that a DLIB library (C/EC++ library) is used.

## **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.

This option requires that a DLIB library (C/EC++ library) is used.

## **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 MSP430 IAR C/C++ Compiler to be able to accept MSP430-specific keywords as extensions to the standard C or Embedded C++ language. In the IAR Embedded Workbench, 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 *MSP430 IAR C/C++ Compiler Reference Guide*.

### 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 radio button **Signed**, 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 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 or C++ style comments, in string literals, and in character constants. They are transferred untouched to the generated code.

### ENABLE IAR MIGRATION PREPROCESSOR EXTENSIONS

Migration preprocessor extensions extend the preprocessor in order to ease migration of code from earlier IAR compilers. If you need to port code written for the MSP430 IAR C Compiler version 1.x, you may want to use this option.

**Note:** If you use this option, not only will the compiler accept code that is not standard conformant, but it will also reject some code that *does* conform to standard.

**Important!** Do not depend on these extensions in newly written code. Support for them may be removed in future compiler versions.

## Code

The **Code** options determine the type and level of optimization for generation of object code.

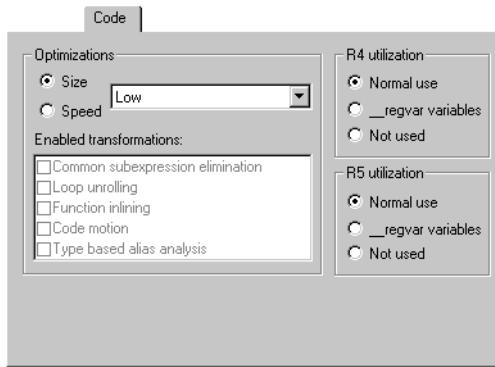


Figure 172: Compiler code options

## OPTIMIZATIONS

### Size and speed

The MSP430 IAR C/C++ Compiler supports two optimization models—size and speed—at different optimization levels.

Select the optimization model using either the **Size** or **Speed** radio button. Then choose the optimization level—None, Low, Medium, or High—from the drop-down list next to the radio buttons.

By default, a debug project will have a size optimization that is fully debuggable, while a release project will have a size optimization that generates an absolute minimum of code.

For a list of optimizations performed at each optimization level, see the *MSP430 IAR C/C++ Compiler Reference Guide*.

### Enabled transformations

The MSP430 IAR C/C++ Compiler supports the following types of transformations:

- Common sub-expression elimination
- Loop unrolling
- Function inlining
- Code motion

- Type-based alias analysis

In a *debug* project, the transformations are by default disabled. You can enable a transformation by selecting its check box. The compiler will then determine if this transformation is feasible.

In a *release* project, the transformations are by default enabled. You can disable a transformation by deselecting its check box.

For a brief description of the transformations that can be individually disabled, see the *MSP430 IAR C/C++ Compiler Reference Guide*.

## R4 UTILIZATION

This option controls how register R4 can be used. There are three possible settings:

- **Normal use.** This setting allows the compiler to use the register in generated code.
- **\_\_regvar variables.** When this setting is selected, the compiler uses the register for locating global register variables declared with the extended keyword `__regvar`.
- **Not used.** If you select this setting, R4 is locked and can be used for a special purpose by the application.

## R5 UTILIZATION

This option controls how register R5 can be used. There are three possible settings:

- **Normal use.** This setting allows the compiler to use the register in generated code.
- **\_\_regvar variables.** When this setting is selected, the compiler uses the register for locating global register variables declared with the extended keyword `__regvar`.
- **Not used.** If you select this setting, R5 is locked and can be used for a special purpose by the application.

## Output

The **Output** options determine the output format of the compiled file, including the level of debugging information in the object code.

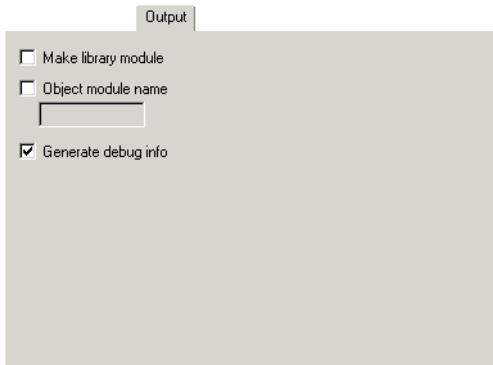


Figure 173: Compiler output options

### MAKE LIBRARY MODULE

By default the compiler generates *program* modules, which are always included during linking. Use this option to make a *library* module that will only be included if it is referenced in your application.

Select the **Make library module** option to make the object file be treated as a library module rather than as a program module.

For information about working with libraries, see the XLIB and XAR chapters in the *IAR Linker and Library Tools Reference Guide*, available from the **Help** menu.

### OBJECT MODULE NAME

Normally, the internal name of the object module is the name of the source file, without a directory name or extension. Use this option to set the object module name explicitly.

First select the **Object module name** check box, then type a name in the entry field.

This option is particularly useful when several modules have the same filename, because the resulting duplicate module name would normally cause a linker error; for example, when the source file is a temporary file generated by a preprocessor.

## GENERATE DEBUG INFO

This option causes the compiler to include additional information in the object modules that is required by C-SPY™ and other symbolic debuggers.

In a debug configuration, the **Generate debug info** option is specified 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.

## List

The **List** options determine whether a list file is produced, and the information included in the list file.

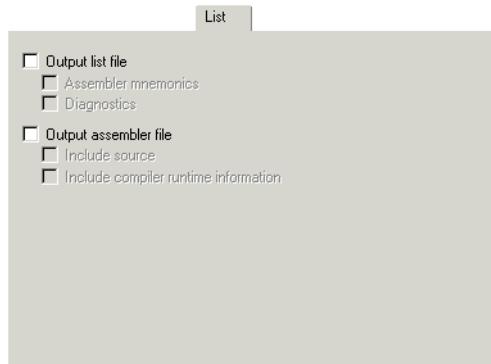


Figure 174: Compiler list file options

Normally, the compiler does not generate a list file. Select one of the following options to generate a list file or an assembler file:

Option	Description
Output list file	Generates a list file with the extension .lst
Assembler mnemonics	Includes assembler mnemonics in the list file
Diagnostics	Includes diagnostic information in the list file
Output assembler file	Generates an assembler file with the extension .s43
Include source	Includes source code in the assembler file
Include compiler runtime information	Includes compiler-generated information for runtime model attributes, call frame information, and frame size information.

Table 96: Compiler list file options

The file will be saved in the `List` directory, and its filename will consist of the source filename, plus the filename extension. You can open the output files directly from the **Output** folder which is available in the workspace window.

## Preprocessor

The **Preprocessor** options allow you to define symbols and include paths for use by the compiler.

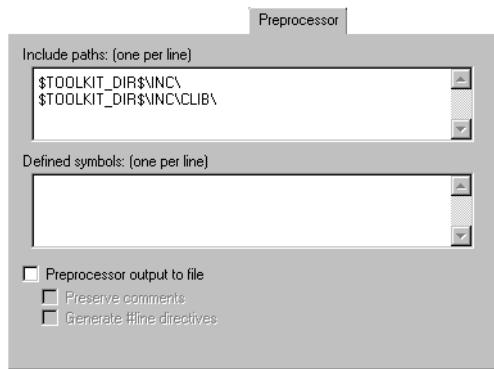


Figure 175: Compiler preprocessor options

### INCLUDE PATHS

The **Include paths** 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.

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 59, *Argument variables*, page 244.

### 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.

This 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.

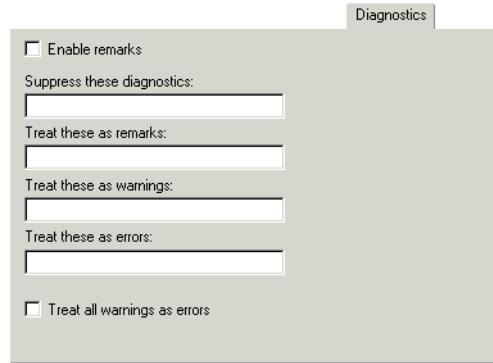


Figure 176: Compiler diagnostics options

### 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 Embedded 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.

# Assembler options

This chapter describes the assembler options available in the IAR Embedded Workbench™.

For information about how to set options, see [Setting options, page 89](#).

---

## Language

The **Language** options control the code generation of the assembler.

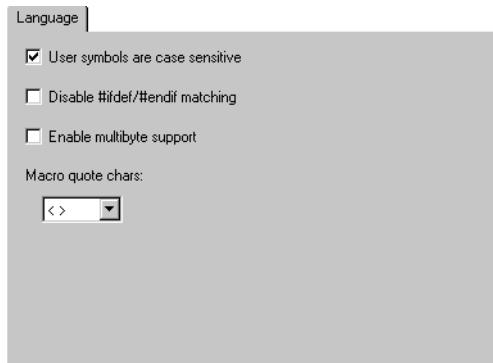


Figure 177: Assembler language options

### 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.

### DISABLE #IFDEF/#ENDIF MATCHING

Use this option to disable the matching between `#ifdef` and `#endif`.

### 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 C or C++ style comments, in string literals, and in character constants. They are transferred untouched to the generated code.

## MACRO QUOTE CHARACTERS

The **Macro quote chars** 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:

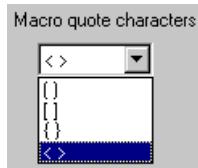


Figure 178: Choosing macro quote characters

---

## Output

The **Output** options allow you to generate information to be used by a debugger such as the IAR C-SPY™ Debugger.

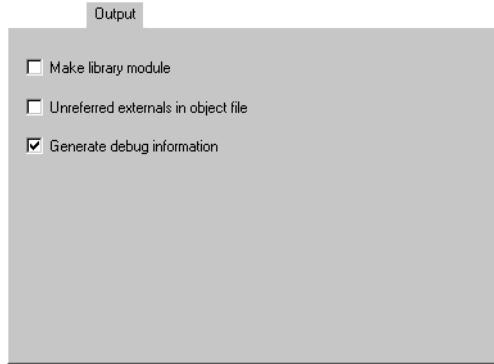


Figure 179: Assembler output options

## MAKE LIBRARY MODULE

By default, the assembler produces *program* modules, which are always included during linking. Use this option to make a *library* module that will only be included if it is referenced in your application.

Select the **Make library module** option to make the object file be treated as a library module rather than as a program module.

**Note:** If the `NAME` directive is used in the source code (to specify the name of the program module), the **Make library module** option is ignored. This means that the assembler produces a program module regardless of the **Make library module** option.

For information about working with libraries, see the XLIB and XAR chapters in the *IAR Linker and Library Tools Reference Guide*, available from the **Help** menu.

## UNREFERRED EXTERNALS IN OBJECT FILE

Use this option to keep external symbols included in the output, even if they are never referenced by your application.

## 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.

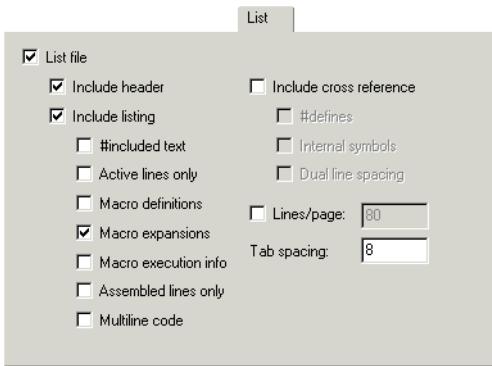


Figure 180: Assembler list file options

By default, the assembler does not generate a list file. Selecting **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 302, 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.
Active lines only	Includes active lines only.
Macro definitions	Includes macro definitions in the list file.
Macro expansions	Includes macro expansions in the list file.

Table 97: Assembler list file options

Option	Description
Macro execution info	Prints macro execution information on every call of a macro.
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 97: 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 *MSP430 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.

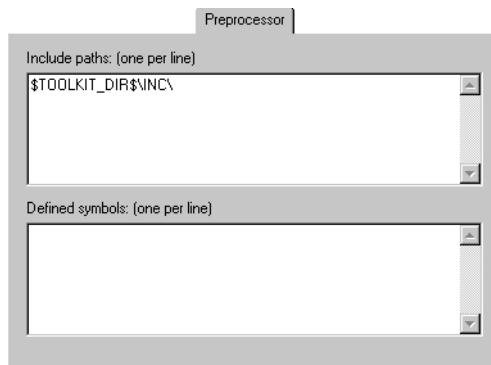


Figure 181: Assembler preprocessor options

## INCLUDE PATHS

The **Include paths** 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 59, *Argument variables*, page 244.

See the *MSP430 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 A430\_INC environment variable. We do not, however, recommend the use of environment variables in the IAR Embedded Workbench.

## 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.

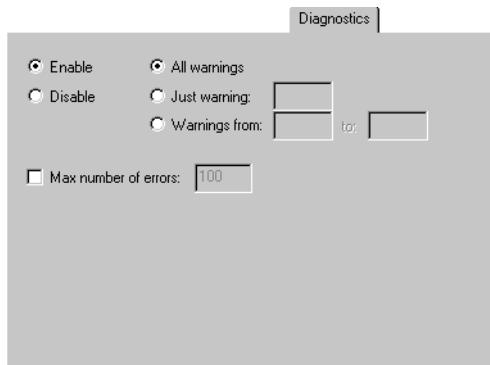


Figure 182: Assembler diagnostics options

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 *MSP430 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.



# Custom build options

This chapter describes the Custom Build options available in the IAR Embedded Workbench™.

For information about how to set options, see *Setting options*, page 89.

---

## Custom Tool Configuration

To set custom build options in the IAR Embedded Workbench, choose **Project>Options** to display the **Options** dialog box. Then select **Custom Build** in the **Category** list to display the **Custom Tool Configuration** page:

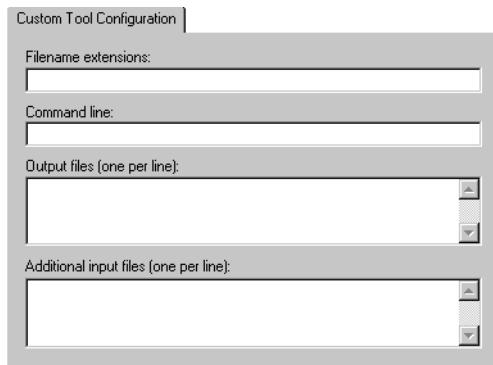


Figure 183: Custom tool options

In the **Filename extensions** text box, specify the filename extensions for the types of files that are to be processed by this custom tool, including the period. You can enter several filename extensions. Use commas, semicolons, or blank spaces as separators.

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 92.



# XLINK options

This chapter describes the XLINK options available in the IAR Embedded Workbench™.

For information about how to set options, see *Setting options, page 89*.

Note that the XLINK command line options that are used for defining segments in a linker command file are described in the *IAR Linker and Library Tools Reference Guide*.

---

## Output

The **Output** options are used for specifying the output format and the level of debugging information included in the output file.

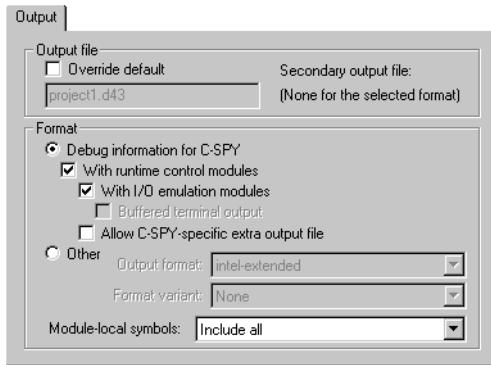


Figure 184: XLINK output file options

### OUTPUT FILE

Use **Output file** to specify the name of the XLINK 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. If you choose **Debug information for C-SPY**, the output file will have the filename extension `d43`.

**Note:** If you select a format that generates two output files, the filename extension that you specify will only affect the primary output file (first format).

## **Override default**

Use this option to specify a filename (including filename extension, if you want to) other than the default.

