



Introduction

Welcome to **ST Visual Develop (STVD)**. STVD is a powerful, integrated development environment for ST microcontrollers. This Windows-based environment provides interfaces for all phases of application development, from building and debugging your application, to programming your ST microcontroller.

New features described in this document are not present in earlier versions of STVD.

This document will help you get started developing your application with STVD. It includes:

- Detailed descriptions of STVD's user interface and features
- Information about STVD features that are specific to your debugging hardware
- Tutorials to help you learn to use STVD's build, debug and program features

Features

- **For building your application**, STVD supports ST Assembler-Linker, Cosmic C, Raisonance C and Metrowerks C toolsets. Its graphics interface provides easy access to a range of options when building your application for debugging or programming your microcontroller.
- **For debugging your application**, STVD supports a complete range of in-circuit debugging, emulation hardware, and advanced debugging features, including advanced breakpoints and trace recording that will make application development easy and fast. It can also be used as a stand-alone tool, providing a software simulation of your microcontroller's behavior as it runs your application.
- **For programming**, STVD provides a programming interface that is based on ST Visual Programming (STVP) software and supports a complete range of hardware for in-circuit, in-situ and socket programming.

Contents

1	Introduction	18
1.1	Building and debugging applications	18
1.2	Programming your microcontroller	19
1.3	Advanced debugging features	19
1.4	Getting assistance	21
1.5	Associated documentation	21
1.6	Conventions	22
2	Getting started with STVD	23
2.1	Set the toolset and path information	23
2.2	Create a workspace with a new project	23
2.2.1	Create a new workspace	24
2.2.2	Create a project in your workspace	25
2.2.3	Specify a target MCU for your project	25
2.3	Add source files (.asm, .c, .s) to your project	27
2.4	Build your application	28
2.4.1	Configuring project settings for Debug or Release	28
2.4.2	Build commands and output	29
2.5	Select and connect to your debug instrument	30
2.6	Debug your application	32
2.7	Program your application to your microcontroller	32
3	Your STVD graphical environment	34
3.1	The main application window	34
3.2	The main menus and their commands	36
3.3	View windows	40
3.4	Workspace window	41
3.5	Editor windows	43
3.5.1	Editor window contextual menu	49
3.5.2	Editing features	49
3.6	Output window	56
3.6.1	Build tab and Tools tab	57

3.6.2	Find in Files 1 & Find in Files 2 tabs	58
3.6.3	Debug tab	58
3.6.4	Console tab	58
3.7	Customizing your work environment	60
3.7.1	Customizing editor features	61
3.7.2	Customizing toolbars	65
3.7.3	Adding custom commands	67
3.8	Tooltips	69
3.9	Help and support features	70
3.10	Migrating old workspaces (STVD7 2.5.4 and prior versions)	71
4	Project creation and build	75
4.1	Specifying a toolset	75
4.2	Loading and creating workspaces (.stw)	76
4.2.1	Loading an existing workspace	76
4.2.2	Creating a new workspace	78
4.3	Creating and loading projects in a workspace	80
4.3.1	Loading an existing project	80
4.3.2	Creating new projects	80
4.3.3	Adding and removing folders and files	83
4.4	Project build configurations	83
4.5	Configuring project settings	84
4.5.1	General settings tab	85
4.5.2	Debug settings tab	86
4.5.3	MCU selection tab	86
4.5.4	Pre-link settings tab	87
4.5.5	Post-build settings tab	87
4.6	Customizing build settings for ST Assembler/Linker toolset	88
4.6.1	ST ASM tab	89
4.6.2	ST Link tab	90
4.6.3	ST Post-Link tab	92
4.7	Customizing build settings for Cosmic C toolsets	93
4.7.1	Cosmic C compiler tab	95
4.7.2	Cosmic C Assembler tab	107
4.7.3	Cosmic C linker tab	110
4.8	Customizing build settings for Raisonance C toolset	116

4.8.1	Raisonance C compiler tab	118
4.8.2	Raisonance Assembler tab	125
4.8.3	Raisonance C linker tab	127
4.9	Customizing build settings for Metrowerks C toolset	131
4.9.1	Metrowerks C Compiler tab	133
4.9.2	Metrowerks Assembler tab	143
4.9.3	Metrowerks linker tab	148
4.10	Configuring folder and file settings	156
4.10.1	General settings for files and folders	156
4.10.2	Custom build tab	157
4.11	Specifying dependencies between projects	158
4.12	Build commands	158
5	Basic debugging features	160
5.1	Selecting the debug instrument	161
5.1.1	Identify your debug instrument	161
5.1.2	Add communication ports	162
5.2	Configuring your target MCU	164
5.2.1	Memory map	164
5.2.2	On-chip peripherals	166
5.3	Running an application	167
5.3.1	Run commands	167
5.3.2	Stepping modes	168
5.3.3	Program and STVD status bar display	169
5.3.4	Monitoring execution in source windows	169
5.4	Editor debug actions	169
5.4.1	Editor debug margin	172
5.4.2	QuickWatch window	174
5.5	Disassembly window	175
5.6	Online assembler	178
5.7	Memory window	180
5.7.1	Viewing memory contents	180
5.7.2	Viewing features	181
5.8	Instruction breakpoints	183
5.8.1	Setting an instruction breakpoint	183
5.8.2	Viewing the instruction breakpoints	184

5.8.3	Setting counters and conditions	184
5.8.4	Showing breakpoints	185
5.9	Data breakpoints	186
5.9.1	Inserting data breakpoints	186
5.9.2	Using the Data Breakpoints window	187
5.10	Call stack window	188
5.11	Local variables window	190
5.12	Watch window	191
5.13	Core registers window	192
5.14	MSCI tools window	194
5.15	Symbols browser	195
5.16	Peripheral registers window	196
5.17	Memory trace window	198
5.18	Online commands	199
5.18.1	The <code>load</code> command	199
5.18.2	The <code>output_file</code> command	199
5.18.3	The <code>stimuli</code> command	199
5.18.4	The <code>symbol-file</code> command	200
5.18.5	The <code>save_memory</code> command	200
5.18.6	The <code>help</code> command	200
5.19	Limitations and discrepancies	200
6	Simulator features	202
6.1	Using the stack control features	202
6.2	Using the input pin stimulator	203
6.3	Using the simulation plotter	207
6.3.1	Plotter selection window	208
6.3.2	Plotter window	209
6.4	Using read/write on-the-fly	220
6.5	Forcing interrupts	221
7	In-circuit debugging	223
7.1	Connecting to and configuring the microcontroller	224
7.1.1	Connecting the hardware for in-circuit debugging	224
7.1.2	Selecting your MCU	225

7.1.3	Ignoring option bytes (ICC only)	226
7.1.4	Configuring option byte settings	226
7.2	Using breakpoints	228
7.2.1	Software breakpoints	228
7.2.2	Hardware breakpoints	229
7.2.3	Setting advanced breakpoints	230
7.2.4	Advanced breakpoint options	232
7.3	Creating a break on trigger input (TRIGIN)	233
7.4	In-circuit debugging in hot plug mode (SWIM only)	233
7.5	In-circuit debugging limitations	234
8	DVP and EMU2 (HDS2) emulator features	237
8.1	Working with output triggers	237
8.2	Using hardware events	237
8.2.1	The hardware event window and contextual menu	238
8.2.2	Adding a hardware event	239
8.3	Trace recording	242
8.3.1	Trace contextual menu	242
8.3.2	Viewing trace contents	245
8.4	Using hardware testing	246
8.5	Logical analyser (EMU2 emulators only)	248
8.5.1	Defining logical analyser events	248
8.5.2	Advanced breaks using the logical analyser	251
8.5.3	Trace filtering using the logical analyser	252
8.6	Stack control window (DVP emulators)	254
8.7	Trigger/trace settings (DVP emulators)	255
8.7.1	Working with output triggers (DVP)	255
9	STice features	258
9.1	Trace recording	258
9.1.1	Trace buffer fields	258
9.1.2	Trace contextual menu	262
9.1.3	Emulator commands	265
9.2	Coverage and profiling	266
9.2.1	Configuring the coverage and profiling settings	267
9.2.2	Running a coverage and profiling session	271

	9.2.3	Reading coverage and profiling results	272
	9.2.4	Typical examples of use	276
10		EMU3 emulator features	284
	10.1	Trace recording	284
	10.1.1	Trace buffer fields	284
	10.1.2	Trace contextual menu	286
	10.1.3	Emulator commands	290
	10.2	Using advanced breakpoints	291
	10.2.1	Defining advanced breakpoints	292
	10.2.2	Memory access events	296
	10.2.3	Other types of events	300
	10.2.4	Enabling advanced breakpoints	301
	10.2.5	Starting with trace ON	301
	10.2.6	The configuration summary	301
	10.2.7	Synoptic representation of advanced breakpoints	302
	10.2.8	Saving and loading advanced breakpoints	304
	10.2.9	Advanced breakpoint examples	304
	10.3	Programming trace recording	309
	10.4	Using output triggers	313
	10.4.1	Trigger programming examples	314
	10.5	Using analyzer input signals	317
	10.5.1	Analyzer input examples	318
	10.6	Performance analysis	319
	10.6.1	Running a performance analysis	319
	10.6.2	Using the Run/Stop/Continue commands	320
	10.6.3	Viewing results	321
	10.7	Read/write on the fly	323
	10.8	Performing automatic firmware updates	324
11		Program	325
	11.1	Configuring general programming settings	325
	11.2	Assigning files to memory areas	328
	11.3	Configuring option byte settings	329
	11.4	Starting the programming sequence	330

12	STM8 C tutorial	332
12.1	Setup	333
12.1.1	Install the toolset and configure the STVD	333
12.1.2	Create a workspace	334
12.1.3	Create a project	334
12.1.4	Add the source files	337
12.1.5	Create a folder to organize files in the project	338
12.2	Build	339
12.2.1	Specify the target STM8	339
12.2.2	Customize the C compiler options	340
12.2.3	Change build settings for a specific file	342
12.2.4	Build the application	343
12.3	Start debugging	344
12.3.1	Start the debugging session	344
12.3.2	Run and stop the application	345
12.3.3	Step through the application	346
12.4	Instruction breakpoints	347
12.4.1	Set an instruction breakpoint	348
12.4.2	Set a counter on an instruction breakpoint	349
12.4.3	Set a condition on an instruction breakpoint	350
12.5	View execution	351
12.5.1	View calls to the stack	351
12.5.2	View and change local variables	351
12.5.3	View variables using the Watch, QuickWatch and Memory windows	352
12.6	Perform memory mapping	355
12.7	Advanced emulator features for EMU3 (ST7) and STice (STM8)	356
12.7.1	View program execution history	357
12.7.2	Use read/write on-the-fly	358
12.7.3	Set an advanced breakpoint	360
12.7.4	Run a coverage and profiling session	363
12.7.5	Run a performance analysis	363
13	ST Assembler/Linker build tutorial	366
13.1	Create a new workspace with the New Workspace wizard	366
13.2	Create a project using Add New Project to Workspace command	367
13.3	Inserting files in your project	368

13.4	Creating a folder to organize files in the project	370
13.5	Project settings	370
13.6	Build a version for debugging	372
13.7	Building a version for programming	373
Appendix A Product support.....		377
A.1	Software updates	377
A.2	Hardware spare parts	377
A.2.1	Sockets.....	377
A.2.2	Connectors	377
A.3	Getting prepared before you call.....	378
Revision history		379

List of tables

Table 1.	Debugging configurations and supported hardware	18
Table 2.	Programming configurations and supported hardware	19
Table 3.	Advanced debugging features	19
Table 4.	Associated documentation	21
Table 5.	Default files and folders for projects	27
Table 6.	Edit menu commands	36
Table 7.	Project menu commands	37
Table 8.	Debug Instrument menu commands	38
Table 9.	Tools menu commands	39
Table 10.	Window menu commands	39
Table 11.	Help menu commands	39
Table 12.	View windows	40
Table 13.	Common contextual menu commands	42
Table 14.	Window-specific contextual menu commands	43
Table 15.	Editor supported file formats	44
Table 16.	Regular expressions	55
Table 17.	Options window tabs	60
Table 18.	Edit and print options	62
Table 19.	Word wrap options	62
Table 20.	Debug options	63
Table 21.	Long line indicator options	63
Table 22.	Line endings options	64
Table 23.	Syntax highlighting options	64
Table 24.	Toolbar options	65
Table 25.	Migration strategy advantages and disadvantages	72
Table 29.	Supported toolsets and file formats	75
Table 30.	Formats for OBSEND output	92
Table 31.	Cosmic C compiler memory models for ST7	96
Table 32.	Cosmic C compiler memory models for STM8	97
Table 33.	Cosmic C compiler debug information options	100
Table 34.	Cosmic C compiler optimization options	101
Table 35.	Cosmic C compiler language options	103
Table 36.	Cosmic C compiler listing options	104
Table 37.	Cosmic C compiler output options	107
Table 38.	Cosmic Assembler debug options	109
Table 39.	Cosmic Assembler language options	109
Table 40.	Cosmic Assembler listing options	110
Table 41.	Cosmic linker output options	115
Table 42.	Raisonance C compiler optimization options	122
Table 43.	Raisonance C compiler language options	123
Table 44.	Raisonance C compiler listing options	124
Table 45.	Raisonance assembler listing options	126
Table 46.	Raisonance linker output options	130
Table 47.	Metrowerks C compiler memory models	134
Table 48.	Metrowerks C compiler optimization options	137
Table 49.	Metrowerks C compiler input options	139
Table 50.	Metrowerks C compiler output options	141
Table 51.	Metrowerks C compiler listing options	142

Table 52.	Metrowerks C compiler language options	143
Table 53.	Metrowerks assembler language options	147
Table 54.	Metrowerks linker optimization options	151
Table 55.	Metrowerks linker output options	153
Table 56.	Toolchain specific tabs	156
Table 57.	Build commands	159
Table 58.	Debug instrument types	161
Table 59.	Memory types	164
Table 60.	Debug menu run commands	167
Table 61.	Editor contextual menu commands	170
Table 62.	Stimuli file syntax	207
Table 63.	Tab contextual menu	211
Table 64.	Display area contextual menu	211
Table 65.	Name column contextual menu	212
Table 66.	Zoom commands	213
Table 67.	Marker commands	215
Table 68.	Plotter tab management commands	216
Table 69.	Item display controls	217
Table 70.	Interrupt mapping	222
Table 71.	ICD-incompatible option byte settings	227
Table 72.	Flash memory limitation on HDFlash devices	235
Table 73.	Advanced breakpoint programming summary	307
Table 74.	Advanced breakpoints summary	307
Table 75.	Advanced breakpoints summary	309
Table 76.	Advanced breakpoints summary	310
Table 77.	Advanced breakpoints summary	311
Table 78.	Advanced breakpoints summary	311
Table 79.	Advanced breakpoints summary	311
Table 80.	Advanced breakpoints summary	312
Table 81.	Advanced breakpoints summary	315
Table 82.	Advanced breakpoints summary	315
Table 83.	Advanced breakpoints summary	317
Table 84.	Advanced breakpoints summary	319
Table 85.	Tutorial sections and applicable debug instrument	332
Table 86.	Advanced breakpoint configuration	361
Table 87.	Document revision history	379

List of figures

Figure 1.	New Workspace window	24
Figure 2.	New project window	25
Figure 3.	MCU selection dialog box	26
Figure 4.	Workspace window	26
Figure 5.	Open window	27
Figure 6.	Workspace window	28
Figure 7.	Project configurations window	29
Figure 8.	Output window Build tab	29
Figure 9.	Debug instrument selection	30
Figure 10.	Debug instrument target port selection	31
Figure 11.	Hot Plug start debug option	31
Figure 12.	Restart with SWIM off option	32
Figure 13.	Programmer window	33
Figure 14.	Main application window	35
Figure 15.	Status bar	35
Figure 16.	Workspace window tabs	42
Figure 17.	Scintilla text editor copyright notice	44
Figure 18.	Editor window file display and editing features	45
Figure 19.	File folding display and controls	46
Figure 20.	Left margin features	47
Figure 21.	Bookmark icons	47
Figure 22.	Status bar elements	48
Figure 23.	Window controls	48
Figure 24.	Contextual menu edit commands	49
Figure 25.	Auto completion pop-up list	50
Figure 26.	Parameter information tip	51
Figure 27.	Auto indentation	51
Figure 28.	Brace matching	52
Figure 29.	Find dialog box	53
Figure 30.	Quick find for strings or variables in a source file	53
Figure 31.	Find/replace window	54
Figure 32.	Find in Files dialog box	54
Figure 33.	Look in additional folders	54
Figure 34.	Find in Files tabs	55
Figure 35.	Go To window	55
Figure 36.	Output window	56
Figure 37.	Output window contextual menu	57
Figure 38.	Output window Build tab	57
Figure 39.	Parser error selection options	58
Figure 40.	Output window Find in Files tab	58
Figure 41.	Output window, Console tab	59
Figure 42.	Console tab contextual menu	59
Figure 43.	Options and Customize windows	60
Figure 44.	Edit/Debug and Styles/Languages tabs	61
Figure 45.	Standard toolbars	65
Figure 46.	Adding new custom toolbars	66
Figure 47.	Options window, Commands tab	67
Figure 48.	Customize option	67

Figure 49.	Adding custom commands	68
Figure 50.	Customized command in Tools menu	69
Figure 51.	Tooltips	69
Figure 52.	Tool description	70
Figure 53.	Generate Information File for Support window	71
Figure 54.	Toolset options	76
Figure 55.	Workspace window	77
Figure 56.	Project parameters	81
Figure 57.	Default project contents	81
Figure 58.	MCU selection tab	86
Figure 59.	ST ASM tab	89
Figure 60.	ST Link tab	90
Figure 61.	ST Post-Link tab	92
Figure 62.	Standard options.	94
Figure 63.	Customizing options	94
Figure 64.	Cosmic C compiler general view	95
Figure 65.	Cosmic C debug information view	100
Figure 66.	Cosmic C compiler optimizations view	101
Figure 67.	Cosmic C language view	103
Figure 68.	Cosmic C compiler listing view	104
Figure 69.	Cosmic C compiler preprocessor view	105
Figure 70.	Cosmic C compiler input view	106
Figure 71.	Cosmic C compiler output view for ST7(left) and STM8 (right).	106
Figure 72.	Cosmic Assembler general view	107
Figure 73.	Cosmic Assembler debug view	108
Figure 74.	Cosmic linker general view	110
Figure 75.	Cosmic linker input view	112
Figure 76.	Cosmic linker output view	115
Figure 77.	Standard options.	117
Figure 78.	Customizing options	117
Figure 79.	Raisonance C compiler general view for ST7 (left) and STM8 (right).	118
Figure 80.	Raisonance C compiler optimizations view.	121
Figure 81.	Raisonance C compiler language view.	122
Figure 82.	Raisonance C compiler listings view	123
Figure 83.	Raisonance C compiler preprocessor view.	124
Figure 84.	Raisonance Assembler general view	125
Figure 85.	Raisonance linker general view	127
Figure 86.	Raisonance linker input view	129
Figure 87.	Raisonance linker output view	130
Figure 88.	Standard options.	132
Figure 89.	Customizing options	133
Figure 90.	General category view	133
Figure 91.	Metrowerks C compiler optimizations view	137
Figure 92.	Metrowerks C compiler input view	139
Figure 93.	Metrowerks C compiler output view	140
Figure 94.	Metrowerks C compiler listing view.	142
Figure 95.	Customizing C language view	143
Figure 96.	Metrowerks Assembler general view	144
Figure 97.	Metrowerks Assembler language view	146
Figure 98.	Metrowerks Assembler listings view	147
Figure 99.	Metrowerks linker general view	148
Figure 100.	Metrowerks linker optimizations view	151

Figure 101. Customizing Metrowerks linker output view	152
Figure 102. Metrowerks linker PRM view	153
Figure 103. Identify debug instrument and connection	161
Figure 104. Add an Ethernet connection	163
Figure 105. Add a USB connection	163
Figure 106. Memory mapping	165
Figure 107. Specifying a new memory zone	166
Figure 108. On-chip peripherals	166
Figure 109. Debugger status on status bar	169
Figure 110. Watch pop-up for variables	171
Figure 111. Editor window margin icons and contextual menu	172
Figure 112. Brown breakpoint icon in Disassembly window	173
Figure 113. Call Stack frame indicator	174
Figure 114. QuickWatch window	175
Figure 115. Disassembly window	175
Figure 116. PC in editor window	176
Figure 117. Program halted at breakpoint	177
Figure 118. “Grayed” breakpoint	177
Figure 119. “Brown” breakpoint	178
Figure 120. Disassembly window contextual menu	178
Figure 121. Online assembler dialog box	179
Figure 122. Online ST7 instruction set	179
Figure 123. Online assembler dialog box	180
Figure 124. Memory window	181
Figure 125. Memory contextual menu	181
Figure 126. Fill Memory dialog box	182
Figure 127. Save Memory to File dialog box	182
Figure 128. Restore memory configuration dialog box	183
Figure 129. Instruction breakpoints window	184
Figure 130. Instructions breakpoints contextual menu	185
Figure 131. Inserting write data breakpoint	187
Figure 132. Data breakpoints window (EMU2)	188
Figure 133. Data breakpoints window (DVP and Simulator)	188
Figure 134. Call stack window and interaction with other debug windows	189
Figure 135. Stepping backwards in call stack	190
Figure 136. Local Variables and Call Stack windows	190
Figure 137. Watch window	191
Figure 138. Watch contextual menu	192
Figure 139. Concurrent IT registers	193
Figure 140. Nested IT registers	193
Figure 141. Simulator time registers (Simulator only)	193
Figure 142. Simulator instruction counter (Simulator only)	193
Figure 143. MSCI Tools	194
Figure 144. Symbols Browser — variables	195
Figure 145. Symbols browser—types	196
Figure 146. Peripheral registers window	197
Figure 147. Peripheral registers contextual menu	197
Figure 148. Forced read warning	198
Figure 149. ST7 Emulation Discrepancies window	201
Figure 150. Stack control window	202
Figure 151. Configuration setup window	203
Figure 152. Define a value and delay for an input signal	204

Figure 153. Define a periodic signal	205
Figure 154. Trigger an input signal on-the-fly	206
Figure 155. Select elements for plotting	208
Figure 156. View plotted elements.	210
Figure 157. No information to plot	213
Figure 158. Scroll controls	214
Figure 159. Select items for export in VCD format	218
Figure 160. Select items for printing	220
Figure 161. Memory and Watch windows with Read/Write on the fly	221
Figure 162. MCU configuration window	225
Figure 163. Setting the option byte values.	228
Figure 164. ST7-ICD emulator advanced breakpoints window	230
Figure 165. Disable instruction breakpoints prompt.	231
Figure 166. Disabled instruction breakpoints.	231
Figure 167. Hardware events window for EMU2 emulator.	238
Figure 168. Hardware event settings window for EMU2 emulator.	239
Figure 169. Hardware event settings window for DVP2	240
Figure 170. Hardware events window (EMU2 emulators)	240
Figure 171. Hardware events window (DVP2 emulators)	241
Figure 172. Hardware events settings dialog box (EMU2 emulators)	241
Figure 173. Hardware events settings dialog box (DVP2 emulators)	241
Figure 174. Trace contextual menu	243
Figure 175. Columns dialog box for DVP emulators	243
Figure 176. Columns dialog box for EMU2 emulators	243
Figure 177. Trace display filter dialog box	244
Figure 178. Save Trace Contents dialog box.	245
Figure 179. EMU2 emulator trace window.	245
Figure 180. DVP2/DVP3 trace window	245
Figure 181. Hardware Test dialog box (DVP version)	247
Figure 182. Hardware test underway (EMU2 emulator version)	247
Figure 183. Logical Analyser mode selection	248
Figure 184. Logical analyser window.	249
Figure 185. Defining an event	250
Figure 186. Event 1 as programmed	251
Figure 187. Defining an advanced breakpoint	252
Figure 188. Defining an advanced breakpoint (continued)	252
Figure 189. Trace filtering options	253
Figure 190. Trace filtering event schematics	254
Figure 191. Stack Control window	254
Figure 192. Configuration setup window	255
Figure 193. Trigger/Trace Settings window	256
Figure 194. DVP1 trigger settings	257
Figure 195. DVP2 trigger/trace settings.	257
Figure 196. STice trace window.	259
Figure 197. Trace contextual menu	262
Figure 198. Trace Display Filter dialog box	263
Figure 199. Saving trace contents	264
Figure 200. Show/hide columns.	265
Figure 201. Customized trace window.	265
Figure 202. Data coverage and occurrence profiling settings window.	268
Figure 203. Code coverage and profiling settings window	268
Figure 204. Code coverage and profiling settings window	270

Figure 205. Code coverage and profiling analysis: functions/instructions view	273
Figure 206. Sort results	274
Figure 207. Data coverage and profiling analysis: data view	275
Figure 208. Data coverage and profiling analysis: source view	276
Figure 209. Lines not covered by test suite	277
Figure 210. Source view of test suite	278
Figure 211. Coverage and Profiling Analysis of test suite after a longer wait	278
Figure 212. Stack memory in Data View tab of the Coverage and Profiling Analysis	279
Figure 213. Stack space details	279
Figure 214. Bottleneck detection: top level	281
Figure 215. Bottleneck detection: intermediate level	281
Figure 216. Bottleneck detection: bottom level	282
Figure 217. EMU3 trace window	285
Figure 218. Trace contextual menu	287
Figure 219. Trace Display Filter dialog box	288
Figure 220. Saving trace contents	289
Figure 221. Show/hide columns	290
Figure 222. Customized trace window	290
Figure 223. Setting timestamp clock	291
Figure 224. Advanced breakpoints window—level 1 programming	293
Figure 225. Defining a counter	294
Figure 226. Defining Level 1, Event 1	295
Figure 227. Then window	296
Figure 228. Memory events	297
Figure 229. Memory Access dialog box	297
Figure 230. Memory Access with Data dialog box	298
Figure 231. Opcode Fetch dialog box	299
Figure 232. Opcode Fetch with Data dialog box	300
Figure 233. Other events	300
Figure 234. Emulator commands in trace contextual menu	301
Figure 235. Advanced breakpoints configuration summary	302
Figure 236. Graphical synoptic sequence	302
Figure 237. Configuration enabled	303
Figure 238. Configuration disabled	304
Figure 239. Programming the memory access	305
Figure 240. Memory access dialog box	305
Figure 241. Programming Then	306
Figure 242. Enabling the configuration	306
Figure 243. Programming memory access with data	308
Figure 244. Configuration summary of program	308
Figure 245. Configuration summary	309
Figure 246. Calling function my_lib	310
Figure 247. Output trigger action modes	314
Figure 248. Programmed window	315
Figure 249. Programming level 1	316
Figure 250. Programming Advanced Breakpoints window	317
Figure 251. Defining events using Analyzer probe signals	318
Figure 252. Specifying an Analyzer probe value with bit mask	318
Figure 253. Performance Analysis dialog box	320
Figure 254. Statistics compiled by performance analysis	321
Figure 255. Graphical representation of performance analysis	322
Figure 256. Memory and Watch windows with read/write on the fly	323

Figure 257. Emulator update window	324
Figure 258. Programmer settings view	325
Figure 259. Programmer, memory areas view	328
Figure 260. Programmer, option byte view	329
Figure 261. Programmer program view	330
Figure 262. Options menus	333
Figure 263. Workspace information	334
Figure 264. Project parameters	335
Figure 265. MCU selection	336
Figure 266. Default project contents	336
Figure 267. Project with source file and dependencies	338
Figure 268. Select the MCU	340
Figure 269. C compiler options for Raisonance	341
Figure 270. C compiler options for Cosmic	342
Figure 271. Exclude the makefile from build	343
Figure 272. Select the debug instrument	344
Figure 273. Activate the Watch Pop-up feature	345
Figure 274. Application is stopped at the cursor position	346
Figure 275. Step over line 242 to line 243	347
Figure 276. View the value of nbOfTransitions	348
Figure 277. Instruction breakpoints	349
Figure 278. Setting counter	349
Figure 279. Setting condition	350
Figure 280. Condition met	350
Figure 281. Call Stack window	351
Figure 282. QuickWatch window	352
Figure 283. QuickWatch window	352
Figure 284. Watch window	353
Figure 285. Memory and Watch windows	354
Figure 286. Watch window	354
Figure 287. MCU configuration	356
Figure 288. Trace contextual menu	357
Figure 289. Saving trace contents	358
Figure 290. Memory and Watch windows	359
Figure 291. Evolution of variables in Watch window	359
Figure 292. Location of bookmarks	360
Figure 293. Setting advanced breakpoints	361
Figure 294. Synoptic view of advanced breakpoint configuration	362
Figure 295. Advanced breakpoint info in trace	363
Figure 296. Setting up a performance analysis	364
Figure 297. Performance analysis results	364
Figure 298. QuickWatch	365
Figure 299. New performance analysis results	365
Figure 300. Create a new project	367
Figure 301. Arrange the files	369
Figure 302. Define a value (-d)	371
Figure 303. Exclude file from build	372
Figure 304. Change the build configuration	374
Figure 305. Enter a post build copy command	375

1 Introduction

This chapter summarizes the features that are detailed later in this manual.

1.1 Building and debugging applications

For building your application, STVD supports ST Assembler-Linker, Cosmic C, Raisonance C and Metrowerks C toolsets. Its graphic interface provides easy access to a range of options when building your application for debugging or programming your microcontroller.

When debugging your application, STVD supports a complete range of in-circuit debugging and emulation hardware. It can also be used as a stand-alone tool, providing a software simulation of your microcontroller's behavior as it runs your application.

Table 1. Debugging configurations and supported hardware

Debugging configuration	Hardware development tools	Application board connection
Simulation	No hardware necessary	
In-circuit debugging (ICC or SWIM)	ST7-ICD (STMC) debugger ST7-DVP3 series emulator ST7-EMU3 series emulator with ICC add-on Raisonance RLink in-circuit debugger/programmer ST7-STICK ST-LINK in-circuit debugger/programmer for STM8 (SWIM) STice advanced emulation system	Connect to the MCU with on-chip Debug Module, installed on your application board.
Emulation	ST7-DVP2 series emulator ST7-DVP3 series emulator ST7-EMU3 series emulator ST7-EMU2 series emulator (formerly HDS2) ST7-STICK STice advanced emulation system	Connect to your application board in place of your microcontroller.

Depending on your microcontroller and hardware configuration you choose, STVD provides you with [Advanced debugging features](#) that make application development easy and fast.

1.2 Programming your microcontroller

When you are ready to program your application to your microcontroller, STVD provides a programming interface based on ST Visual Programming (STVP) software which supports a complete range of hardware for in-circuit, in-situ and socket programming.