## **FORMAT**

The output options determine the format of the output file generated by the IAR XLINK Linker. The output file is used as input to either a debugger or as input for programming the target system. The IAR Systems proprietary output format is called UBROF, Universal Binary Relocatable Object Format.

The default output settings are:

- In a *debug* project, **Debug information for C-SPY, With runtime control modules**, and **With I/O emulation modules**
- In a *release* project, msp430-txt

**Note:** For debuggers other than C-SPY, check the user documentation supplied with that debugger for information about which format/variant should be used.

### **Debug information for C-SPY**

This option creates a UBROF output file, with a d43 filename extension, to be used with the IAR C-SPY Debugger.

#### **With runtime control modules**

This option produces the same output as the **Debug information for C-SPY** option, but also includes debugger support for handling program abort, exit, and assertions. Special C-SPY variants for the corresponding library functions are linked with your application. For more information about the debugger runtime interface, see the *MSP430 IAR C/C++ Compiler Reference Guide*.

#### **With I/O emulation modules**

This option produces the same output as the **Debug information for C-SPY** and **With runtime control modules** options, but also includes debugger support for I/O handling, which means that `stdin` and `stdout` are redirected to the Terminal I/O window, and that it is possible to access files on the host computer during debugging.

For more information about the debugger runtime interface, see the *MSP430 IAR C/C++ Compiler Reference Guide*.

## Buffered terminal output

During program execution in C-SPY, instead of instantly printing each new character to the C-SPY Terminal I/O window, this option will buffer the output. This option is useful when you use hardware debugger systems with a low communication speed.

## Allow C-SPY-specific extra output file

Use this option to enable the options available on the **Extra Output** page.

If you choose any of the options **With runtime control modules** or **With I/O emulation modules**, the generated output file will contain dummy implementations for certain library functions, such as `putchar`, and extra debug information required by C-SPY to handle those functions. In this case, the options available on the **Extra Output** page are disabled, which means you cannot generate an extra output file. The reason is that the extra output file would still contain the dummy functions, but would lack the required extra debug information, and would therefore normally be useless.

However, for *some* debugger solutions, two output files from the same build process are required—one with the required debug information, and one that you can burn to your hardware before debugging. This is useful when you want to debug code that is located in non-volatile memory. In this case, you must choose the **Allow C-SPY-specific extra output file** option to make it possible to generate an extra output file.

## Other

Use this option to generate output other than those generated by the option **Debug information for C-SPY**.

Use the **Output format** drop-down list to select the appropriate output. If applicable, use **Format variant** to select variants available for some of the output formats. The alternatives depend on the output format chosen.

When you specify the **Other>Output format** option as either **debug (ubrof)**, or **ubrof**, a UBROF output file with the filename extension `dbg` will be created. The generated output file will not contain debugging information for simulating facilities such as stop at program exit, long jump instructions, and terminal I/O. If you need support for these facilities during debugging, use the **Debug information for C-SPY**, **With runtime control modules**, and **With I/O emulation modules** options, respectively.

For more information, see the *IAR Linker and Library Tools Reference Guide*.

## Module-local symbols

Use this option to specify whether local (non-public) symbols in the input modules should be included or not by the linker. If suppressed, the local symbols will not appear in the listing cross-reference and they will not be passed on to the output file. By default, all symbols are included.

You can choose to ignore just the compiler-generated local symbols, such as jump or constant labels. Usually these are only of interest when debugging at assembler level.

**Note:** Local symbols are only included in files if they were compiled or assembled with the appropriate option to specify this.

---

## Extra Output

The **Extra Output** options are used for generating an extra output file and for specifying its format.

**Note:** If you have chosen any of the options **With runtime control modules** or **With I/O emulation modules** available on the **Output** page, you must also choose the option **Allow C-SPY-specific extra output file** to enable the **Extra Output** options.

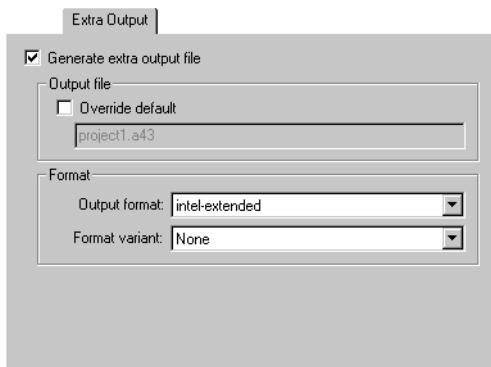


Figure 185: XLINK extra output file options

Use the **Generate extra output file** option to generate an extra output file from the build process.

Use the **Override default** option to override the default file name. If a name is not specified, the linker will use the project name and a filename extension which depends on the output format you choose.

Use the **Output format** drop-down list to select the appropriate output. If applicable, use **Format variant** to select variants available for some of the output formats. The alternatives depend on the output format you have chosen.

When you specify the **Output format** option as either **debug (ubrof)**, or **ubrof**, a UBROF output file with the filename extension `.dbg` will be created.

---

## #define

You can define symbols with the **#define** option.



Figure 186: XLINK defined symbols options

### DEFINE SYMBOL

Use **Define symbol** to define absolute symbols at link time. This is especially useful for configuration purposes.

Any number of symbols can be defined in a linker command 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.

XLINK will display an error message if you attempt to redefine an existing symbol.

## Diagnostics

The **Diagnostics** options determine the error and warning messages generated by the IAR XLINK Linker.

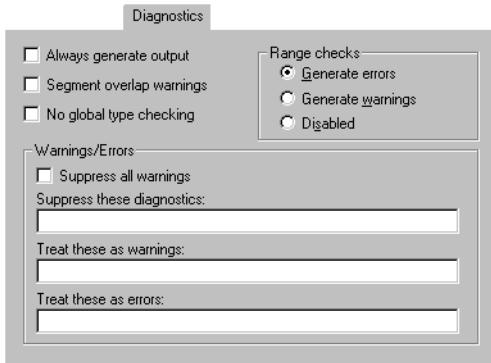


Figure 187: XLINK diagnostics options

### ALWAYS GENERATE OUTPUT

Use **Always generate output** to generate an output file even if a non-fatal error was encountered during the linking process, such as a missing global entry or a duplicate declaration. Normally, XLINK will not generate an output file if an error is encountered.

**Note:** XLINK always aborts on fatal errors, even when this option is used.

The **Always generate output** option allows missing entries to be patched in later in the absolute output image.

### SEGMENT OVERLAP WARNINGS

Use **Segment overlap warnings** to reduce segment overlap errors to warnings, making it possible to produce cross-reference maps, etc.

### NO GLOBAL TYPE CHECKING

Use **No global type checking** to disable type checking at link time. While a well-written application should not need this option, there may be occasions where it is helpful.

By default, XLINK performs link-time type checking between modules by comparing the external references to an entry with the PUBLIC entry (if the information exists in the object modules involved). A warning is generated if there are mismatches.

## RANGE CHECKS

Use **Range checks** to specify the address range check. The following table shows the range check options in the IAR Embedded Workbench:

IAR Embedded Workbench	Description
Generate errors	An error message is generated
Generate warnings	Range errors are treated as warnings
Disabled	Disables the address range checking

Table 98: XLINK range check options

If an address is relocated outside address range of the target CPU —code, external data, or internal data address—an error message is generated. This usually indicates an error in an assembler language module or in the segment placement.

## WARNINGS/ERRORS

By default, the IAR XLINK Linker generates a warning when it detects that something may be wrong, although the generated code might still be correct. The **Warnings/Errors** options allow you to suppress or enable all warnings, and to change the severity classification of errors and warnings.

Refer to the *IAR Linker and Library Tools Reference Guide* for information about the different warning and error messages.

Use the following options to control the generation of warning and error messages:

### Suppress all warnings

Use this option to suppress all warnings.

### Suppress these diagnostics

This option suppresses the output of diagnostics for the tags that you specify.

For example, to suppress the warnings w117 and w177, type `w117,w177`.

### Treat these as warnings

Use this option to specify errors that should be treated as warnings instead. For example, to make error 106 become treated as a warning, type `e106`.

### Treat these as errors

Use this option to specify warnings that should be treated as errors instead. For example, to make warning 26 become treated as an error, type `w26`.

## List

The **List** options determine the generation of an XLINK cross-reference listing.

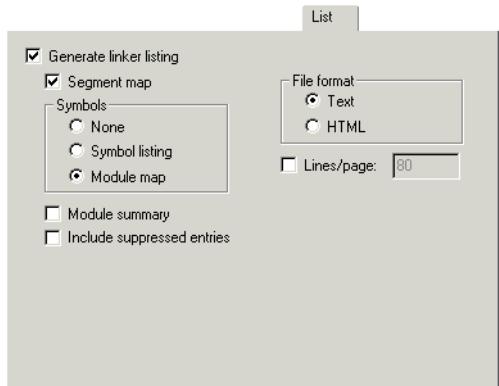


Figure 188: XLINK list file options

### GENERATE LINKER LISTING

Causes the linker to generate a listing and send it to the file *project.map*.

### Segment map

Use **Segment map** to include a segment map in the XLINK listing file. The segment map will contain a list of all the segments in dump order.

### Symbols

The following options are available:

Option	Description
None	Symbols will be excluded from the linker listing.
Symbol listing	An abbreviated list of every entry (global symbol) in every module. This entry map is useful for quickly finding the address of a routine or data element.
Module map	A list of all segments, local symbols, and entries (public symbols) for every module in the application.

Table 99: XLINK list file options

## Module summary

Use the **Module summary** option to generate a summary of the contributions to the total memory use from each module.

Only modules with a contribution to memory use are listed.

## Include suppressed entries

Use this option to include all segment parts in a linked module in the list file, not just the segment parts that were included in the output. This makes it possible to determine exactly which entries that were not needed.

## File format

The following options are available:

Option	Description
Text	Plain text file
HTML	HTML format, with hyperlinks

Table 100: XLINK list file format options

## Lines/page

The number of lines per page for the XLINK listings must be in the range 10 to 150.

## Config

With the **Config** options you can specify the path and name of the linker command file, override the default program entry, and specify the library search path.

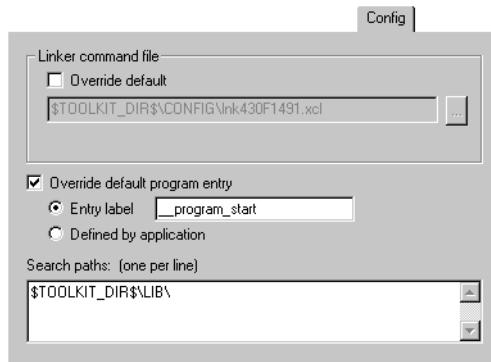


Figure 189: XLINK include files options

## LINKER COMMAND FILE

A default linker command file is selected automatically for the chosen **Target** settings in the **General Options** category. You can override this by selecting the **Override default** option, and then specifying an alternative file.

The argument variables `$TOOLKIT_DIRS` or `$PROJ_DIR$` can be used here too, to specify a project-specific or predefined linker command file.

## OVERRIDE DEFAULT PROGRAM ENTRY

By default, the program entry is the label `__program_start`. The linker will make sure that a module containing the program entry label is included, and that the segment part containing the label is not discarded.

The default program handling can be overridden by selecting **Override default program entry**.

Selecting the option **Entry label** will make it possible to specify a label other than `__program_start` to use for the program entry.

Selecting the option **Defined by application** will disable the use of a start label. The linker will, as always, include all program modules, and enough library modules to satisfy all symbol references, keeping all segment parts that are marked with the `root` attribute or that are referenced, directly or indirectly, from such a segment part.

## SEARCH PATHS

The **Search paths** option specifies the names of the directories which XLINK will search if it fails to find the object files to be linked in the current working directory. Add the full paths of any further directories that you want XLINK to search.

The paths required by the product are specified by default, depending on your choice of runtime library. If the box is left empty, XLINK searches for object files only in the current working directory.

Type the full file path of your `#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 59, *Argument variables*, page 244.

## Processing

With the **Processing** options you can specify details about how the code is generated.

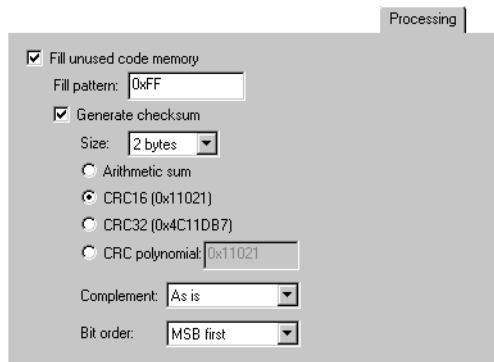


Figure 190: XLINK processing options

### FILL UNUSED CODE MEMORY

Use **Fill unused code memory** to fill all gaps between segment parts introduced by the linker with the value you enter. The linker can introduce gaps either because of alignment restriction, or at the end of ranges given in segment placement options.

The default behavior, when this option is not used, is that these gaps are not given a value in the output file.

#### Fill pattern

Use this option to specify the size, in hexadecimal notation, of the filler to be used in gaps between segment parts.

#### Generate checksum

Use **Generate checksum** to checksum all generated raw data bytes. This option can only be used if the **Fill unused code memory** option has been specified.

**Size** specifies the number of bytes in the checksum, which can be 1, 2, or 4.

One of the following algorithms can be used:

Algorithms	Description
Arithmetic sum	Simple arithmetic sum
CRC16	CRC16, generating polynomial 0x11021 (default)

Table 101: XLINK checksum algorithms

Algorithms	Description
CRC32	CRC32, generating polynomial 0x104C11DB7
Crc polynomial	CRC with a generating polynomial of the value you enter

Table 101: XLINK checksum algorithms (Continued)

You can also specify that the one's complement or two's complement should be used.

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.

The CRC checksum is calculated as if the following code was called for each bit in the input, starting with a CRC of 0:

```
unsigned long
crc(int bit, unsigned long oldcrc)
{
    unsigned long newcrc = (oldcrc << 1) ^ bit;
    if (oldcrc & 0x80000000)
        newcrc ^= POLY;
    return newcrc;
}
```

POLY is the generating polynomial. The checksum is the result of the final call to this routine. If the complement is specified, the checksum is the one's or two's complement of the result.

The linker places the checksum byte(s) at the `__checksum` label in the `CHECKSUM` segment. This segment must be placed using the segment placement options like any other segment.

For additional information about segment control, see the *IAR Linker and Library Tools Reference Guide*.

# Library builder options

This chapter describes the XAR Library Builder options available in the IAR Embedded Workbench™.

For information about how to set options, see [Setting options, page 89](#).

---

## Output

To make the library builder options available, you must first set up the project for building a library. Choose **Project>Options** to display the **Options** dialog box, and select the **General Options** category. On the **Output** page, select the **Library** option.

Now **Library Builder** appears as a category in the **Options** dialog box. As a result of the build process, the XAR Library Builder will create a library output file.

Before you create the library you can set **Library Builder** options.

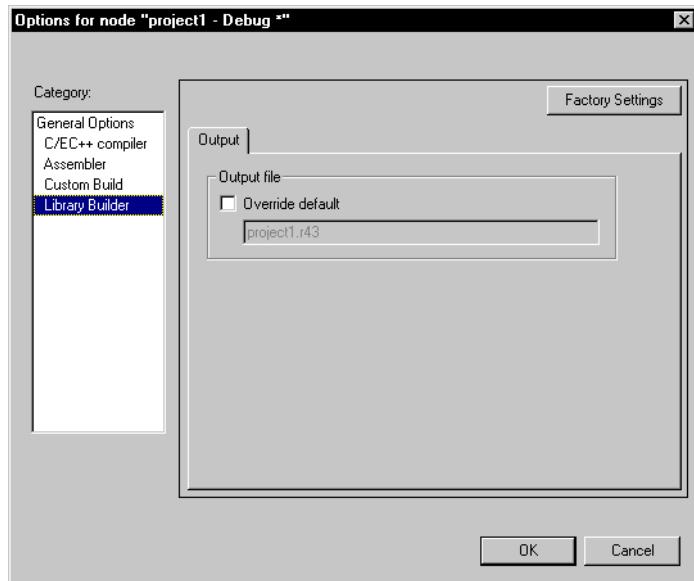


Figure 191: XAR Library Builder output options

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.

For information about how options can be set, see *Setting options*, page 89.

# Debugger options

This chapter describes the C-SPY Debugger options available in the IAR Embedded Workbench™.

For information about how to set options, see *Setting options, page 89*.

In addition, options specific to the C-SPY FET Debugger are described in the chapter *C-SPY FET driver-specific characteristics*.

---

## Setup

To set C-SPY Debugger options in the IAR Embedded Workbench, 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.

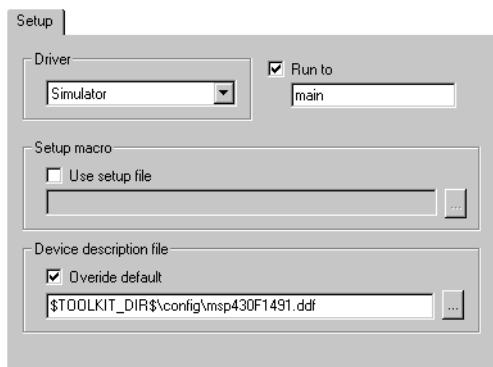


Figure 192: Generic C-SPY options

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	430sim.dll
FET Debugger	430fet.dll

Table 102: C-SPY driver options

Contact your distributor or IAR representative, or visit the IAR website at [www.iar.com](http://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 you leave the option deselected, the program counter will contain the regular hardware reset address at each reset.

## SETUP MACRO

To register the contents of a setup macro file in the C-SPY startup sequence, select **Use setup 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.

## DEVICE DESCRIPTION FILE

Use this option to override the default selection of device description file, a file that contains device-specific information.

For details about the device description file, see *Device description file*, page 109.

Device description files for each MSP430 device are provided in the directory `430\config` and have the filename extension `ddf`.

## Plugins

On the **Plugins** page you can specify C-SPY plugin modules you want to use 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 web site, for information about available modules.

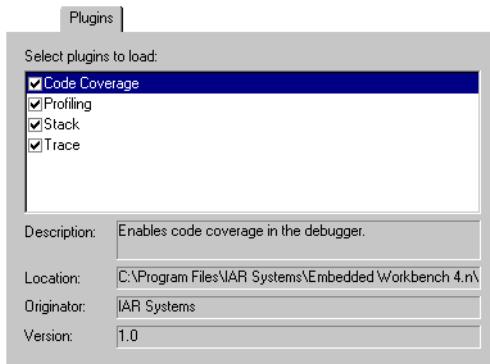


Figure 193: C-SPY plugin options

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.

For you to be able use a plugin module, it must also be supported by the C-SPY debugger system you are using.

The `common\plugins` directory is intended for generic plugin modules. The `430\plugins` directory is intended for target-specific plugin modules.



# 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 352.

## 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 117.

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 keeps its value and type through 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
<code>myvar = 3.5;</code>	<code>myvar</code> is now type <code>float</code> , <code>value 3.5.</code>
<code>myvar = (int*)i;</code>	<code>myvar</code> is now type pointer to <code>int</code> , and the value is the same as <code>i</code> .

Table 103: 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.

## 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 117.

### 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
}
```

## Printing messages

The \_\_message statement allows you to print messages while executing a macro function. The value of expression arguments or strings are printed to the Log window. Its definition is as follows:

```
__message argList;
```

where *argList* is a list of C-SPY expressions or strings separated by commas, 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.
```

### **Overriding default 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
```

### **Writing to a file**

There is also a statement `__fmessage` which is similar to `__message`, except that the first argument must be a file handle. The output is written to the designated file. For example:

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

## **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.

Table 104: C-SPY setup macros

Macro	Description
execUserSetup ()	Called once after the target application is downloaded. Implement this macro to set up the memory map, breakpoints, interrupts, register macro files, etc.
execUserReset ()	Called each time the reset command is issued. Implement this macro to set up and restore data.
execUserExit ()	Called once when the debug session ends. Implement this macro to save status data etc.

Table 104: C-SPY setup macros (Continued)

**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 interrupts*, page 159.

The reason for this is that the simulator saves breakpoint and 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
__cancelAllInterrupts	Cancels all ordered interrupts.
__cancelInterrupt	Cancels an interrupt.
__clearBreak	Clears a breakpoint.
__closeFile	Closes a file that was open by __openFile.
__disableInterrupts	Disables generation of interrupts.
__driverType	Verifies the driver type.
__enableInterrupts	Enables generation of interrupts.
__openFile	Opens a file for I/O operations.
__orderInterrupt	Generates an interrupt.
__readFile	Reads from the specified file.
__readFileByte	Reads one byte from the specified file.
__readMemoryByte	Reads one byte from the specified memory location.
__readMemory8	Reads one byte from the specified memory location.
__readMemory16	Reads two bytes from the specified memory location.

Table 105: Summary of system macros

Macro	Description
<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>__setCodeBreak</code>	Sets a code breakpoint.
<code>__setDataBreak</code>	Sets a data breakpoint.
<code>__setSimBreak</code>	Sets a simulation breakpoint.
<code>__strFind</code>	Searches a given string for the occurrence of another string.
<code>__subString</code>	Extracts a substring from another string.
<code>__toLower</code>	Returns a copy of the parameter string where all the characters have been converted to lower case.
<code>__toUpper</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>__writeMemoryByte</code>	Writes one byte to the specified memory location.
<code>__writeMemory8</code>	Writes one byte to the specified memory location.
<code>__writeMemory16</code>	Writes two bytes to the specified memory location.
<code>__writeMemory32</code>	Writes four bytes to the specified memory location.

Table 105: 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` `__cancelAllInterrupts()`

For details of this macro function, see *Description of interrupt system macros*, page 168.

---

`__cancelInterrupt` `__cancelInterrupt(interrupt_id)`

For details of this macro function, see *Description of interrupt system macros*, page 168.

---

```
__clearBreak __clearBreak(break_id)
```

### Parameter

*break\_id* The value returned by any of the set breakpoint macros

### Return value

int 0

### Description

Clears a user-defined breakpoint. For additional information, see *Defining breakpoints*, page 124.

---

```
__closeFile __closeFile(filehandle)
```

### Parameter

*filehandle* The macro variable used as filehandle by the \_\_openFile macro

### Return value

int 0.

### Description

Closes a file previously opened by \_\_openFile.

---

```
__disableInterrupts __disableInterrupts()
```

For details of this macro function, see *Description of interrupt system macros*, page 168.

---

```
__driverType __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  
"emul" corresponds to the FET Debugger driver

**Return value**

<b>Result</b>	<b>Value</b>
Successful	1
Unsuccessful	0

Table 106: `__driverType` return values**Description**

Checks to see if the current IAR C-SPY Debugger 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.

---

```
__enableInterrupts __enableInterrupts()
```

For details of this macro function, see *Description of interrupt system macros*, page 168.

---

```
__openFile __openFile(file, access)
```

**Parameters**

<code>file</code>	The filename as a string
<code>access</code>	The access type (string); one of the following: "r" ASCII read "w" ASCII write

**Return value**

<b>Result</b>	<b>Value</b>
Successful	The file handle
Unsuccessful	An invalid file handle, which tests as False

Table 107: `__openFile` return values

## Description

Opens a file for I/O operations. The default base directory of this macro is where the currently open project file (\*.ewp) is located. The argument to \_\_openFile can specify a location relative to this directory. In addition, you can use argument variables such as \$PROJ\_DIR\$ and \$TOOLKIT\_DIR\$ in the path argument; see Table 59, *Argument variables*, page 244.

### Example

```
--var filehandle;           /* The macro variable to contain */
                           /* the file handle */
filehandle = __openFile("Debug\\Exe\\test.tst", "r");
if (filehandle)
{
    /* successful opening */
}
```

---

```
--orderInterrupt __orderInterrupt(specification, activation_time,
                                repeat_interval, variance, infinite_hold_time, hold_time,
                                probability)
```

For details of this macro function, see *Description of interrupt system macros*, page 168.

---

```
____readFileByte ____readFileByte(file)
```

### Parameter

<i>file</i>	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
}
```

---

```
__readMemoryByte __readMemoryByte(address, zone)
```

### 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 129

### Return value

The macro returns the value from memory.

### Description

Reads one byte from a given memory location.

#### Example

```
__readMemoryByte(0x0108, "Memory");
```

---

```
__readMemory8 __readMemory8(address, zone)
```

### 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 129

### Return value

The macro returns the value from memory.

### Description

Reads one byte from a given memory location.

#### Example

```
__readMemory8(0x0108, "Memory");
```

---

```
__readMemory16  __readMemory16(address, zone)
```

### 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 129

### Return value

The macro returns the value from memory.

### Description

Reads two bytes from a given memory location.

#### Example

```
__readMemory16(0x0108, "Memory");
```

---

```
__readMemory32  __readMemory32(address, zone)
```

### 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 129

### Return value

The macro returns the value from memory.

### Description

Reads four bytes from a given memory location.

#### Example

```
__readMemory32(0x0108, "Memory");
```

---

```
__registerMacroFile __registerMacroFile(filename)
```

**Parameter**

*filename* A file containing the macros to be registered (string)

**Return value**

int 0

**Description**

Registers macros from a setup macro file. This function lets you register multiple macro files during C-SPY startup.

For additional information, see *Registering and executing using setup macros and setup files*, page 139.

**Example**

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

---

```
__resetFile __resetFile(filehandle)
```

**Parameter**

*filehandle* The macro variable used as filehandle by the \_\_openFile macro

**Return value**

int 0

**Description**

Rewinds the file previously opened by \_\_openFile.

---

```
--setCodeBreak  __setCodeBreak(location, count, condition, cond_type, action)
```

## Parameters

<i>location</i>	A string with a location description. This can be either: A source location on the form <code>{filename}.line.col</code> (for example <code>{D:\\src\\prog.c}.12.9</code> ) An absolute location on the form <code>zone:hexaddress</code> or simply <code>hexaddress</code> (for example <code>Memory:0x42</code> ) An expression whose value designates a location (for example <code>main</code> )
<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>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 108: `_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 assembler source:

```
_setCodeBreak("#main", 0, "1", "TRUE", "");
```

For additional information, see *Defining breakpoints*, page 124.

---

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

For details about this macro function, see *Using a macro to set a conditional breakpoint*, page 199.

---

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

## Parameters

<i>location</i>	A string with a location description. This can be either: 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
	An <i>absolute location</i> on the form <code>zone:hexaddress</code> or simply <code>hexaddress</code> (for example <code>Memory:0x42</code> )
	An expression whose value designates a location (for example <code>my_global_variable</code> ).
<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 109: `__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.

This macro is only applicable to the simulator version of C-SPY.

## Example

```
--var brk;
brk = __setDataBreak("Memory:0x4710", 3, "d>6", "TRUE",
                      "W", "ActionData()");
...
__clearBreak(brk);
```

For additional information, see *Defining breakpoints*, page 124.

---

<code>--setRangeBreak</code>	<code>--setRangeBreak(start_loc, end_loc, end_cond, type, access,                   action, action_when)</code>
------------------------------	---

For details about this macro function, see *Using a system macro to set a range breakpoint*, page 194.

---

<code>--setSimBreak</code>	<code>--setSimBreak(location, access, action)</code>
----------------------------	--

For details about this macro function, see *Using breakpoints*, page 156.

---

<code>--strFind</code>	<code>--strFind(string, pattern, position)</code>
------------------------	---

## Parameters

<i>string</i>	The 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**

```
--strFind("Compiler", "pile", 0) = 3
--strFind("Compiler", "foo", 0) = -1
```

---

`--subString` `--subString(string, position, length)`

**Parameters**

<i>string</i>	The string from which to extract a substring
<i>position</i>	The start position of the substring
<i>length</i>	The length of the substring

**Return value**

A substring extracted from the given string.

**Description**

This macro extracts a substring from another string.

**Example**

```
--subString("Compiler", 0, 2) = "Co"
--subString("Compiler", 3, 4) = "pile"
```

---

`--toLower` `--toLower(string)`

**Parameter**

*string* is any string.

**Return value**

The converted string.

**Description**

This macro returns a copy of the parameter string where all the characters have been converted to lower case.

**Example**

```
--toLower("IAR") = "iar"
--toLower("Mix42") = "mix42"
```

---

```
__ToUpper __ToUpper(string)
```

### Parameter

*string* is any string.

### Return value

The converted string.

### Description

This macro returns a copy of the parameter string where all the characters have been converted to upper case.

### Example

```
__ToUpper("string") = "STRING"
```

---

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

### Parameters

<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 *file*.

**Note:** The `__fmessage` statement can do the same thing. The `__writeFile` macro is provided for symmetry with `__readFile`.

---

```
__writeFileByte __writeFileByte(file, value)
```

### Parameters

<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 *file*.`__writeMemoryByte __writeMemoryByte(value, address, zone)`**Parameters**

<i>value</i>	The value to be written (integer)
<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 129

**Return value**

int 0

**Description**

Writes one byte to a given memory location.

**Example**`__writeMemoryByte(0x2F, 0x1F, "Memory");``__writeMemory8 __writeMemory8(value, address, zone)`**Parameters**

<i>value</i>	The value to be written (integer)
<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 129

**Return value**

int 0



## Description

Writes one byte to a given memory location.

### Example

```
__writeMemory8(0x2F, 0x8020, "Memory");
```

---

```
__writeMemory16 __writeMemory16(value, address, zone)
```

## Parameters

<i>value</i>	The value to be written (integer)
<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 129

## Return value

int 0

## Description

Writes two bytes to a given memory location.

### Example

```
__writeMemory16(0x2FF, 0x8020, "Memory");
```

---

```
__writeMemory32 __writeMemory32(value, address, zone)
```

## Parameters

<i>value</i>	The value to be written (integer)
<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 129

## Return value

int 0

## Description

Writes four bytes to a given memory location.

### Example