Table 2. Programming configurations and supported hardware

Programming configuration	Hardware development tools	Application board connection
In-circuit programming (ICC)	ST7-STICK ST7-DVP3 series emulator ST7-EMU3 series emulator with ICC add-on ST7xxx-EPB Raisonance RLink in-circuit debugger/programmer	Connect to the ST7(2)Fxxxx microcontroller on your application board.
In-situ programming	ST7-DVP2 series emulator ST7-EPB2 series emulator	Connect to the ST72Cxxxx microcontroller on your application board.
Socket programming	ST7SB socket board with an ICP capable programming tool ST7xxx-EPB ST7xxx-DVP and DVP2 series emulators	

1.3 Advanced debugging features

Depending on the debugging configuration you choose, STVD supports a full range of advanced debugging features, including advanced breakpoints and trace recording. [Table 3](#) provides a summary of STVD's advanced debugging features for common debugging configurations.

Table 3. Advanced debugging features

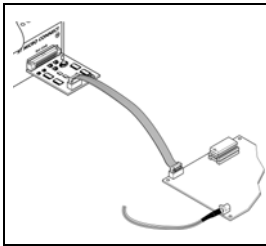
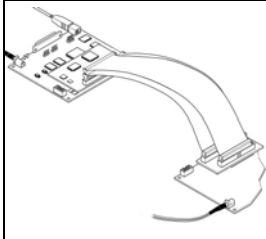
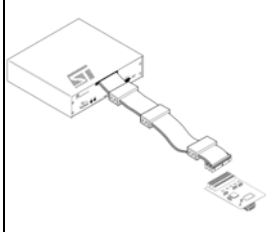
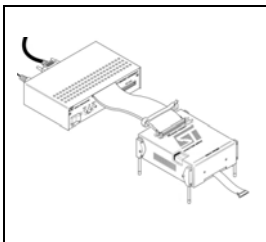
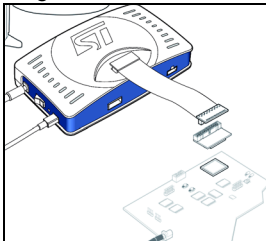
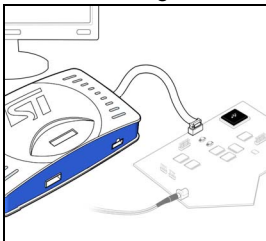
Debugging configuration	Features
Simulator (STVD software only)	<ul style="list-style-type: none"> – Data breakpoints on read/write access – Specify the address for the stack overflow or underflow point – Input stimulator for pin level simulations – Plotter for view the evolution of simulated signals and the values of variables and registers
ST7-ICD with ST7 	<ul style="list-style-type: none"> – Unlimited software-based breakpoints – Hardware breakpoints that can be combined to generate advanced breakpoints, including break on: read/write access, data value at an address, a stack write, Opcode fetch, and some combinations such as an Opcode fetch OR data access at an address.

Table 3. Advanced debugging features

Debugging configuration	Features
ST7-DVP series emulator 	<ul style="list-style-type: none"> – Output signals triggered by advanced breakpoints – Trace recording and trace filtering capability (256 hardware cycles for DVP2, 512 hardware cycles for DVP3) – Specify the address for the stack overflow point – Data breakpoint on address read and/or write access – Hardware test feature for troubleshooting your emulator – In-circuit debugging and programming via ICC connection
ST7-EMU2 series emulator 	<ul style="list-style-type: none"> – Output signals triggered by hardware events – Trace recording and trace filtering capability (1K hardware cycles) – Data breakpoint on address read and/or write access – Logical analyzer controls breaks in application execution or trace recording based on multiple conditions – Hardware Test feature for trouble shooting your emulator
ST7-EMU3 series emulator 	<ul style="list-style-type: none"> – Output signals triggered by advanced breakpoints – Trace recording and trace filtering capability (256K hardware cycles) – Instruction breakpoints on Opcode fetch – Advanced breakpoints control breaks in application execution or trace recording based on multiple, nested conditions – Non-intrusive read/write of variable during emulation with Read/Write on-the-fly – Application performance analysis with graphical display measures real execution time of specified portions of code – In-circuit debugging and programming via ICC add-on
STice emulation configuration 	<ul style="list-style-type: none"> – Advanced breakpoints with up to 4 levels of user-configured conditions – Application profiling for execution time or number of executions at instruction, source code or function level – Unlimited instruction breakpoints for the entire MCU memory space – Control of application memory accesses configurable at byte level – Trace of 128K records with time stamp – Data breakpoints – Non-intrusive read/write to memory and internal registers during emulation – CPU frequency from 250Hz up to 50MHz. – Power supply follower managing application voltages in the range 1.65V to 5.5V (0.8V is possible for MEB with a specific TEB)
STice ICD configuration 	<ul style="list-style-type: none"> – In-circuit debugging/programming via the SWIM protocol

1.4 Getting assistance

For more information, application notes, FAQs and software updates on all the ST microcontroller families, check out the CD-ROM or our web site: www.st.com.

For assistance on all ST microcontroller subjects, or for help using your emulator, refer to [Appendix A: Product support](#).

1.5 Associated documentation

This manual is released with STVD version 4.0.1 and contains some features that are not present in earlier versions of STVD. For information about setting up and connecting the hardware intended for use with STVD, refer to the documents listed in [Table 4](#).

Table 4. Associated documentation

Hardware		Supporting documentation
MCUs	ST7	ST7xxx datasheet
	STM8	STM8xxx datasheet
ICD devices	ST7-ICD (STMC) debugger	ST7-ICD User Manual
	ST7-EMU3 series emulator with ICC Add-on	ST7-EMU3 User manual
	ST7-DVP series emulator	ST7-DVPx Emulator User Manual
	ST7-STICK	ST7 Flash STICK User Manual
	STice advanced emulation system	STice advanced emulation system for ST microcontrollers User Manual
Emulators	ST7-DVP series emulator	ST7-DVPx Emulator User Manual ST7xxx-DVPx Probe User Guide
	ST7-EMU3 series emulator	ST7-EMU3 Emulator User Manual ST7xxx-EMU3 Probe User Guide
	ST7-EMU2 series emulator (formerly the HDS2)	ST7xxx-EMU2x Emulator User Manual
	STice advanced emulation system	STice advanced emulation system for ST microcontrollers User Manual
ICP devices	ST7-STICK	ST7 Flash STICK User Manual
	ST7-DVP3 series emulator	ST7-DVP3 Emulator User Manual
	ST7-EMU3 series emulator with ICC Add-on	ST7-EMU3 Emulator User Manual
	ST7 EPB	ST7 EPB User Manual
	STice advanced emulation system	STice advanced emulation system for ST microcontrollers User Manual

1.6 Conventions

The following conventions are used in the documentation:

- **Bold text** highlights key terms and phrases, and is used when referring to names of dialog boxes and windows, as well as tabs and entry fields within windows or dialog boxes.
- ***Bold italic*** text denotes menu commands (or sequence of commands), options, buttons or checkboxes which you must click with your mouse in order to perform an action.
- The > symbol is used in a sequence of commands to mean “then”. For example, to open an application in Windows, we would write: “Click **Start>Programs>ST7 Tool Chain>...**”.
- `Courier` font designates file names, programming commands, path names and any text or commands you must type.
- *Italicized* type is used for value substitution. *Italic* type indicates categories of items for which you must substitute the appropriate values, such as arguments, or hypothetical filenames. For example, if the text was demonstrating a hypothetical command line to compile and generate debugging information for any file, it might appear as:
`cxst7 +mods +debug file.c`
- Items enclosed in [brackets] are optional. For example, the line:
`[options]`
means that zero or more options may be specified because options appears in brackets. Conversely, the line:
`options`
means that one or more options must be specified because options is not enclosed by brackets.
As another example, the line:
`file1. [o|st7]`
means that one file with the extension `.o` or `.st7` may be specified, and the line:
`file1 [file2...]`
means that additional files may be specified.
- ***Blue italicized*** text indicates a cross-reference—you can link directly to the reference by clicking on it while viewing with Acrobat Reader.

2 Getting started with STVD

Once you have installed and launched **ST Visual Develop (STVD)**, the following section will provide you with information to help you get started building, debugging and programming your application to your target microcontroller.

The following sections will explain how to:

- [Section 2.1: Set the toolset and path information](#)
- [Section 2.2: Create a workspace with a new project](#)
- [Section 2.3: Add source files \(.asm, .c, .s\) to your project](#)
- [Section 2.4: Build your application](#)
- [Section 2.5: Select and connect to your debug instrument](#)
- [Section 2.6: Debug your application](#)
- [Section 2.7: Program your application to your microcontroller](#)

2.1 Set the toolset and path information

When you first open STVD, you should confirm the default toolset and path information that will be used when building your application. STVD will automatically set path and subpath information for standard installations of the supported toolsets. However, you should confirm that these paths are correct for your particular installation.

Note: Even though you have specified these default settings in STVD, which are applied to all projects using a toolset, you can still change the toolset and paths for a specific project in the General tab of the Project Settings window (see [Section 4.5: Configuring project settings on page 84](#)).

To change default toolset paths:

1. Select **Tools > Options**.
2. In the **Options** window click on the **Toolset** tab.
3. Select your toolset from the **Toolset** list box.
If the path is incorrect you can type the correct path in the **Root Path** field, or use the browse button to locate it.
4. In the subpath fields, type the correct subpath if necessary.
These paths are relative to the toolset path in the Root Path field.

Note: For Metrowerks toolset users, some versions use the filename “prog” instead of “bin” for the binary path.

2.2 Create a workspace with a new project

The workspace is a generic, global environment for STVD, which can contain one or more projects and their dependencies. Projects, however, contain the specific information for building your application including the toolset information, build configuration and target device information. When you started debugging, the project also stores the debug context each time you save it.

Note: Workspace files for previous versions of STVD (.wsp) can be opened in STVD 3.0 and later versions. For information on migrating your workspaces to the new workspace format, refer to [Migrating old workspaces \(STVD7 2.5.4 and prior versions\) on page 71](#).

With the **Create Workspace with New Project** wizard you can quickly and easily:

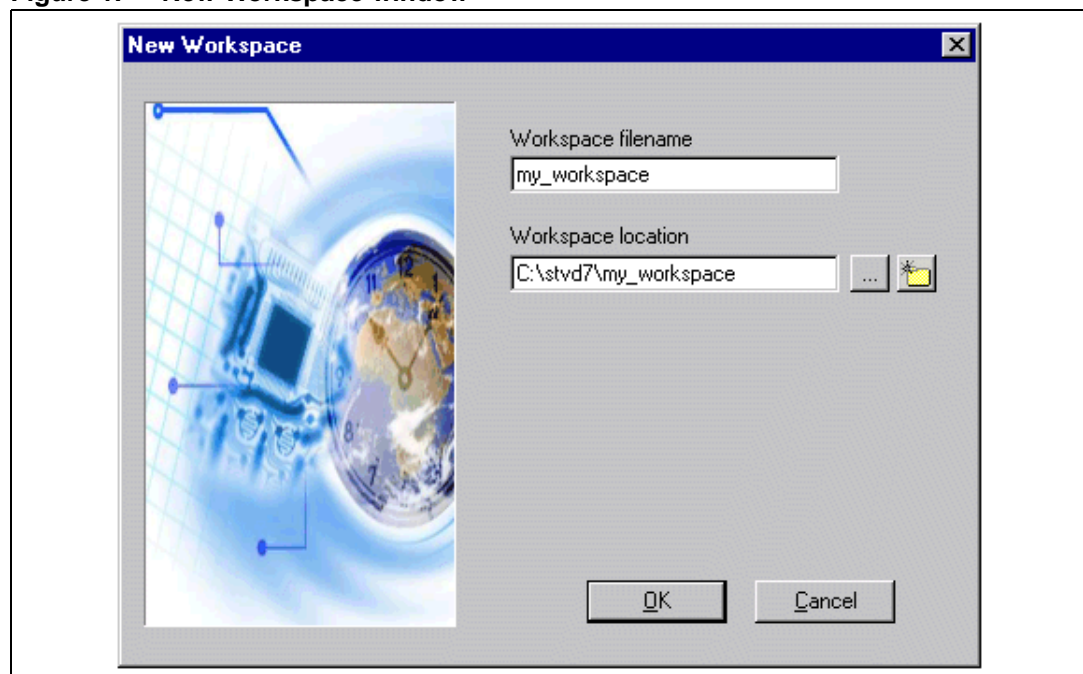
- [Create a new workspace](#)
- [Create a project in your workspace](#)
- [Specify a target MCU for your project](#)

2.2.1 Create a new workspace

Create a new workspace by selecting **File > New Workspace**. The **New Workspace** window opens. Here, select the **Create Workspace and Project** icon and click on **OK**.

Note: For information on creating workspaces from a project, an executable file or a makefile, refer to [Section 4.2.2: Creating a new workspace on page 78](#).

Figure 1. New Workspace window

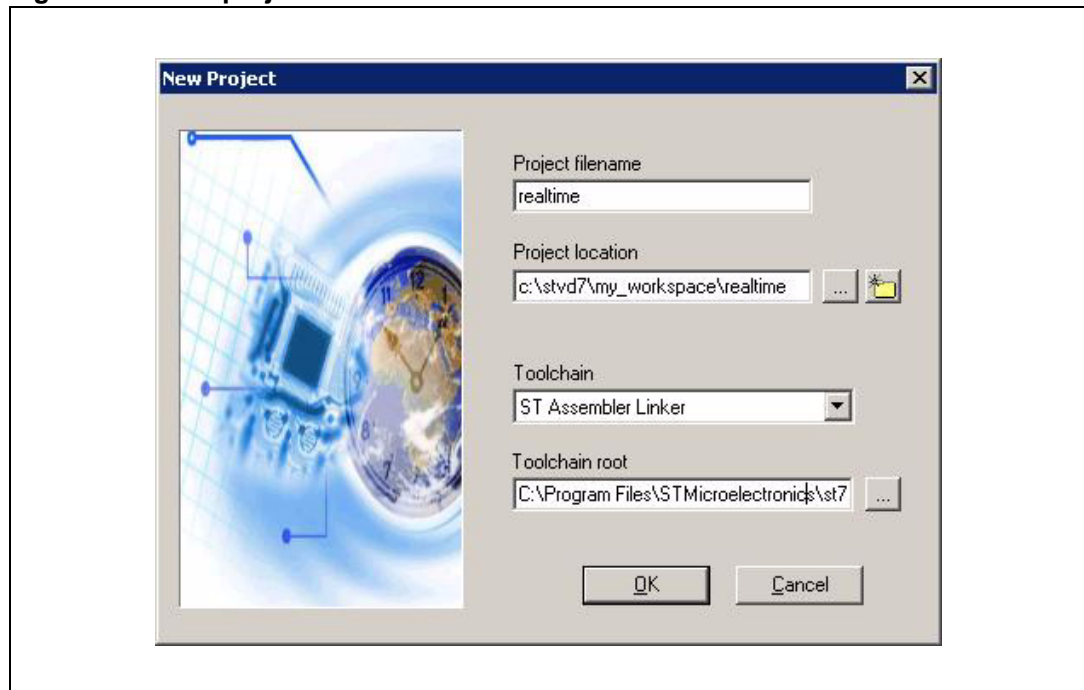


In the **New Workspace** window:

1. Enter the name in the **Workspace Filename** field.
2. Enter the pathname where you want to store your workspace by typing it, or using the **Browse** button to locate a file. Click on the **Create a New Folder** button if you need to create a folder for your workspace, then enter the new folder name in the pop-up window.
3. Click on **OK**, and the **New Project** window opens.

2.2.2 Create a project in your workspace

Figure 2. New project window



In the **New Project** window:

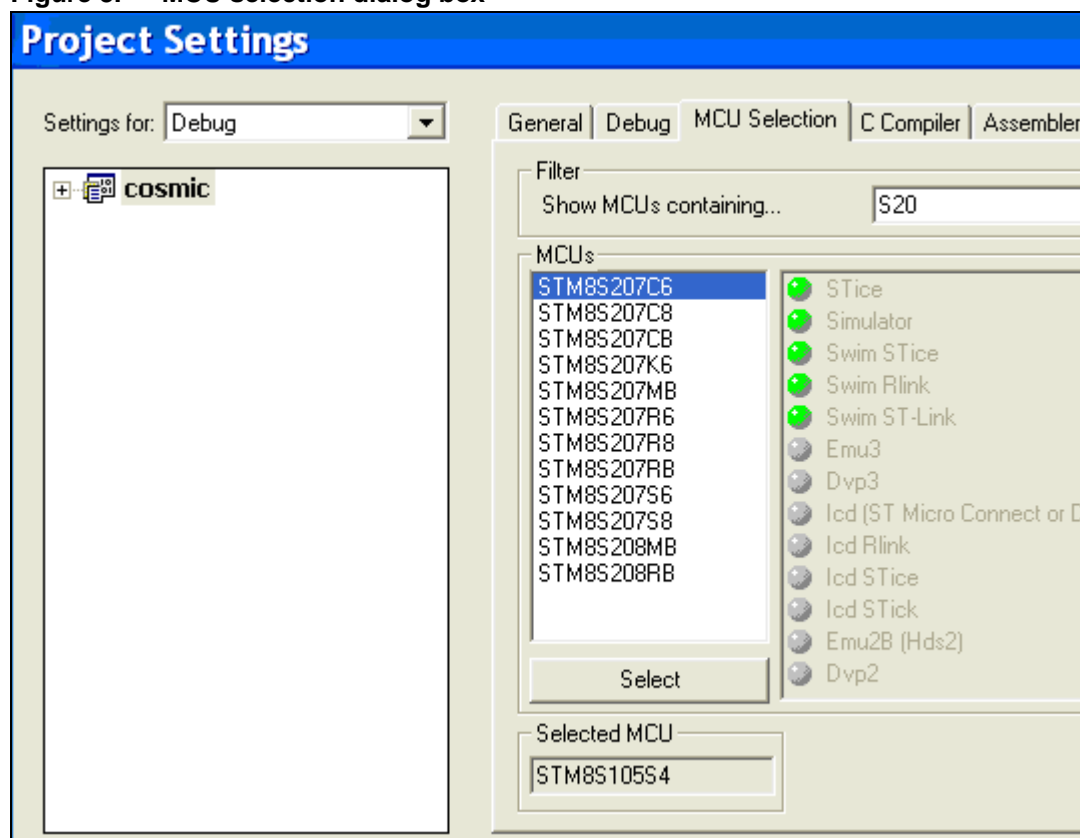
1. Enter the name of your project in the **Project Filename** field.
2. Type the pathname where you want to store it.
By default, this is the pathname you entered for your workspace.
3. Select your toolset (ST Assembler/Linker, Cosmic C or Metrowerks C) from the **Toolchain** list box.
4. Enter the pathname for your toolset by typing the path or using the **Browse** button to find it.
5. Click on **OK**.
The **Select an MCU** dialog box opens.

2.2.3 Specify a target MCU for your project

In the **MCU Selection** dialog box:

1. Choose the target device for your application from the complete list of supported MCUs.
2. Click on **Select**.
The name of the MCU appears in the **Selected MCU** field.
3. Click on **OK**.
Your new workspace and project are created. Only the project is associated with the selected MCU. Your workspace can contain multiple projects that are to be built for different target MCUs.

Figure 3. MCU selection dialog box



Note: If necessary, you can change the MCU selection later in the [MCU selection tab](#) of the **Project Settings** window.

RESULT

A project file (.stp) and its associated folders have been created in your workspace and appear in the **Workspace** window. Depending on your toolset, the folders and files listed in [Table 5](#) are created in your project by default.

Figure 4. Workspace window



Table 5. Default files and folders for projects

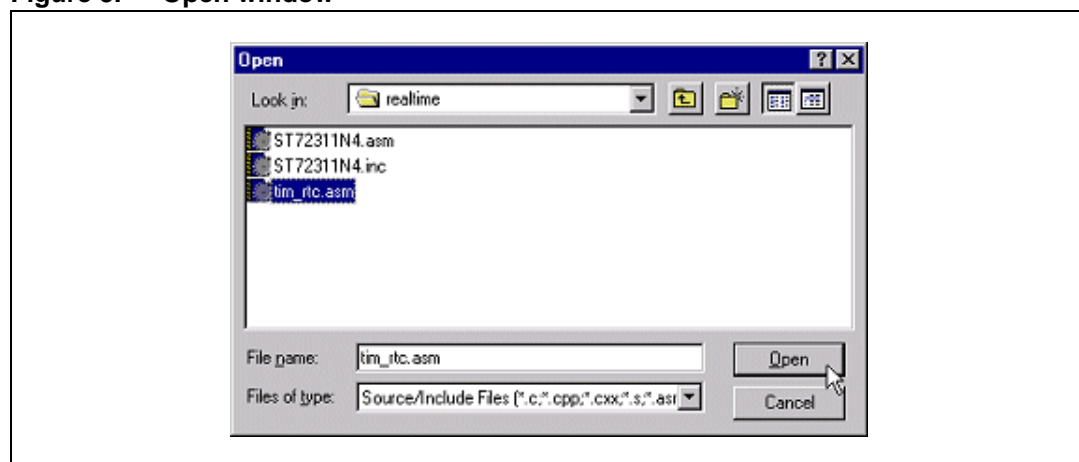
Project default Folders	Files	ST Assembler Linker	Cosmic C	Metrowerks C	Raisonance C
Source files		X	X	X	X
	interrupt_vector.c		X		
	vector.c			X	
	main.c		X	X	X
Include files		X	X	X	X
External dependencies		X	X	X	X
	modm.h		X		

2.3 Add source files (.asm, .c, .s) to your project

Source files (.asm, .c, .s) contain the code for building your application. To add the application source files to your project:

1. Right-click on the **Source Files** folder, then select **Add Files to Folder** from the contextual menu.
A browse window opens, allowing you to find the source files to add to your project.
2. Highlight the file(s) and click on **Open**.

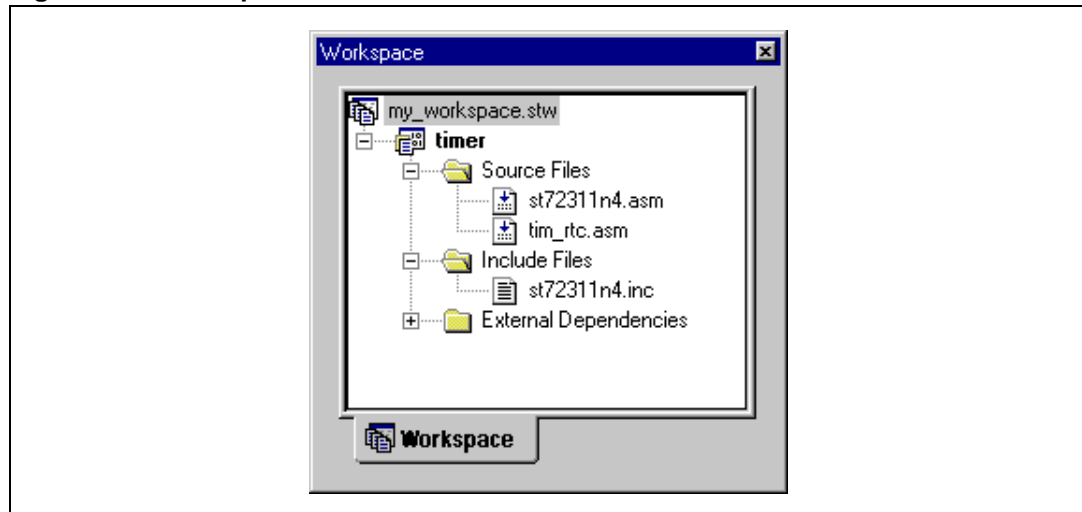
Figure 5. Open window



RESULT

Your project's **Source Files** folder now contains the source code that you want to use to build your application.

Figure 6. Workspace window



Caution: When building with ST Assembler Linker, your application source files may need to be assembled in a specific order. To change the order of the files, you need to disable the **Add Sorted Elements** option in the workspace contextual menu. For more information, refer to [Ordering source files before assembly on page 88](#).

2.4 Build your application

Once you have created your project, you are in the **Build context** by default. You have access to all the commands necessary for setting up, customizing and building your application. You do not yet have access to the commands for debugging your application, or programming it to your microcontroller.

In the build context, you can select default settings for building your application in order to **debug** it, or to generate a **release** version to be programmed to your microcontroller.

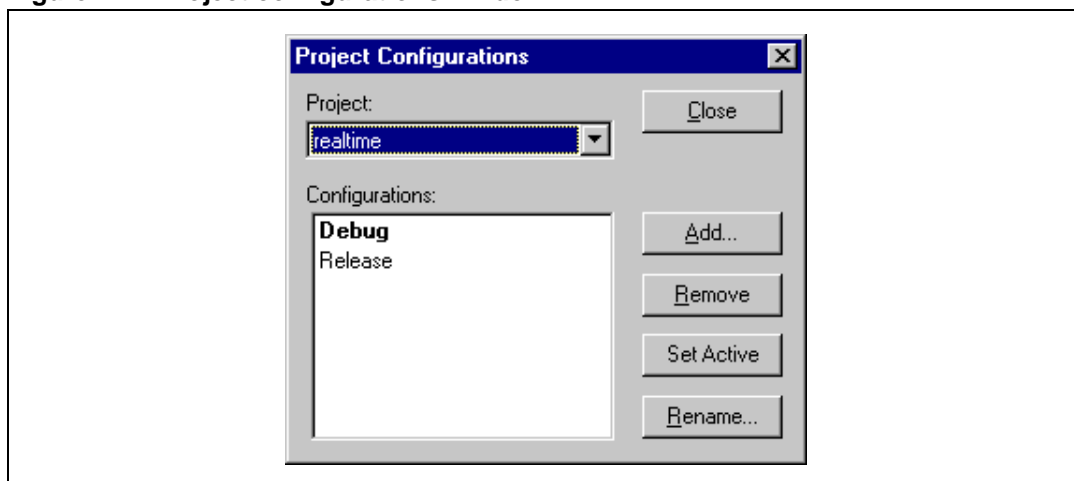
In addition, you have access to the interface for customizing the building of your application and identifying the **target microcontroller**. When using STVD, you must identify the target MCU to which you will program your application.

2.4.1 Configuring project settings for Debug or Release

Build configurations allow you to change project settings quickly and easily. STVD has two preset configurations:

- **Debug:** creates a version of your application that allows you to use all of STVD's advanced debugging features. When using this configuration, output files are saved in the **Debug** folder in your workspace directory.
 - **Release:** creates a version of your application using the default optimizations for your toolset. This version of your application is ready to be programmed to your target microcontroller. Output files are saved in the **Release** folder in your workspace directory.
1. To choose a build configuration, select **Build > Configurations**. The **Project Configurations** window is displayed.
 2. Highlight the configuration you want to use and then click on **Set Active**. Click on **Close**, to close the window.

Figure 7. Project configurations window

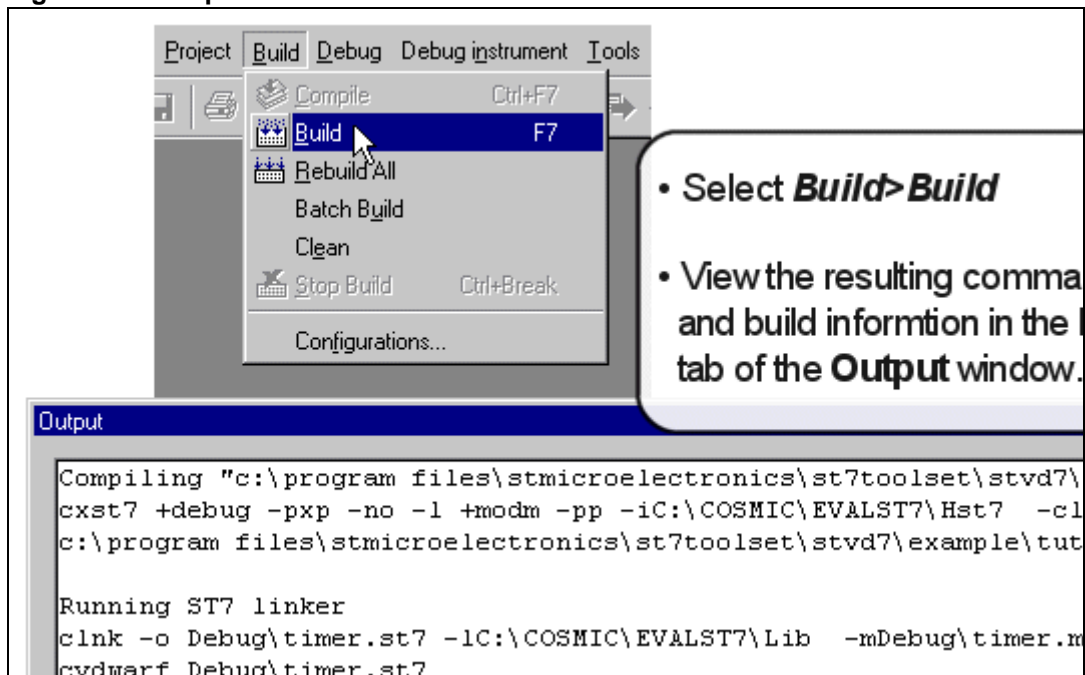


- To view the project settings that have been applied, select **Project > Settings**. The **Project Settings** window opens and you can view the options for your toolset's Compiler, Assembler and Linker, among others. You can also customize these options. For more information about the available options for your toolset, refer to [Section 4.5: Configuring project settings on page 84](#).

2.4.2 Build commands and output

Once you have selected your MCU and configured all the options for building, use the commands in the **Build** menu to build and rebuild your application, or compile your source files. When STVD builds your application the command lines and any warnings, errors or other build information are displayed in the **Build** tab of the **Output** window.

Figure 8. Output window Build tab



2.5 Select and connect to your debug instrument

STVD allows you to debug your application using the Simulator (software only), or a range of debug instruments (emulators and in-circuit debugging hardware).