```
__writeMemory32(0x5555FFFF, 0x8020, "Memory");
```

# A

**Absolute location.** A specific memory address for an object specified in the source code, as opposed to the object being assigned a location by the IAR XLINK Linker.

**Absolute segments.** Segments that have fixed locations in memory before linking.

**Address expression.** An expression which has an address as its value.

**Application.** The program developed by the user of the IAR toolkit and which will be run as an embedded application on a target processor.

**Architecture.** A term used by computer designers to designate the structure of complex information-processing systems. It includes the kinds of instructions and data used, the memory organization and addressing, and the methods by which the system is implemented. The two main architecture types used in processor design are *Harvard* and *von Neumann*.

**Assembler directives.** The set of commands that control how the assembler operates.

**Assembler options.** Parameters you can specify to change the default behavior of the assembler.

**Assembler language.** A machine-specific set of mnemonics used to specify operations to the target processor and input or output registers or data areas. Assembler language might sometimes be preferred over C/Embedded C++ to save memory or to enhance the execution speed of the application.

**Auto variables.** The term refers to the fact that each time the function in which the variable is declared is called, a new instance of the variable is created automatically. This can be compared with the behavior of local variables in systems using static overlay, where a local variable only exists in one instance, even if the function is called recursively. Also called local variables. Compare *Register variables*.

# B

**Backtrace.** Information that allows the IAR C-SPY™ Debugger to show, without any runtime penalty, the complete stack of function calls wherever the program counter is, provided that the code comes from compiled C functions.

**Bank.** See *Memory bank*.

**Bank switching.** Switching between different sets of memory banks. This software technique is used to increase a computer's usable memory by allowing different pieces of memory to occupy the same address space.

**Banked code.** Code that is distributed over several banks of memory. Each function must reside in only one bank.

**Banked data.** Data that is distributed over several banks of memory. Each data object must fit inside one memory bank.

**Banked memory.** Has multiple storage locations for the same address. See also *Memory bank*.

**Bank-switching routines.** Code that selects a memory bank.

**Batch files.** A text file containing operating system commands which are executed by the command line interpreter. In Unix, this is called a "shell script" because it is the Unix shell which includes the command line interpreter. Batch files can be used as a simple way to combine existing commands into new commands.

**Bitfield.** A group of bits considered as a unit.

**Breakpoint.** 1. Code breakpoint. A point in a program that, when reached, triggers some special behavior useful to the process of debugging. Generally, breakpoints are used for stopping program execution or dumping the values of some or all of the program variables. Breakpoints can be part of the program itself, or they can be set by the programmer as part of an interactive session with a debugging tool for scrutinizing the program's execution.

2. Data breakpoint. A point in memory that, when accessed, triggers some special behavior useful to the process of debugging. Generally, data breakpoints are used to stop program execution when an address location is accessed either by a read operation or a write operation.

3. Immediate breakpoint. A point in memory that, when accessed, trigger some special behavior useful in the process of debugging. Immediate breakpoints are generally used for halting the program execution in the middle of a memory access instruction (before or after the actual memory access depending on the access type) while performing some user-specified action. The execution is then resumed. This feature is only available in the simulator version of C-SPY.

## C

**Calling convention.** A calling convention describes the way one function in a program calls another function. This includes how register parameters are handled, how the return value is returned, and which registers that will be preserved by the called function. The compiler handles this automatically for all C and C++ functions. All code written in assembler language must conform to the rules in the calling convention in order to be callable from C or C++, or to be able to call C and C++ functions. The C calling convention and the C++ calling conventions are not necessarily the same.

**Cheap.** As in *cheap memory access*. A cheap memory access either requires few cycles to perform, or few bytes of code to implement. A cheap memory access is said to have a low cost. See *Memory access cost*.

**Checksum.** A computed value which depends on the contents of a block of data and which is stored along with the data in order to detect corruption of the data. Compare *CRC*.

**Code banking.** See *Banked code*.

**Code model.** The code model controls how code is generated for an application. All object files of a system must be compiled using the same code model.

**Code pointers.** A code pointer is a function pointer. As many microcontrollers allow several different methods of calling a function, compilers for embedded systems usually provide the users with the ability to use all these methods.

Do not confuse code pointers with data pointers.

**Compilation unit.** See *Translation unit*.

**Compiler function directives.** The compiler function directives are generated by the compiler to pass information about functions and function calls to the IAR XLINK Linker. To view these directives, you must create an assembler list file. These directives are primarily intended for compilers that support static overlay, a feature which is useful in smaller microcontrollers.

**Compiler options.** Parameters you can specify to change the default behavior of the compiler.

**Cost.** See *Memory access cost*.

**CRC (cyclic redundancy checking).** A number derived from, and stored with, a block of data in order to detect corruption. A CRC is based on polynomials and is a more advanced way of detecting errors than a simple arithmetic checksum. Compare *Checksum*.

**C-SPY options.** Parameters you can specify to change the default behavior of the IAR C-SPY Debugger.

**Cstartup.** Code that sets up the system before the application starts executing.

**C-style preprocessor.** A preprocessor is either a stand-alone application or an integrated part of a compiler, that performs preprocessing of the input stream before actual compilation takes place. A C-style preprocessor follows the rules set up in the ANSI specification of the C language and implements commands like #define, #if, and #include, which are used to handle textual macro substitution, conditional compilation, and inclusion of other files.

## D

**Data banking.** See *Banked data*.

**Data pointers.** Many microcontrollers have different addressing modes in order to access different memory types or address spaces. Compilers for embedded systems usually have a set of different data pointer types so they can access the available memory efficiently.

**Data representation.** How different data types are laid out in memory and what value ranges they represent.

**Declaration.** A specification to the compiler that an object, a variable or function, exists. The object itself must be defined in exactly one translation unit (source file). An object must either be declared or defined before it is used. Normally an object that is used in many files is defined in one source file. A declaration is normally placed in a header file that is included by the files that use the object.

For example:

```
/* Variable "a" exists somewhere. Function
   "b" takes two int parameters and returns an
   int. */

extern int a;
int b(int, int);
```

**Definition.** The variable or function itself. Only one definition can exist for each variable or function in an application. See also *Tentative definition*.

For example:

```
int a;
int b(int x, int y)
{
    return x + y;
}
```

**Derivative.** One of two or more processor variants in a series or family of microprocessors or microcontrollers.

**Device description file.** A file used by the IAR C-SPY Debugger that contains various device-specific information such as I/O registers (SFR) definitions, interrupt vectors, and control register definitions.

**Device driver.** Software that provides a high-level programming interface to a particular peripheral device.

**Digital signal processor (DSP).** A device that is similar to a microprocessor, except that the internal CPU has been optimized for use in applications involving discrete-time signal processing. In addition to standard microprocessor instructions, digital signal processors usually support a set of complex instructions to perform common signal-processing computations quickly.

**Disassembly window.** A C-SPY window that shows the memory contents disassembled as machine instructions, interspersed with the corresponding C source code (if available).

**Dynamic initialization.** Variables in a program written in C are initialized during the initial phase of execution, before the main function is called. These variables are always initialized with a static value, which is determined either at compile-time or at link-time. This is called static initialization. In Embedded C++, variables might require initialization to be performed by executing code, for example, running the constructor of global objects, or performing dynamic memory allocation.

**Dynamic memory allocation.** There are two main strategies for storing variables: statically at link-time, or dynamically at runtime. Dynamic memory allocation is often performed from the heap and it is the size of the heap that determines how much memory that can be used for dynamic objects and variables. The advantage of dynamic memory allocation is that several variables or objects that are not active at the same time can be stored in the same memory, thus reducing the memory need of an application. See also *Heap memory*.

**Dynamic object.** An object that is allocated, created, destroyed, and released at runtime. Dynamic objects are almost always stored in memory that is dynamically allocated. Compare *Static objects*.

# E

**EEPROM.** Electrically Erasable, Programmable Read-Only Memory. A type of ROM that can be erased electronically, and then be re-programmed.

**EPROM.** Erasable, Programmable Read-Only Memory. A type of ROM that can be erased by exposing it to ultraviolet light, and then be re-programmed.

**Embedded C++.** A subset of the C++ programming language, which is intended for embedded systems programming. The fact that performance and portability are particularly important in embedded systems development was considered when defining the language.

**Embedded system.** A combination of hardware and software, designed for a specific purpose. Embedded systems are often part of a larger system or product.

**Emulator.** An emulator is a hardware device that performs emulation of one or more derivatives of a processor family. An emulator can often be used instead of the actual microcontroller and connects directly to the printed circuit board—where the microcontroller would have been connected—via a connecting device. An emulator always behaves exactly as the processor it emulates, and is used when debugging requires all systems actuators, or when debugging device drivers.

**Enumeration.** A type which includes in its definition an exhaustive list of possible values for variables of that type. Common examples include Boolean, which takes values from the list [true, false], and day-of-week which takes values [Sunday, Monday, Tuesday, Wednesday, Thursday, Friday, Saturday]. Enumerated types are a feature of typed languages, including C and Ada.

Characters, (fixed-size) integers, and even floating-point types might be (but are not usually) considered to be (large) enumerated types.

**Exceptions.** An exception is an interrupt initiated by the processor hardware, or hardware that is tightly coupled with the processor, for instance, a memory management unit (MMU). The exception signals a violation of the rules of the architecture (access to protected memory), or an extreme error condition (division by zero).

Do not confuse this use of the word exception with the term *exception* used in the C++ language (but not in Embedded C++!).

**Expensive.** As in *expensive memory access*. An expensive memory access either requires many cycles to perform, or many bytes of code to implement. An expensive memory access is said to have a high cost. See *Memory access cost*.

**Extended keywords.** Non-standard keywords in C and C++. These usually control the definition and declaration of objects (that is, data and functions). See also *Keywords*.

# F

**Format specifiers.** Used to specify the format of strings sent by library functions such as printf. In the following example, the function call contains one format string with one format specifier, %c, that prints the value of a as a single ASCII character:

```
printf("a = %c", a);
```

# G

**General options.** Parameters you can specify to change the default behavior of all tools that are included in the IAR Embedded Workbench IDE.

**Generic pointers.** Pointers that have the ability to point to all different memory types in for instance a microcontroller based on the Harvard architecture

# H

**Harvard architecture.** A microcontroller based on the Harvard architecture has separate data and instruction buses. This allows execution to occur in parallel. As an instruction is being fetched, the current instruction is executing on the data bus. Once the current instruction is complete, the next instruction is ready to go. This theoretically allows for much faster execution than a von Neumann architecture, but there is some added silicon complexity. Compare *von Neumann architecture*.

**Heap memory.** The heap is a pool of memory in a system that is reserved for dynamic memory allocation. An application can request parts of the heap for its own use; once memory has been allocated from the heap it remains valid until it is explicitly released back to the heap by the application. This type of memory is useful when the number of objects is not known until the application executes. Note that this type of memory is risky to use in systems with a limited amount of memory or systems that are expected to run for a very long time.

**Heap size.** Total size of memory that can be dynamically allocated.

**Host.** The computer that communicates with the target processor. The term is used to distinguish the computer on which the debugger is running from the microcontroller the embedded application you develop runs on.

# I

**IDE (integrated development environment).** A programming environment with all necessary tools integrated into one single application.

**Include file.** A text file which is included into a source file. This is often performed by the preprocessor.

**Inline assembler.** Assembler language code that is inserted directly between C statements.

**Inlining.** An optimization that replaces function calls with the body of the called function. This optimization increases the execution speed and can even reduce the size of the generated code.

**Instruction mnemonics.** A word or acronym used in assembler language to represent a machine instruction. Different processors have different instruction sets and therefore use a different set of mnemonics to represent them, such as, ADD, BR (branch), BLT (branch if less than), MOVE, LDR (load register).

**Interrupt vector.** A small piece of code that will be executed, or a pointer that points to code that will be executed when an interrupt occurs.

**Interrupt vector table.** A table containing interrupt vectors, indexed by interrupt type. This table contains the processor's mapping between interrupts and interrupt service routines and must be initialized by the programmer.

**Interrupts.** In embedded systems, the use of interrupts is a method of detecting external events immediately, for example a timer overflow or the pressing of a button.

Interrupts are asynchronous events that suspend normal processing and temporarily divert the flow of control through an “interrupt handler” routine. Interrupts can be caused by both hardware (I/O, timer, machine check) and software (supervisor, system call or trap instruction). Compare *Trap*.

**Intrinsic.** An adjective describing native compiler objects, properties, events, and methods.

**Intrinsic functions.** 1. Function calls that are directly expanded into specific sequences of machine code. 2. Functions called by the compiler for internal purposes (that is, floating point arithmetic etc.).

# K

**Key bindings.** Key shortcuts for menu commands used in the IAR Embedded Workbench IDE.

**Keywords.** A fixed set of symbols built into the syntax of a programming language. All keywords used in a language are reserved—they cannot be used as identifiers (in other words, user-defined objects such as variables or procedures). See also *Extended keywords*.

## L

**L-value.** A value that can be found on the left side of an assignment and thus be changed. This includes plain variables and de-referenced pointers. Expressions like  $(x + 10)$  cannot be assigned a new value and are therefore not L-values.

**Language extensions.** Target-specific extensions to the C language.

**Library.** See *Runtime library*.

**Linker command file.** A file used by the IAR XLINK Linker. It contains command line options which specify the locations where the memory segments can be placed, thereby assuring that your application fits on the target chip.

Because many of the chip-specific details are specified in the linker command file and not in the source code, the linker command file also helps to make the code portable.

In particular, the linker specifies the placement of segments, the stack size, and the heap size.

**Local variable.** See *Auto variables*.

**Location counter.** See *Program location counter*.

**Logical address.** See *Virtual address*.

## M

**MAC (Multiply and accumulate).** A special instruction, or on-chip device, that performs a multiplication together with an addition. This is very useful when performing signal processing where many filters and transforms have the form:

$$y_j = \sum_{i=0}^N c_i \cdot x_{i+j}$$

The accumulator of the MAC usually has a higher precision (more bits) than normal registers. See also *Digital signal processor*.

**Macro.** 1. Assembler macros are user-defined sets of assembler lines that can be expanded later in the source file by referring to the given macro name. Parameters will be substituted if referred to.

2. C macro. A text substitution mechanism used during preprocessing of source files. Macros are defined using the `#define` preprocessing directive. The replacement text of each macro is then substituted for any occurrences of the macro name in the rest of the translation unit.

3. C-SPY macros are programs that you can write to enhance the functionality of the IAR C-SPY Debugger. A typical application of C-SPY macros is to associate them with breakpoints; when such a breakpoint is hit, the macro is run and can for example be used to simulate peripheral devices, to evaluate complex conditions, or to output a trace.

The C-SPY macro language is like a simple dialect of C, but is less strict with types.

**Mailbox.** A mailbox in an RTOS is a point of communication between two or more tasks. One task can send messages to another task by placing the message in the mailbox of the other task. Mailboxes are also known as message queues or message ports.

**Memory access cost.** The cost of a memory access can be in clock cycles, or in the number of bytes of code needed to perform the access. A memory which requires large instructions or many instructions is said to have a higher access cost than a memory which can be accessed with few, or small instructions.

**Memory area.** A region of the memory.

**Memory bank.** The smallest unit of continuous memory in banked memory. One memory bank at a time is visible in a microcontroller's physical address space.

**Memory map.** A map of the different memory areas available to the microcontroller.

**Memory model.** Specifies the memory hierarchy and how much memory the system can handle. Your application must use only one memory model at a time, and the same model must be used by all user modules and all library modules. Most default behaviors originating from the selected memory model can be altered by the use of extended keywords and #pragma directives.

**Microcontroller.** A microprocessor on a single integrated circuit intended to operate as an embedded system. As well as a CPU, a microcontroller typically includes small amounts of RAM, PROM, timers, and I/O ports.

**Microprocessor.** A CPU contained on one (or a small number of) integrated circuits. A single-chip microprocessor can include other components such as memory, memory management, caches, floating-point unit, I/O ports and timers. Such devices are also known as microcontrollers.

**Module.** The basic unit of linking. A module contains definitions for symbols (exports) and references to external symbols (imports). When compiling C/C++, each translation unit produces one module. In assembler, each source file can produce more than one module.

## N

**Nested interrupts.** A system where an interrupt can be interrupted by another interrupt is said to have nested interrupts.

**Non-banked memory.** Has a single storage location for each memory address in a microcontroller's physical address space.

**Non-initialized memory.** Memory that can contain any value at reset, or in the case of a soft reset, can remember the value it had before the reset.

**Non-volatile storage.** Memory devices such as battery-backed RAM, ROM, magnetic tape and magnetic disks that can retain data when electric power is shut off. Compare *Volatile storage*.

**NOP.** No operation. This is an instruction that does not perform anything, but is used to create a delay. In pipelined architectures, the NOP instruction can be used for synchronizing the pipeline. See also *Pipeline*.

## O

**Operator.** A symbol used as a function, with infix syntax if it has two arguments (+, for example) or prefix syntax if it has only one (for instance, bitwise negation, ~). Many languages use operators for built-in functions such as arithmetic and logic.

**Operator precedence.** Each operator has a precedence number assigned to it that determines the order in which the operator and its operands are evaluated. The highest precedence operators are evaluated first. Use parentheses to group operators and operands to control the order in which the expressions are evaluated.

## P

**Parameter passing.** See *Calling convention*.

**Peripheral.** A hardware component other than the processor, for example memory or an I/O device.

**Pipeline.** A structure that consists of a sequence of stages through which a computation flows. New operations can be initiated at the start of the pipeline even though other operations are already in progress through the pipeline.

**Pointer.** An object that contains an address to another object of a specified type.

**pragma directive.** During compilation of a C/C++ program, pragma directives cause the compiler to behave in an implementation-defined manner. This can include, for example, producing output on the console, changing the declaration of a subsequent object, changing the optimization level, or enabling/disabling language extensions.

**Pre-emptive multitasking.** An RTOS task is allowed to run until a higher priority process is activated. The higher priority task might become active as the result of an interrupt. The term preemptive indicates that although a task is allotted to run a given length of time (a timeslice), it might lose the processor at any time. Each time an interrupt occurs, the task scheduler looks for the highest priority task that is active and switches to that task. If the located task is different from the task that was executing before the interrupt, the previous task is suspended at the point of interruption.

Compare *Round Robin*.

**Preprocessing directives.** A set of directives that are executed before the parsing of the actual code is started.

**Preprocessor.** See *C-style preprocessor*.

**Processor variant.** The different chip setups that the compiler supports. See *Derivative*.

**Program counter (PC).** A special processor register that is used to address instructions. Compare *Program location counter*.

**Program location counter (PLC).** Used in the IAR Assembler to denote the code address of the current instruction. The PLC is represented by a special symbol (typically \$) that can be used in arithmetic expressions. Also called simply location counter (LC).

**PROM.** Programmable Read-Only Memory. A type of ROM that can be programmed only once.

**Project.** The user application development project.

**Project options.** General options that apply to an entire project, for example the target processor that the application will run on.

## Q

**Qualifiers.** See *Type qualifiers*.

## R

**R-value.** A value that can be found on the right side of an assignment. This is just a plain value. See also *L-Value*.

**Real-time operating system (RTOS).** An operating system which guarantees the latency between an interrupt being triggered and the interrupt handler starting, as well as how tasks are scheduled. An RTOS is typically much smaller than a normal desktop operating system. Compare *Real-time system*.

**Real-time system.** A computer system whose processes are time-sensitive. Compare *Real-time operating system*.

**Register constant.** A register constant is a value that is loaded into a dedicated processor register when the system is initialized. The compiler can then generate code that assumes that the constants are present in the dedicated registers.

**Register.** A small on-chip memory unit, usually just one or a few bytes in size, which is particularly efficient to access and therefore often reserved to function as a temporary storage area during program execution.

**Register locking.** Register locking means that the compiler can be instructed that some processor registers shall not be used during normal code generation. This is useful in a number of situations. For example, some parts of a system might be written in assembler language to gain speed. These parts might be given dedicated processor registers. Or the register might be used by an operating system, or by other third-party software.

**Register variables.** Typically, register variables are local variables that have been placed in registers instead of on the (stack) frame of the function. Register variables are much more efficient than other variables because they do not require memory accesses, so the compiler can use shorter/faster instructions when working with them. See also *Auto variables*.

**Relocatable segments.** Segments that have no fixed location in memory before linking.

**Reset.** A reset is a restart from the initial state of a system. A reset can originate from hardware (hard reset), or from software (soft reset). A hard reset can usually not be distinguished from the power-on condition, which a soft reset can be.

**ROM-monitor.** A piece of embedded software that has been designed specifically for use as a debugging tool. It resides in the ROM of the evaluation board chip and communicates with a debugger via a serial port or network connection. The ROM-monitor provides a set of primitive commands to view and modify memory locations and registers, create and remove breakpoints, and execute your application. The debugger combines these primitives to fulfill higher-level requests like program download and single-step.

**Round Robin.** Task scheduling in an operating system, where all tasks have the same priority level and are executed in turn, one after the other. Compare *Preemptive multitasking*.

**RTOS.** See *Real-time operating system*.

**Runtime library.** A collection of useful routines, stored as an object file, that can be linked into any application.

**Runtime model attributes.** A mechanism that is designed to prevent modules that are not compatible to be linked into an application. A runtime attribute is a pair constituted of a named key and its corresponding value.

Two modules can only be linked together if they have the same value for each key that they both define.

## S

**Saturated mathematics.** Most, if not all, C and C++ implementations use mod- $2^N$  2-complement-based mathematics where an overflow wraps the value in the value domain, that is,  $(127 + 1) = -128$ . Saturated mathematics, on the other hand, does *not* allow wrapping in the value domain, for instance,  $(127 + 1) = 127$ , if 127 is the upper limit. Saturated mathematics is often used in signal processing, where an overflow condition would have been fatal if value wrapping had been allowed.

**Scheduler.** The part of an RTOS that performs task-switching. It is also responsible for selecting which task that should be allowed to run. There are many different scheduling algorithms, but most of them are either based on static scheduling (performed at compile-time), or on dynamic scheduling (where the actual choice of which task to run next is taken at runtime, depending on the state of the system at the time of the task-switch). Most real-time systems use static scheduling, because it makes it possible to prove that the system will not violate the real-time requirements.

**Scope.** The section of an application where a function or a variable can be referenced by name. The scope of an item can be limited to file, function, or block.

**Segment.** A chunk of data or code that should be mapped to a physical location in memory. The segment can either be placed in RAM (read-and-writeable memory) or in ROM (read-only memory).

**Segment map.** A set of segments and their locations.

**Semaphore.** A semaphore is a type of flag that is used for guaranteeing exclusive access to resources. The resource can be a hardware port, a configuration memory, or a set of variables. If several different tasks have to access the same resource, the parts of the code (the critical sections) that access the resource have to be made exclusive for every task. This is done by obtaining the semaphore that protects that resource, thus blocking all other tasks from it. If another task wishes to use the resource, it also has to obtain the semaphore. If the semaphore is already in use, the second task has to wait until the semaphore is released. After the semaphore is released, the second task is allowed to execute and can obtain the semaphore for its own exclusive access.

**Severity level.** The level of seriousness of the diagnostic response from the assembler, compiler, or debugger, when it notices that something is wrong. Typical severity levels are remarks, warnings, errors, and fatal errors. A remark just points to a possible problem, while a fatal error means that the programming tool exits without finishing.

**Short addressing.** Many microcontrollers have special addressing modes for efficient access to internal RAM and memory mapped I/O. Short addressing is therefore provided as an extended feature by many compilers for embedded systems. See also *Data pointers*.

**Side effect.** An expression in C or C++ is said to have a side-effect if it changes the state of the system. Examples are assignments to a variable, or using a variable with the post-increment operator. The C and C++ standards state that a variable that is subject to a side-effect should not be used more than once in an expression. As an example, this statement violates that rule:

```
*d++ = *d;
```

**Signal.** Signals provide event-based communication between tasks. A task can wait for one or more signals from other tasks. Once a task receives a signal it waits for, execution continues. A task in an RTOS that waits for a signal does not use any processing time, which allows other tasks to execute.

**Simulator.** A debugging tool that runs on the host and behaves as similar to the target processor as possible. A simulator is used to debug the application when the hardware is unavailable, or not needed for proper debugging. A simulator is usually not connected to any physical peripheral devices. A simulated processor is often slower, or even much slower, than the real hardware.

**Single stepping.** Executing one instruction or one C statement at a time in the debugger.

**Skeleton code.** An incomplete code framework that allows the user to specialize the code.

**Special function register (SFR).** A register that is used to read and write to the hardware components of the microcontroller.

**Stack frames.** Data structures containing data objects as preserved registers, local variables, and other data objects that need to be stored temporary for a particular scope (usually a function).

Earlier compilers usually had a fixed size and layout on a stack frame throughout a complete function, while modern compilers might have a very dynamic layout and size that can change anywhere and anytime in a function.

**Stack segments.** The segment or segments that reserve space for the stack(s). Most processors use the same stack for calls and parameters, but some have separate stacks.

**Statically allocated memory.** This kind of memory is allocated once and for all at link-time, and remains valid all through the execution of the application. Variables that are either global or declared static are allocated this way.

**Static objects.** An object whose memory is allocated at link-time and is created during system startup (or at first use). Compare *Dynamic objects*.

**Static overlay.** Instead of using a dynamic allocation scheme for parameters and auto variables, the linker allocates space for parameters and auto variables at link time. This generates a worst-case scenario of stack usage, but might be preferable for small chips with expensive stack access or no stack access at all.

**Structure value.** A collecting names for structs and unions. A struct is a collection of data object placed sequentially in memory (possibly with pad bytes between them). A union is a collection of data sharing the same memory location.

**Symbol.** A name that represents a register, an absolute value, or a memory address (relative or absolute).

**Symbolic location.** A location that uses a symbolic name because the exact address is unknown.

## T

**Target.** 1. An architecture. 2. A piece of hardware. The particular embedded system you are developing the application for. The term is usually used to distinguish the system from the host system.

**Task (thread).** A task is an execution thread in a system. Systems that contain many tasks that execute in parallel are called multitasking systems. Because a processor only executes one instruction stream at the time, most systems implement some sort of task-switch mechanism (often called context switch) so that all tasks get their share of processing time. The process of determining which task that should be allowed to run next is called scheduling. Two common scheduling methods are preemptive multitasking and Round Robin.

**Tentative definition.** A variable that can be defined in multiple files, provided that the definition is identical and that it is an absolute variable.

**Terminal I/O.** A simulated terminal window in the IAR C-SPY Debugger.

**Timeslice.** The (longest) time an RTOS allows a task to run without running the task-scheduling algorithm. It is possible that a task will be allowed to execute during several consecutive timeslices before being switched out. It is also possible that a task will not be allowed to use its entire time slice, for example if, in a preemptive system, a higher priority task is activated by an interrupt.

**Timer.** A peripheral that counts independent of the program execution.

**Translation unit.** A source file together with all the header files and source files included via the preprocessor directive #include, with the exception of the lines skipped by conditional preprocessor directives such as #if and #ifdef.

**Trap.** A trap is an interrupt initiated by inserting a special instruction into the instruction stream. Many systems use traps to call operating system functions. Another name for trap is software interrupt.

**Type qualifiers.** In standard C/C++, const or volatile. IAR compilers usually add target-specific type qualifiers for memory and other type attributes.

## U

**UBROF (Universal Binary Relocatable Object Format).** File format produced by the IAR programming tools.

## V

**Virtual address (logical address).** An address that needs to be translated by the compiler, linker or the runtime system into a physical memory address before it is used. The virtual address is the address seen by the application, which can be different from the address seen by other parts of the system.

**Virtual space.** An IAR Embedded Workbench Editor feature which allows you to place the insertion point outside of the area where there are actual characters.

**Volatile storage.** Data stored in a volatile storage device is not retained when the power to the device is turned off. In order to preserve data during a power-down cycle, you should store it in non-volatile storage. This should not be confused with the C keyword `volatile`. Compare *Non-volatile storage*.

**von Neumann architecture.** A computer architecture where both instructions and data are transferred over a common data channel. Compare *Harvard architecture*.

## W

**Watchpoints.** Watchpoints keep track of the values of C variables or expressions in the C-SPY Watch window as the application is being executed.

## X

**XAR options.** The set of commands that control how the IAR XAR Library Builder™ operates.

**XLIB options.** The set of commands that control how the IAR XLIB Librarian™ operates.

**XLINK options.** Parameters you can specify to change the default behavior of the IAR XLINK Linker.

## Z

**Zero-overhead loop.** A loop in which the loop condition, including branching back to the beginning of the loop, does not take any time at all. This is usually implemented as a special hardware feature of the processor and is not available in all architectures.

**Zone.** Different processors have widely differing memory architectures. *Zone* is the term C-SPY uses for a named memory area. For example, on processors with separately addressable code and data memory there would be at least two zones. A processor with an intricate banked memory scheme might have several zones.

# A

absolute location, definition of ..... 367  
 absolute segments, definition of ..... 367  
 Access Type (Breakpoints dialog) ..... 240  
 Action (Breakpoints dialog) ..... 241  
 address expression, definition of ..... 367  
 address range check, specifying in XLINK ..... 335  
 Allow C-SPY-specific output file (XLINK option) ..... 331  
 Always generate output (XLINK option) ..... 334  
 application  
     built outside the IDE ..... 108  
     definition of ..... 367  
     hardware-dependent aspects ..... 81  
     testing ..... 91  
 architecture, definition of ..... 367  
 argument variables ..... 264  
     in #include file paths ..... 316, 324, 338  
     summary ..... 244  
 asm (filename extension) ..... 18  
 assembler  
     documentation ..... 20  
     on the Help menu ..... 268  
     features ..... 11  
 assembler directives ..... 70  
     definition of ..... 367  
 assembler files  
     generating from compiler ..... 315  
 assembler language, definition of ..... 367  
 assembler list files  
     compiler runtime information, including ..... 315  
     conditional information, specifying ..... 322  
     cross-references, generating ..... 323  
     format ..... 51  
     generating ..... 322  
     header, including ..... 322  
     lines per page, specifying ..... 323  
     tab spacing, specifying ..... 323  
 Assembler mnemonics (compiler option) ..... 315

Assembler only project (target option) ..... 302  
 assembler options ..... 319  
     Cross-reference ..... 323  
     Defined symbols ..... 324  
     definition of ..... 367  
     Diagnostics ..... 325  
     Disable #ifdef/#endif matching ..... 319  
     Enable multibyte support ..... 319  
     Generate debug information ..... 321  
     Include header ..... 322  
     Include listing ..... 322  
     Include paths ..... 324  
     Language ..... 319  
     Lines/page ..... 323  
     List ..... 322  
     Macro quote characters ..... 320  
     Make library module ..... 321  
     Output ..... 320  
     Preprocessor ..... 323  
     Tab spacing ..... 323  
     Unreferred externals in object file ..... 321  
     User symbols are case sensitive ..... 319  
 assembler output, including debug information ..... 320  
 assembler preprocessor ..... 323  
 Assembler Reference Guide (Help menu) ..... 268  
 assembler symbols  
     defining ..... 324  
     using in C-SPY expressions ..... 118  
 assumptions, programming experience ..... xxix  
 Auto indent (editor option) ..... 257  
 auto variables, definition of ..... 367  
 Auto window ..... 282  
     context menu ..... 282  
 Automatic (compiler option) ..... 310  
 Autostep settings dialog box (Debug menu) ..... 296  
 a43 (filename extension) ..... 18

# B

backtrace information	
definition of	367
generated by compiler	115
bank switching, definition of	367
banked code, definition of	367
banked data, definition of	367
banked memory, definition of	367
bank-switching routines, definition of	367
Batch Build Configuration dialog box (Project menu)	250
Batch Build dialog box (Project menu)	249
batch files	
definition of	367
specifying in Embedded Workbench	80, 265
bin, common (subdirectory)	17
bin, 430 (subdirectory)	16
bitfield, definition of	367
blocks, in C-SPY macros	349
bookmarks	
adding	96
showing in editor	257
brackets, matching (in editor)	97
Break At Location Type (Breakpoints dialog)	239
Break (button)	273
breakpoint condition, example	125–126
Breakpoint Usage dialog box (Simulator menu)	154
breakpoints	114
code, example	359
conditional	
example	64
in FET Debugger	196
using	183
connecting a C-SPY macro	141
C-SPY FET specifics	179
data	239
example	195, 201, 361
definition of	367
immediate, example	64

in the simulator	156
range, using in FET	192
setting in memory window	124
settings	237, 247
system, description of	123
toggling	124
using system macros	126
using the dialog box	125
Breakpoints dialog box (Edit menu)	237
Buffered terminal output (XLINK option)	331
build configuration, definition of	83
build messages, option	256
Build window (View menu)	227
building	89, 91
from the command line	92
options	259

# C

C compiler. <i>See</i> compiler	
C function information, in C-SPY	115
C symbols, using in C-SPY expressions	117
C variables, using in C-SPY expressions	117
c (filename extension)	18
Call stack information	115
Call Stack window	115, 283
context menu	284
example	63
calling convention, definition of	368
_cancelAllInterrupts (C-SPY system macro)	352
_cancelInterrupt (C-SPY system macro)	168, 352
Category, in Options dialog box	90, 248
cfg (filename extension)	18, 258
characters, in assembler macro quotes	320
cheap memory access, definition of	368
checksum	
definition of	368
generating in XLINK	339
_clearBreak (C-SPY system macro)	353

CLIB.....	10
documentation .....	21
Close Workspace (File menu).....	231
_closeFile (C-SPY system macro) .....	353
Code	
Compiler options .....	312
code	
banked, definition of .....	367
skeleton, definition of .....	376
testing .....	91
Code Coverage	
commands .....	286
context menu .....	287
using .....	146
viewing the figures.....	147
window .....	286
code generation	
assembler .....	319
compiler.....	10
code memory, filling unused.....	339
code model, definition of .....	368
Code page (compiler options).....	312
code pointers, definition of .....	368
command line options	
specifying in Embedded Workbench .....	80, 265
Common Fonts (IDE Options dialog box) .....	252
common (directory) .....	17
Compact math libraries (general option).....	305
compiler	
command line version .....	4, 75
documentation .....	11, 20
features .....	9
compiler diagnostics.....	315
suppressing .....	318
compiler function directives, definition of .....	368
compiler list files	
assembler mnemonics, including .....	315
example .....	33
generating .....	315
source code, including .....	315
compiler options.....	309
definition of .....	368
factory settings.....	90
Assembler mnemonics.....	315
Automatic .....	310
Defined symbols .....	316
Diagnostics .....	317
Diagnostics (in list file) .....	315
Disable language extensions .....	310
Embedded C++ .....	310
Enable multibyte support.....	311
Enable remarks .....	317
Extended Embedded C++ syntax.....	310
Generate debug info.....	315
Include compiler runtime information .....	315
Include paths .....	316
Include source .....	315
Language .....	309
List .....	315
Make library module .....	314
Object module name .....	314
Output .....	314
Output assembler file .....	315
Output list file .....	315
Plain ‘char’ is.....	311
Preprocessor .....	316
Preprocessor output to file .....	317
Relaxed ISO/ANSI .....	310
Require prototypes.....	310
R4 utilization .....	313
R5 utilization .....	313
Strict ISO/ANSI.....	310
Suppress these diagnostics .....	318
Treat all warnings as errors .....	318
Treat these as errors .....	318
Treat these as remarks .....	318
Treat these as warnings .....	318

compiler output	
debug information, including	315
module name	314
compiler preprocessor	316
Compiler Reference Guide (Help menu)	268
compiler runtime information	
including in assembler list file	315
compiler symbols, defining	316
conditional breakpoints	
example	64
Conditional breakpoints dialog box (Edit menu)	196
conditional statements, in C-SPY macros	348
Conditions (Breakpoints dialog)	241
Config options (XLINK)	337
Configuration file (general option)	305
Configurations for project dialog box (Project menu)	245
Configure Tools (Tools menu)	263
config, common (subdirectory)	17
config, 430 (subdirectory)	16
context menus	
Build window	227