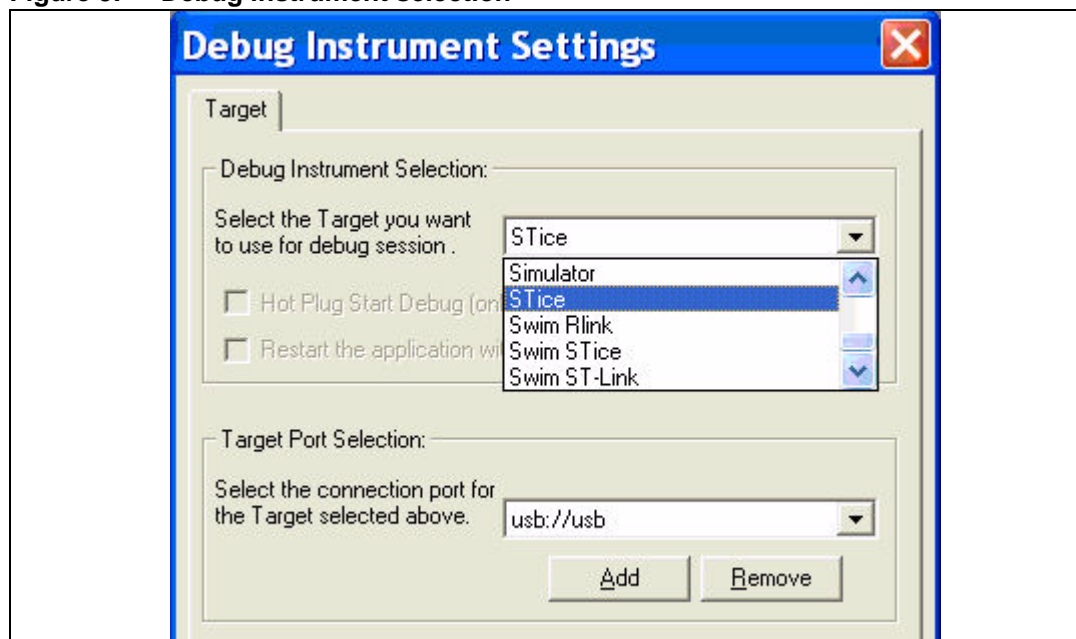
1. Once you have built your application for debugging, select **Debug Instrument > Target Settings**.

The **Debug Instrument Settings** window will open.

2. In this window, use the **Debug Instrument Selection** list box to select your debugging hardware (emulator or other device), as shown in [Figure 9](#).

This list box contains the list of debug instruments that support your target microcontroller.

Figure 9. Debug instrument selection



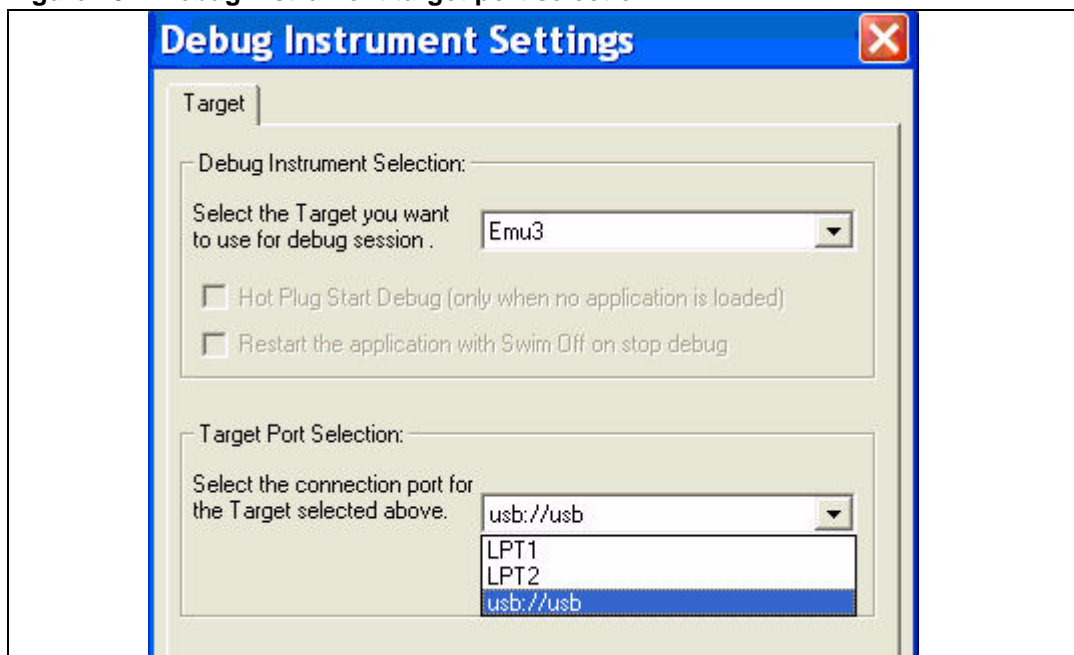
3. Use the **Target Port Selection** list box to identify the port (parallel, USB, or Ethernet) that your debug hardware is connected to, as shown in [Figure 10](#).

To add a port that is not in the list, click on **Add** to open the **Add Connection** dialog box. For more information about adding a connection, see [Section 5.1: Selecting the debug instrument on page 161](#).

4. To confirm your selection, click on **Apply**. Close the window by clicking on **OK**.

When you start your debugging session (**Debug > Start Debugging**), STVD will ask you to confirm your selection and then establish the connection with your debug instrument.

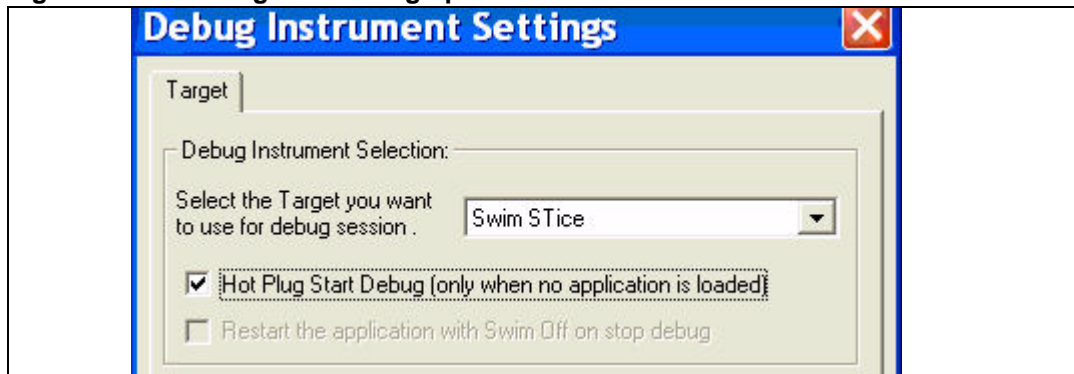
Figure 10. Debug instrument target port selection



The SWIM debugger has two additional parameters to define the target connection:

1. Out of workspace, you can connect in Hot Plug mode, which means the microcontroller is not put under reset while initializing the connection. For more details on Hot Plug mode refer to [Section 7.4: In-circuit debugging in hot plug mode \(SWIM only\)](#).

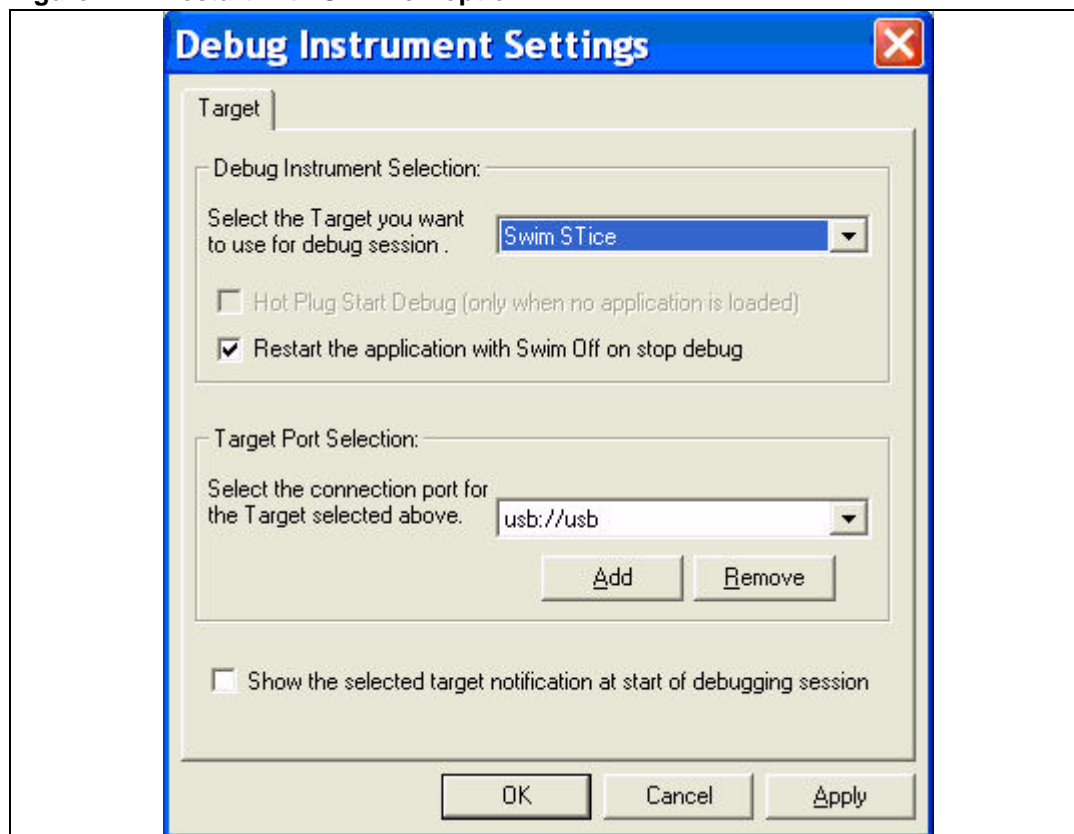
Figure 11. Hot Plug start debug option



2. When you are inside a workspace and stop debug, by default STVD removes the software breakpoints (if any, and if possible) and makes the application restart with the SWIM module OFF on the microcontroller side (see [Figure 12](#)). You may change this default behavior by unchecking the **Restart Application** checkbox.

Caution: You must choose the behavior before the target is connected (start debug), even though the behavior only affects the target disconnection (stop debug). Software breakpoints, if any, remain in Flash memory until reprogrammed. After a reset the microcontroller needs a SWIM host before it can continue. A power-on reset is required to escape this state.

Figure 12. Restart with SWIM off option



2.6 Debug your application

In the **Debug** context, STVD provides access to the commands and windows that are specific to debugging applications, and that are specific to your debugging instrument.

Select **Debug > Start Debugging** to change from the Build context to the Debug context.

RESULT:

The commands in the **Build** menu are no longer available. Instead, commands specific to debugging, view windows for debugging and emulator features are now available.

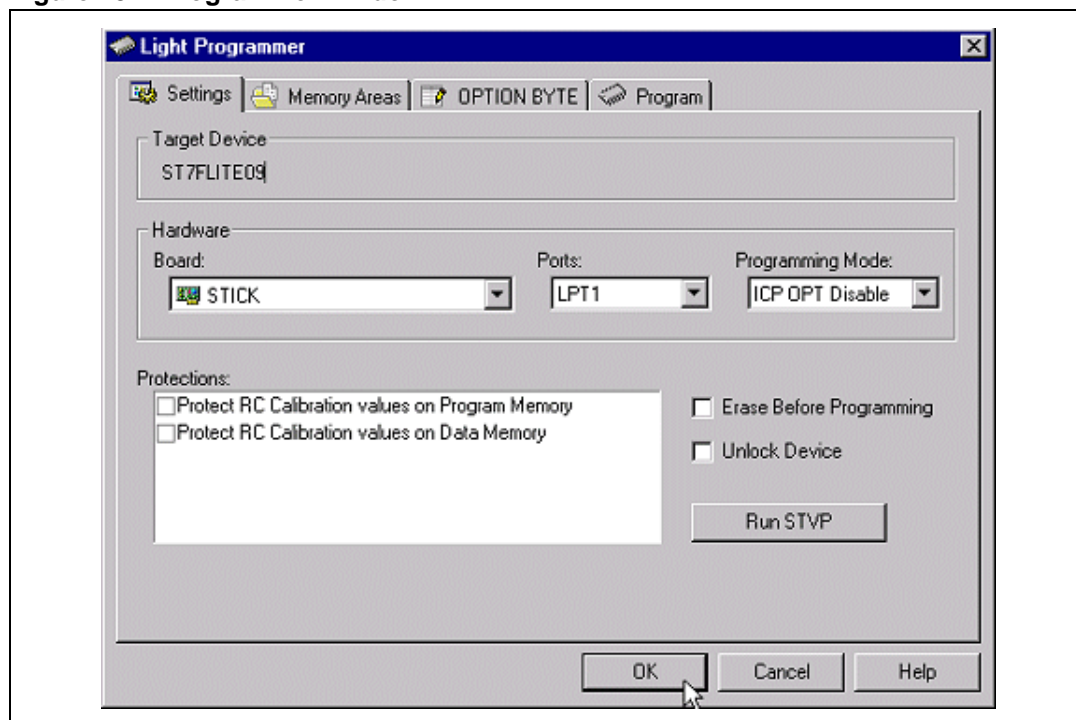
For more information about debugging features, you can refer to [Section 5: Basic debugging features on page 160](#).

2.7 Program your application to your microcontroller

Once you have finished debugging your application and have built the final version, you can program it to your target microcontroller using STVD's **Programmer**.

To access the Programmer, select **Tools > Programmer** from the main menu bar. The **Programmer** window opens.

Figure 13. Programmer window



From this window, you have interfaces that allow you to:

- Select your programming hardware and mode
- Identify files to be programmed to specific Memory areas
- Configure the option bytes for your Flash microcontroller
- Monitor the status while your application is being programmed to your microcontroller

For a complete description of features, refer to [Section 11: Program on page 325](#).

3 Your STVD graphical environment

These sections describe the layout of STVD: the various menus, toolbars and status indicators that will help you, whether you are building, or debugging your application. In particular the following sections provide information about:

- [Section 3.1: The main application window](#)
- [Section 3.2: The main menus and their commands](#)
- [Section 3.3: View windows](#)
- [Section 3.4: Workspace window](#)
- [Section 3.5: Editor windows](#)
- [Section 3.6: Output window](#)
- [Section 3.7: Customizing your work environment](#)
- [Section 3.8: Tooltips](#)
- [Section 3.9: Help and support features](#)
- [Section 3.10: Migrating old workspaces \(STVD7 2.5.4 and prior versions\)](#)

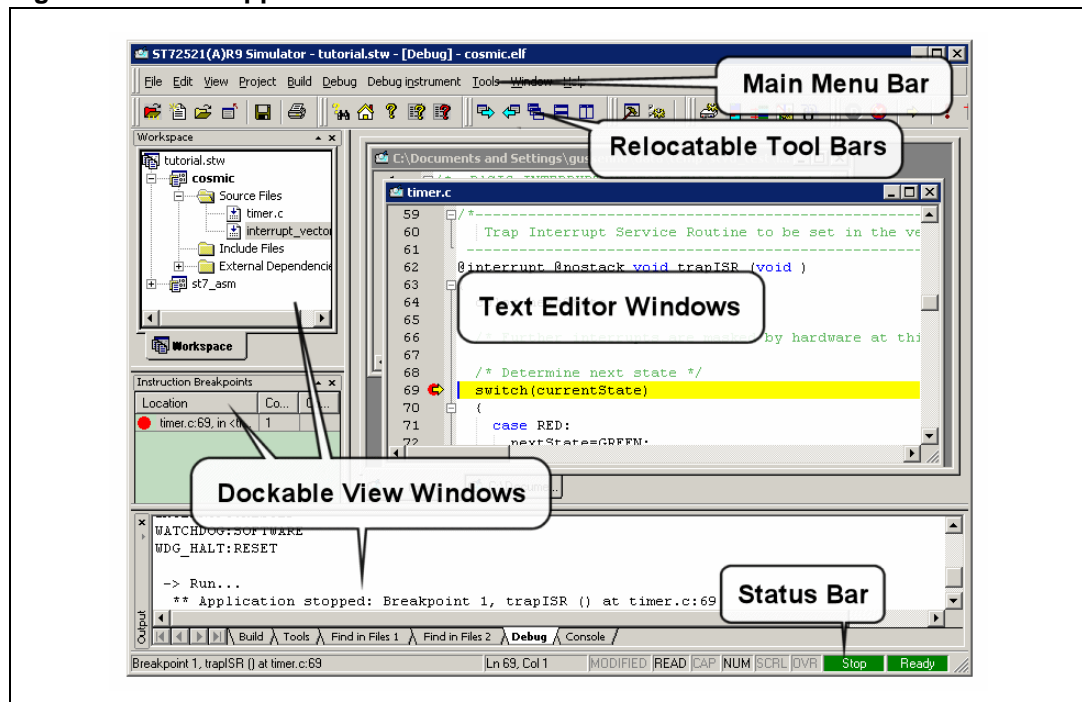
3.1 The main application window

[Figure 14](#) shows STVD's main application window. An application has been opened and several of the debugging windows are seen in a typical configuration. This is one possible arrangement of window elements; you can change the position of all windows, open or close individual elements, change the windows that are visible at startup, and save the configuration.

The different areas on this screen are:

- **Main menu bar:** The main menu bar is always visible. It contains pull-down menus to access the principal STVD functions (see [Section 3.2: The main menus and their commands](#)).
- **Relocatable toolbars:** Under the main menu, control buttons are grouped by subject into relocatable toolbars (default position). Toolbars may be shown or hidden independently. The contents of each toolbar can be user-defined, and new toolbars created as required (see [Section 3.7.2: Customizing toolbars on page 65](#)).
- **Text Editor windows:** You can open up one or all of your application's source files in text editor windows. These windows can be cascaded, floated, minimized or maximized (see [Section 3.5: Editor windows on page 43](#)).
- **Docking windows:** The various STVD view windows are dockable for greater ease of use when several windows are open at the same time. They may be moved to different docking positions, undocked and placed independently on the desktop, or free-floating within the main STVD window area. The name of each window appears in a caption at the top of the window (see [Section 3.5: Editor windows on page 43](#)).
- **The Status bar:** Displays current attributes associated with the **Edit** window and STVD status.

Figure 14. Main application window

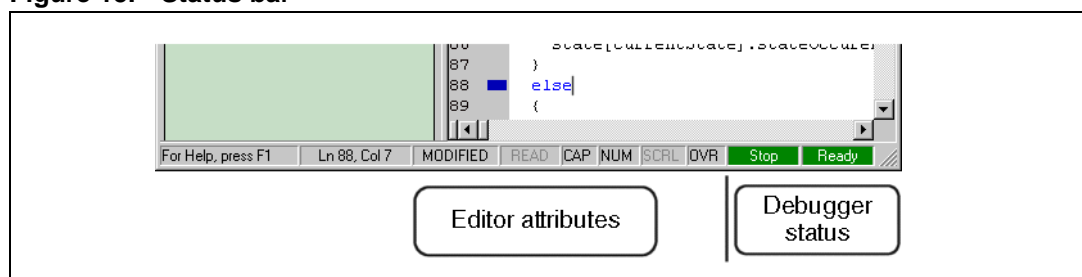


Status bar

The status bar in the lower frame of the STVD main application window contains useful information about editor attributes and debugger status.

Editor attributes such as keyboard states (for example CAPS On/Off, Type Over On/Off) are shown in the center of the bar. Also displayed are attributes of the active file in the Edit window, such as the current location of the cursor in the editor window, and whether the file has been modified or not. These attributes and their functions are described in more detail in [Editor status bar information on page 48](#).

Figure 15. Status bar



The two right-most panels of the status bar show the current status. **Stop** and **Ready** shown in green mean that the program is halted and the debugger awaits further instruction. **Run** and **Debugger**, shown in red and orange, respectively, mean that the program is currently running on the debugger.

Note: The status bar does not explicitly indicate whether you are in the Build or the Debug context. The Stop and Ready messages appear when you are in either context. However the Run and Debugger messages only appear when you are in the Debug context.

3.2 The main menus and their commands

STVD's menus provide a range of common commands for general activities such as opening projects, or loading, editing and saving files, as well as access to build and debug commands.

File menu

The **File** menu provides standard commands for managing workspace and text files, including: New, Open, Close, Close All, Save, Save as, Save All, Print and Recent.

The **New Workspace** and **Open Workspace** commands are very important because before you start creating projects, building or debugging your application, you **must** create or open a workspace. For more information, refer to [Loading and creating workspaces \(.stw\) on page 76](#).

Edit menu

The Edit menu provides file editing and text search commands that are available whenever an Editor window is open, in both the Build and Debug context. In addition, this menu provides breakpoint commands, and access to the **QuickWatch** window, which you will use in the Debug context.

This section provides a summary of available commands. For more information about editing features and commands, refer to [Editor windows on page 43](#). For more information about using debug commands from the Editor windows, refer to [Basic debugging features on page 160](#).

Table 6. Edit menu commands

Command	Description
Undo/Redo	Undoes the last edit (text processing) command, or redoes last undone command.
Cut, Copy, Paste	Standard clipboard operations.
Find..., Find Next and Replace	Finds and/or replaces text in the text editor. The <i>Find</i> command is also available in the contextual menu.
Go to	Opens the Go To window. Define a line, address or function to go to.
Find in Files	Finds a string within any of the files in a given directory.
Breakpoints	Inserts breakpoint symbols into the source code in the active Edit window. The program can then be run taking into account the symbols that you inserted. For details, see Editor debug actions on page 169 .
Bookmarks	Insert/remove and navigate through the bookmarks in the margins of the current source file.
QuickWatch	Opens a QuickWatch window , which allows rapid access of common watch functions.
Refresh	Refreshes all windows.
Match Brace	Goes to the brace that corresponds to the brace you have highlighted in the Editor window.

Table 6. Edit menu commands (continued)

Command	Description
Complete Word	Opens the auto completion pop-up for a list of context specific keywords to complete the word you are typing.
Parameter Info	Opens the parameter information pop-up to see the syntax for the ASM instruction that you are typing.

View menu

The commands in the **View** menu open the **Workspace**, **Output** and **Instruction Breakpoint** windows when you are in the Build or the Debug context. When you are in the Debug context this menu also allows you to open the different view windows for viewing disassembled code, registers, memory, stack and variables. These views are not available when you are in the Build context. They are explained in greater detail in [Basic debugging features on page 160](#).

This menu may offer some different commands depending on your debug instrument. For information about commands that are specific to your debugging hardware, refer to the sections on target-specific debugging features.

Project menu

The **Project** menu gives you access to all of the following project-related commands and activities.

Table 7. Project menu commands

Command	Description
Set Active Project	Allows you to select which project you want to make active for building or configuration.
Insert Files into Project	Opens a browse window from which you can locate an existing file to insert in the active project (see Section 4.3.3: Adding and removing folders and files on page 83).
Dependencies...	Allows you to set dependencies among the projects in the current workspace (see Section 4.12: Build commands on page 158).
Settings...	Opens the Project Settings window (see Section 4.5: Configuring project settings on page 84).
Export Makefile...	Generates a makefile script that can be launched with the gmake utility, separately from STVD full IDE.
Insert Project into Workspace	Opens a browse window from which you can locate an existing project to insert in the current workspace (see Section 4.3.1: Loading an existing project on page 80).
Add New Project to Workspace	Allows you to create a new project and add it to the current workspace (see Add a project (.stp) to your workspace on page 81).
Remove Project from Workspace	Deletes the active project from the workspace.

Build menu

The **Build** menu contains the commands that allow you to set the configuration for building your application, and to start and stop the build. These commands, with the exception of **Compile** and **Batch Build**, apply to the active project in your workspace. **Compile** applies to the source file that you have selected in the **Workspace** window. **Batch Build** applies to the projects that you identify in the **Batch Build** window. For more information, see [Build commands on page 158](#).

Note: You do not have access to the commands in this menu until you have created a workspace with a project.

Debug menu

This menu provides access to the commands associated with running and stopping a loaded program (**Run**, **Restart**, **Continue**, **Run to Cursor** and **Stop**), single stepping through it (**Step Into**, **Step Over**, **Step Out**), and the commands **Go To PC**, and **Set PC**. These commands are specific to running your program while debugging. For more information about using these commands refer to [Running an application on page 167](#).

Debug instrument menu

This menu gives you access to options that are specific to your target debugging hardware. As a result, the menu contents will change when you start a debugging session, depending on the features supported by the debugging tool that you have specified.

Before you identify your debug instrument, this menu contains the following command by default.

Table 8. Debug Instrument menu commands

Command	Description
Target Settings	Opens the Debug Instrument Settings window, which allows you to choose your debug instrument and configure the parallel, USB or Ethernet connection (see Selecting the debug instrument on page 161).

For more information about other commands that can appear in this menu, refer to the debugging features that are specific to your debug instrument.

Tools menu

The **Tools** menu offers customization and setup options for defining the look and layout of STVD and to set a number of user options. The options of the **Tools** menu are described in [Table 9](#).

Table 9. Tools menu commands

Command	Description
Customize	Allows you to specify functions to add to the Tools menu (see Adding custom commands on page 67).
Options	Allows you to customize user options including: <ul style="list-style-type: none"> – Toolbars (see Creating user-defined toolbars on page 66) – Commands (see Rearranging toolbar icons on page 67) – Edit/Debug options (see Customizing editor features on page 61) – Workspaces (see Customizing your work environment on page 60)
Programmer	Opens the Programmer interface that you will use to program the release version of your application to your target MCU (see Program on page 325).

Window menu

This menu gives you access to commands for arranging and navigating in open [Editor windows](#).

Table 10. Window menu commands

Command	Description
Next/Previous	Cycles through the open Editor windows and any other windows that are floating inside the main application window.
Cascade/Tile	Arranges all open windows within the main application area according the option selected. Independent and docked windows are not affected.
Arrange Icons	Arranges any minimized window icon.
Status Bar	Controls the displaying of the status bar.
List of opened files	Creates a list of all files in the Main window. A checkmark indicates which is the active window. A single left-mouse-click on any of the listed windows sets it as the active window and brings it to the front of the group in the main application area. Double-click on the filename to view.

Help menu

This menu gives you access to commands for accessing help features.

Table 11. Help menu commands

Command	Description
Search	Allows you to search the contents of the online Help.
Help Home Page	Opens the online Help Home Page.
About...	Provides version information about STVD and your target debugging instrument.
Help On Instruction...	Opens the Index to the Instruction Set online assembler reference pages.

Table 11. Help menu commands (continued)

Command	Description
Instruction Set Contents...	Opens the Table of Contents of the Instruction Set online assembler reference pages.
Generate Support File	Opens the Generate Information File for Support window for outputting log files that you can send to support when seeking assistance after an application crash.

For details about these features, refer to [Help and support features](#) on page 70.

Contextual menus

STVD provides contextual menus in the view, editor and workspace windows. Contextual menus contain commands that are specific to a window or STVD feature. In some cases, commands are only accessible in the contextual menu and not the main menu. You can access contextual menus for a window by right-clicking within that window. To find out more about the commands that are available in a contextual menu, refer to the section that describes the feature or window that you are interested in.

3.3 View windows

Within STVD's main window, positionable view windows are provided to give you easy access to debugging information and features. These view windows are specific to debugging features and activities. A list of these windows and summary description is provided in [Table 12](#).

Table 12. View windows

Command	Description
Workspace window	Allows you to manage your project, add projects and source files and define dependencies. (refer to Workspace window on page 41).
Output window	Contains tabs that display messages and information related to specific activities or features. For example, the Build tab displays the commands errors and messages that are the result of building your application. In addition, the Console tab is a powerful tool that allows you to view the commands sent by the core debugger when you use an STVD feature, or bypass the GUI and send commands directly to the core debugger (refer to Output window on page 56).
Disassembly window	Displays disassembled application code (refer to Disassembly window on page 175).
Register windows	
Core registers	Displays the contents of all internal registers (Accumulator, registers X and Y, Program counter, Stack pointer and Condition code register) at the location of the program counter (refer to Core registers window on page 192).
Peripheral registers	Displays all the peripheral registers of the target MCU and their values (refer to Peripheral registers window on page 196).
Memory window	Allows you to monitor the contents of the microcontroller's entire memory (refer to Memory window on page 180).

Table 12. View windows (continued)

Command	Description
Instruction breakpoints window	Provides an interface for activating and inactivating instruction breakpoints and applying conditions and counters to them (refer to Instruction breakpoints on page 183).
Watch window	Displays the current value of selected registers or variables in decimal, hexadecimal or binary format (refer to Watch window on page 191).
Call stack window	Allows you to keep track of stack utilization, by displaying the function calls that are stored in the microcontroller's Stack (refer to Call stack window on page 188).
Local variables window	Displays program variables that are local to a function, as well as their values at particular points in the running of the application (refer to Local variables window on page 190).
Symbols browser window	Allows you to find functions in the application files (refer to Symbols browser on page 195).

Docking view windows

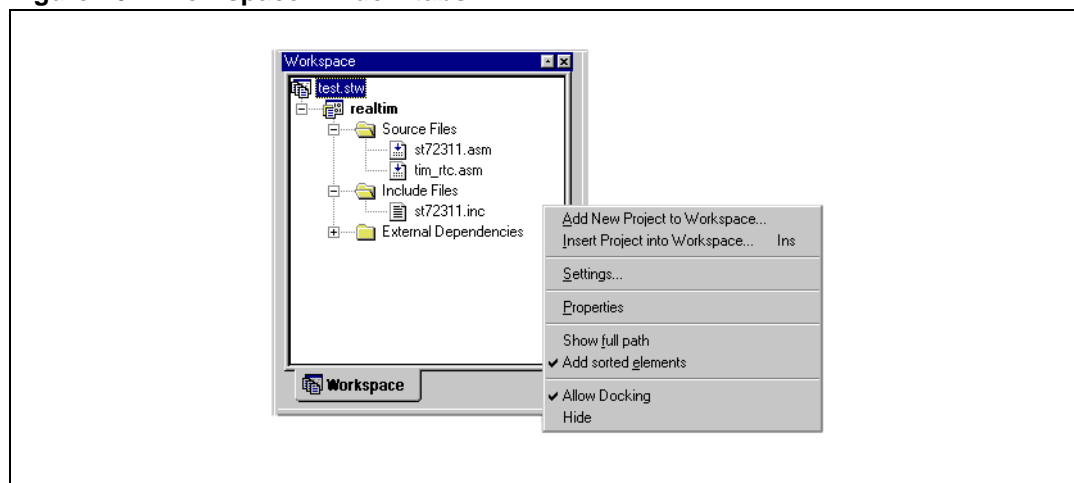
All of the View windows (the windows accessible from the **View** menu or toolbar) are **docking windows**.

Docking for each window is turned on/off from the contextual menu associated with that window (except for **Editor** windows—see [Editor windows on page 43](#) for more information). **Allow Docking** turns docking on/off for that window. When docking is on, a black rectangular outline presents possible docking positions as the window is moved. When released the window fits into the docking position currently outlined. A docked window always has one or more of its sides contiguous with the frame of the STVD main application window.

3.4 Workspace window

The **Workspace** window shows a structured view of your project. The workspace is represented in a typical directory structure. The root item is the workspace (`.stw`), and its nodes are the projects (`.stp`) that contain the source files for your application.

Figure 16. Workspace window tabs



The project file contains the target file name, the settings for building the project and the files needed to build the application. Files can be organized in sub-folders. By default, a new project contains three folders: **Source files**, **Include files** and **Dependencies**.

To open the Workspace window, either click on the Workspace window icon  in the View toolbar or, from the main menu, select **View>Workspace**.

Workspace contextual menus

From the **Workspace** window you have access to the contextual menus that allow you to add or remove files, view file properties and the build settings for projects, folders or files. In addition, the contextual menus provide access to build commands that are specific to projects folders and files.