Call Stack window	284
Debug Log window	230
Disassembly window	275
Editor window	223
Find in Files window	228
Memory window	277
Messages window	227–230
Tool Output window	229
Trace	291
Watch window	280
Workspace window	221
conventions, typographic	xxxiii
cost. <i>See</i> memory access cost	
cpp (filename extension)	18
CPU variant, definition of	369
CRC, definition of	368
Create New Project dialog box (Project menu)	247
Cross-reference (assembler option)	323
cross-references, in map files	36
Cstartup, definition of	368
current position, in C-SPY Disassembly window	274
cursor, in C-SPY Disassembly window	274
\$CUR_DIR\$ (argument variable)	244
\$CUR_LINE\$ (argument variable)	244
Custom Build options	
Custom Tool Configuration	327
custom build, using	93
Custom Keyword File (editor option)	257
Custom Tool Configuration (Custom Build options)	327
C++	
enabling in compiler	310
tutorial	53
C-SPY	105
characteristics	
FET Debugger	173
Simulator	153
debugger systems	8
IDE reference information	271
overview	5
starting the debugger	108
C-SPY expressions	117
evaluating	120
in C-SPY macros	348
Quick Watch, using	120
Tooltip watch, using	120
C-SPY macros	135, 347
blocks	349
conditional statements	348
C-SPY expressions	348
definition of the system	135
dialog box	296
dialog box, using	138
examples	
checking latest value	136
checking status of register	140
checking the status of WDT	140
creating a log macro	141

execUserExit() . . . . .	351
execUserSetup()	
example . . . . .	61, 67
executing . . . . .	137
connecting to a breakpoint . . . . .	141
using Quick Watch . . . . .	140
using setup macro and setup file . . . . .	139
functions . . . . .	118, 347
loop statements . . . . .	349
macro statements . . . . .	348
printing messages . . . . .	349
setup macro file	
definition of . . . . .	137
executing . . . . .	139
setup macro function	
definition of . . . . .	137
execUserPreload() . . . . .	350
execUserReset() . . . . .	351
execUserSetup() . . . . .	351
setup macro functon	
summary . . . . .	350
using . . . . .	135
variables . . . . .	118, 348
<u>_cancelAllInterrupts</u> (system macro) . . . . .	168
<u>_clearBreak</u> (system macro) . . . . .	353
<u>_closeFile</u> (system macro) . . . . .	353
<u>_disableInterrupts</u> (system macro) . . . . .	169
<u>_driverType</u> (system macro) . . . . .	354
<u>_enableInterrupts</u> (system macro) . . . . .	169
<u>_openFile</u> (system macro) . . . . .	354
<u>_orderInterrupt</u> (system macro) . . . . .	169
<u>_readFileByte</u> (system macro) . . . . .	355
<u>_readMemoryByte</u> (system macro) . . . . .	356
<u>_registerMacroFile</u> (system macro) . . . . .	358
<u>_resetFile</u> (system macro) . . . . .	358
<u>_setCodeBreak</u> (system macro) . . . . .	359
<u>_setConditionalBreak</u> (system macro) . . . . .	199
<u>_setDataBreak</u> (system macro) . . . . .	360–361
<u>_setRangeBreak</u> (system macro) . . . . .	195
<u>_setSimBreak</u> (C-SPY system macro) . . . . .	157
<u>_strFind</u> . . . . .	361
<u>_subString</u> . . . . .	362
<u>_toLower</u> . . . . .	362
<u>_toUpper</u> . . . . .	363
<u>_writeFile</u> (system macro) . . . . .	363
<u>_writeFileByte</u> (system macro) . . . . .	364
<u>_writeMemoryByte</u> (system macro) . . . . .	364
<u>_writeMemory16</u> (system macro) . . . . .	365
<u>_writeMemory32</u> (system macro) . . . . .	366
<u>_writeMemory8</u> (system macro) . . . . .	365
C-SPY options . . . . .	248, 343
definition of . . . . .	368
Device description file . . . . .	344
Driver . . . . .	343
FET Debugger . . . . .	190
Run to . . . . .	107, 344
Setup . . . . .	343, 345
Setup macro . . . . .	344
C-SPY windows	
Auto . . . . .	282
Call Stack . . . . .	283
Code Coverage . . . . .	146, 286
Disassembly . . . . .	273
Live Watch . . . . .	282
Locals . . . . .	281
main . . . . .	105
Memory . . . . .	276
using . . . . .	130
Profiling . . . . .	287
Register . . . . .	279
example . . . . .	44
Register, using . . . . .	132
Stack . . . . .	292
Terminal I/O . . . . .	285
example . . . . .	46
Trace . . . . .	121, 289
Watch . . . . .	280
C-style preprocessor, definition of . . . . .	368

C/C++ syntax styles, options ..... 258

## D

data breakpoints .....	239
data pointers, definition of .....	369
data representation, definition of.....	369
dbg (filename extension).....	18
dbg7 (filename extension).....	18
ddf (filename extension) .....	18, 107
debug information	
generating in assembler .....	321
in compiler, generating .....	315
Debug information for C-SPY (XLINK option) .....	330
Debug Log window (View menu).....	230
Debug menu .....	294
debugger concepts, definitions of .....	103
Debugger (IDE Options dialog box).....	260
debugging projects	
externally built applications .....	108
in disassembly mode	
example.....	43
declaration, definition of.....	369
default installation path.....	15
XLINK options	
#define .....	333
#define options (XLINK) .....	333
#define statement, in compiler .....	316
Define symbol (XLINK option) .....	333
Defined symbols (assembler option).....	324
Defined symbols (compiler option).....	316
definition, definition of .....	369
demo application, running with C-SPY FET.....	176
dep (filename extension).....	18
derivative, definition of .....	369
design considerations (FET)	
boot-strap loader .....	209
device signals.....	209
external power .....	210
development environment, introduction .....	75
Device description file (C-SPY option).....	344
device description files .....	16, 107
definition of .....	109, 369
memory zones .....	110, 129
modifying .....	110
register zone.....	110, 129
specifying interrupts .....	169
using with memory maps.....	155
device driver, definition of .....	369
Device (target option) .....	301
diagnostics	
compiler	
including in list file .....	315
suppressing .....	318
XLINK	
suppressing .....	335
Diagnostics (assembler option) .....	325
Diagnostics (compiler option).....	317
Diagnostics (XLINK option) .....	334
Diagnostics, in list file (compiler option) .....	315
dialog boxes	
Autostep settings (Debug menu) .....	296
Batch Build Configuration (Project menu).....	250
Batch Build (Project menu) .....	249
Breakpoint Usage (Simulator menu) .....	154
Breakpoints (Edit menu) .....	237
Common fonts (IDE Options dialog box) .....	252
Conditional breakpoints dialog box (Edit menu) .....	196
Configurations for project (Project menu) .....	245
Create New Project (Project menu) .....	247
Debugger (IDE Options dialog box) .....	260
Edit Filename Extensions (Tools menu). ....	266
Editor Colors and Fonts (IDE Options dialog) .....	258
Editor (IDE Options dialog) .....	256
Embedded Workbench Startup (Help menu) .....	269
Enter Location (Breakpoints dialog box) .....	239
External Editor (IDE Options dialog box) .....	254
Filename Extensions Overrides (Tools menu) .....	266

Filename Extensions (Tools menu) . . . . .	265
Fill (Memory window context menu) . . . . .	278
Find in Files (Edit menu) . . . . .	235
Find (Edit menu) . . . . .	234
Interrupt Setup . . . . .	163
Interrupts . . . . .	162
Key Bindings (IDE Options dialog box) . . . . .	253
Log File (Debug menu) . . . . .	298
Macro Configuration (Debug menu) . . . . .	296
Messages (IDE Options dialog box) . . . . .	255
New Configuration (Project menu) . . . . .	246
Options (Project menu) . . . . .	248
Quick Watch (Debug menu) . . . . .	295
Range breakpoints dialog box (Edit menu) . . . . .	192
Register Filter (IDE Options dialog box) . . . . .	261
Replace (Edit menu) . . . . .	234
Sequencer Control (Emulator menu) . . . . .	206
Set Log file (Debug menu) . . . . .	295
Stack settings . . . . .	293
Terminal I/O Log File (Debug menu) . . . . .	299
Terminal I/O (IDE Options dialog) . . . . .	262
digital signal processor, definition of . . . . .	369
directories	
common\bin . . . . .	17
common\config . . . . .	17
common\doc . . . . .	17
common\plugins . . . . .	17
common\src . . . . .	18
settings . . . . .	19
430\bin . . . . .	16
430\config . . . . .	16
430\doc . . . . .	16
430\FET_examples . . . . .	16
430\inc . . . . .	16
430\lib . . . . .	16
430\plugins . . . . .	17
430\src . . . . .	17
430\tutor . . . . .	17
directory structure . . . . .	15
Disable language extensions (compiler option) . . . . .	310
Disable #ifdef/#endif matching (assembler option) . . . . .	319
__disableInterrupts (C-SPY system macro) . . . . .	353
disassembly mode debugging, example . . . . .	43
Disassembly window . . . . .	273
context menu . . . . .	275
definition of . . . . .	369
DLIB . . . . .	10
documentation . . . . .	21
specifying . . . . .	305
dni (filename extension) . . . . .	18–19
do (macro statement) . . . . .	349
document conventions . . . . .	xxxiii
documentation . . . . .	15
assembler . . . . .	11
compiler . . . . .	11
online . . . . .	16–17
other documentation . . . . .	xxxiii
product . . . . .	20
runtime libraries . . . . .	21
this guide . . . . .	xxix
XAR . . . . .	13
XLINK . . . . .	12
doc, common (subdirectory) . . . . .	17
doc, FET_examples (subdirectory) . . . . .	16
doc, 430 (subdirectory) . . . . .	16
Double floating point size (target option) . . . . .	302
Download control (C-SPY FET options) . . . . .	191
drag-and-drop, text in editor window . . . . .	97
Driver (C-SPY option) . . . . .	343
__driverType (C-SPY system macro) . . . . .	353
DSP. <i>See</i> digital signal processor	
Dynamic Data Exchange (DDE), calling external editor . . . . .	254
dynamic initialization, definition of . . . . .	369
dynamic memory allocation, definition of . . . . .	369
dynamic object, definition of . . . . .	369
d43 (filename extension) . . . . .	18

# E

Edit Filename Extensions dialog box (Tools menu) . . . . .	266
Edit menu . . . . .	232
editing source files . . . . .	95
editor	
commands . . . . .	96
customizing the environment . . . . .	98
features . . . . .	5
indentation . . . . .	97
keyboard commands . . . . .	224
matching brackets . . . . .	97
options . . . . .	256
shortcut to functions . . . . .	98, 223
splitter controls . . . . .	222
status bar, using in . . . . .	98
using . . . . .	95
using external . . . . .	99
Editor Colors and Fonts (IDE Options dialog) . . . . .	258
Editor window . . . . .	222
context menu . . . . .	223
Editor (IDE Options dialog) . . . . .	256
EEC++ syntax (compiler option) . . . . .	310
EEPROM, definition of . . . . .	370
Embedded C++	
definition of . . . . .	370
enabling in compiler . . . . .	310
tutorial . . . . .	53
Embedded C++ (compiler option) . . . . .	310
embedded system, definition of . . . . .	370
Embedded Workbench	
editor . . . . .	95
exiting from . . . . .	77
main window . . . . .	76, 218
reference information . . . . .	217
running . . . . .	76
version number, displaying . . . . .	268
Embedded Workbench Startup dialog box (Help menu) . . . . .	269
Embedded Workbench User Guide (Help menu) . . . . .	268
Emulator menu . . . . .	201, 299
emulator (C-SPY version)	
definition of . . . . .	370
third-party . . . . .	4
Enable multibyte support (assembler option) . . . . .	319
Enable multibyte support (compiler option) . . . . .	311
Enable remarks (compiler option) . . . . .	317
Enable Virtual Space (editor option) . . . . .	257
enabled transformations, in compiler . . . . .	312
__enableInterrupts (C-SPY system macro) . . . . .	354
Enter Location dialog box . . . . .	239
enumeration, definition of . . . . .	370
EPROM, definition of . . . . .	370
error messages	
assembler . . . . .	325
compiler . . . . .	318
XLINK . . . . .	335
ewd (filename extension) . . . . .	18
ewp (filename extension) . . . . .	18
eww (filename extension) . . . . .	18
\$EW_DIR\$ (argument variable) . . . . .	244
examples	
assembler	
mixing C and assembler . . . . .	49
running project with C-SPY FET . . . . .	177
viewing list file . . . . .	51
breakpoints	
executing up to . . . . .	42
setting . . . . .	42
setting, using macro . . . . .	67
setting, using options dialog . . . . .	64
C example, running with C-SPY FET . . . . .	176
calling convention, examining . . . . .	49
compiling . . . . .	32
conditional breakpoint triggering state storage . . . . .	183
ddf file, using . . . . .	63
debugging a program . . . . .	37
disassembly mode debugging . . . . .	43
displaying function calls in C-SPY . . . . .	63

displaying Terminal I/O	46
interrupts	
timer interrupt	166
using macro	67
linking	
a compiler program	35
viewing the map file	36
macros	
checking latest value	136
checking status of register	140
checking status of WDT	140
creating a log macro	141
for interrupts and breakpoints	67
using Quick Watch	140
Memory window, using	44
mixing C/C++ and assembler	50
monitoring memory	44
monitoring registers	44
options	
setting project	29
performing tasks without stopping execution	126
periodically monitoring data	126
project	
adding files	28
creating	25, 27
setting options	29
reaching program exit	46
Scan for Changed Files (editor option), using	34
state storage, using	182
stepping	38
tracing incorrect function arguments	125
using libraries	69
variables	
setting a watch point	41
watching in C-SPY	40
viewing compiler list files	33
workspace, creating a new	26
exceptions, definition of	370
execUserExit() (C-SPY setup macro)	351
execUserPreload() (C-SPY setup macro)	350
execUserReset() (C-SPY setup macro)	351
execUserSetup() (C-SPY setup macro)	351
example	61, 67
Executables (output directory)	303
executing a program up to a breakpoint	42
\$EXE_DIR\$ (argument variable)	244
exit	
example, reaching program	46
Exit (File menu)	77
expensive memory access, definition of	370
expressions. <i>See</i> C-SPY expressions	
Extended Embedded C++	
syntax, enabling in compiler	310
extended keywords, definition of	370
extended linker command line file. <i>See</i> linker command file	
extensions. <i>See</i> filename extensions or language extensions	
External Editor (IDE Options dialog box)	254
Extra Output (XLINK options)	332

## F

factory settings	
compiler	90
restoring default settings	91
XLINK	341
features	
assembler	11
compiler	9
editor	5
FET Debugger	
characteristics	179
design considerations	209
features	9
functionality	
breakpoints	179
state storage	182
stepping	182
options	190

file extensions. <i>See</i> filename extensions	
File menu . . . . .	231
file types	
device description . . . . .	16
specifying in Embedded Workbench . . . . .	107
documentation . . . . .	16
header . . . . .	16
include . . . . .	16
library . . . . .	16
linker command file templates . . . . .	16
macro . . . . .	107, 344
map . . . . .	336
project templates . . . . .	16
special function registers description files . . . . .	16
syntax coloring configuration . . . . .	16
filename extensions. . . . .	18
asm . . . . .	18
a43 . . . . .	18
c . . . . .	18
cfg . . . . .	18, 258
cpp . . . . .	18
dbg . . . . .	18
dbg <sub>t</sub> . . . . .	18
ddf . . . . .	18
dep . . . . .	18
dni . . . . .	18–19
d43 . . . . .	18
ewd . . . . .	18
ewp . . . . .	18
eww . . . . .	18
fmt . . . . .	18
h . . . . .	18
i . . . . .	19
inc . . . . .	19
ini . . . . .	19
lst . . . . .	19
mac . . . . .	19
map . . . . .	19
pbd . . . . .	19
pbi . . . . .	19
prj . . . . .	19
r43 . . . . .	19
sfr . . . . .	19
register definitions for C-SPY . . . . .	110
s43 . . . . .	19
wsdt . . . . .	19
xcl . . . . .	19
xlb . . . . .	19
Filename Extensions dialog box (Tools menu) . . . . .	265
Filename Extensions Overrides dialog box (Tools menu) . . . . .	266
files	
adding to a project . . . . .	28
compiling	
example . . . . .	32
editing . . . . .	95
navigating . . . . .	85
readme.htm . . . . .	20
\$FILE_DIR\$ (argument variable) . . . . .	244
\$FILE_FNAME\$ (argument variable) . . . . .	244
\$FILE_PATH\$ (argument variable) . . . . .	244
Fill dialog box . . . . .	278
using . . . . .	131
Fill pattern (XLINK option) . . . . .	339
Fill unused code memory (XLINK option) . . . . .	339
Find dialog box (Edit menu) . . . . .	234
Find in Files dialog box (Edit menu) . . . . .	235
Find in Files window (View menu) . . . . .	228
Find (button) . . . . .	219
First activation time, definition of . . . . .	160
fmt (filename extension) . . . . .	18
for (macro statement) . . . . .	349
Forced Interrupt window (Simulator menu) . . . . .	164
Forced Interrupts (Simulator menu) . . . . .	153
format specifiers, definition of . . . . .	370
Format (XLINK option) . . . . .	330
formats	
assembler list file . . . . .	51
compiler list file . . . . .	33

C-SPY input.....	8
standard IEEE (floating point).....	302
XLINK output	
default, overriding.....	333
XLINK output formats	
default, overriding.....	331
specifying.....	330
function calls	
displaying in C-SPY .....	63
functions	
C-SPY running to when starting .....	107, 344
intrinsic, definition of.....	371
shortcut to in editor windows.....	98, 223

<b>G</b>	
general options .....	301
definition of.....	370
Library Configurations.....	304
Library Options .....	306
Output .....	302
specifying, example.....	29
Stack/Heap Options .....	307
Target.....	301
Generate checksum (XLINK option) .....	339
Generate debug info (compiler option).....	315
Generate debug information (assembler option) .....	321
Generate extra output file (XLINK option).....	332
Generate linker listing (XLINK option) .....	336
generating extra output file .....	331
generic pointers, definition of .....	370
Getting started, using the C-SPY FET .....	176
glossary.....	367
Go to function (editor button).....	98, 223
Go to (button).....	219
Go (button) .....	273
Go (Debug menu).....	114
groups, definition of .....	83

## H

h (filename extension).....	18
Hardware multiplier (target option) .....	302
hardware multiplier, enabling support for .....	158
Harvard architecture, definition of .....	371
header files .....	16
heap memory, definition of .....	371
Heap size (general option) .....	307
heap size, definition of .....	371
Help menu .....	268
Assembler Reference Guide.....	268
Compiler Reference Guide.....	268
Embedded Workbench User Guide .....	268
Linker and Library Tools Reference Guide .....	268
highlight color, paler variant of .....	114
Hold time, definition of .....	160
host, definition of .....	371

## I

i (filename extension) .....	19
IAR Assembler Reference Guide .....	20
IAR Compiler Reference Guide .....	20
IAR Linker and Library Tools Reference Guide .....	20
IAR Systems website .....	21
IarIdePm.exe .....	76
IDE .....	3–4
definition of .....	371
IEEE format, floating-point values .....	302
if else (macro statement) .....	348
if (macro statement) .....	348
illegal access .....	154
inc (filename extension) .....	19
Include compiler runtime information (compiler option) ..	315
include files .....	16
assembler, specifying path .....	324
compiler, specifying path .....	316
definition of .....	371

XLINK, specifying path	338
Include header (assembler option)	322
Include listing (assembler option)	322
Include paths (assembler option)	324
Include paths (compiler option)	316
Include source (compiler option)	315
Include suppressed entries (XLINK option)	337
inc, 430 (subdirectory)	16
Indent Size (editor option)	256
indentation, in Editor	97
information, product	20
ini (filename extension)	19
inline assembler, definition of	371
inlining, definition of	371
input	
redirecting to Terminal I/O window	285
special characters in Terminal I/O window	285
input formats, C-SPY	8
installation path, default	15
installed files	15
documentation	16–17
executables	17
include	16
library	16
instruction mnemonics, definition of	371
Integrated Development Environment (IDE)	3–4
definition of	371
Intel-extended, C-SPY input format	8, 105
Internet, IAR Systems website	21
Interrupt Log window (Simulator menu)	166
Interrupt Setup dialog box (Simulator menu)	163
interrupt vector table, definition of	371
interrupt vector, definition of	371
interrupts	
adapting C-SPY system for target hardware	161
definition of	371
definition of simulated interrupts	159
in ddf	110
nested, definition of	373
options	163
simulating	162
timer interrupt, example	166
using system macros	165
Interrupts dialog box (Simulator menu)	162
Interrupts (Simulator menu)	153
intrinsic functions, definition of	371
intrinsic, definition of	371
ISO/ANSI C	10
adhering to	310
I/O emulation modules, linking for C-SPY	116

## K

Key bindings (IDE Options dialog box)	253
key bindings, definition of	371
key summary, editor	224
keywords, definition of	372

## L

language extensions	
definition of	372
disabling in compiler	310
language facilities, in compiler	10
Language (assembler options)	319
Language (compiler options)	309
libraries, creating a project	70
libraries, runtime	10
library builder. <i>See</i> XAR	
Library Configurations (general options)	304
Library file (general option)	305
library files	16
library functions, configurable	17
library modules	
example	69
specifying in assembler	321
specifying in compiler	314
using	69

Library Options (general options) . . . . .	306
Library (general option) . . . . .	304
library, definition of . . . . .	375
lib, 430 (subdirectory) . . . . .	16
#line directives, generating	
in compiler . . . . .	317
Lines/page (assembler option) . . . . .	323
Lines/page (XLINK option) . . . . .	337
Linker and Library Tools Reference Guide (Help menu) .	268
linker command file	
definition of . . . . .	372
path, specifying . . . . .	338
specifying in XLINK . . . . .	338
templates . . . . .	16
Linker command file (XLINK option) . . . . .	338
linker. <i>See</i> XLINK	
list files	
assembler . . . . .	51
compiler runtime information, including . . . . .	315
conditional information, specifying . . . . .	322
cross-references, generating . . . . .	323
header, including . . . . .	322
lines per page, specifying . . . . .	323
tab spacing, specifying . . . . .	323
compiler	
assembler mnemonics, including . . . . .	315
example . . . . .	33
generating . . . . .	315
source code, including . . . . .	315
XLINK	
generating . . . . .	336
including segment map . . . . .	336
specifying lines per page . . . . .	337
list files, option for specifying destination . . . . .	303
List (assembler options) . . . . .	322
List (compiler options) . . . . .	315
List (XLINK options) . . . . .	336
\$LIST_DIR\$ (argument variable) . . . . .	245
Live Watch window . . . . .	282
context menu . . . . .	283
local variables. <i>See</i> auto variables	
Locals window . . . . .	281
context menu . . . . .	281
location counter, definition of . . . . .	374
Log File dialog box (Debug menu) . . . . .	298
logical address, definition of . . . . .	377
loop statements, in C-SPY macros . . . . .	349
lst (filename extension) . . . . .	19
L-value, definition of . . . . .	372
<b>M</b>	
mac (filename extension) . . . . .	19
Macro Configuration dialog box (Debug menu) . . . . .	296
macro files, specifying . . . . .	107, 344
Macro quote characters (assembler option) . . . . .	320
macro statements . . . . .	348
macros	
definition of . . . . .	372
executing . . . . .	137
MAC, definition of . . . . .	372
mailbox (RTOS), definition of . . . . .	372
main function, C-SPY running to when starting . . . . .	107, 344
main.s43 (assembler tutorial file) . . . . .	69
Make library module (assembler option) . . . . .	321
Make library module (compiler option) . . . . .	314
managing projects . . . . .	4
map files . . . . .	336
example . . . . .	36
viewing . . . . .	36
map (filename extension) . . . . .	19
Marginal read check (C-SPY FET option) . . . . .	190
memory	
filling unused . . . . .	339
filling with value . . . . .	131
monitoring . . . . .	130, 134
example . . . . .	44

memory access cost, definition of	373
memory area, definition of	373
memory bank, definition of	373
memory map	154 <ul style="list-style-type: none"> <li>definition of</li> </ul>
memory model, definition of	373
memory usage, summary of	337
Memory window	276 <ul style="list-style-type: none"> <li>context menu</li> <li>operations</li> <li>using</li> </ul>
memory zones	129
in ddf	110
menu bar	218 <ul style="list-style-type: none"> <li>C-SPY-specific</li> </ul>
menus	<ul style="list-style-type: none"> <li>Emulator (FET)</li> <li>File</li> <li>Project</li> <li>Tools</li> <li>View</li> </ul>
message (C-SPY macro statement)	349
messages	<ul style="list-style-type: none"> <li>displaying in Embedded Workbench</li> <li>printing during macro execution</li> </ul>
Messages window	<ul style="list-style-type: none"> <li>amount of output, configuring</li> </ul>
Messages (IDE Options dialog box)	255
microcontroller, definition of	373
microprocessor, definition of	373
migration from earlier IAR compilers	311
module map, in map files	36
module name, specifying in compiler	314
Module summary (XLINK option)	337
MODULE (assembler directive)	70
modules	<ul style="list-style-type: none"> <li>definition of</li> <li>including local symbols in input</li> </ul>
Module-local symbols (XLINK option)	332

Motorola, C-SPY input format	8, 105
Multiply and accumulate, definition of	372
multitasking, definition of	374

## N

nested interrupts, definition of	373
New Configuration dialog box (Project menu)	246
Next Statement (button)	273
No global type checking (XLINK option)	334
non-banked memory, definition of	373
non-initialized memory, definition of	373
non-volatile storage, definition of	373
NOP, definition of	373

## O

object files, specifying output directory	303
Object module name (compiler option)	314
\$OBJ_DIR\$ (argument variable)	245
online documentation	<ul style="list-style-type: none"> <li>guides</li> <li>help</li> </ul>
Online help	21
Open Workspace (File menu)	231 <ul style="list-style-type: none"> <li>_openFile (C-SPY system macro)</li> </ul>
operator precedence, definition of	373
operators, definition of	373
optimization levels	312
optimization models	312
optimizations	119, 312
options	<ul style="list-style-type: none"> <li>assembler</li> <li>compiler</li> <li>custom build</li> <li>C-SPY</li> <li>C-SPY FET</li> <li>editor</li> <li>general</li> </ul>
	319
	309
	327
	248, 343
	190
	256
	29, 301

XAR .....	341
XLINK .....	329
Options dialog box (Project menu) .....	248
using .....	90
_orderInterrupt (C-SPY system macro) .....	355
output	
assembler	
generating library modules .....	321
including debug information .....	320
compiler	
including debug information .....	315
preprocessor, generating .....	317
from C-SPY, redirecting to a file .....	109
XLINK	
generating .....	334
specifying filename .....	329
Output assembler file (compiler option) .....	315
Output file (XLINK option) .....	329
Output format (XLINK option) .....	331, 333
output formats	
debug (ubrof) .....	330
XLINK .....	330–331, 333
Output list file (compiler option) .....	315
Output (assembler option) .....	320
Output (compiler options) .....	314
Output (general options) .....	302
Output (XAR options) .....	341
Output (XLINK options) .....	329
output, generating extra file .....	331
<b>P</b>	
Parallel Port (C-SPY FET option) .....	191
paths	
assembler include files .....	324
compiler include files .....	316
relative, in Embedded Workbench .....	85, 224
source files .....	224
XLINK include files .....	338
pbd (filename extension) .....	19
pbi (filename extension) .....	19
peripherals, definition of .....	374
pipeline, definition of .....	374
Plain ‘char’ is (compiler option) .....	311
plugins, common (subdirectory) .....	17
plugins, 430 (subdirectory) .....	17
pointers, definition of .....	374
Position-independent code (target option) .....	301
pragma directive, definition of .....	374
precedence, definition of .....	373
preemptive multitasking, definition of .....	374
preprocessing directives, definition of .....	374
preprocessor	
definition of. <i>See</i> C-style preprocessor	
Preprocessor output to file (compiler option) .....	317
Preprocessor (assembler option) .....	323
Preprocessor (compiler options) .....	316
prerequisites, programming experience .....	xxix
Printf formatter (general option) .....	306
prj (filename extension) .....	19
Probability, definition of .....	160
Processing options (XLINK) .....	339
processor variant, definition of .....	374
product information, obtaining detailed .....	268
product overview	
assembler .....	11
compiler .....	9
C-SPY Debugger .....	5
directory structure .....	15
documentation .....	20
FET Debugger .....	9
IAR Embedded Workbench IDE .....	3
XAR .....	12
XLINK .....	11
profiling .....	287
program counter, definition of .....	374
program execution, in C-SPY .....	111
program location counter, definition of .....	374

programming experience.....	xxix
Project Make, options .....	259
Project menu.....	243
Project model .....	81
project options, definition of.....	374
Project page (IDE Options dialog) .....	259
projects	
adding files to .....	84, 243
example.....	28
build configuration, creating .....	84
building .....	91
compiling, example .....	32
creating .....	27, 84
example.....	70
definition of.....	81, 374
excluding groups and files .....	84
for debugging externally built applications .....	108
groups, creating .....	84
managing .....	4, 81
moving files.....	84
organization.....	81
removing items .....	84
setting options .....	89
testing .....	91
workspace, creating .....	84
\$PROJ_DIR\$ (argument variable) .....	245
\$PROJ_FNAME\$ (argument variable).....	245
\$PROJ_PATH\$ (argument variable) .....	245
PROM, definition of .....	374
PUBLIC (assembler directive) .....	70