The following commands and options are common to all contextual menus in the **Workspace** window.

Table 13. Common contextual menu commands

Command	Description
Settings...	Provides access to the build settings for the active project, folder or file that you have selected (see Configuring project settings on page 84).
Properties	Opens properties window where you will find information about the selected file that can include its directory locations, date of modification, associated build tools and dependencies.
Show full path	Activates "full path" feature so that files in the window are displayed with full path name.
Add Sorted Elements	Protects the ordering of files in the project when activated. To change the ordering, uncheck this option. This option is particularly useful when building projects with the Assembler Linker, when the source files must be assembled in a specific order (see Ordering source files before assembly on page 88).
Allow Docking	Activates docking for the Workspace window.
Hide	Hides the Workspace window.

Depending on the element that you select in the **Workspace** window (workspace, project, folder, file), you will also find in the contextual menu the commands listed in [Table 14](#).

Table 14. Window-specific contextual menu commands

Command	Description
Workspace contextual menu	
Add new Project to Workspace	Opens the New Project Wizard , which leads you through the process of setting up a new project in the current workspace (see Creating new projects on page 80).
Insert Project into Workspace	Opens browse window from which you can search for existing project files (.stp, .mak) to insert in the current workspace (see Loading an existing project on page 80).
Project contextual menu	
Build	Compiles, assembles and links the source code in the selected project (see Build commands on page 158).
Clean	Removes files generated by previous builds of the selected project (see Build commands on page 158).
New folder	Opens the dialog box for creating a new folder (see Build commands on page 158).
Add Files to Project	Opens a browse window from which you can search for existing source (.c, .cpp, .cxx, .s, .spp, .asm) and include files (.h, .hpp, .hxx, .inc) to insert in the current project (see Adding and removing folders and files on page 83).
Set as active project	In a workspace that contains more than one project, this command allows you to select the project that you want to build, debug or program to your microcontroller.
Remove from Workspace	Deletes the selected project from the workspace (see Adding and removing folders and files on page 83).
File and Folder contextual menus	
Add Files to Folder	Opens a browse window from which you can search for existing source (.c, .cpp, .cxx, .s, .spp, .asm) and include files (.h, .hpp, .hxx, .inc) to insert in the selected folder.
Remove from Project	Deletes the selected folder from the project. For more information about these commands, refer to Adding and removing folders and files on page 83 .
Open	Opens the selected file so you can view it in the Editor window.
Compile	Compiles the selected file. This command is not available for object files in the Include and External Dependencies folders (see Build commands on page 158).

3.5 Editor windows

STVD's integrated text editor helps you edit source files by providing advanced editing and navigation features. In addition, its full integration into the STVD environment makes it a powerful tool during debugging for controlling and viewing the execution of your application.

This section provides a description of **Editor** window features, including:

- [Editor window contextual menu](#)
- [Editing features](#)
- [Customizing editor features](#)

For more information about using the Editor window during debugging, refer to [Editor debug actions on page 169](#).

The integrated **Editor** windows are *client* windows that float in the main application window (see [Figure 14 on page 35](#)). When an Editor window is maximized, any other open Editor windows are hidden and the name of the open application file contained in the maximized editor window is shown across the top of the STVD main application window. Editor windows for any open files can be accessed by clicking on them in the **Window** menu.

Note: The text editor in STVD 3.2 and later versions is adapted from the Scintilla text editor.

Figure 17. Scintilla text editor copyright notice

© Copyright 1998-2003 by Neil Hodgson (neilh@scintilla.org) All Rights Reserved.

Permission to use, copy, modify, and distribute the Scintilla software and its documentation for any purpose and without fee is hereby granted, provided that the above copyright notice appears in all copies and that both that copyright notice and this permission notice appear in supporting documentation. NEIL HODGSON DISCLAIMS ALL WARRANTIES WITH REGARD TO THIS SOFTWARE, INCLUDING ALL IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS, IN NO EVENT SHALL NEIL HODGSON BE LIABLE FOR ANY SPECIAL, INDIRECT OR CONSEQUENTIAL DAMAGES OR ANY DAMAGES WHATSOEVER RESULTING FROM LOSS OF USE, DATA OR PROFITS, WHETHER IN AN ACTION OF CONTRACT, NEGLIGENCE OR OTHER TORTIOUS ACTION, ARISING OUT OF OR IN CONNECTION WITH THE USE OR PERFORMANCE OF THIS SOFTWARE.

Supported file types

Any ASCII text file can be loaded into the text editor. The Editor is able to distinguish and color-code the source code (for example, comments, keywords, functions, variables) for the file types listed in [Table 15](#).

Table 15. Editor supported file formats

Language	Supported file extensions
Assembler	*.asm, *.s, *.S, *.inc
C-language	*.c, *.h

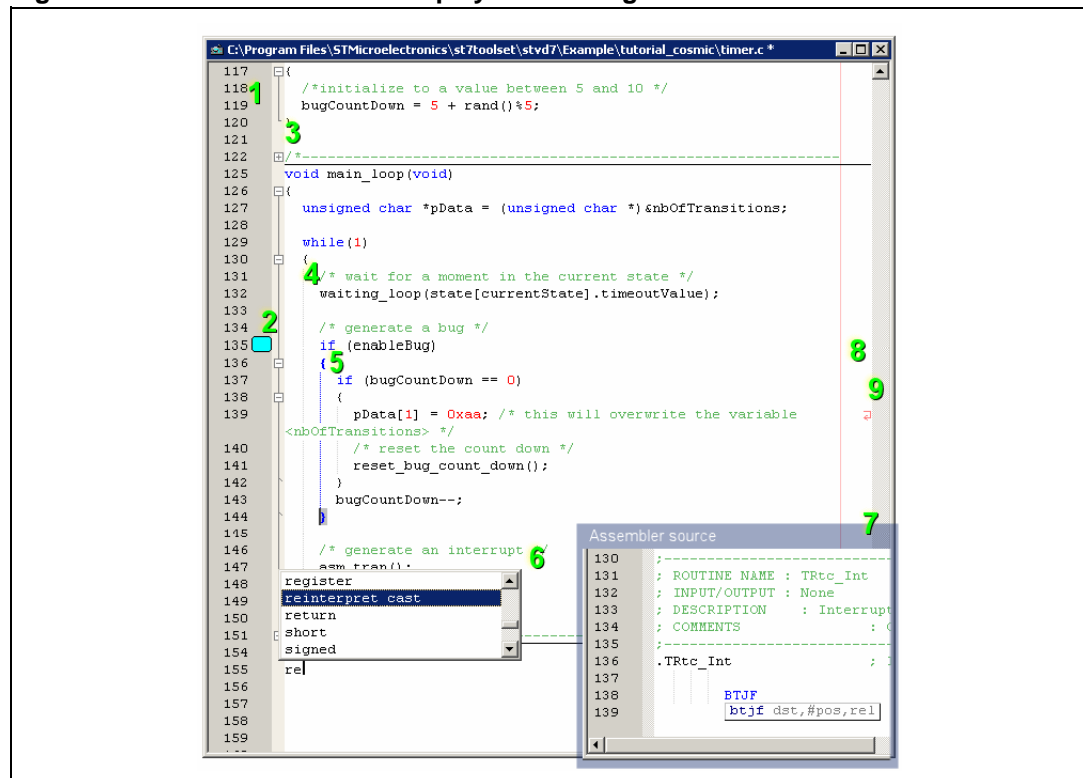
In addition to syntax highlighting, it provides advanced, language-specific features such as **auto-completion** for C and Assembler source files, and **parameter information** for source files in Assembler.

File display and editing features

The **Editor** window has a number of advanced features to help you view, navigate and edit your C and Assembler source files. Many of these features can be configured and enabled/disabled in the **Options** window (**Tools>Options**). The following features are illustrated in [Figure 18](#):

1. **Margin** controls and information including line numbers, line selection and enabling/disabling of instruction breakpoints (see [Margin features](#)).
2. **Margin icons** for navigation and debugging, including bookmarks (shown in [Figure 18](#)), instruction breakpoints, position counter and stack position indicator (see [Editor debug margin on page 172](#)).
3. **File folding** for display of specific sections of code in the source file.
4. **Configurable indentation** features include the size of indents (in characters), indentation guides and automatic indentation upon typing an opening or closing brace (see [Indentation](#)).
5. **Brace matching** to highlight opening and closing braces (see [Brace matching](#)).
6. **Auto-completion** for quick prompting of language-specific keywords when coding in C or Assembler (see [Auto-completion](#)).
7. **Parameter information** for quick prompting of ST Assembler instruction syntax in a tip pop-up (see [Parameter information](#)).
8. **Long line** indicator shows lines that exceed the maximum number of characters that you specify (see [Customizing editor features on page 61](#)).
9. **Line wrapping** indicator shows where long lines have been forced to a new line when wrapping is enabled (see [Customizing editor features on page 61](#)).

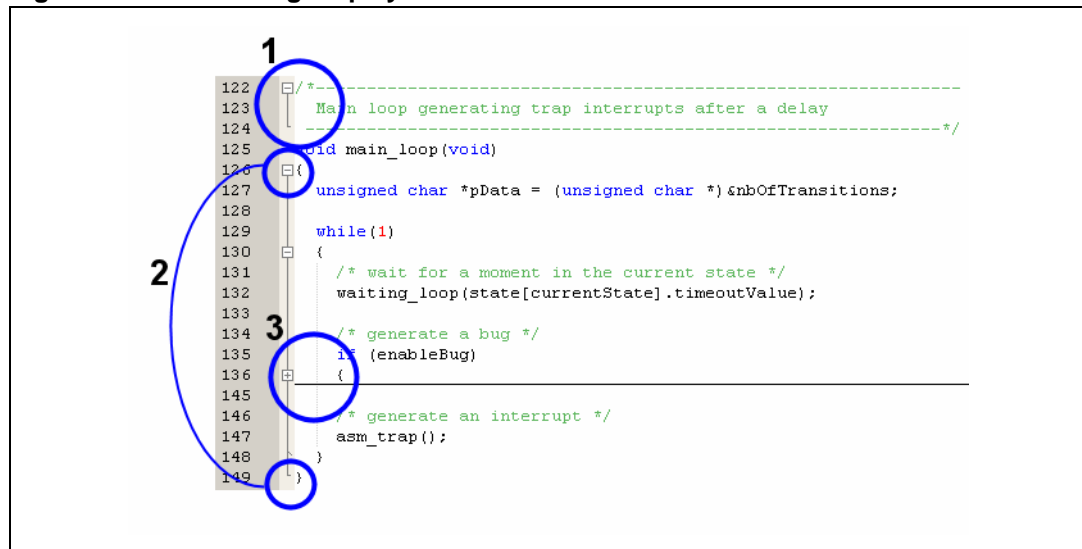
Figure 18. Editor window file display and editing features



File folding

This feature allows you to hide or display sections of C source code. Folds are created automatically for comments and {} braces as you write your code.

Figure 19. File folding display and controls



Folds are indicated in the margin by a minus (-) symbol and a vertical line ending in a short horizontal leg (see 1 & 2, [Figure 19](#)). Folded/hidden sections of code are indicated by a plus (+) sign and an unbroken horizontal line (see 3, [Figure 19](#)). When nested within another fold, the nested fold begins with the minus symbol and ends with a short, diagonal line.

Margin features

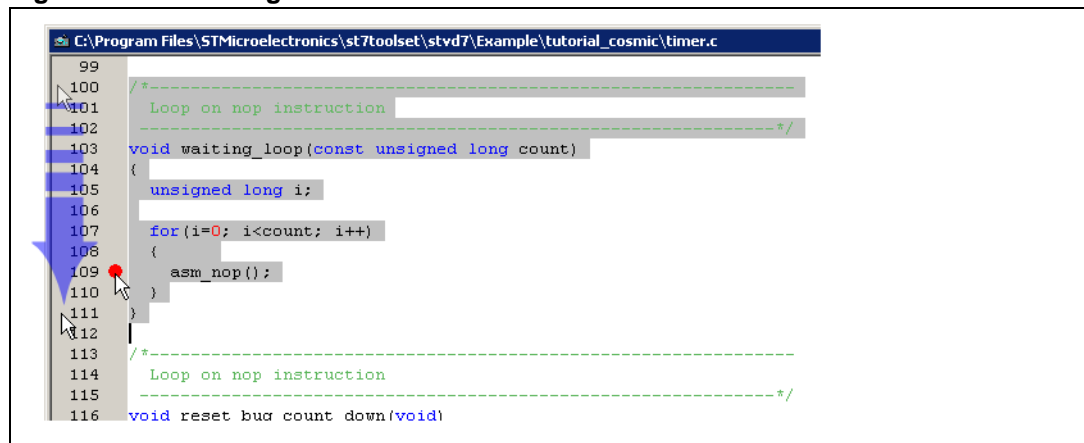
The Editor window margin offers several standard and advanced features to help you view, navigate and debug your source files, including line numbering, bookmarks and file folding. You can customize the Editor window to display or hide these features. For more information, refer to [Section 3.7: Customizing your work environment on page 60](#).

Standard features include line numbering that you can choose to display and print, and line selection.

When **line numbering** is enabled, each line ending in a return is numbered. Wrapping of lines does not change the line numbering.

Margin line selection allows you to select complete lines by clicking the left button on your mouse and dragging the cursor in the left margin as shown in [Figure 20](#). Selecting lines in this way ensures selection of the entire line. The right side of the margin is also used to enable/disable instruction breakpoints (see [Section 5.4: Editor debug actions on page 169](#)). However, Margin line selection can be set to use the entire left margin (see [Section 3.7: Customizing your work environment on page 60](#)). When the Margin selection option is checked in the Edit/Debug tab of the Options window, the margin can no longer be used to enable/disable instruction breakpoints.

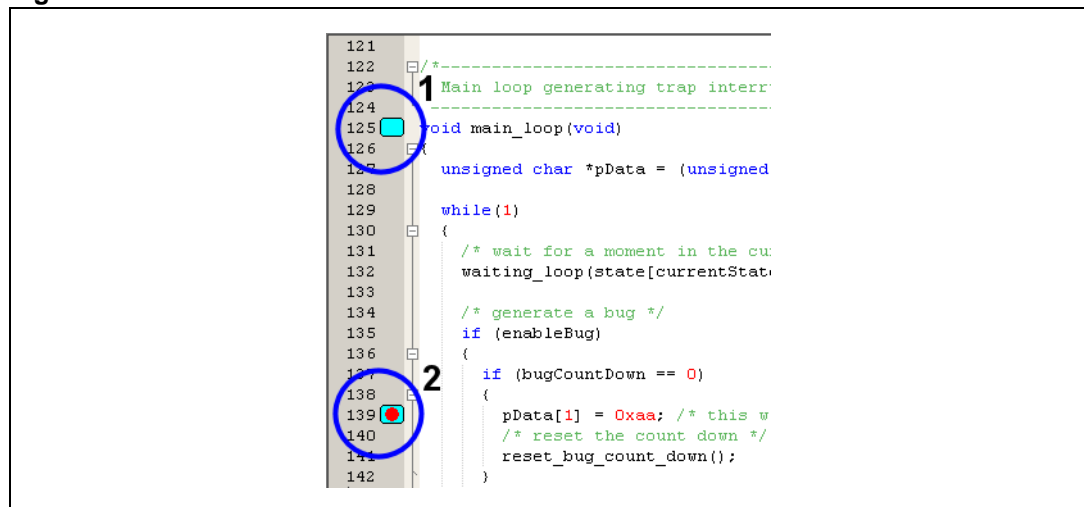
Figure 20. Left margin features



Bookmarks are particularly useful when editing large files. Bookmarks set in the Editor margin permit easy navigation between marked lines in a source file. They also serve as references for other tasks such as setting advanced breakpoints and conducting performance analysis.

The **bookmark icon** is a rectangular, blue icon in the left margin of the Editor window (see 1, [Figure 21](#)). A bookmark may be located on the same line as other margin icons, such as breakpoint icons, (see 2, [Figure 21](#)).

Figure 21. Bookmark icons



Set a bookmark by placing the cursor on a line in the file. In the main menu, select **Edit>Toggle Bookmark**. A blue rectangle appears in the margin.

Remove the bookmark by using the **Toggle Bookmark** command a second time.

Remove all bookmarks in the file that is currently being edited with the **Clear All Bookmarks** command.

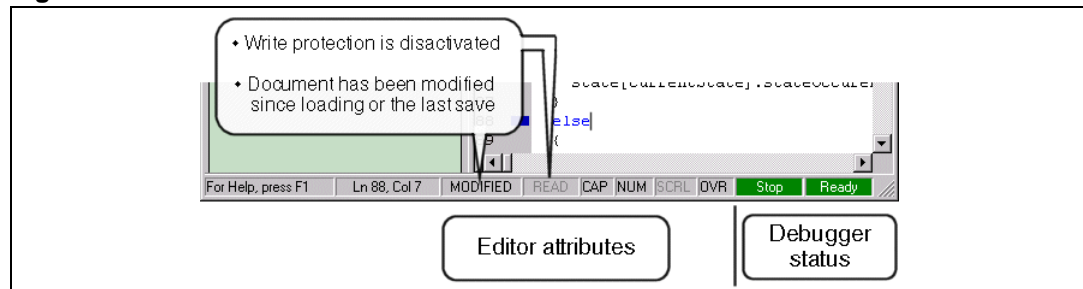
Navigate through bookmarks using the **Next** and **Previous Bookmark** commands.

Editor status bar information

The status information for each file in the Editor window is preserved when the file is deselected (including cursor column/line position), and displayed when the file is reselected.

The status bar is divided into seven sections reporting current attributes of the Editor window, plus two sections reporting the debugger status. A black keyword indicates the attribute is active. When the keyword is gray, the attribute is inactive.

Figure 22. Status bar elements



In [Figure 22](#), the status elements, ordered from left to right, are:

- **Ln xx, Col yy**: Shows the line and column position of the cursor.
- **MODIFIED**: Shows if the current document has been changed since loading or last save.
- **READ**: Shows the read/write status of the current document.
- **CAP**: Caps Lock
- **NUM**: Number Lock
- **SCRL**: Scroll Lock
- **OVR**: Overwrite is active; text insertion status.

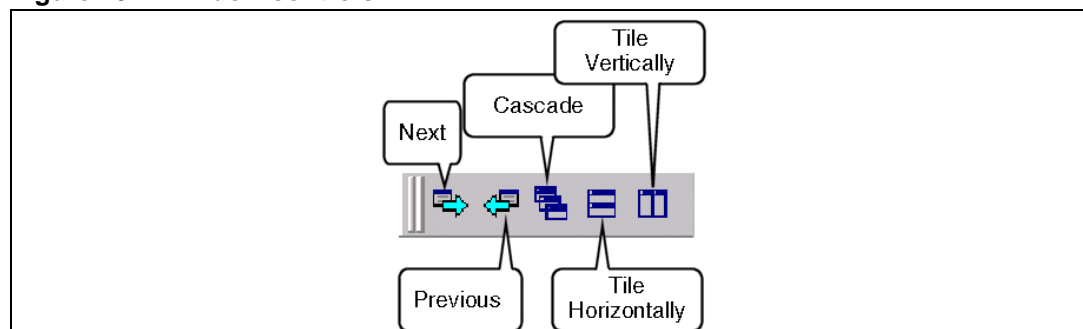
The status bar can be hidden by unchecking the **Status Bar** option in the [Window menu](#).

Editor window positioning

The Editor windows can be maximized or minimized. When maximized, only one Editor window is visible. When other windows are docked around a maximized Editor window, it can no longer be resized by dragging on the border. Instead, it is resized automatically when the docking windows are resized.

Commands to **tile** or **cascade** windows and switch between windows are available from the **Window** menu. These commands are available on the Window toolbar, shown in [Figure 23](#).

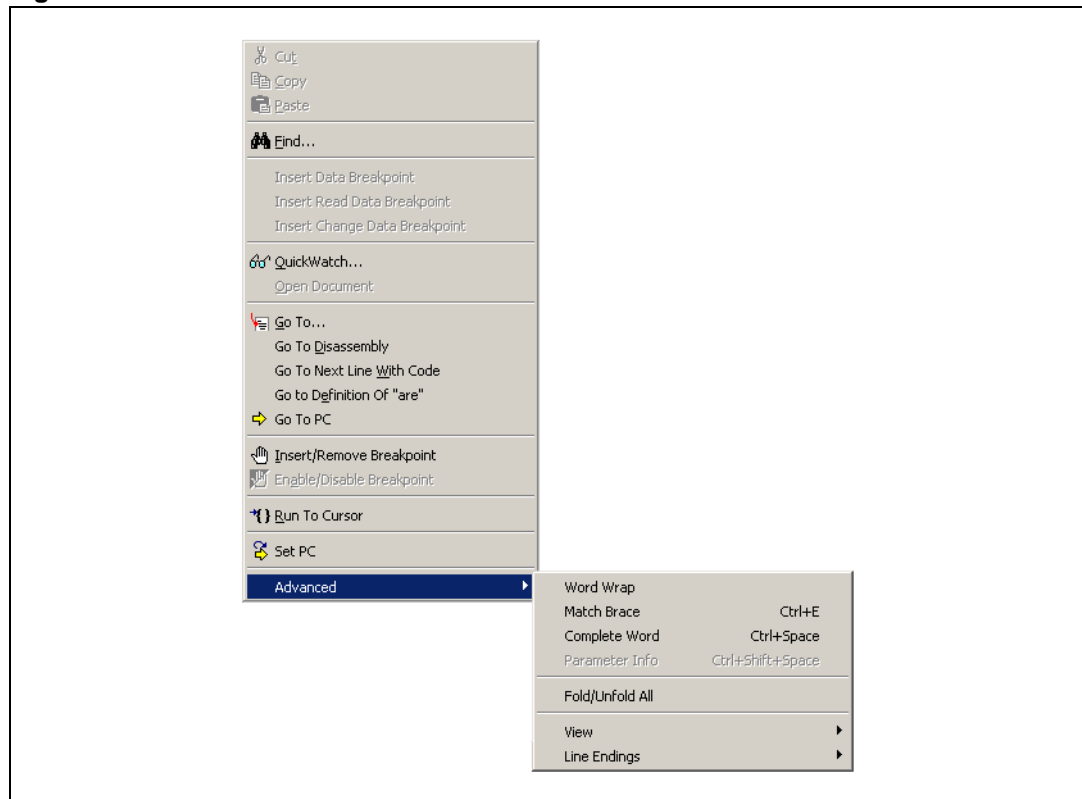
Figure 23. Window controls



3.5.1 Editor window contextual menu

By right-clicking in any Editor window, you can access the contextual menu with options and commands that are specific to the editor functions.

Figure 24. Contextual menu edit commands



In the Editor window, you have access to the basic editing commands: **Cut**, **Copy**, **Paste**, **Open Document** and **Find**. In addition, you can access advanced editing features (**Match Brace**, **Complete Word**, **Parameter Info**) and controls for advanced display conditions (**Word Wrap**, and display of **Whitespace**, **Line Numbers**, **Margins**, **Indentations Guide** and **Long Line Marker**). For more about these features, refer to [Section 3.5.2: Editing features](#).

During debugging, the contextual menu also provides access to various debug commands (**Go To**, **Run to Cursor**, **Insert/Remove Breakpoint**, **Enable/Disable Breakpoint**, **Set PC**), which are described in detail in [Section 5.4: Editor debug actions on page 169](#).

3.5.2 Editing features

The Editor offers a complete range of text-viewing, text-editing and print features for editing text and source code, including:

- [Syntax highlighting](#)
- [Auto-completion](#)
- [Parameter information](#)
- [Indentation](#)
- [Brace matching](#)
- [Text selection and clipboard operations](#)
- [Find features](#)

Note: To edit source files during a debugging session, ensure that the **Read-only source files** feature has been disabled. To do this, select **Tools>Options** and click on the **Edit/Debug** tab, then uncheck the **Read-only source files** option. Then click on **Apply** and **OK**. Note that while this allows you to modify the application source files, you must recompile the executable file so that your changes are taken into account when running the application.

Syntax highlighting

STVD offers configurable syntax highlighting to improve the readability of your C and Assembly source files. This feature allows you to define the source code display conditions (text color, background color, character style) for a syntactic group. In addition to highlighting for commands, variables and comments, the Editor recognizes language-specific keywords for C and Assembler source files.

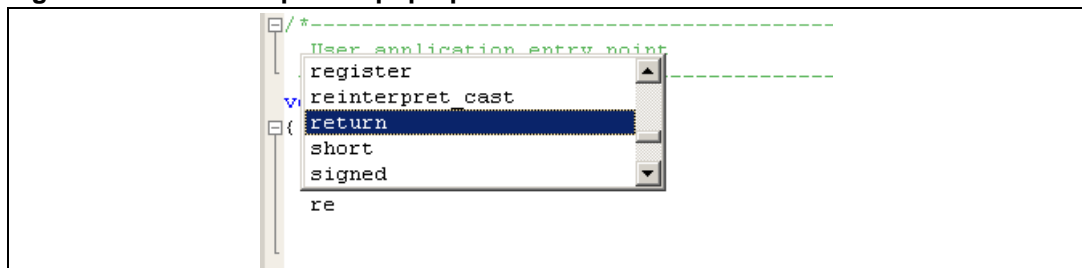
Note: Language-specific highlighting is established at the file level, therefore no language-specific highlighting is applied to Assembler code in C language files (*.c, *.h or *.ssp).

A basic highlighting configuration is set by default, which you can modify to meet your specific needs or preferences by using the **Styles/Languages** tab of the **Options** window (**Tools>Options**). For more information about modifying the syntax highlighting configuration, refer to [Section 3.7.1: Customizing editor features on page 61](#).

Auto-completion

While typing your code, use the **auto-completion** feature to see a list of language-specific keywords that match what you have started typing. For example, when typing a command in a C source file, type the first letters followed by **CTRL+Space**, or select **Edit>Complete Word**. A pop-up window opens with the alphabetical listing of C language keywords, at the section of the list that matches what you have started typing. Highlight the keyword you want and press enter (see [Figure 25](#)).

Figure 25. Auto completion pop-up list



If you have just created a new C or Assembler source file, the auto-completion feature is not available until you have saved the file with the file extension that is appropriate to the language (for C: *.c, *.h, or for Assembly: *.asm, *.s, *.S or *.inc).

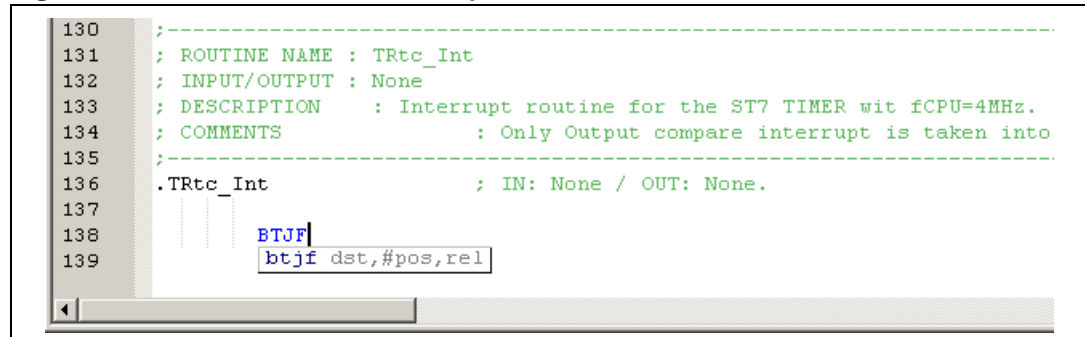
Parameter information

While typing your code in Assembly, the **parameter information** tips prompt you with the syntax for the instruction that you are typing. Syntax information is provided for the ST ASM instruction set.

For example, when typing an instruction in an Assembler source file, type the instruction then press **CTRL+Shift+Space**, or select **Edit>Parameter Info**. A tip window appears with the syntax of the instruction that you have named (see [Figure 26](#)).

If you have just created a new Assembler source file, the parameter information feature is not available until you have saved the file with the appropriate file extension: *.asm, *.s, *.S or *.inc.

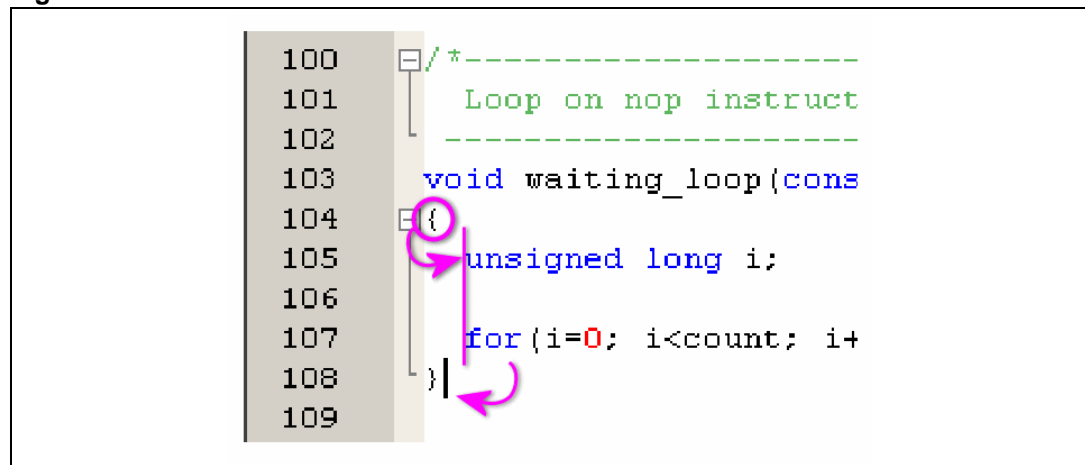
Figure 26. Parameter information tip



Indentation

During text entry, **auto-indent** returns the cursor to the column in which the previous line of text starts, so that the indent of the base line is preserved in subsequent lines. In addition, whenever you type the opening brace { for a function, the subsequent line is automatically indented. When you type the closing brace } for your function, the line with the closing brace and subsequent lines are un-indented by one level.