## Q

qualifiers, definition of. *See* type qualifiers

Quick Watch	
executing C-SPY macros .....	140
using .....	120
Quick Watch dialog box (Edit menu) .....	295

## R

Range breakpoints dialog box (Edit menu).....	192
Range checks (XLINK option) .....	335
__readFile (C-SPY system macro).....	355
__readFileByte (C-SPY system macro).....	355
readme files.....	
readme.htm .....	20
__readMemoryByte (C-SPY system macro) .....	356
__readMemory16 (C-SPY system macro) .....	357
__readMemory32 (C-SPY system macro) .....	357
__readMemory8 (C-SPY system macro) .....	356
real-time operating system, definition of .....	374
real-time system, definition of .....	374
reference information	
C-SPY .....	271
Embedded Workbench.....	217
guides.....	20
register constant, definition of .....	374
Register Filter (IDE Options dialog box) .....	261
Register groups.....	
application-specific, defining .....	133
pre-defined, enabling .....	133
register locking, definition of .....	375
register variables, definition of .....	375
Register window .....	
example .....	44
using .....	132
register zone .....	
__registerMacroFile (C-SPY system macro) .....	358
registers	
definition of .....	374
in device description file .....	110
relative paths .....	
85, 224	
Relaxed ISO/ANSI (compiler option) .....	310
relocatable segments, definition of .....	375
remarks, compiler diagnostics .....	318
Repeat interval, definition of .....	160
Replace dialog box (Edit menu) .....	234

Replace (button) . . . . .	219
Require prototypes (compiler option) . . . . .	310
Reset (button) . . . . .	273
Reset (Debug menu)	
example . . . . .	47
_resetFile (C-SPY system macro) . . . . .	358
reset, definition of . . . . .	375
return (macro statement) . . . . .	349
ROM-monitor, definition of . . . . .	375
root directory . . . . .	15
Round Robin, definition of . . . . .	375
RTOS, definition of . . . . .	374
Run to Cursor (button) . . . . .	273
Run to (C-SPY option) . . . . .	107, 344
runtime libraries . . . . .	10
documentation . . . . .	21
runtime library, definition of . . . . .	375
runtime model attributes	
definition of . . . . .	375
in map files . . . . .	36
R-value, definition of . . . . .	374
R4 utilization (compiler option) . . . . .	313
r43 (filename extension) . . . . .	19
R5 utilization (compiler option) . . . . .	313
<b>S</b>	
saturated mathematics, definition of . . . . .	375
Save All (File menu) . . . . .	232
Save As (File menu) . . . . .	232
Save Current Layout As Default (Debug menu) . . . . .	153, 299
Save Workspace (File menu) . . . . .	231
Save (File menu) . . . . .	231
Scan for Changed Files (editor option) . . . . .	257
using . . . . .	34
Scancf formatter (general option) . . . . .	306
scheduler (RTOS), definition of . . . . .	375
scope, definition of . . . . .	375
Search paths (XLINK option) . . . . .	338
searching . . . . .	98
Segment map (XLINK option) . . . . .	336
segment map, definition of . . . . .	375
Segment overlap warnings (XLINK option) . . . . .	334
segment parts, including all in list file . . . . .	337
segments	
definition of . . . . .	375
overlap errors, reducing . . . . .	334
range checks, controlling . . . . .	335
section in map files . . . . .	36
semaphores, definition of . . . . .	376
Sequencer Control window (Emulator menu) . . . . .	206
Set Log file dialog box (Debug menu) . . . . .	295
Set Terminal IO Log File (Debug menu) . . . . .	116
_setCodeBreak (C-SPY system macro) . . . . .	359
_setConditionalBreak (C-SPY system macro) . . . . .	199
_setDataBreak (C-SPY system macro) . . . . .	360–361
_setRangeBreak (C-SPY system macro) . . . . .	195
_setSimBreak (C-SPY system macro) . . . . .	157
settings (directory) . . . . .	19
Setup macro (C-SPY option) . . . . .	344
setup macros, in C-SPY	
<i>See also</i> C-SPY macros	
setup macros, in C-SPY. <i>See</i> C-SPY macros	
Setup (C-SPY options) . . . . .	343, 345
severity level, definition of . . . . .	376
SFR	
definition of . . . . .	376
header files . . . . .	16
sfr (filename extension) . . . . .	19
shifts.s34 (assembler tutorial file) . . . . .	69
short addressing, definition of . . . . .	376
shortcut keys . . . . .	96
Show Bookmarks (editor option) . . . . .	257
Show Line Number (editor option) . . . . .	257
showing build messages, in Embedded Workbench . . . . .	256
side-effect, definition of . . . . .	376
signals, definition of . . . . .	376

simulator	
definition of	376
features	8
Simulator menu	299
size optimization	312
Size (Breakpoints dialog)	241
skeleton code, definition of	376
Source Browser window (View menu)	226
using	87
source code, in compiler list file	315
source file paths	85, 224
source files	83
adding to a project	28
editing	95
special function registers (SFR)	
definition of	376
description files	16, 110
header files	16
using as assembler symbols	118
speed optimization	312
src, common (subdirectory)	18
src, 430 (subdirectory)	17
stack frames, definition of	376
stack segments, definition of	376
Stack Settings (dialog box)	293
Stack size (general option)	307
Stack window	292
using	134
Stack/Heap (general options)	307
starting the Embedded Workbench	76
State Storage Control (Emulator menu)	203
State Storage window (Emulator menu)	205
state storage, using	182
static objects, definition of	376
static overlay, definition of	377
statically allocated memory, definition of	376
status bar	219
Step Into	
button	273
example	40
Step Out (button)	273
Step Over (button)	273
step points, definition of	112
stepping	112
definition of	376
example	38
using C-SPY FET	182
Stop Debugging (button)	273
_strFind (C-SPY system macro)	361
Strict ISO/ANSI (compiler option)	310
structure value, definition of	377
_subString (C-SPY system macro)	362
support, technical	21
Suppress all warnings (XLINK option)	335
Suppress these diagnostics (compiler option)	318
Suppress these diagnostics (XLINK option)	335
symbolic location, definition of	377
symbols	
<i>See also</i> user symbols	
defining in assembler	324
defining in compiler	316
defining in XLINK	333
definition of	377
in input modules	332
using in C-SPY expressions	117
syntax coloring	
configuration files	16
in editor	97
Syntax Highlighting (editor option)	257
syntax highlighting, in Editor window	97
System breakpoints on (C-SPY FET option)	191
s43 (filename extension)	19
T	
Tab Key Function (editor option)	256
Tab Size (editor option)	256
Tab spacing (assembler option)	323

Target options	
Assembler only project	302
Device	301
Double floating point size	302
Hardware multiplier	302
Position-independent code	301
target processors	81
Target (general options)	301
target, definition of	377
\$TARGET_BNAME\$ (argument variable)	245
\$TARGET_BPATH\$ (argument variable)	245
\$TARGET_DIR\$ (argument variable)	245
\$TARGET_FNAME\$ (argument variable)	245
\$TARGET_PATH\$ (argument variable)	245
task, definition of	377
technical support	21
tentative definition, definition of	377
terminal I/O	
definition of	377
simulating	330
Terminal I/O Log File dialog box (Debug menu)	299
Terminal I/O window	285
example	46
Terminal I/O (IDE Options dialog)	262
terminology	367
testing, of code	91
thread, definition of	377
timer, definition of	377
timeslice, definition of	377
Toggle Breakpoint (button)	273
Toggle Breakpoint (Edit menu)	
example	42, 66
_toLower (C-SPY system macro)	362
tool chain	
extending	92
specifying	27
Tool Output window (View menu)	229
toolbar	219
debug	273
Trace	291
\$TOOLKIT_DIR\$ (argument variable)	245
Tools menu	251
tools, user-configured	263
_ToUpper (C-SPY system macro)	363
Trace (window)	289
using	121
transformations, enabled in compiler	312
translation unit, definition of	377
trap, definition of	377
Treat all warnings as errors (compiler option)	318
Treat these as errors (compiler option)	318
Treat these as errors (XLINK option)	335
Treat these as remarks (compiler option)	318
Treat these as warnings (compiler option)	318
Treat these as warnings (XLINK option)	335
tutor, 430 (subdirectory)	17
type qualifiers, definition of	377
type-checking	10, 12
disabling at link time	334
typographic conventions	xxxiii
<b>U</b>	
UBROF	8, 12
definition of	377
Universal Binary Relocatable Object Format (UBROF)	8, 12
definition of	377
Unreferred externals in object file (assembler option)	321
Use Default Size (editor option)	257
Use virtual breakpoints (C-SPY FET option)	191
User symbols are case sensitive	
(assembler option)	319
<b>V</b>	
variables	
effects of optimizations	119
information, limitation on	119

using in arguments . . . . .	264
using in C-SPY expressions . . . . .	117
watching in C-SPY . . . . .	120
example . . . . .	40
Variance, definition of . . . . .	160
Verify download (C-SPY FET option) . . . . .	190
version number, of Embedded Workbench . . . . .	268
View menu . . . . .	242
virtual address, definition of . . . . .	377
virtual space, definition of . . . . .	377
volatile storage, definition of . . . . .	378
von Neumann architecture, definition of . . . . .	378

## W

warnings	
assembler . . . . .	325
compiler . . . . .	318
XLINK . . . . .	335
Warnings/Errors (XLINK option) . . . . .	335
Watch window . . . . .	280
context menu . . . . .	280
watchpoints	
definition of . . . . .	378
setting . . . . .	40
website, IAR Systems . . . . .	21
while (macro statement) . . . . .	349
Window menu . . . . .	267
windows	
<i>See also</i> Project Manager windows or C-SPY windows	
Build . . . . .	227
Debug Log . . . . .	230
Editor . . . . .	222
Find in Files . . . . .	228
Forced Interrupt . . . . .	164
Interrupt Log . . . . .	166
Source Browser . . . . .	226
Tool Output . . . . .	229

Workspace . . . . .	220
With I/O emulation modules (XLINK option) . . . . .	285, 330
With runtime control modules (XLINK option) . . . . .	330
Workspace window (View menu) . . . . .	220
example . . . . .	28
workspaces . . . . .	84
creating . . . . .	26
definition of . . . . .	81
__writeFile (C-SPY system macro) . . . . .	363
__writeFileByte (C-SPY system macro) . . . . .	363
__writeMemoryByte (C-SPY system macro) . . . . .	364
__writeMemory16 (C-SPY system macro) . . . . .	365
__writeMemory32 (C-SPY system macro) . . . . .	365
__writeMemory8 (C-SPY system macro) . . . . .	364
wsdt (filename extension) . . . . .	19
www.iar.com . . . . .	21

## X

XAR	
documentation . . . . .	20
features . . . . .	13
overview . . . . .	12
XAR options . . . . .	341
definition of . . . . .	378
Output . . . . .	341
xcl (filename extension) . . . . .	19
xlb (filename extension) . . . . .	19
XLIB options, definition of . . . . .	378
XLIB, documentation . . . . .	20
XLINK	
diagnostics, suppressing . . . . .	335
documentation . . . . .	20
example . . . . .	35
overview . . . . .	11
XLINK list files	
generating . . . . .	336
including segment map . . . . .	336
specifying lines per page . . . . .	337

XLINK options .....	329, 339
definition of .....	378
factory settings .....	341
Allow C-SPY-specific output file .....	331
Always generate output .....	334
Buffered terminal output .....	331
Config .....	337
Debug information for C-SPY .....	330
Define symbol .....	333
Diagnostics .....	334
Extra Output .....	332
Fill pattern .....	339
Fill unused code memory .....	339
Format .....	330
Generate checksum .....	339
Generate extra output file .....	332
Generate linker listing .....	336
Include suppressed entries .....	337
Lines/page .....	337
Linker command file .....	338
List .....	336
Module summary .....	337
Module-local symbols .....	332
No global type checking .....	334
Output .....	329
Output file .....	329
Output format .....	331, 333
Range checks .....	335
Search paths .....	338
Segment map .....	336
Segment overlap warnings .....	334
Suppress all warnings .....	335
Suppress these diagnostics .....	335
Treat these as errors .....	335
Treat these as warnings .....	335
Warnings/Errors .....	335
With I/O emulation modules .....	330
With runtime control modules .....	330

XLINK output, overriding default format .....	331, 333
XLINK symbols, defining .....	333

## Z

zero-overhead loop, definition of .....	378
zones	
definition of .....	378
in C-SPY .....	129

# Symbols

#define options (XLINK) .....	333
#define statement, in compiler .....	316
#line directives, generating in compiler .....	317
\$CUR_DIR\$ (argument variable) .....	244
\$CUR_LINE\$ (argument variable) .....	244
\$EW_DIR\$ (argument variable) .....	244
\$EXE_DIR\$ (argument variable) .....	244
\$FILE_DIR\$ (argument variable) .....	244
\$FILE_FNAME\$ (argument variable) .....	244
\$FILE_PATH\$ (argument variable) .....	244
\$LIST_DIR\$ (argument variable) .....	245
\$OBJ_DIR\$ (argument variable) .....	245
\$PROJ_DIR\$ (argument variable) .....	245
\$PROJ_FNAME\$ (argument variable) .....	245
\$PROJ_PATH\$ (argument variable) .....	245
\$TARGET_BNAME\$ (argument variable) .....	245
\$TARGET_BPATH\$ (argument variable) .....	245
\$TARGET_DIRS\$ (argument variable) .....	245
\$TARGET_FNAME\$ (argument variable) .....	245
\$TARGET_PATH\$ (argument variable) .....	245
\$TOOLKIT_DIR\$ (argument variable) .....	245
_cancelAllInterrupts (C-SPY system macro) .....	168, 352
_cancelInterrupt (C-SPY system macro) .....	168, 352
_clearBreak (C-SPY system macro) .....	353
_closeFile (C-SPY system macro) .....	353
_disableInterrupts (C-SPY system macro) .....	169, 353
_driverType (C-SPY system macro) .....	353

_enableInterrupts (C-SPY system macro) . . . . .	169, 354
_openFile (C-SPY system macro) . . . . .	354
_orderInterrupt (C-SPY system macro) . . . . .	169, 355
_readFile (C-SPY system macro) . . . . .	355
_readFileByte (C-SPY system macro) . . . . .	355
_readMemoryByte (C-SPY system macro) . . . . .	356
_readMemory16 (C-SPY system macro) . . . . .	357
_readMemory32 (C-SPY system macro) . . . . .	357
_readMemory8 (C-SPY system macro) . . . . .	356
_registerMacroFile (C-SPY system macro) . . . . .	358
_resetFile (C-SPY system macro) . . . . .	358
_setCodeBreak (C-SPY system macro) . . . . .	359
_setConditionalBreak (C-SPY system macro) . . . . .	199
_setDataBreak (C-SPY system macro) . . . . .	360–361
_setRangeBreak (C-SPY system macro) . . . . .	195
_strFind (C-SPY system macro) . . . . .	361
_subString (C-SPY system macro) . . . . .	362
_toLower (C-SPY system macro) . . . . .	362
_toUpper (C-SPY system macro) . . . . .	363
_writeFile (C-SPY system macro) . . . . .	363
_writeFileByte (C-SPY system macro) . . . . .	363
_writeMemoryByte (C-SPY system macro) . . . . .	364
_writeMemory16 (C-SPY system macro) . . . . .	365
_writeMemory32 (C-SPY system macro) . . . . .	365
_writeMemory8 (C-SPY system macro) . . . . .	364

# Numerics

430 (directory) . . . . .	16
---------------------------	----