Figure 27. Auto indentation



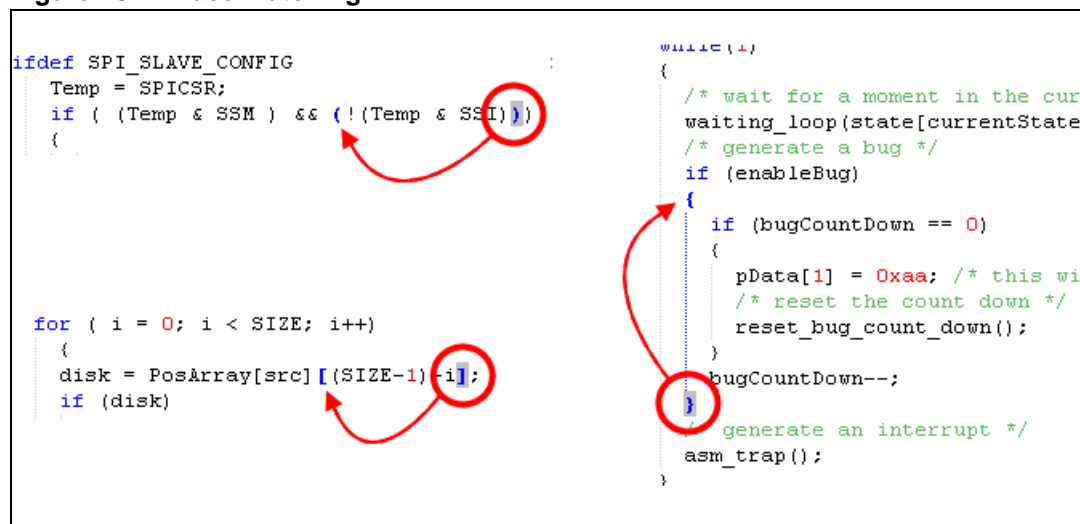
When enabled, the editor also provides indentation guides to help you see code lines that have the same indent level. For more information about configuring these options, refer to [Section 3.7: Customizing your work environment on page 60](#).

You can also **manually indent lines** by selecting a line or placing the cursor at the beginning of the line and pressing the Tab key. **Block indent** allows any block of text highlighted with the text cursor to be indented with the Tab key in the same way that you would indent a single line. Pressing **SHIFT+Tab** un-indents the selected text block or line, moving it to the left.

Brace matching

When viewing and navigating in source code, brace matching highlights matching opening and closing braces – {}, (), [], <>. If you highlight a brace with the cursor, then the matching brace is highlighted in the same color. If you press **CTRL+E**, or select **Edit>Match Brace**, then the cursor moves to the corresponding brace.

Figure 28. Brace matching



Text selection and clipboard operations

The editor offers the same clipboard features (**Cut**, **Copy** and **Paste**) used by standard Windows-based text editors using the same keystroke combinations. You can select text with the mouse, using the left mouse button alone or using **<Alt + left mouse button>** to define any rectangle in the text area and select the text it contains.

Drag and drop

Any selected block of text (highlighted using the text cursor) can also be dragged while holding down the left mouse button and moved to another window in STVD, another application, or to a new location inside the Editor window. When dragged, a cross-hatch square symbol accompanies the mouse cursor to indicate the transfer of text.

The same action (drag with left mouse button) with the **Control** key depressed, copies the highlighted text to the new location. In this case, a square symbol containing a plus sign accompanies the mouse cursor.

Find features

The Editor window provides standard find and replace commands to help you locate and update specific strings. The **Find**, Find next, **Replace** and **Find in files** functions are available from the **Edit** menu. The Find command is also available in the contextual menu.

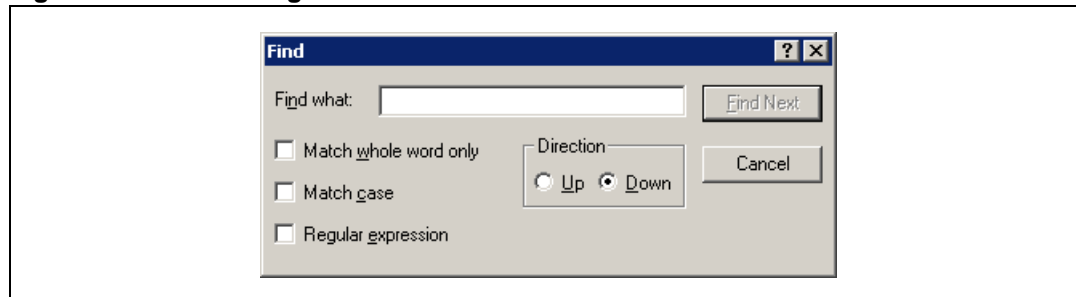
In addition to searches for strings or variables, the Editor allows you to search for patterns of characters by using **Regular expressions**.

Find

You can select the string for a search by placing the cursor anywhere in the string, highlighting it or typing it directly in the search window.

If you place the cursor to select the string, the string for the search is delimited by white space or any other non-alphabetic and non-numeric characters. On the other hand, when you select by highlighting a string, the full string of characters regardless of the type of characters appears in the **Find** window.

Figure 29. Find dialog box

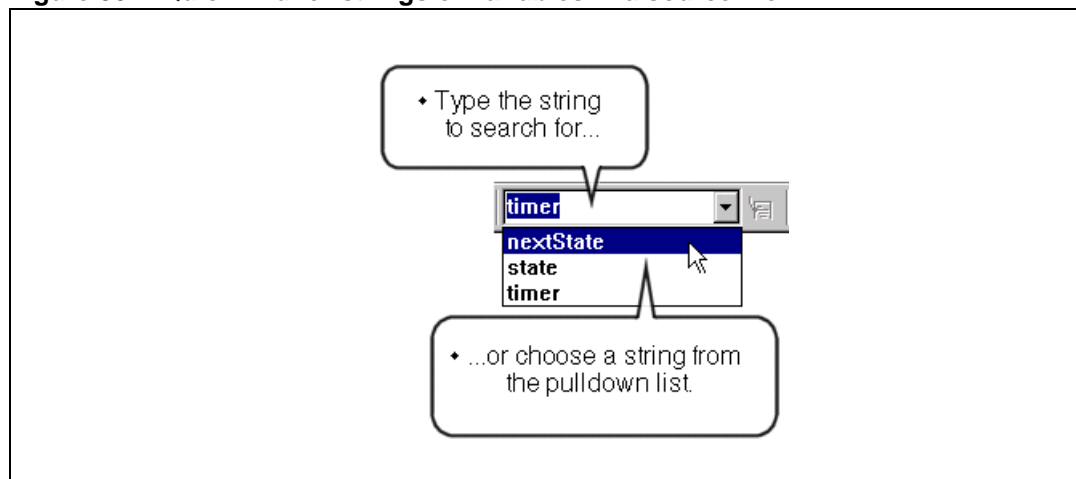


In addition to common find options (Match whole word, Match case), the Find dialog box allows you to search for patterns of characters using [Regular expressions](#).

Find pull-down list

The **Edit** toolbar in the active window contains a text entry box for quick string searches (see [Figure 30](#)). Type the string you want to search for in the field, or select a string that you have already searched for from the pull-down list.

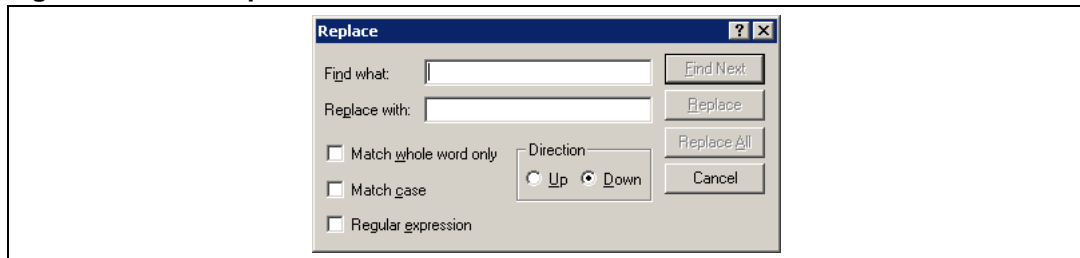
Figure 30. Quick find for strings or variables in a source file



Replace

In the active Editor window, a string at the cursor position (delimited by white space and/or other usual separators) or any highlighted string, can be sought in the whole of the text. Pull down the **Edit** menu from the main menu bar, and select the **Replace** command to open the **Replace** window.

Figure 31. Find/replace window



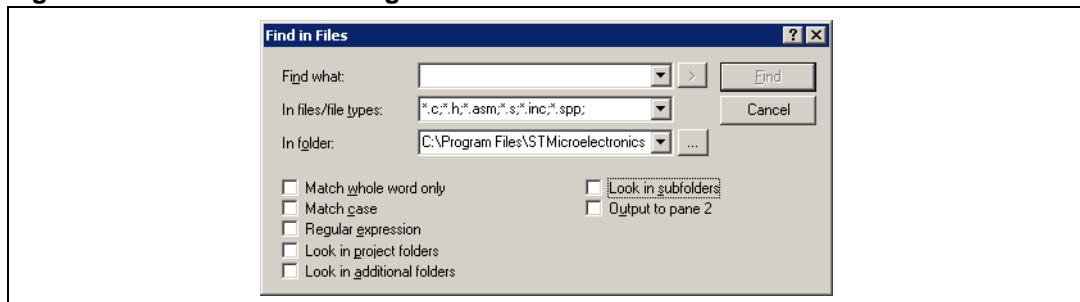
In addition to common find options (Match whole word, Match case), the **Replace** dialog box allows you to search for patterns of characters using [Regular expressions](#).

Find in files

The **Find in Files** command allows you to search for any string in any number of file types within a given source directory. In addition to common find features and directory search options, the **Find in Files** dialog box allows you to search for patterns of characters using [Regular expressions](#).

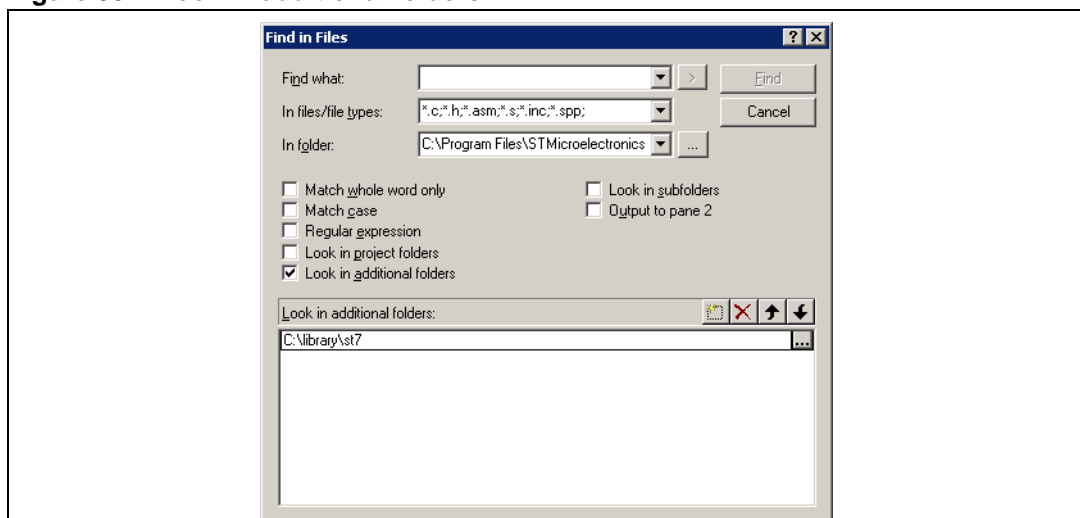
For example, [Figure 32](#) shows the search for the variable “alpha” in files with a variety of extensions.

Figure 32. Find in Files dialog box



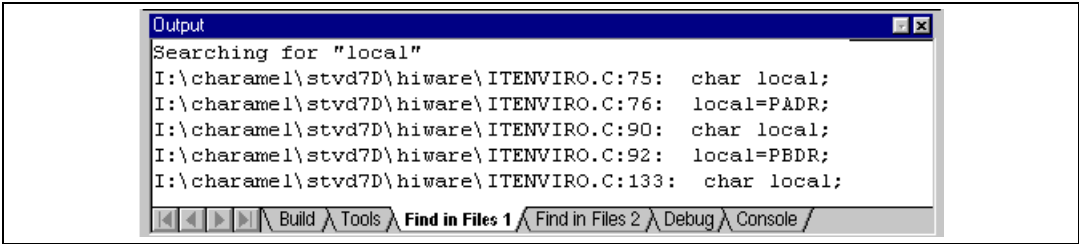
If you put a checkmark beside the **Look in additional folders** option, a dialog box like the one shown in [Figure 33](#) is displayed. You can add additional folders to search.

Figure 33. Look in additional folders



The results of your search are displayed in the [Output window](#), by default in the **Find in Files 1** tab, or in the **Find in Files 2** tab if you select the option **Output to pane 2**.

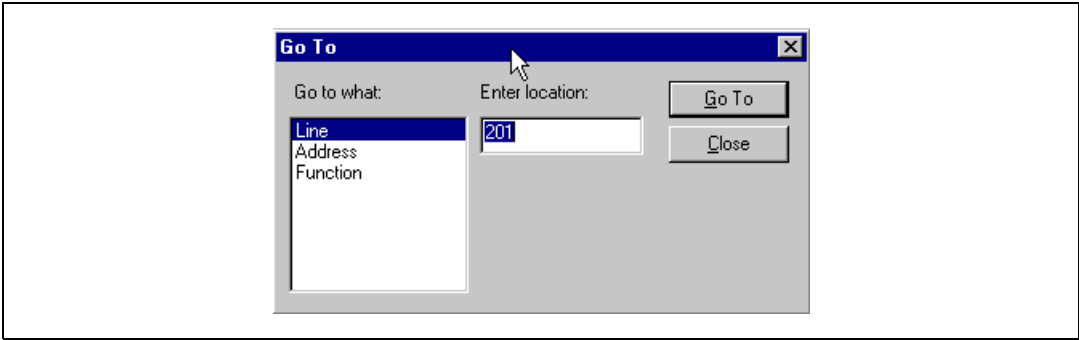
Figure 34. Find in Files tabs



Go to

This command opens the **Go To** window, permitting a line in a source file to be defined as the target for the **Go To** action.

Figure 35. Go To window



Regular expressions

Regular expressions make it possible to search ASCII text for patterns of characters instead of literal strings of characters. For example, they allow you to search for all expressions consisting of a character followed by any character in a defined set, or to search only for occurrences of a string of characters at the end of a line.

Checking the **Regular Expressions** checkbox in the [Find](#), [Replace](#) or [Find in files](#) windows enables the use of meta characters for regular expressions such as those listed in [Table 16](#).

Table 16. Regular expressions

Meta character	Use	Example
[]	Find any characters in the defined set.	X[A-B] finds any occurrences of XA or XB
[^]	Find characters except those in the defined set	X[^C-Z] finds any occurrences of XA or XB and excludes combinations with any other letters (XC, XD, ... XZ)
.	Any character	X.[A-B] finds any occurrences of X followed by any two characters and then an A or B (X--A, or X--B where - is equal to any ASCII character)
+	Matches one or more occurrences of the specified characters	X[A-B]+ finds any occurrences of X followed by one or more occurrences of the defined set (XA, XB, XAA, XAB, XBB, XAAA, XAAB, ...)

Table 16. Regular expressions (continued)

Meta character	Use	Example
*	Matches zero or more occurrences of the specified characters	x[A-B]* finds any occurrences of X and X followed by one or more occurrences of the defined set (X, XA, XB, XAA, XAB, XBB, XAAA, XAAB, ...)
^	Beginning of line indicator (when it precedes a search string and is not enclosed by brackets)	^x[A-B] finds only the occurrences of XA and XB at the beginning of a line.
\$	End of line indicator (except when preceded by a \)	x[A-B]\$ finds only the occurrences of XA and XB at the end of a line.
\	The meta character following the \ is treated as an ASCII character	Changing the preceding example to x[A-B]\\$ finds all occurrences of XA\$ and XB\$.
\<	Restricts the search to the beginnings of words	\<x[A-B] finds only the occurrences of XA and XB that are at the beginning of a word (that are followed by other alpha-numeric characters – XAVIER or XA123)
\>	Restricts search to the ends of words	x[A-B]\> finds only the occurrences of XA and XB that are at the end of a word (that are preceded by other alpha-numeric characters – MIXA or 123XA)

These are just a few of the regular expressions that you might use to search your files. For more information, you can refer to any reference about Unix regular expressions.

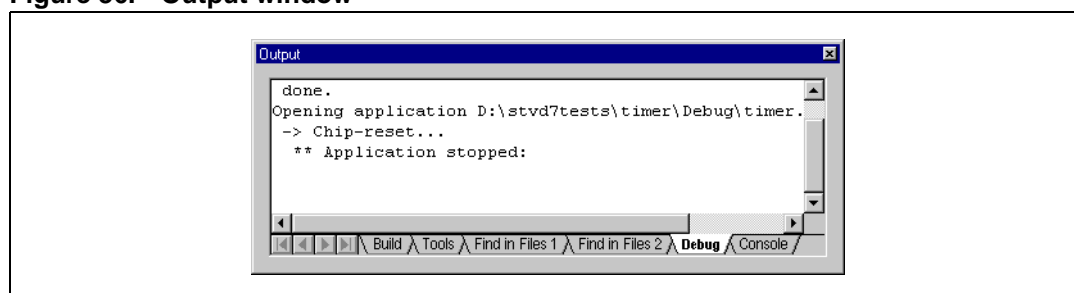
Note: When regular expressions are enabled in the **Replace** dialog box, they are used in the **Find what** field. A regular expression in the **Replace with** field is treated as a string of ASCII characters, not as a regular expression.

3.6 Output window

To open the **Output** window, either click on the Output window icon  in the **View** toolbar or from the main menu select **View>Output Window**.

The Output window is opened from the View toolbar or View menu.

Figure 36. Output window

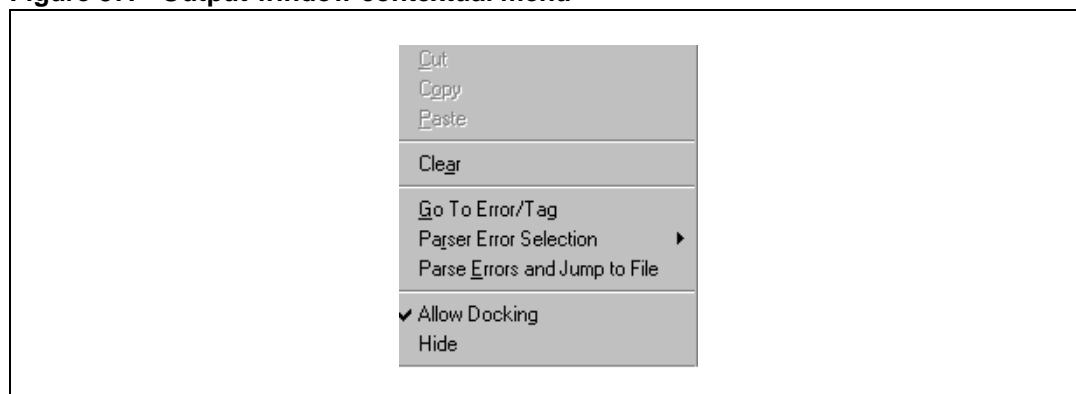


The tabs located at the bottom of the Output window area are used to flip through the different views available in the window. These tabs include:

- [Build tab and Tools tab](#)
- [Find in Files 1 & Find in Files 2 tabs](#)
- [Debug tab](#)
- [Console tab](#)

If you right-click the mouse in the Output window tabs a contextual menu appears specific to that tab. For example, the contextual menu for the **Build** and **Tools** tabs is shown in [Figure 37](#).

Figure 37. Output window contextual menu



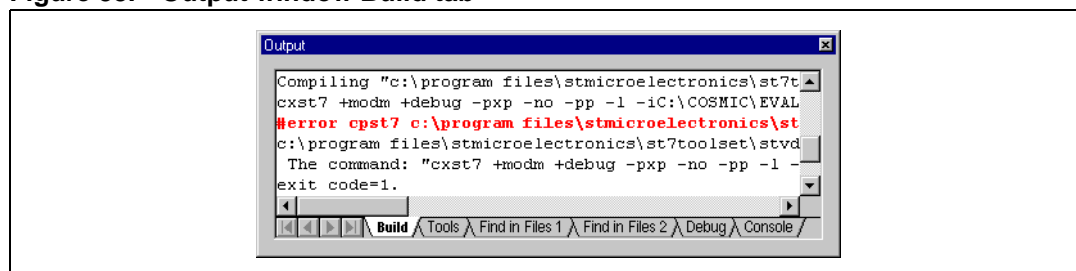
Note: Depending on the tab view you are in, the contextual menu may have fewer or different options than the one shown above.

3.6.1 Build tab and Tools tab

These tabs display information on the current build or tools session.

For example, the view of the **Build** tab in [Figure 38](#) shows the messages resulting during build session using the Cosmic toolset. Errors are shown in **bold text**, and refer to the source file and line number where the error was found.

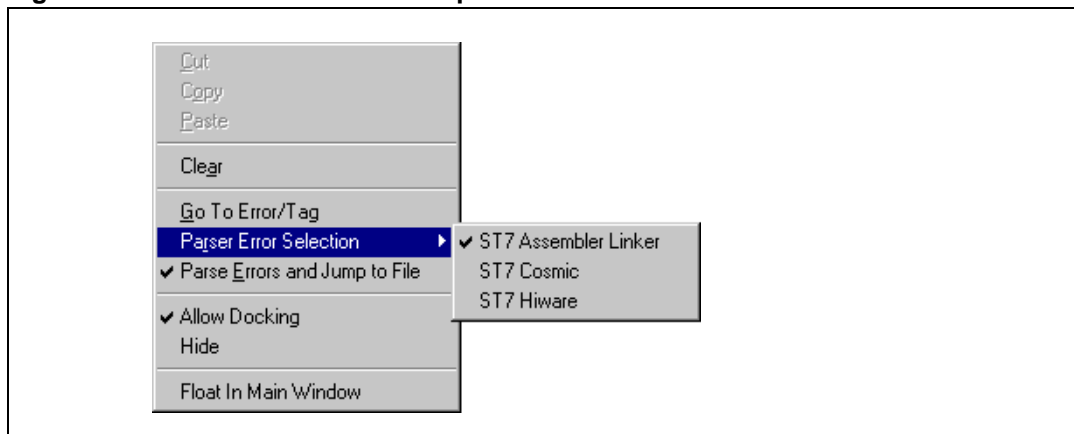
Figure 38. Output window Build tab



The contextual menu for the Build and Tools tabs has options that make it easier to jump to any errors found during the build.

In the **Tools** tab contextual menu, first choose the parser that corresponds to your toolset. The various parsers installed appear when **Parser Error Selection** is selected. A checkmark appears beside the active parser. [Figure 39](#) shows that the ST Assembler Linker parser has been activated.

Figure 39. Parser error selection options



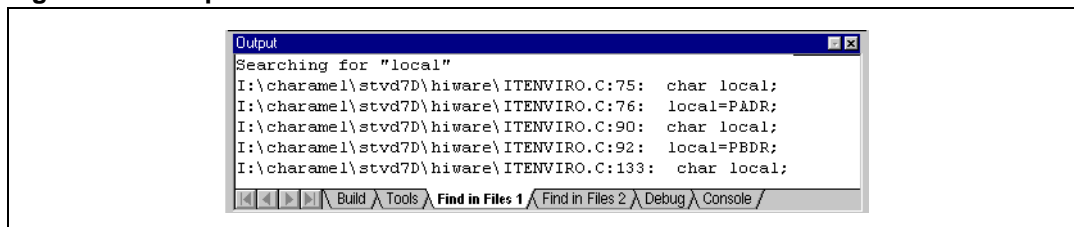
Once a parser is selected, you can use the **Parse Errors and Jump to File** option (a checkmark indicates that it is enabled). This option allows you to automatically jump to the source of any errors found during the build process by:

- Double-clicking on the error line in the Build tab.
- Selecting the error line in the Build tab, then right-clicking the mouse to bring up the contextual menu, then selecting **Go To Error/Tag**.

3.6.2 Find in Files 1 & Find in Files 2 tabs

These tabs show the results of any [Find in files](#) searches. By default, results are posted to Find in Files 1, unless you select the option **Output to pane 2** in the Find in Files dialog box.

Figure 40. Output window Find in Files tab



3.6.3 Debug tab

This tab gives information on the current debugging session, including:

- Emulator/Connection information
- Run/Stop information
- Warning messages

In addition to this information about the debugger and debug instrument, STVD also provides a [Console tab](#) that allows you to enter debugging commands.

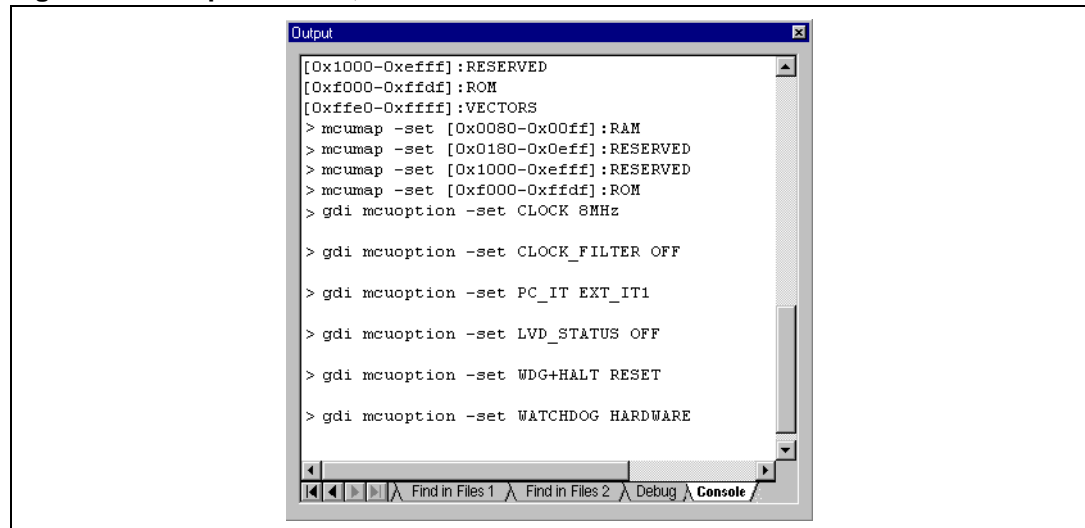
3.6.4 Console tab

This tab gives you access to a command console where you can enter debugging commands for the STVD debugger (also referred to as GDB). Normally, you should not have

to enter commands directly. However, in [Section 5.18: Online commands on page 199](#), you will find several basic online commands for reference.

This command console (shown in [Figure 41](#)) displays online output and allows you to enter commands at the command prompt, marked by '`->`'.

Figure 41. Output window, Console tab

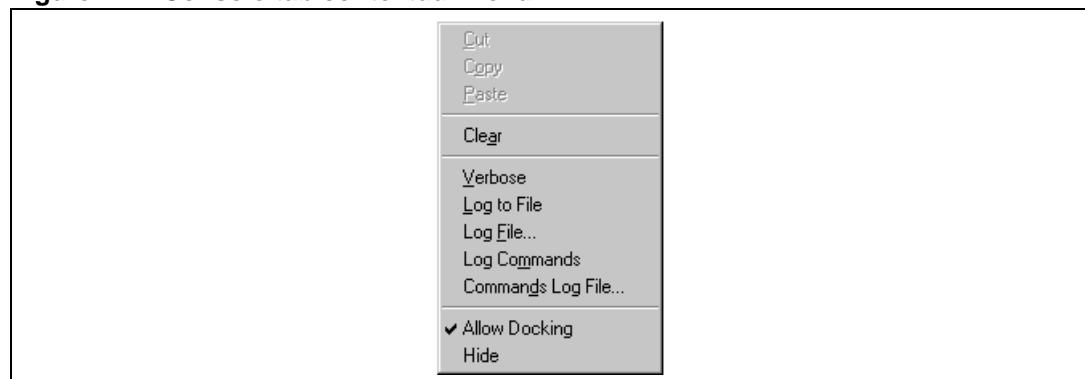


Caution: Under normal circumstances, the workspace must **not** be opened or closed, or a **Quit Debugger** command issued, from within this window. The STVD graphical user interface must be used for these operations.

The STVD graphical user interface is not refreshed systematically following direct entry of commands, therefore use of the **Run** command within the command console should be approached with caution.

Right-click anywhere in the **Console** tab to open the console contextual menu.

Figure 42. Console tab contextual menu



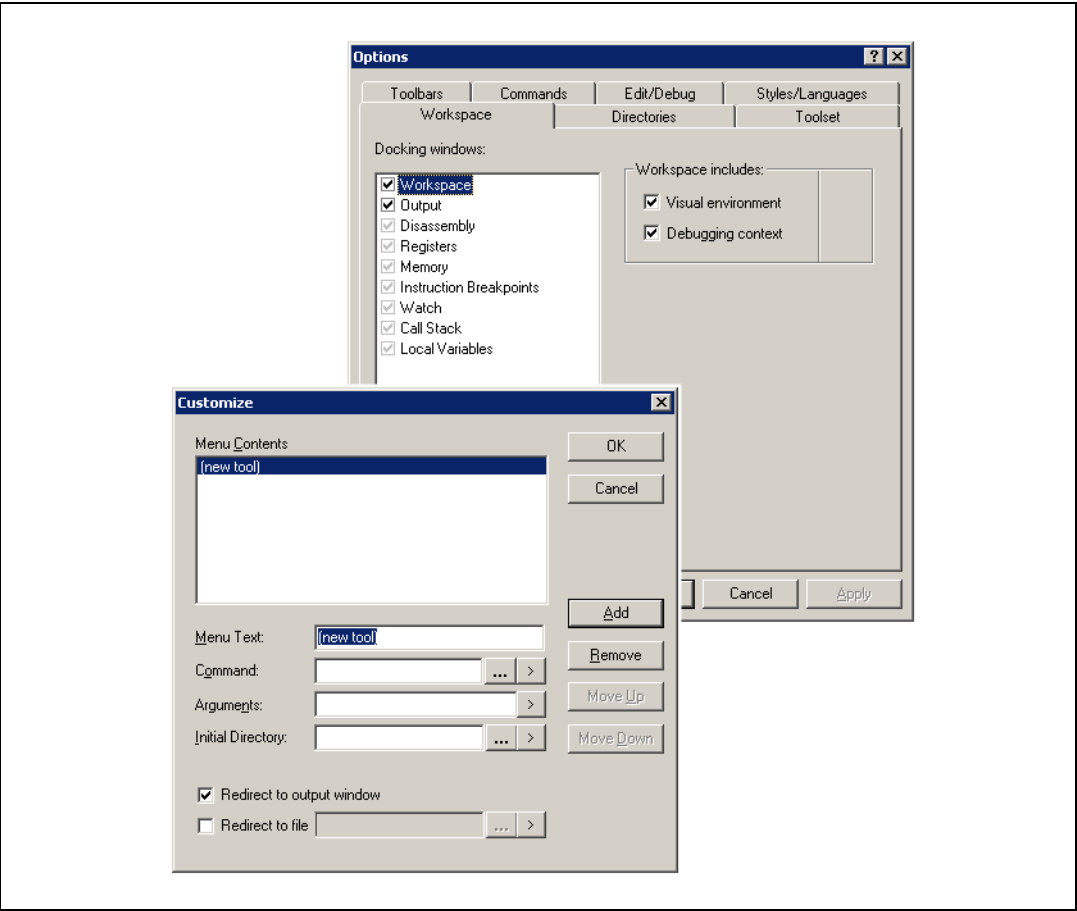
The contextual menu provides options to edit online commands (**Cut**, **Copy**, **Paste**) plus logging options (**Log to file** and **Log commands to file**) with file-creation options in each case (**Log file** and **Commands log file**).

Option **Verbose On/Off** defines the level of comment displayed on screen.

3.7 Customizing your work environment

STVD offers a flexible working environment for developing your application with **Options** and **Customize** windows to help you tailor STVD to your needs.

Figure 43. Options and Customize windows



The **Options** window (**Tools>Options**) provides several tabs, listed in [Table 17](#), that allow you to enable, disable and configure STVD features.

Table 17. Options window tabs

Tab name	Description
Workspace	Configure docking windows and information stored in the workspace file.
Directories	Define global directories to use when building and debugging your application. Refer to Section 2.1: Set the toolset and path information on page 23 .
Toolset	Specify the toolset and paths to use when building your application. Refer to Section 2.1: Set the toolset and path information on page 23 .
Toolbars	Enable, disable, and configure toolbars. Refer to Section 3.7.2: Customizing toolbars on page 65 .
Commands	View and customize the commands on the various toolbars.

Table 17. Options window tabs (continued)

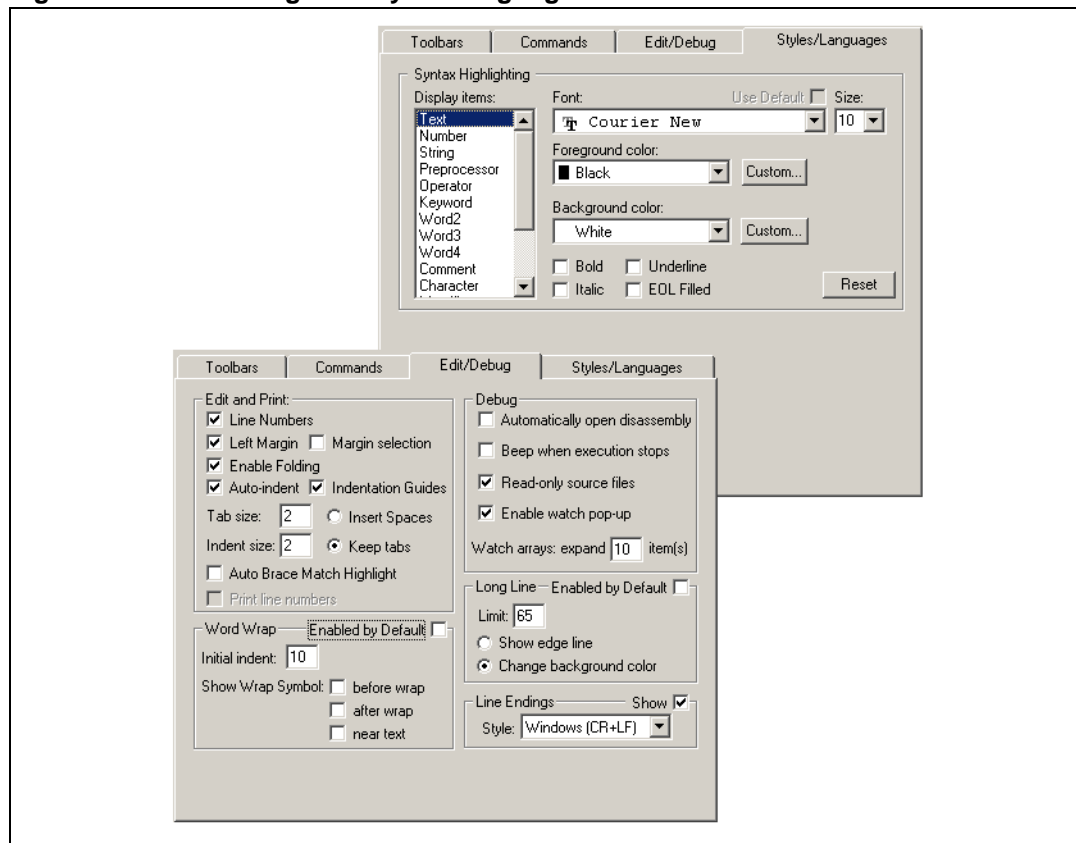
Tab name	Description
Edit/Debug	Enable and disable Editor features related to debugging and editing. Refer to Section 3.7.1: Customizing editor features on page 61 .
Styles/Languages	Configure the language specific syntax highlighting. Refer to Section 3.7.1: Customizing editor features on page 61 .

The **Customize** window (**Tools>Customize**) provides an interface for creating your own commands to add to STVD's menus and toolbars. For example, you can create commands that launch other applications that you use frequently during application development. Refer to [Section 3.7.3: Adding custom commands on page 67](#).

3.7.1 Customizing editor features

You can define the attributes and functions of the integrated Editor in the [Edit/Debug tab](#) and [Styles/Languages tab](#) of the **Options** window. To access this window, select **Tools>Options** in the main menu bar.

Figure 44. Edit/Debug and Styles/Languages tabs



Edit/Debug tab

Use the **Edit/Debug** tab to enable/disable and configure editor features related to editing source files and debugging your application. The tab is divided into the following areas:

- Edit and print
- Word wrap
- Debug
- Long line
- Line ending

Edit and Print allows you to access the editing and printing options described in [Table 18](#).

Table 18. Edit and print options

Option	Description
Line numbers	When checked, the line numbers are displayed in the line number margin on the left side of the Editor window.
Left Margin	When checked, the margin used for enabling/disabling of instruction breakpoints is visible.
Margin selection	When checked, clicking in the left margin of the Editor window selects the entire line. When unchecked, a mouse click sets/disables/removes a breakpoint on the indicated line. Even when this box is not checked, you can select a full line by clicking in the Line number margin.
Enable Folding	When checked, folding of source code is enabled and the fold margin is visible.
Auto-Indent	When checked, automatic indentation upon typing an opening or closing brace for a function is enabled.
Indentation Guides	When checked, the thin vertical lines indicating indents and tabs are visible.
Tab size	Specifies the size for tabs in number of characters.
Indent size	Specifies the size for indents in number of characters.
Insert spaces	When checked, tabs in files that are opened with the editor are replaced with spaces.
Keep Tabs	When checked, the editor retains the tabs in any files that are opened.
Auto Brace Match Highlighting	When checked, the highlighting of matching braces is enabled. Even when this option is disabled, users can still find matching braces by placing the cursor next to a brace and selecting Edit>Match Brace , or pressing CTRL+E .

Word Wrap allows you to access the line wrapping options described in [Table 19](#).

Table 19. Word wrap options

Option	Description
Enable	When checked, wrapping is enabled. This option is checked by default.
Initial Indent	Specifies the indent at the beginning of each line that has been wrapped from the preceding line. Enter 0 for no indent.

Table 19. Word wrap options (continued)

Option	Description
Show Wrap Symbol	These checkboxes specify the position of the red arrow symbol indicating a wrapped line: Before wrap - symbol is at right edge of the window before the wrap. After wrap - symbol is on the left side of the window after the wrap. Near text - symbol is next to the last text symbol before the wrap.

Debug allows you to access the debugging options described in [Table 20](#).

Table 20. Debug options

Option	Description
Automatically Open Disassembly	When checked, STVD automatically opens the Disassembly window when you start a debug session
Beep When Execution Stops	When checked, STVD generates an audible signal whenever the execution of the application stops while debugging.
Read-Only Source Files	When checked, the Editor will not allow editing of open source files during a debug session. Source files are not protected from editing if you are not in a debug session. When disabled, if a source file is changed during the debugging session, the change is not taken into account until the application has been rebuilt.
Enable Watch Pop-Up	When checked, during debugging you will be able to view the current value of a variable in a Watch pop-up field by hovering the mouse over a variable in the source code.
Watch Arrays	This field allows you to specify the number of levels (1-256) of an array to display for variables in the Watch window.

Long Line allows you to access the line length options described in [Table 21](#).

Table 21. Long line indicator options

Option	Description
Enable	When checked, this option enables the long line specification and display of the long line indicator or background color.
Limit	This field allows you to specify the maximum length of code lines in number of characters.
Show Edge Line	When checked, this option displays a red vertical line indicating the maximum line length limit for each line of code. Symbols and spaces/indent associated with wrapping lines are not subtracted from the maximum number of characters in a line of code.
Change Background Color	When checked, this option changes the text background color for all lines exceeding the maximum line length.

Line Ending character formats vary from one operating system to another. Failure to recognize end of line characters can cause compilation errors. If in doubt, you should confirm that your source files use the Windows end of line character style (CRLF).

When you open a file with end of line characters that are different from the default line character style (the Editor is set to Windows style by default), STVD warns you and asks if you want to change them to the default style.

If your file is read-only, STVD warns you that the characters are different from the default, but cannot give you the option to change them.

In the **Line Endings** section of the **Options** window **Debug/Edit** tab, you can enable the display of line endings and choose the line ending style to use. [Table 22](#) summarizes the line ending options.

Table 22. Line endings options

Option	Description
Show	When checked, an icon representing the end of each text line is displayed. The icons are: <ul style="list-style-type: none">– CRLF for text files created under Windows– LF for text files created under Unix– CR for text files created under Mac
Style	This option allows you to indicate the type of end of line character to use (Windows, Unix or Mac), by choosing the line character style from the Style list. By default the Style is set to Windows. When you change the style, all of the end of line characters in open files are changed to the new default setting.

Styles/Languages tab

The **Syntax highlighting** section allows you to modify the display characteristics used for the display of syntactic items such as language-specific keywords, operators, comments, and numbers. This is a machine-level configuration for STVD, which is applied to editable files regardless of the project or workspace that you are in.

When making changes to the Editor display options, use the **Apply** button to see the effect of the changes you have made. The **Apply** button is only available once you have clicked in or modified a display option.

You can return to the original configuration at any time by clicking on the **Reset** button.

Table 23. Syntax highlighting options

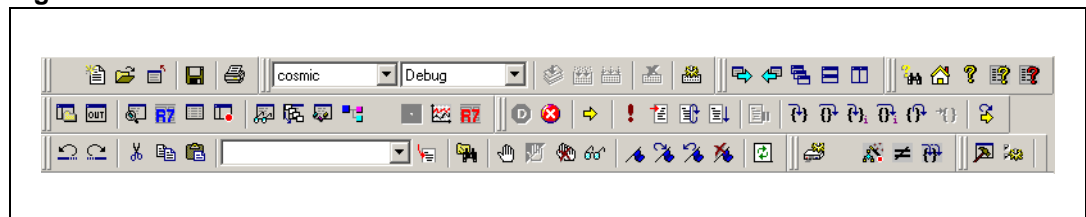
Option	Description
Display Items	This listing allows you to select the syntactic items whose display characteristics you would like to view or modify. Current characteristics are shown when you select any display item. The font characteristics (font, size, bold, italic, underline) of the Default item are applied to all other items that have the Use Default option checked. The foreground and background color of the Default item are applied to all other items that have foreground and/or background set to Default.
Use Default	When this box is checked for any display item, the settings of the Default item are applied.

Table 23. Syntax highlighting options (continued)

Option	Description
Font and Size	These fields allow you to select the font and the font size (in points) to be applied to all text items. To change this configuration you can select <i>Text</i> as the display item and the changes you make will be applied to all the display items where Use Default is checked. You can select any display Item, uncheck <i>Use Default</i> and select a font and/or size that will be applied only to the selected display item.
Foreground and Background colors	Select the text color (foreground) and background colors from a list of available colors. These can be configured for each display item.
Bold, Underline and Italic	These checkboxes allow you to apply text characteristics to any of the display items.
EOL Filled	The <i>End Of Line Filled</i> checkbox enables filling of the background color to the edge of the Editor window.

3.7.2 Customizing toolbars

A number of predefined relocatable toolbars are present by default. These toolbars may be modified or others may be created. All toolbars are dockable and may be moved independently, shown or hidden.

Figure 45. Standard toolbars

The toolbar options dialog box

From the main menu, select **Tools>Options** to open the **Options** dialog box. Click on the **Toolbars** tab (see [Figure 43](#)) to access the customizable toolbar options.

Table 24. Toolbar options

Option	Description
Toolbars list	This list of toolbars with checkboxes allows you to configure which toolbars are visible.
Show/hide tooltips	This checkbox enables/disables the Tooltips pop-ups.
Cool look and Large Buttons	Use checkboxes to enable and disable button styles – Cool look (beveled buttons), Large buttons (icon and text to indicate button's functionality)
New button	Opens a window that allows you to specify the name of a new user-defined toolbar. To make your own toolbars refer to Creating user-defined toolbars . To add or remove toolbar commands Rearranging toolbar icons .

Moving toolbars

Toolbars can be transformed to independent always-on-top windows, by placing them over (or *in part* over) the main application window or on the desktop outside the frame of the STVD main application window. In this floating form they may be relocated by mouse dragging on the window title bar or any unoccupied area in the window. These toolbar windows may be resized in the standard manner by dragging on the edge of the frame (double arrow cursor indicates resize dimension).

Individual toolbars may be picked up and repositioned anywhere in the STVD main application window.

When docking is active, toolbars locate themselves in a window border region close to the drop point. This causes automatic rearrangement of the local windows to accommodate the new toolbar position.

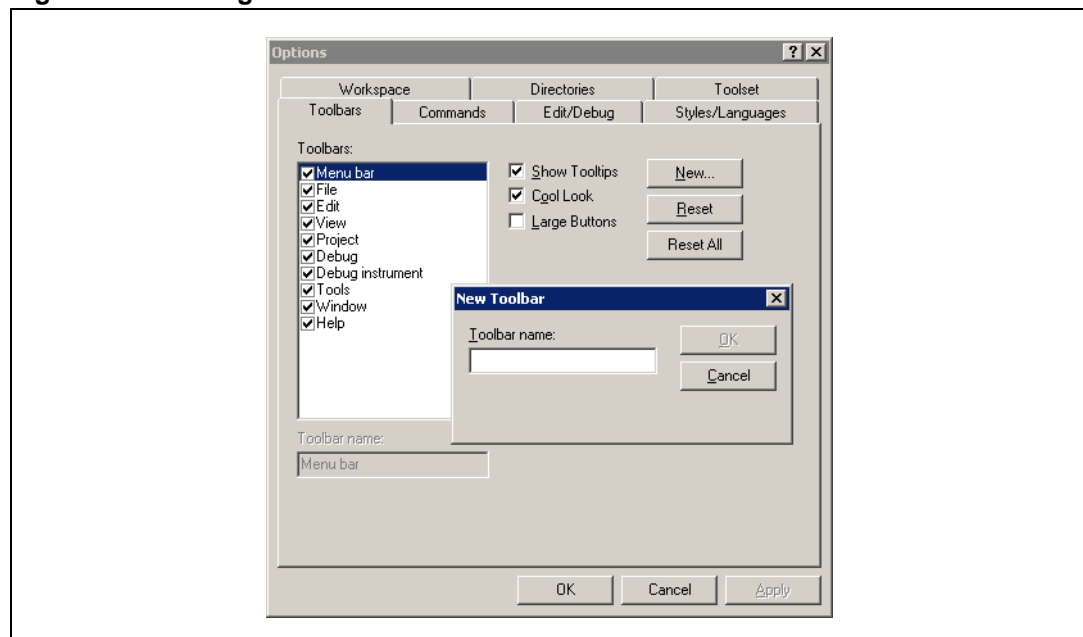
When docking is not active, the toolbar stays where dropped.

When a toolbar is placed over the main application window, it will remain where placed and appear as a floating window with its own header. (In this case, it is located by absolute screen position, and remains stationary when the Editor or STVD main application window is moved or resized.)

Creating user-defined toolbars

1. From the main menu, select **Tools>Options** to open the **Options** window.
2. Click on the **Toolbars** tab and then **New** to open the **New Toolbar** window.

Figure 46. Adding new custom toolbars

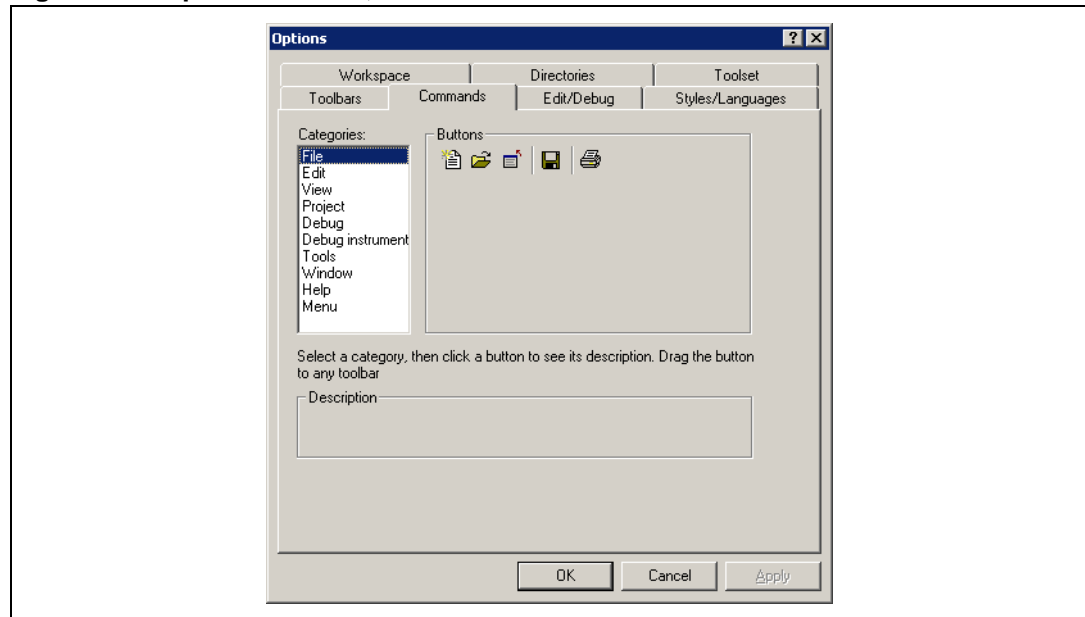


3. Enter a name for the new toolbar at the prompt.
This name is then added to the list of current toolbars on the **Toolbars** tab. By default, the new toolbar is marked as visible. However, even though it has been named, the new toolbar is still empty. Add elements to it as described in [Rearranging toolbar icons](#).

Rearranging toolbar icons

1. From the main menu select **Tools>Options**.
2. Select the **Commands** tab and select a **Category** from the list (see [Figure 47](#)).
The icons for commands in the selected category are displayed.
3. To add a button to a toolbar, drag an icon from the **Buttons** field to any toolbar. To remove a button from a toolbar, drag an icon off the toolbar.

Figure 47. Options window, Commands tab



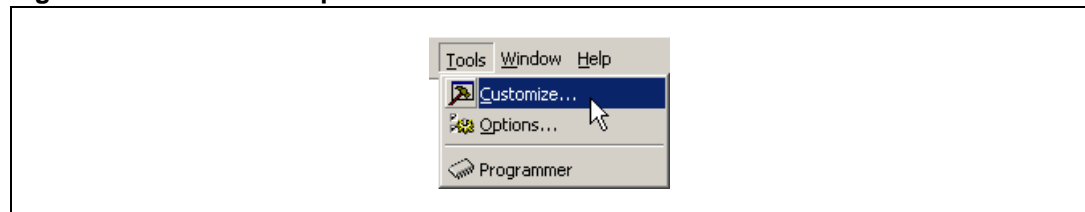
4. While the **Commands** tab is open, displace a button slightly left or right on the toolbar to create a separator between toolbar elements at that point.

3.7.3 Adding custom commands

You can use the **Tools>Customize** command to define and add custom commands that link to external add-on software directly from within STVD.

This interface allows you to define commands and parameters, and name the new command. The new command name is then added to the **Tools** menu.

Figure 48. Customize option

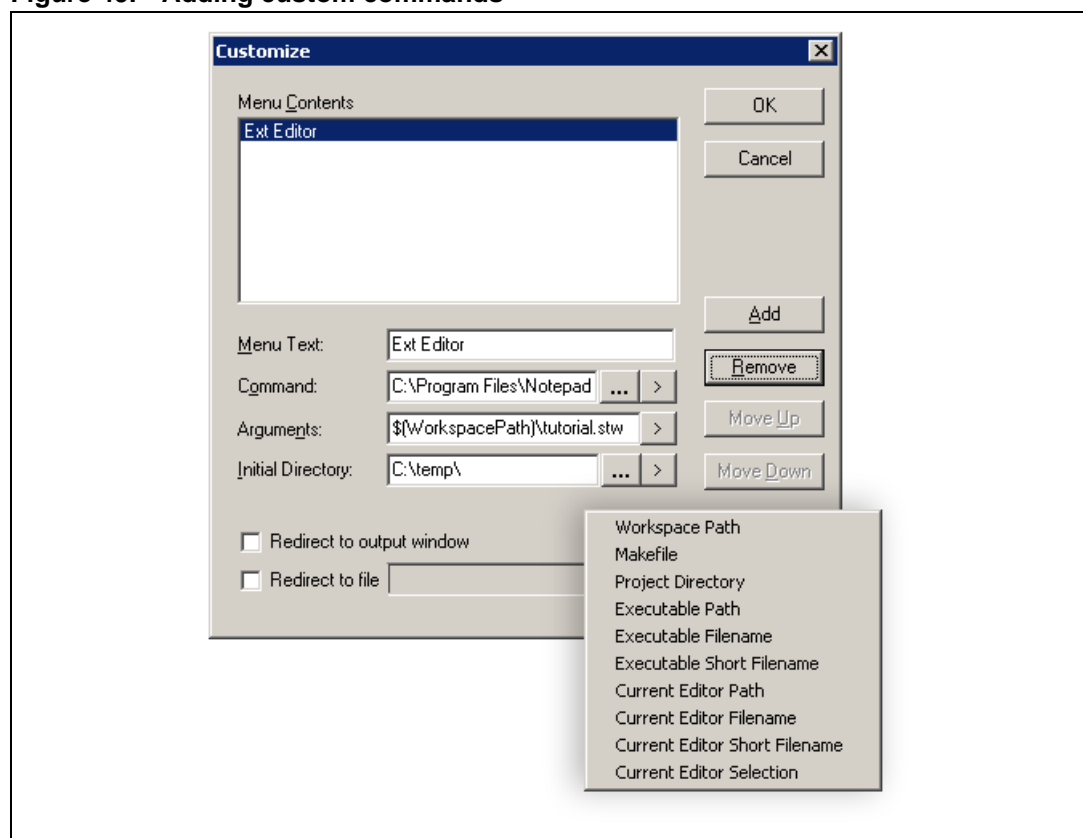


For example, imagine that you want to add a command that would link directly to an external editor (such as Windows® Notepad):

1. From the main menu, select **Tools>Customize** to open the **Customize** dialog box (shown in [Figure 49](#)).
2. Click **Add**.
3. In the **Menu Text** field, enter the new command keyword, which appears in the Tools menu—for example, *Ext. Editor*.
4. In the **Command** field, enter or browse to the executable file for the custom command, *notepad.exe* in the example.
5. In the **Arguments** field, you can specify an argument to go along with the executable. This is useful if, for example, you always want to open a certain file that is located in particular directory. Click **>** for a list of arguments. Possible arguments that can be added include:

- | | |
|-----------------------|---------------------------------|
| – Workspace Path | – Executable Short Filename |
| – Makefile | – Current Editor Path |
| – Project Directory | – Current Editor Filename |
| – Executable Path | – Current Editor Short Filename |
| – Executable Filename | – Current Editor Selection |

Figure 49. Adding custom commands

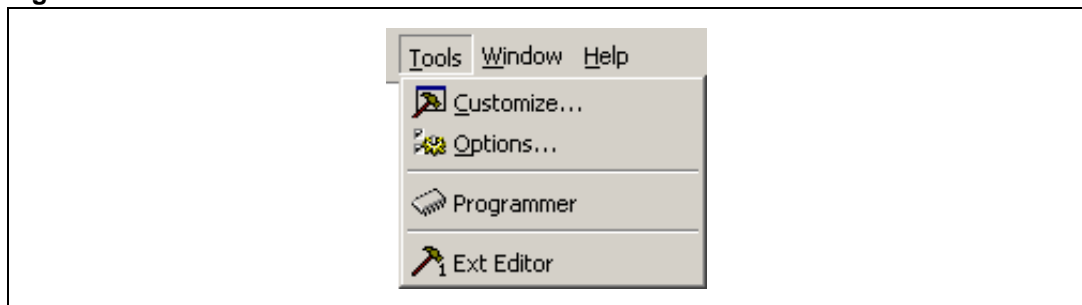


In the example, Notepad opens *sample.wsp*, which is located in the directory of the open workspace. This is specified using the **Workspace Path** argument.

6. In the **Initial Directory** field, you can specify the directory from which you want the command to be initialized. You can type the directory, browse to it, or specify it using the argument list shown.
In our example, Notepad is initialized from `C:\temp\`.
7. Finally, there are two other options that can be used with Win32 console applications (DOS applications) only. **Redirect to output window** sends any output to the **Output** window within STVD and **Redirect to file** allows you to send your output to the file you specify (by typing in its name and path, browsing to it or using arguments to specify it).

Once you have finished, a new function, called Ext Edit, is added to the Tools menu, which can be accessed like any other tool as shown in [Figure 50](#).

Figure 50. Customized command in Tools menu



3.8 Tooltips

A local banner describing each tool button can be activated from the Tools menu. In the **Toolbars** tab of the **Options** window, the checkbox **Show Tooltips** turns on the Tooltips function. The mouse pointer must remain inactive on the tool button for a short delay to activate the tooltip.

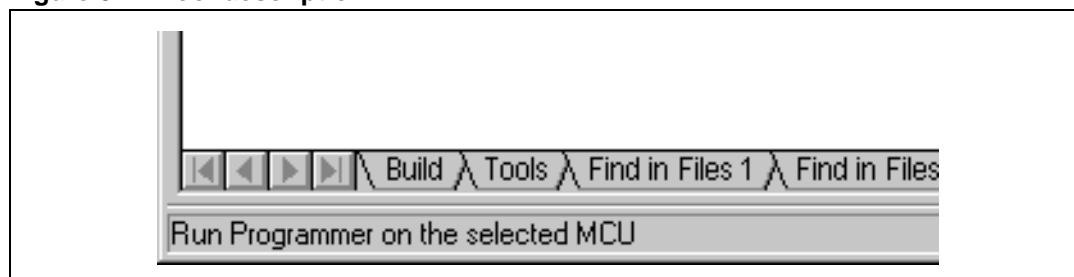
Figure 51. Tooltips



A mouse-over on any tool button can be set to open a tooltips banner with a short description of the tool. At the same time, a longer descriptive statement on the selected tool appears to the far left of the status bar, in the lower left border of the STVD main application window.

This report offers more detail on the selected tool, and is active even when the tooltips banner is inactive. The status bar report on a mouse-over of the **Run** command button is shown in [Figure 52](#). The corresponding tooltips message is illustrated in [Figure 51](#).

Figure 52. Tool description



3.9 Help and support features

From the Help menu you can access a variety of STVD features that have been designed to help learn about and use its most advanced features. These features include:

- **STVD Online Help and Tutorial**— provides detail information on using STVD with supported debug instruments. Includes a tutorial with everything you need to create an application that you can debug with STVD (see [Section 12: STM8 C tutorial on page 332](#) or [Section 13: ST Assembler/Linker build tutorial on page 366](#)).
- **About...**— provides version information for STVD and the various components of your debug instrument
- **Help on Instruction**— provides information about ST assembly instructions
- **Instruction Set Contents...**— allows you to access the ST Instruction Set online reference
- **Generate support file**— allows you rapidly transfer error log files to a single file that you can send to support when seeking assistance after an application crash

Generate support file

An application crash during debugging could be the result of a combination of factors involving the different components of STVD and/or your debug instrument. If an application crash occurs, STVD and your debug instrument generate log files in various STVD directories. These logs can help support representatives determine the source of the problem.

The **Generate Support File** feature retrieves these error logs and allows you to save them in an archive (zip) so that they can be sent to your support representative.

Following a crash, to retrieve all the error logs for support:

1. Select **Help>Generate Support Information**.
2. In the **Destination Zip File** field, you can enter the path and the name for the archive where the logs will be gathered together.

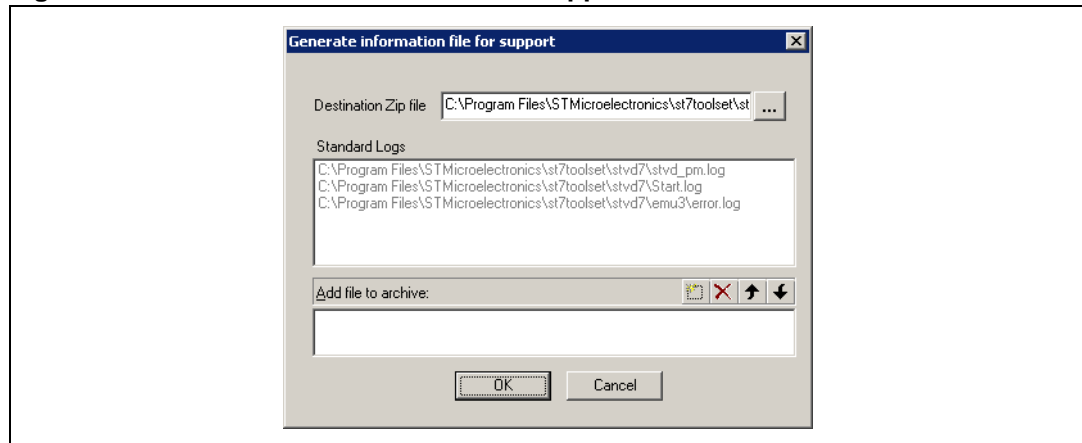
In the resulting **Generate Information File for Support** window (see [Figure 53](#)) you will see the list of standard error logs that have been generated as a result of the crash.

This list may include:

- **stvd_pm.log**— Contains information about any crashes of STVD that have occurred since its installation (provided that it has not been modified manually). Information is stored in chronological order from the first to the last crash.

- **Start.log**– Contains the target debug and STVD version information. Old information is overwritten with the most recent, each time you enter a debug session (*Debug>Start Debugging*).
- **error.log**– Contains information about errors that are specific to your debug instrument. This information is cleared each time you enter a debug session (*Debug>Start Debugging*).
- **downloader.log**– Contains information regarding the last update of your debug instrument's firmware.

Figure 53. Generate Information File for Support window



3. In the **Add File to Archive** field, specify any additional files that you would like to add to the archive.
4. Click on **OK** to retrieve the logs and place them in the archive.
You can mail this file to your support representative, who will use it to help you resolve your problem.

3.10 Migrating old workspaces (STVD7 2.5.4 and prior versions)

When building ST7 applications with STVD7 prior to version 3.0, the command line options, include files, and library path names used by your toolset were contained in a makefile (.mak). STVD no longer relies on the makefile, which you had to write for previous versions. Instead it offers a graphical interface that allows you to quickly and easily apply and change build options, in addition to storing those options as build configurations.

To benefit fully from STVD's new build features, you must manually transfer the build configuration (options, paths) from the makefile used by your old workspace (.wsp) to the build settings interface for your project (.stp), which is contained in a new workspace format (.stw).

As an expedient, STVD also allows you to automatically create a new workspace (.stw) from the makefile that you used with your old workspace (.wsp). This permits a rapid migration, however, you will not be able to take advantage of the full range of STVD's new build features. [Table 25](#) outlines the advantages and drawbacks of these two migration methods.

Table 25. Migration strategy advantages and disadvantages

Full migration	Partial migration (wrap makefile) ⁽¹⁾
Advantages	
<ul style="list-style-type: none"> • Allows full use of the graphical interface for managing build options and configuration • Facilitates modification of build settings during development • Automatic, complete management of dependencies • No makefile to maintain • Migrate your project one time only 	<ul style="list-style-type: none"> • Automatic migration • Rapid migration • Use exactly the same settings used when building with a previous version and produces exactly the same executable
Disadvantages	
<ul style="list-style-type: none"> • Manual migration • Migration takes a little longer 	<ul style="list-style-type: none"> • Cannot use the STVD graphical interface to manage build options • Must continue to maintain the makefile and the dependencies manually

1. If you want to continue using your makefile to manage the build options for your application, refer to [Create a workspace by wrapping a makefile](#).

For a full migration of your project, follow the 7 steps outlined in this section. This will take a little time up-front during the migration, but will give you full use of all STVD's new build features.

7 steps to full migration

To migrate your project from previous versions, you need to set up a workspace and project in STVD 4.0.1, add the source and include files and then confirm the various options and paths that were previously specified in your makefile.

Table 26. Migration steps 1 and 2

Step	Related information
STEP 1: Create a workspace in STVD. Select File>New Workspace , then select the New Workspace and Project icon. Follow the steps in the different screens to create your workspace and project.	Do not use the "Create from..." options. If you need more help on how to create a workspace, refer to Create a workspace with a new project .
STEP 2: Add the application source files to the project. Select Project>Insert Files into Project , then drag the source files to the Source Files folder in the Workspace window.	Using the same command, you can add any include or header files that your application may require and place them in the appropriate folder in the Workspace window. If you need more help on adding files to your workspace, refer to Add source files (.asm, .c, .s) to your project .

To provide STVD all the information required to build your application, you must now specify the target microcontroller, the build options, additional include and library paths and the link script in the **Project Settings** interface.

From the makefile, you can note the command line options and path information used in previous builds of your application.

Open the **Project Settings** window by selecting **Project>Settings...** from the main menu.

Table 27. Migration steps 3-5

Step	Related information
STEP 3: In the General tab, confirm that the toolset paths correspond to the installation on your PC. Some paths depend on options specified in the project settings.	<p>For <u>Metrowerks users</u>, the library path used corresponds to the output format set in the Compiler tab:</p> <ul style="list-style-type: none"> • for Hiware format - lib\ST7c\lib, • for ELF/DWARF 2.0 format - lib\ST7c\lib.e20 <p>You can specify the ELF/DWARF 2.0 format by selecting Output in the Category list box, and then checking ELF/DWARF 2.0 format. ⁽¹⁾</p>
STEP 4: In the MCU Selection tab, select the target microcontroller.	
STEP 5: In the Compiler tab, confirm the command line options in the gray Command Line window at the bottom of the tab (<u>all compilers</u>).	<p><u>Cosmic users</u> can specify additional include directories. Select Preprocessor in the Category list box and enter the file and the path in the field provided.</p> <p><u>Metrowerks users</u> can specify extra include files and their paths. Select Input in the Category list box and enter the file and the path in the fields provided.</p> <p>For details on the available options for your toolset refer to Section 4.5: Configuring project settings on page 84.</p>

1. It is highly recommended that you use the ELF/DWARF 2.0 format rather than the Hiware format. The ELF/DWARF 2.0 format is more powerful and the debug information is more reliable.

Note: *Most options for the supported toolsets can be set in the Project Settings window. However, if there is an option that is not incorporated in the Project Settings interface, you can add it to the command line using the User Defined Options field.*

Table 28. Migration steps 6 and 7

Step	Related information
STEP 6: In the Assembler tab, confirm the command line options in the gray Command Line window at the bottom of the tab (<u>all compilers</u>).	<p><u>Metrowerks users</u> can specify extra include paths. Select General in the Category list box and enter the pathname.</p> <p><u>ST Assembler Linker</u> users can specify additional include paths in the provided field.</p> <p>For details on the available options for your toolset refer to Section 4.5: Configuring project settings on page 84.</p>

Table 28. Migration steps 6 and 7 (continued)

Step	Related information
<p>STEP 7: In the Linker tab, confirm the command line options in the gray Command Line window at the bottom of the tab (all compilers).</p> <p>For details on the available options for your toolset refer to Section 4.5: Configuring project settings on page 84.</p>	<p>Cosmic users can specify additional Object Library Modules and Library paths to include during linking. Select General in the Category list box and enter the pathnames in the provided fields.</p> <p>Cosmic users can choose to generate an automatic link script (.lkf) file for linking, or to use an existing .lkf file. Select Linker Input in the Category list box. To use an existing .lkf file, uncheck the Auto checkbox, and enter the name of the file to use in the Script LKF Filename field. For more information about automatic script link generation, refer to Section 4.7.3: Cosmic C linker tab on page 110.</p> <p>Metrowerks users can specify additional Object Library Modules and Library paths to include during linking. Select General in the Category list box and enter the pathnames in the provided fields.</p> <p>Metrowerks users can choose to generate an automatic parameter (.prm) file for linking, or to use an existing .prm file. Select Linker PRM File in the Category list box. To use an existing .prm file, uncheck the Auto checkbox, and enter the name of the file to use in the Script PRM Filename field. For more information about automatic parameter file generation, refer to Section 4.9.3: Metrowerks linker tab on page 148.</p> <p>ST Assembler Linker users can specify additional Library paths to include during linking by entering the pathname in the provided field.</p>

Note: [Metrowerks](#) We recommend that users working with the Hiware format migrate their workspaces to STVD 4.0.1. Once you have successfully done this, we recommend switching to the ELF/DWARF 2.0 format to build your application. This gives you access to a greater range of options and will allow you to use both the **Auto PRM file generation** option, or your previous PRM file.

If you choose to use the previous PRM file, note that it must correspond to the syntax used with the ELF/DWARF 2.0 format, notably the ELF syntax uses **SECTION** in place of **SEGMENT**. Also ensure that the variables for the **start07** routine are placed in a **No.Init** section.

Once you have made the necessary entries and confirmed the command line options, click on **OK** to apply the settings and exit the **Project Settings** window.

You have now migrated your project to STVD 4.0.1. You have access to the full range of supported options and features for your toolset and you can now build your application using the same build options used to generate previous versions of your application.

4 Project creation and build

STVD provides a build interface that allows you to control the building of your application for debugging or for programming to your target microcontroller. When you first open STVD, by default you are in the **Build context**. In this context, you will set up your workspace and project(s), configure project settings and build your application.

The following sections provide information about the build interface, including:

- [Section 4.1: Specifying a toolset](#)
- [Section 4.2: Loading and creating workspaces \(.stw\)](#)
- [Section 4.3: Creating and loading projects in a workspace](#)
- [Section 4.4: Project build configurations](#)
- [Section 4.5: Configuring project settings](#)
- [Section 4.6: Customizing build settings for ST Assembler/Linker toolset](#)
- [Section 4.7: Customizing build settings for Cosmic C toolsets](#)
- [Section 4.8: Customizing build settings for Raisonance C toolset](#)
- [Section 4.9: Customizing build settings for Metrowerks C toolset](#)
- [Section 4.10: Configuring folder and file settings](#)
- [Section 4.11: Specifying dependencies between projects](#)
- [Section 4.12: Build commands](#)

4.1 Specifying a toolset

When building your application, you can use one of supported toolsets: ST Assembler/Linker, Metrowerks C (Hiware), or Cosmic C, and their respective options and file formats.

Table 29. Supported toolsets and file formats

Toolset name	Supported microcontrollers	Initial source files	Compiler/ assembler output	Linker output	Final executable
ST assembler/linker	ST7, STM8	.asm	.obj, .lst	.map, .tab	.s19, .hex
ST7 Metrowerks (1.1)	ST7	.c, .asm	.o, .dbg	.abs	.abs, .elf
ST7 Cosmic C	ST7	.c, .s	.o	.st7	.elf
STM8 Cosmic	STM8	.c, .s	.o	.sm8	.elf
Raisonance C	ST7, STM8	.asm, .c	.o	.aof	.elf

The toolset to use during the build is specified at the project level. The **Project Settings** window provides the interface for configuring toolset options, libraries and include paths that are specific to building an application. This organization makes it possible for one workspace to contain projects that are built using different toolsets.

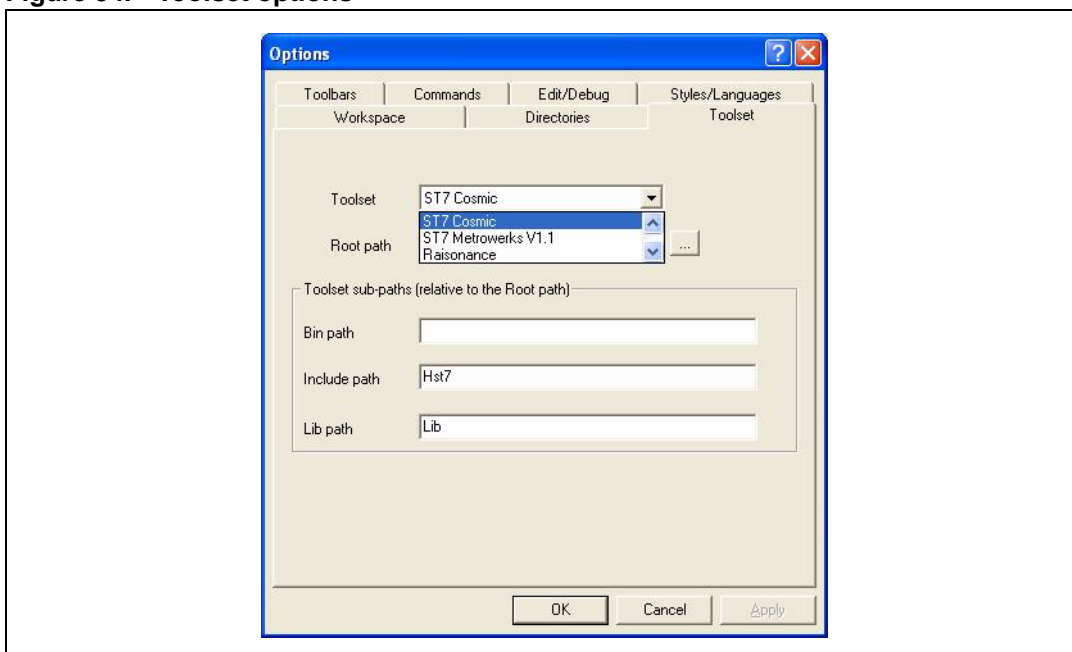
Before you start customizing toolset paths, you will want to ensure that the default paths are correct for the installation of the toolsets on your host PC.

Toolset path information

Upon installation, STVD will attempt to identify the toolsets and their paths. You can confirm or modify default path information for the supported toolsets by selecting **Tools > Options** (see [Figure 54](#)).

The resulting **Options** window provides a **Toolset** tab that allows you to specify the Bin, Include and Library paths for your toolset on your host PC. It also provides a **Directories** tab, where you can specify any additional default subpaths for include files and libraries that you typically use when building your microcontroller applications. The paths that you specify in the Options window will be the default settings applied for any project using a toolset. However, you can also specify a project specific toolset and paths in the **General** tab of the **Project Settings** window for any project (see also [Section 4.5: Configuring project settings on page 84](#)).

Figure 54. Toolset options



4.2 Loading and creating workspaces (.stw)

The workspace (*filename.stw*) is the file that contains the project(s) you are building to generate your application. Typically, a workspace contains one or more projects and their source files and dependencies. This section provides information on:

- [Loading an existing workspace](#)
- [Creating a new workspace](#)

Note: You cannot create a project and build or debug an application until you have created a workspace.

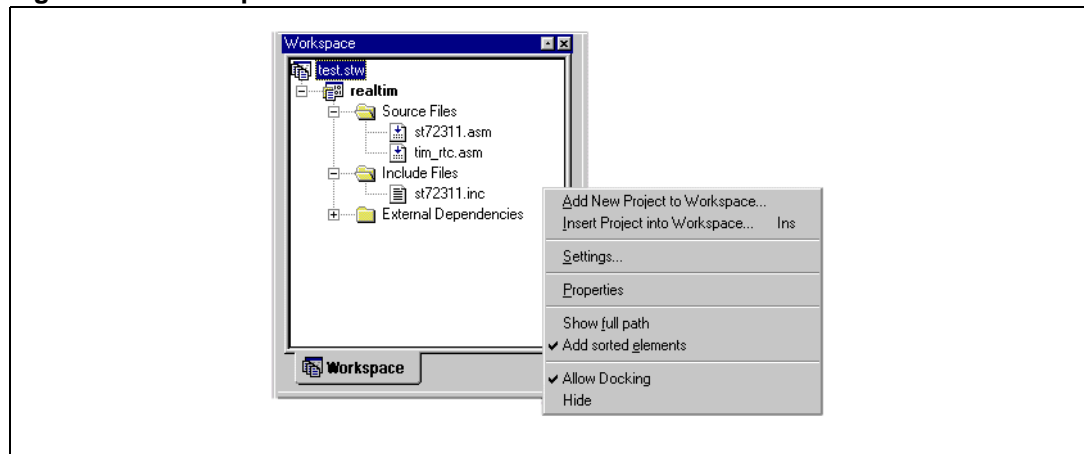
4.2.1 Loading an existing workspace

- [Opening a .stw workspace](#)
- [Opening a .wsp workspace from STVD7 prior to version 3.0](#)

Opening a .stw workspace

To load a workspace (*.stw) created by STVD7 3.0 or later versions, select **File > Open Workspace**. The **Open Workspace** browse window will open. Locate the workspace file (.stw), highlight it, and then click on **Open**. The workspace will appear in the Workspace window, typically on the left side of STVD's main window (see [Figure 55](#)).

Figure 55. Workspace window



Opening a .wsp workspace from STVD7 prior to version 3.0

You can open a workspace that was created by a version of STVD7 prior to version 3.0. This method automatically creates a .stw workspace based on the *makefile* used by your .wsp workspace.

Caution: A .stw workspace created by opening a .wsp and wrapping the *makefile*, does not allow you to take full advantage of STVD's build interface. Instead, you must manually maintain your *makefile* to manage the building of your application. For information about creating a .stw workspace that allows you to use STVD's graphical build interface, refer to [7 steps to full migration on page 72](#).

To open a .wsp workspace:

1. Select **File > Open Workspace** from the main menu bar.
2. In the resulting browse window, select **Old Project Workspace (*.wsp)** in the **Files of Type** field. Then select the .wsp file and click on **Open**.
3. Read the message in the **Import Workspace** message box. Click on **Continue** to open your .wsp. Click on **Cancel** if you want to abandon the import procedure and do a full migration instead (refer to [7 steps to full migration on page 72](#)).

A pop-up dialog box will ask you if you want to add the files as filtered by your .wsp workspace. Click on **Yes** to add the files automatically.

Note: If a pop-up message appears indicating that your makefile was not found, click on **OK** and then check that your makefile is in the same directory as the `.wsp` that you are trying to open.

4. In the **MCU Selection** window, pick the target microcontroller for your application from the list, click on **Select** and then **OK**.
5. Select **File > Save Workspace** from the main menu bar.
A workspace in `.stw` format is created in the working directory. In the future, you will open this file and not the old `.wsp` file. You can now continue building and debugging your application.

The `.stw` workspace that you have created is limited to using your *makefile* when you build and rebuild your application. For this reason you do not have access to the tabs in the **Project Settings** window that provides an interface for controlling your toolset and the building of your application.

4.2.2 Creating a new workspace

STVD provides several options for creating a workspace in the **New Workspace** window. To access this window, select **File > New Workspace**. From this window you can:

- [Create a workspace with a new project on page 23](#)
- [Create an empty workspace](#)
- [Create a workspace from an existing project](#)
- [Create a workspace by wrapping an executable](#)
- [Create a workspace by wrapping a makefile](#)

Create an empty workspace

You can create an empty workspace by selecting the **Create an Empty Workspace** icon.

In the New Workspace window:

1. Type the name of the workspace in the **Workspace Name** field.
2. Enter the working directory where files are to be saved in the **Workspace Location** field, or use the browse button to find a location.
3. Click on **OK** to create your workspace.

Create a workspace from an existing project

This option allows you to create a workspace around an existing STVD project file (`.stp`).

1. From the **New Workspace** window, select the **Create from Project** icon and click on **OK**.

A browse window opens. By default it is set to display `.stp` project files.

2. Highlight the project file that you want to use to create your workspace and click on **Open**.

Your workspace is created and named `project_filename.stw`. The new workspace file appears in the Workspace window, typically on the left side of STVD's main window.

Your workspace contains the project you identified and its contents.

Note: When creating a workspace in this manner, make sure that your source files are in the correct directory relative to the project file. Otherwise you must update the paths by adding the files to the Source Files folder from their actual location (see [Section 4.3.3: Adding and removing folders and files on page 83](#)).

Create a workspace by wrapping an executable

You can create a workspace for an executable file (`.abs`, `.elf`, `.hex`, `.s19`) for debugging. However, during debugging you will only have access to the disassembled code and not the original source code.

1. To create your workspace, select the **Wrap Executable** icon and click on **OK**.
A browse window opens.
2. Highlight the executable file that you want to use to create your workspace and click on **Open**.
The **New Project** window opens.
3. Here, enter a name for the project file (`.stp`) and the project's location.
By default your project is named `executable_filename.stp` and its location is the directory of the executable file.
4. Now, select your toolset from the list box and enter its path. Click on **OK**.
Your workspace is created and named `executable_filename.stw`. The new workspace file appears in the **Workspace** window, typically on the left side of STVD's main window. Your workspace contains a project `executable_filename.stp`.

You can now identify your debug instrument and debug the application.

Create a workspace by wrapping a makefile

Early versions of STVD7 used a makefile (`.mak`) to define the build process for source files and a specified toolset. Wrapping your makefile to create a new workspace allows you to rapidly transition projects to STVD 3.0 and later releases, with certain limitations.

However, if you follow this migration procedure you will not have access to the build options in the **Project Settings** window. Any changes to options must, therefore be made manually to the makefile. STVD will not automatically manage project dependencies either. You, must instead use your makefile to specify dependencies and you must verify these dependencies yourself. Finally, if you add files to the project to edit them, they are not automatically added to the build. In order to be taken into account when building, you must manually update the makefile to include the new files.

For information about migrating your project in a manner that preserves the full range of STVD's build features, refer to [7 steps to full migration on page 72](#).

To create your workspace by wrapping an existing makefile:

1. Highlight the **Wrap Makefile** icon and click on **OK**.
A browse window opens. By default it is set to display `.mak` makefiles.
2. Highlight the makefile that you want to use to create your workspace and click on **Open**.
You will receive an information message, notifying you that STVD is going to wrap the makefile in a workspace.
3. Click on **OK** to continue.
The **New Project** window opens.
4. Here, enter a name for the project file (`.stp`) and the project's location. By default your project is named `makefile_name.stp` and its location is the directory of the makefile. Click on **OK**.
The MCU selection window appears.

5. Select the MCU for your application and then click on **OK**.
Your workspace is created and named *makefile_name.stw*. The new workspace file appears in the **Workspace** window, typically on the left side of STVD's main window. Your workspace contains a project with the makefile that you have identified and the External Dependencies folder.

Before you start debugging your application you will have to identify the executable file to debug and you may have to rebuild your application.

4.3 Creating and loading projects in a workspace

The project file (*.stp*) contains the information that you need to build an application with a specific toolset and for a specific microcontroller. It contains the source files needed to build the application, the build configuration and the target MCU's memory map. This section provides information about:

- [Loading an existing project](#)
- [Creating new projects](#)
- [Adding and removing folders and files](#)

Note: Until you create a project or add an existing one to your workspace, you do not have access to STVD's build and debug commands.

4.3.1 Loading an existing project

To load an existing project:

1. Select **File > Insert Project into Workspace**.
The **Insert Project** browse window will open.
2. Locate the project file (*.stp*), highlight it and then click on **Open**.
The project will appear in the **Workspace** window, typically on the left side of STVD's main window (see [Figure 55 on page 77](#)).

STVD will not allow you to insert a project into a workspace if it already contains a project with the same name.

4.3.2 Creating new projects

STVD provides three options for creating a new project in a workspace.

To create a new project, select **Project > Add New Project to Workspace**. The **New Project window** opens. From this window you can:

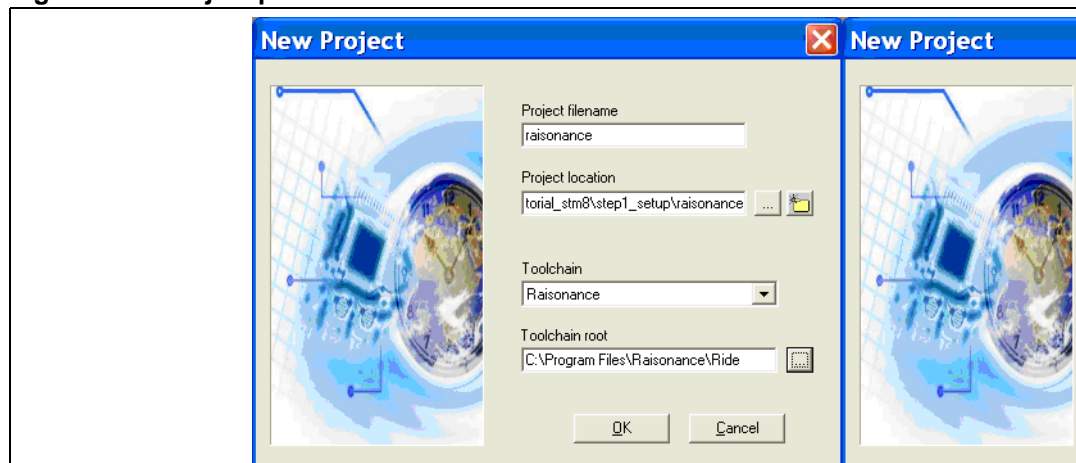
- [Add a project \(.stp\) to your workspace](#)
- [Create a project from an existing executable](#)
- [Create a project from an existing makefile](#)

Add a project (.stp) to your workspace

Click on the **New Project** icon and then **OK**. This launches the **New Project** wizard as shown in [Figure 56](#). In this window you will enter all of the information required to create the project including:

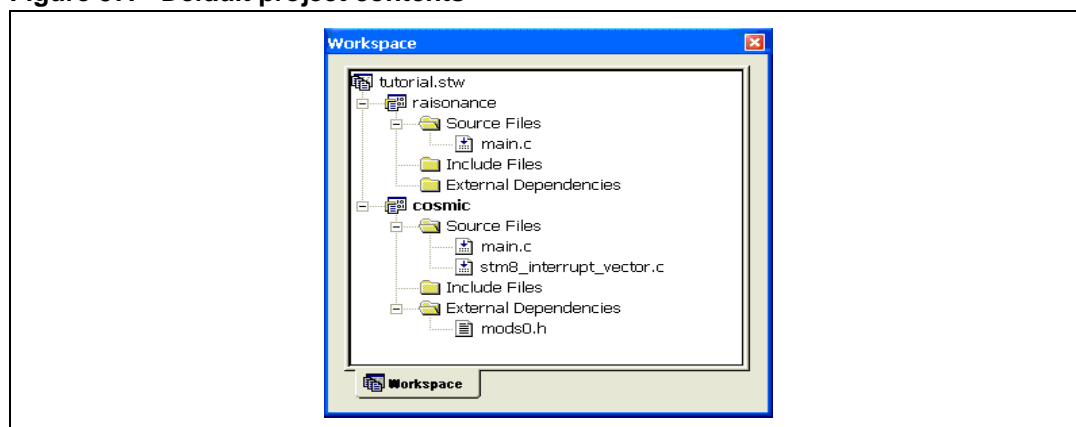
- A project name in the **Project Filename** field.
- The path name where you want to store the project and the resulting files in the **Project Location** field. The path for the workspace is used by default.
- The toolset to use when building and the path for the toolset.

Figure 56. Project parameters



When you click on **OK**, the project (*filename.stp*) is added to the workspace in the STVD **Workspace** window. By default, the project contains folders called **Source Files**, **Include Files** and **External Dependencies**. These folders are used to organize your project within STVD. They do not exist in the working directory. Other files may be added to the project folders automatically based on your toolset's default settings. STVD will change these according to your project build settings and your application.

Figure 57. Default project contents



When you save your workspace, the working directory is updated with the project file and two folders: **Debug** and **Release**. These folders are the default locations for storing your application executable and any intermediate files depending on whether you build using the Debug or Release configuration.

Create a project from an existing executable

To create a project from an executable (.abs, .elf, .hex, .s19):

1. Select the Create from Executable icon and click on **OK**.
You will receive an information message asking you if you want to create a project to wrap the executable.
2. Click on **OK** to continue.
The **New Project** window opens.
3. Here enter a name for the project file (.stp) and the project location.
By default, your project is named *executable_filename.stp* and its location is the directory of the executable file.
4. Next, select the appropriate microcontroller in the **MCU Selection** window and click on **OK**.
Your project is created and appears in the **Workspace** window. It contains the default folders and any files specified by the toolset settings. The **Project Settings** window contains the **Debug** tab and the **MCU Selection** tab where you can change the microcontroller selection.

You can now identify your debug instrument and debug the application.

Create a project from an existing makefile

Early versions of STVD used a makefile (.mak) to define the build process for source files and a specified toolset. To support the migration of projects from earlier versions, STVD allows you to create a project around a makefile. In this way you preserve the information and conditions applied when building previous applications. For more information about this process, refer to [Migrating old workspaces \(STVD7 2.5.4 and prior versions\) on page 71](#).

To create your project:

1. Highlight the **Create from Makefile** icon and click on **OK**.
A browse window opens. By default it is set to display .mak makefiles.
2. Highlight the makefile file that you want to use to create your project and click on **Open**.
You will receive an information message, notifying you that STVD is going to wrap the makefile in a new project.
3. Click on **OK** to continue.
The **New Project** window opens.
4. Here enter a name for the project file (.stp) and the project location.
By default, your project is named *makefile_name.stp* and its location is the directory of the makefile.
5. Now, select your toolset from the list box and enter its path. Click on **OK**.
Your workspace has been created and named *makefile_name.stw*. The new project file appears in the **Workspace** window, typically on the left side of STVD's main window. Your project contains the makefile that you have identified and the External Dependencies folder.

Before you start debugging your application you will have to identify the executable file to debug and you may have to rebuild your application. For more information about this process, refer to [Migrating old workspaces \(STVD7 2.5.4 and prior versions\) on page 71](#).

4.3.3 Adding and removing folders and files

Projects contain the files that you will use to build your application. These can be organized in folders. By default, projects contain the following folders:

- **Source Files** — this folder contains the source files (.asm, .c, .s) that you identify for the building of your application.
- **Include Files** — this folder contains header files required to build your source code. These files are specified by the include statements in your project settings.
- **External Dependencies** — this folder contains any dependencies specified by the source files or the toolset. The folder's contents are updated automatically and cannot be modified by the user.

Adding folders

You can create your own folders in your project to create a virtual project structure that will help you organize the links to your source files. Folders can be created in the project or in any folder except the **Dependencies** folder.

To add folders to your project or in another folder:

1. Right-click on the project or on the folder to get the contextual menu. In the contextual menu, select **New Folder**.
The **New Folder** dialog box opens.
2. Enter the name for the folder in the provided field and click on **OK**.
The folder is added to your project.

Adding files

To add source files to the source folder:

1. Right-click on the folder to get the contextual menu. From the menu, select **Add Files to Folder**.
A browse window opens.
2. Select the file(s) you want to add to the folder and click on **Open**.
STVD will not allow you to add a file to a project if the project already contains a file with the same path name. Files with the same name can be added to the same project if they have different paths. A workspace may contain files with the same path name, as long as they are in different projects.

The folder contextual menu also offers the **Add Sorted Elements** option. When this option is checked, STVD protects the order of the files in the project. Upon build, the command lines for the source files are executed in the same order as the files in the project from the top (first) to the bottom (last). When this option is checked, STVD will not allow you to change the order of the files. If you disable this option, change the order, and then reactivate this option, STVD will maintain the new ordering of the files. This option is important when using the ST Assembler/Linker toolset, which is sensitive to the ordering of source files upon assembly (see [Ordering source files before assembly on page 88](#)).

4.4 Project build configurations

Once you have set up your project with the source files to build your application, you are ready to configure your project for building. STVD offers extensive customizations for the supported toolsets via the **Project Settings** window. In addition, these settings can be

saved as **Build Configurations** so that you can easily switch between two or more configurations.

The name of each build configuration also corresponds to a folder in your project directory that is created automatically by STVD (for example, the Debug configuration corresponds to the Debug folder in your project directory). This folder is the default Output directory for your project and the Intermediate directory for each of your files when you build your application with the corresponding build configuration.

By default, two configurations are automatically created for each project: **Debug** and **Release**. The Debug configuration applies your toolset's default settings to produce a version of your application that is ready to be debugged. The Release configuration applies your toolset's default settings to produce an optimized version of your application that is ready to be programmed to your target microcontroller.

Setting the build configuration

To set the build configuration for your project:

1. Select **Build > Configurations**.
The project configurations window opens.
2. If your workspace contains more than one project, use the **Project** list box to select the project that you want to apply the build configuration to.
3. Select the configuration you want from the list in the **Configurations** field and then click on **Set Active**.
The active configuration is in **bold** type.
4. Click on **Close**.

Once you have set the build configuration, you can customize it in the **Project Settings window**.

Creating a new build configuration

When you change from one build configuration to another, the Output directory is automatically changed to designate the appropriate folder. Likewise, the Intermediate directories for all files are automatically changed to use the corresponding folder, except if you have manually designated an alternate folder for the Intermediate directory. For more information, refer to [Section 4.10: Configuring folder and file settings on page 156](#).

To create a new build configuration based on the settings in any of the existing configuration:

1. Select **Build > Configurations**
2. Click on **Add** in the New Configuration window
3. Type a name for the configuration
4. In the **Copy Settings From** field, select the existing configuration to use as the basis for the new configuration and click on **OK**.

4.5 Configuring project settings

The **Project Settings** window provides the standard interface for customizing the build of a project or specific files within a project. The tabs in the Project Settings window present commonly used options in menus, list boxes and check boxes. In addition, these tabs provide fields for entering custom commands and other supported options.

Some aspects of this interface are common to all toolsets, these common tabs are described in the following sections:

- [General settings tab](#)
- [Debug settings tab](#)
- [MCU selection tab](#)
- [Pre-link settings tab](#)
- [Post-build settings tab](#)

Several tabs are specific to your toolset. STVD's graphical interface adapts automatically to the toolset that you specify for your project in the **General** tab of the Project Settings window. The options in each of the tabs that are specific to your toolset are explained in:

- [Customizing build settings for ST Assembler/Linker toolset](#)
- [Customizing build settings for Cosmic C toolsets](#)
- [Customizing build settings for Raisonance C toolset](#)
- [Customizing build settings for Metrowerks C toolset](#)

4.5.1 General settings tab

This tab allows you to specify general project information, such as the toolset to use and the paths for the output directory, as well as for include, bin and library files.

To access this tab, select **Project > Settings** to view the **Project Settings** window. Then click on the **General** tab. This tab should contain the toolset that you specified when you created the project and the default path information that is specified for it in the **Options** window (see [Section 4.1: Specifying a toolset on page 75](#)).

To replace the defaults with a project specific toolset and/or path options:

1. Change the toolset by selecting one from the **Toolset** list box.

Note: If you change the toolset selection for a toolset that does not support the selected MCU, you will receive an error message to indicate that you must also select an MCU that is supported by the new toolset.

2. Place a checkmark next to **Project Specific Toolset Path**.
The fields change from a protected state (gray) to an active state (white).
3. If necessary, enter the directory path for the toolset in the **Root Path** field by typing the path or using the browse button to locate it.
4. In the **Toolset sub-paths** fields, specify bin paths, include paths and library paths to use when building the application.

These paths are entered relative to the toolset path that you specified in step 3.

Note: Some versions of the Metrowerks toolset use the filename *prog* instead of *bin* for the bin subpath.

5. In the **Output Directory** field you can specify a folder where output will be stored when building.

By default, output is stored in the `Debug` or `Release` folders in the working directory, depending on whether you are using the debug or release configuration (see [Section 4.4: Project build configurations on page 83](#)). The folder specified as the `Output Directory` for your project is also used as the `Intermediate Directory` for all files unless you specify otherwise in the [General settings for files and folders](#).

6. To confirm the settings you have made, click on **OK**.

4.5.2 Debug settings tab

This tab allows you to specify common settings for debugging your application.

To access this tab, select **Project > Settings** to view the **Project Settings** window. Then click on the **Debug** tab. Here you can specify:

- the **Executable** that you want to debug. By default, STVD will look for the target file specified by the linker. However, you can also use the browse button to locate an executable file manually.
- the **Debug working directory**. When using the Debug configuration, by default this is the folder specified in the **Output Directory** field of the **General** settings tab.
- the **Source Directories** where any source files are located. By default, this field will identify the directory path for your project. You can add other directories by clicking on the **New (Ins)** button in the gray field. A field will open that allows you to type the new directory path or use a browse button to locate the path.

To confirm the settings you have made, click on **OK**.

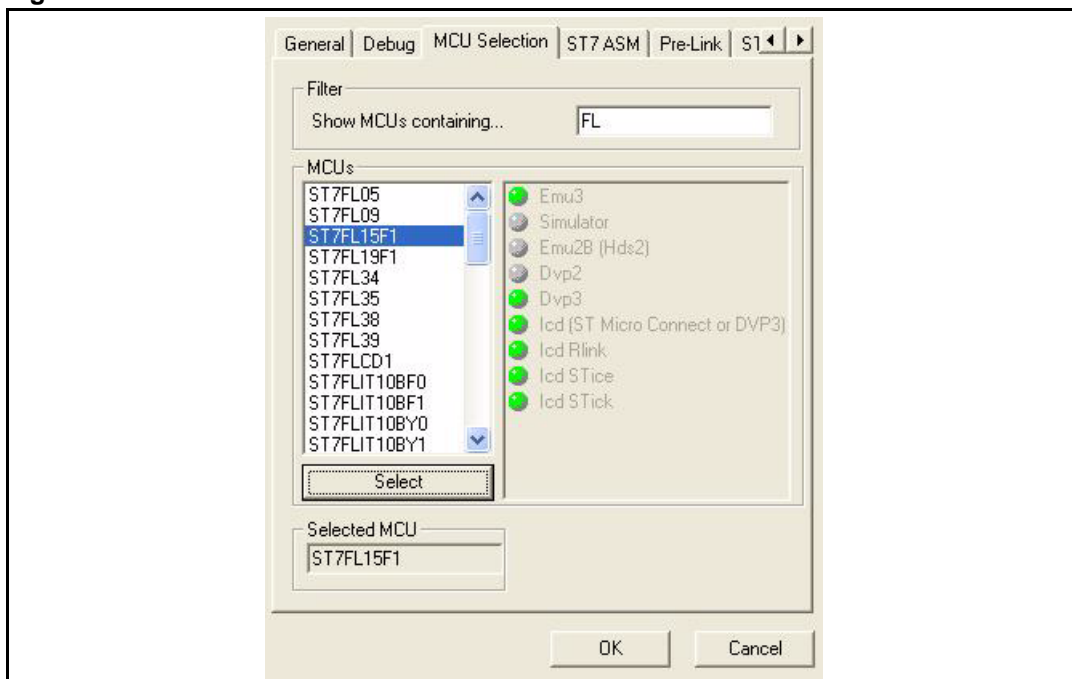
4.5.3 MCU selection tab

The **MCU Selection** tab is very important because when you build an application, your toolset uses the memory map for your specific MCU. Therefore, any time you build an application, the resulting executable is directly associated with the memory map of your target microcontroller. As a result, if you change microcontrollers, you will have to rebuild your application.

This also has an impact on debugging, because STVD only allows you to connect to a debug instrument that supports the target microcontroller that you specified prior to building.

To access this tab, select **Project > Settings** to view the **Project Settings** window. Then click on the **MCU Selection** tab.

Figure 58. MCU selection tab



The current microcontroller setting is displayed in the gray **Selected MCU** field at the bottom of the tab. The MCU field to the left displays a complete listing of supported microcontrollers. When you highlight an MCU name in the **MCU** field, colored indicators tell you which debug instruments support the highlighted MCU.

- **Gray** = Not supported
- **Green** = Supported
- **Yellow** = Supported in beta version

To select a microcontroller, click on the microcontroller name in the MCU list. To help you locate your MCU, as you type the name in the **Filter** field, the choices are progressively reduced according to what you type. To confirm your selection, click on **Select**. Your MCU's name will appear in the **Selected MCU** field.

4.5.4 Pre-link settings tab

The **Pre-link** tab allows you to control what occurs during the pre-link stage of compiling, by entering commands defined by your toolset.

To access this tab, select **Project > Settings** to display the **Project Settings** window. Then click on the **Pre-link** tab. Here you can:

- Type a Description. This description is displayed in the **Build** tab of STVD's **Output** window when the pre-link commands are executed.
- Enter a Pre-link Command. The **Directory** and **File** buttons provide lists of macros to help you formulate commands. Refer to your toolset's user documentation for more information about the pre-link commands that it supports.

To confirm the settings you have made, click on **OK**.

4.5.5 Post-build settings tab

The **Post-build** tab allows you to control what occurs during the post-build stage of compiling by entering commands defined by your toolset.

To access this tab, select **Project > Settings** to display the **Project Settings** window. Then click on the **Post-build** tab. Here you can:

- Type a Description. This description is displayed in the **Build** tab of STVD's **Output** window when the post-build commands are executed.
- Enter a Post-build Command. The **Directory** and **File** buttons provide lists of macros to help you formulate commands. Refer to your toolset's user documentation for more information about the post-build commands that it supports.

To confirm the settings you have made, click on **OK**.

4.6 Customizing build settings for ST Assembler/Linker toolset

The ST Assembler/Linker toolset builds your application from source code that is written in Assembler language. Building requires the following steps:

- **Assembler:** Translates assembler source code into objects and generates listings with relative addresses.
- **Linker:** Links objects from the source code and any included objects from libraries in a `.cod` file.
- **OBSEND:** Translates the `.cod` file into the final executable format, which you specify.
- **Abslist:** Generates listings with absolute addresses from listings with relative addresses, the executable file and the map file.

Ordering source files before assembly

The ST Assembler Linker is sensitive to the ordering of files during assembly. For example, a source file that defines a label must be assembled before other source files that use this label.

To protect the order of the source files for assembly you should:

1. Add the source files to your project (**Project > Insert Files Into Project**)
2. Right-click on the project in the Workspace window and disable the **Add Sorted Elements** option in the project contextual menu.
3. Drag and drop the source files into the correct order. The first file to be assembled in at the top, the last is at the bottom.
4. Right-click on the project in the Workspace window and reactivate the **Add Sorted Elements** option in the project contextual menu.

Once the Add Sorted Elements option is reactivated, you can not drag and drop the files into a new, non-alphabetic order. You can, however move the files individually to different folders in the Workspace window.

Provided ASM include files for ST Assembler/Linker

STVD comes with a complete set of include files (`*.inc` and `*.asm`) that define the registers and vectors for all the supported microcontrollers. They are intended to save you the time that you might spend writing these files yourself.

Caution: The ASM include files provided with STVD are not the same as those provided for the ST7 library. ST7 library users should use the C header and/or ASM include files specific to that library.

These files are installed with the ST Assembler-Linker toolset. For standard installations these files are located at:

```
C:\Program Files\STMicroelectronics\st_toolset\asm\include\
```

You can use these files from their installed location. However, if you intend change these files to meet the requirements for your application, you should copy the files for your target microcontroller to your project's working directory and make amendments to the copied files.

Project settings for ST Assembler/Linker

The **Project Settings** window allows you to configure settings for these steps. This window contains five tabs that are common to all the toolsets (General, MCU selection, Debug, Pre-

link and Post-Build) and three tabs that contain options and settings that are specific to ST Assembler/Linker toolset.

This section provides details about configuring the options that are specific to the ST Assembler/Linker toolset. These options include:

- [ST ASM tab](#)
- [ST Link tab](#)
- [ST Post-Link tab](#)

Note: For more details regarding options for this toolset, refer to the **ST Assembler-Linker User Manual**.

Configuring projects settings in the five common tabs is described in [Section 4.5: Configuring project settings on page 84](#).

4.6.1 ST ASM tab

The ST ASM tab provides an interface for setting the options in the command line for the ST Assembler. The gray Command line field displays the changes to the command line as you make them.

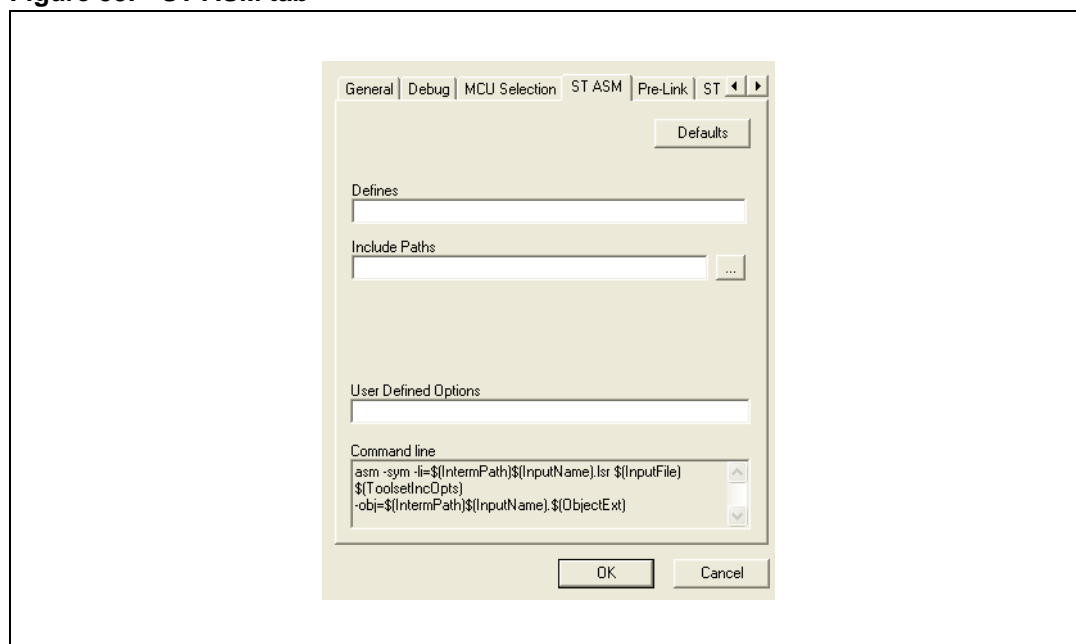
The **ASM** tab allows you to specify **defines** and **include paths**.

Defines

The **Defines** field allows you to apply `-d` option in order to define a string that is to be replaced by another string during assembly. Type the string that is to be replaced in the field. The `-d` option is automatically added to the command line. To enter the string to replace it, type a space then type the replacement string.

To make a second entry, type another space and then the second string that is to be replaced. The `-d` is automatically added to the command line. Type a space followed by the second replacement string.

Figure 59. ST ASM tab



Include paths

The **Include path** field introduces the `-I` option in the command line. This option allows you to specify search paths for files to be included during assembly. To enter a path, start typing the path, or use the browse button to identify the path. When typing the path, the `-I` option is automatically added to the command line.

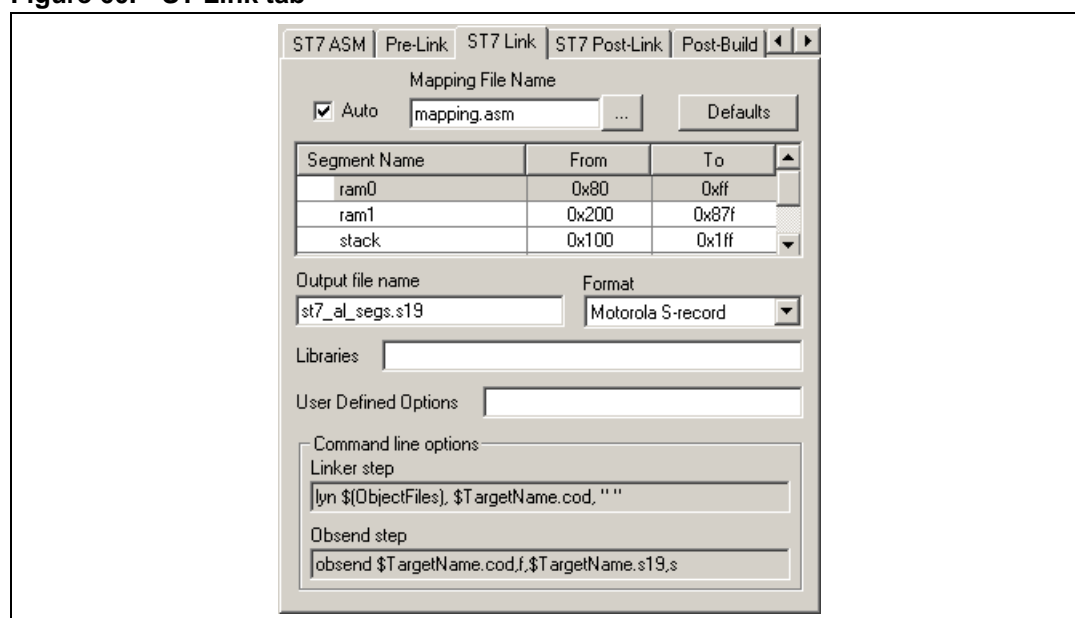
To define a second path, type a semicolon (;) and then the second path. Again, the `-I` option is automatically added to the command line.

4.6.2 ST Link tab

The **ST Link** tab provides you with an interface for setting linker and OBSEND options and for defining segments in your microcontroller's memory to which sections of your code will be assigned.

The gray **Command line** fields display changes as you make them. This tab contains two Command line fields, one for the **Linker step** and the other for the **OBSEND step**.

Figure 60. ST Link tab



Automatic segment declaration

When the **Auto** checkbox is not checked, the Linker requires that segments be declared in your applications source files.

When the **Auto** checkbox is checked, a configurable mapping is displayed in the **Linker** tab. This mapping is used to automatically generate a file in assembly language (default name is **mapping.asm**) that contains the declarations of memory segments to which you can assign sections of code. You assign code to a memory segment by using the segment name that you have specified in the mapping in your source code. For more information about assigning code to memory segments, refer to your *ST Assembler Linker User Manual*.

Note: If you have declared segments in your source code and using the Linker tab, it is up to you to ensure their consistency.

The source file with the segment declarations (default is mapping.asm) is regenerated each time you build the application when the Auto feature is enabled. If you make changes to the file and Auto is enabled, your modifications will be overwritten when STVD updates the file based on the mapping in the Linker tab.

This file must always be assembled first as the segment names are used in the other source files in your project. For this reason it is automatically placed at the top of the list of source files in your project in the **Workspace** window.

You can change the name of the output file assembly source file by typing a new name in the **Mapping File Name** field.

Modify the segment mapping

The segment mapping in the Linker tab can be entirely modified. The interface allows you to:

- Change any segment name and address range.
- Add your own segments and specify their address ranges
- Delete any segment

However, the interface does not allow you to delete segment if it is the last segment in the mapping. At least one segment must always be declared.

Right-clicking on a segment in the mapping opens a contextual menu with the following commands:

- **Add Segment** - Adds a row of empty fields to the mapping. Enter the name of the new segment and press the **Enter** key. If you do not enter a segment name, and only press enter, the new segment is removed from the mapping. You cannot use the same segment name twice. Naming is case sensitive.
- **Change** - Allows you to change the value or entry in the selected cell of the mapping. Address values can be entered in decimal or hexadecimal (use "0x_____" notation) formats.
- **Delete** - Deletes the selected entry. The last segment entry in the mapping cannot be deleted.

Output filename

The **Output filename** field allows you to specify the name of the final output. By default, the output file is *project_name.s19*.

Libraries

The **Libraries** field allows you to enter the path names for object libraries to include when linking your application. These path names appear in the command line for the linker step.

To specify a library to include, type the library path name in the field. To add a second library type a semicolon (;) followed by the second path name.

Format

In the **Format** list box you can specify the file format of the executable. When a format is specified a code appears at the end of the OBTEND command line. In the **Format** list box, you can choose from the options described in [Table 30](#).

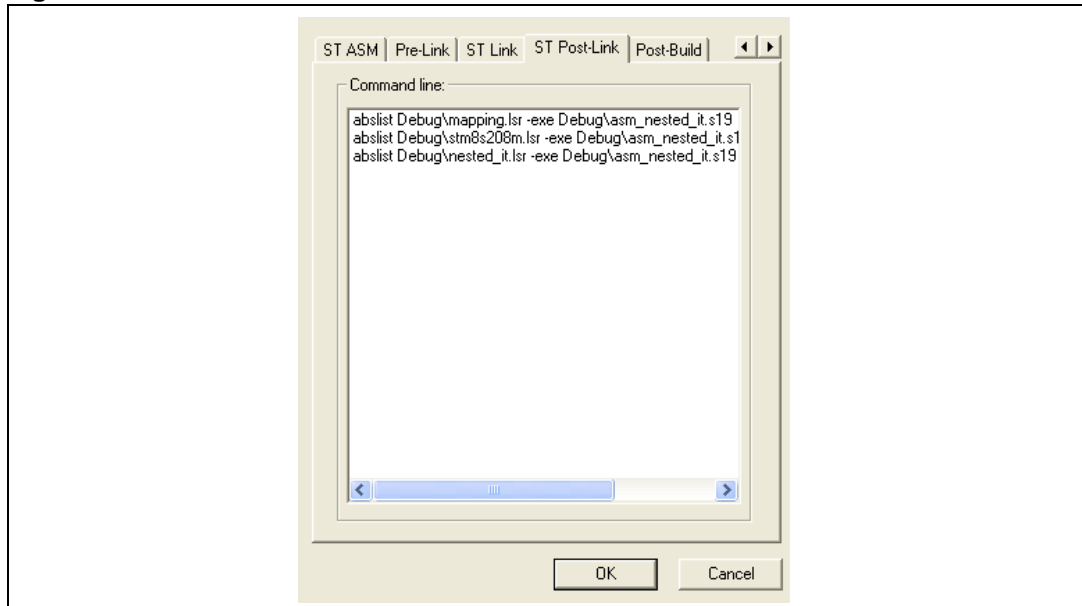
Table 30. Formats for OBSEND output

Format	Description
Intel Hex (i)	Intel hexadecimal format
Intel Hex extended (ix)	Intel hexadecimal format allowing 32 bit addressing
Motorola S-record (s)	Motorola hexadecimal format with 16 bytes of data per line

4.6.3 ST Post-Link tab

The **ST Post-Link** tab allows you to view any instructions that entail post-link actions that result from the options or settings that you have made in the other tabs of the **Project Settings** window. Specifically, during the post-link step, a listing with absolute addresses that are used by the debugger is generated. You cannot edit the entries in this window directly.

Figure 61. ST Post-Link tab



4.7 Customizing build settings for Cosmic C toolsets

The **Project Settings** window for building your application with the Cosmic C toolset contains five tabs that are common to all the toolsets (General, MCU selection, Debug, Pre-link and Post-Build) and three tabs that contain options and settings that are specific to the Cosmic C compiler toolsets.

This section provides details about configuring options that are specific to the **Cosmic C toolset**. These options include:

- [Cosmic C compiler tab](#)
- [Cosmic C Assembler tab](#)
- [Cosmic C linker tab](#)

Configuring projects settings in the five common tabs is described in [Section 4.5: Configuring project settings on page 84](#).

Note: For complete details regarding options for this toolset, refer to the **C Cross Compiler User's Guide** provided by Cosmic.

Provided microcontroller-specific C header files for Cosmic

Depending on your MCU you can include a header file provided with STVD, which defines the device's peripheral registers. This will save you the time of creating this header file yourself.

Caution: The C header files provided with STVD for ST7 and STM8 microcontrollers are not the same as those provided for the ST7 Library and STM8 Library. ST7 and STM8 library users should use the C header and/or ASM include files specific to that library.

Depending on your installation, the C header files provided with STVD are typically located at:

```
C:\Program Files\STMicroelectronics\st_toolset\include\
```

To build your project using the provided header file for your device, specify its inclusion in your application code. The name of the file to use is *device_name.h*.

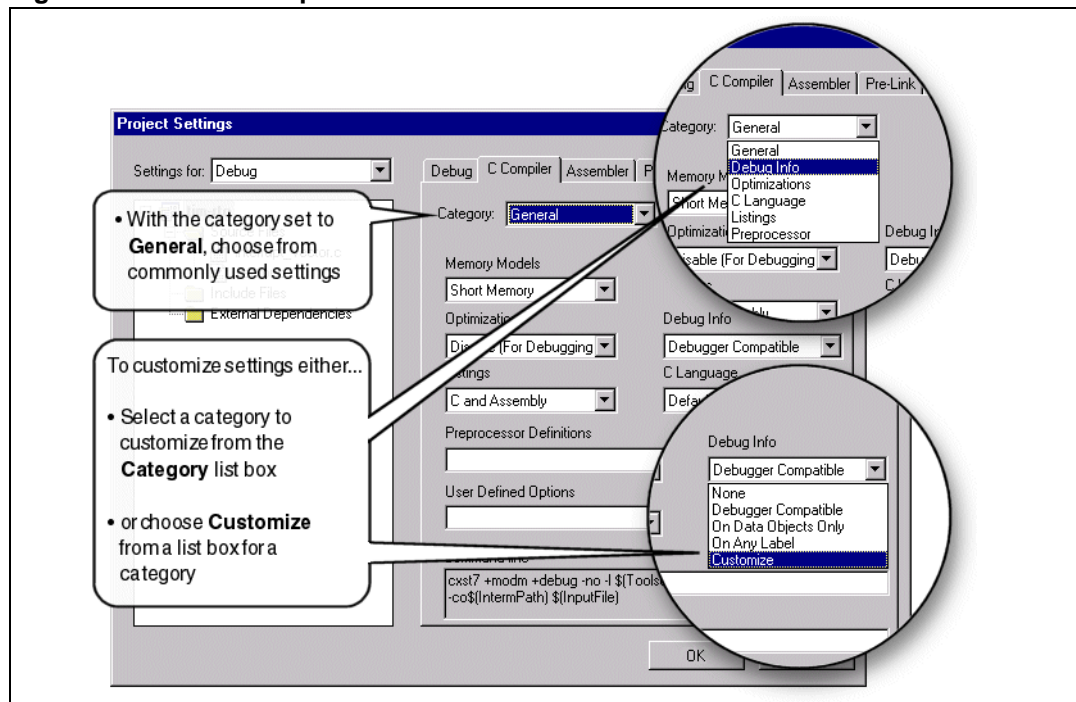
Header files are provided for virtually all the supported devices. If you do not find your device's name in the header files, this may be because the full name is not used. To determine if your device has a header file, you can check the **Peripheral Registers** window (**View > Peripheral Registers**) in the debug context. If peripheral register information is provided in this window, then a header file has been generated for your device. The correct file for your device, is the one with the device name you found in the **Peripheral Registers** window.

Using the Project Settings interface

The tabs (C Compiler, Assembler and Linker) have a common interface that allows you to choose from standard configurations, customize options and view changes to the command line as you apply different options.

When the **Category** List box is set to **General**, the tab provides check boxes and list boxes for easy access to commonly used options and configurations (see [Figure 62](#)).

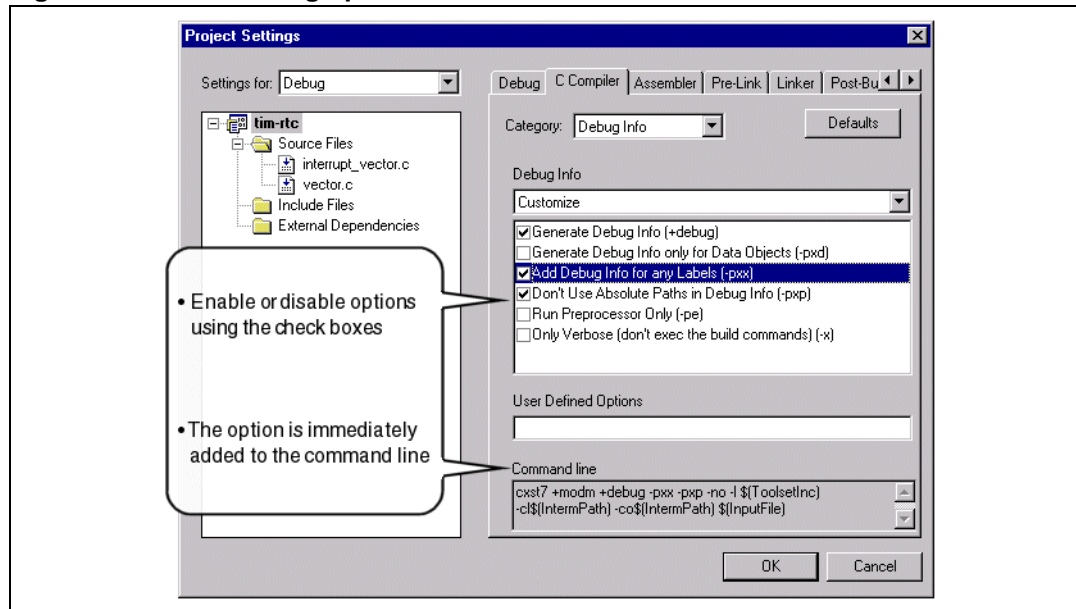
Figure 62. Standard options



To further customize options, select a category in the **Category** list box, or select **Customize** in a list box for a category of settings. The view in the tab will change to present more options (see [Figure 63](#)).

You can enable or disable options by clicking on the check boxes. When you invoke an option, the change to the command line is immediately visible in the gray **Command Line** window at the bottom of the tab.

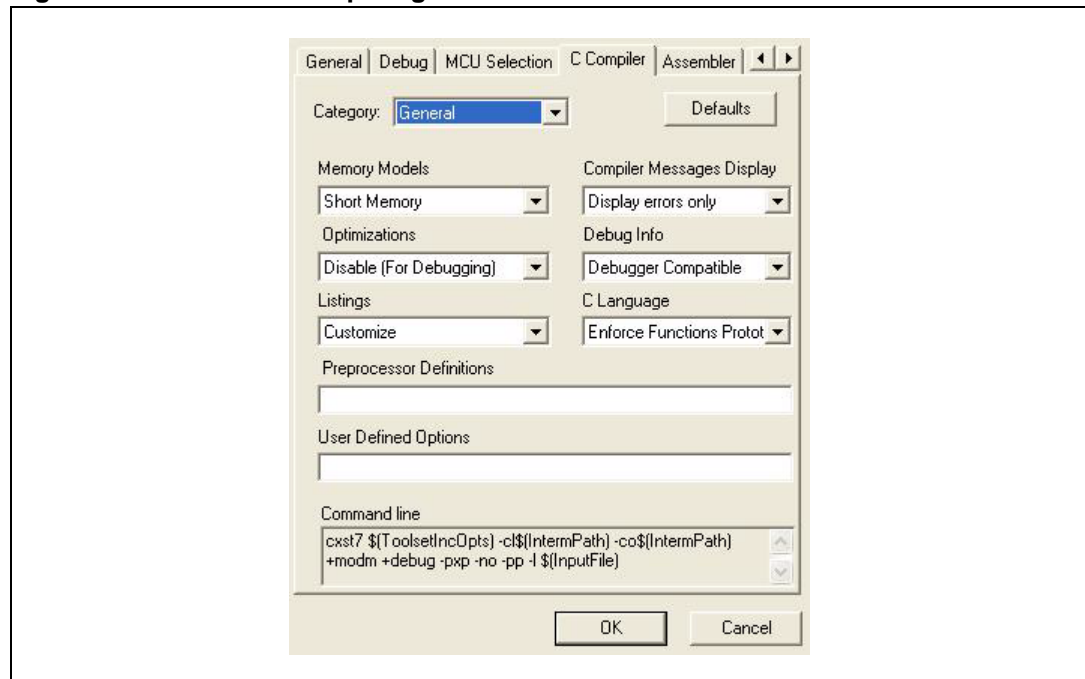
Figure 63. Customizing options



4.7.1 Cosmic C compiler tab

The **C Compiler** tab provides an interface for setting the options in the command line for the Cosmic C compiler for ST7 (cxst7) or for STM8 (cxstm8). The gray **Command line** field, displays all the changes to the command line as you make them.

Figure 64. Cosmic C compiler general view



The **Category** list box allows you to access a general view for easy access to standard settings and options. You can also change the category to access views where you can customize settings in greater detail.

In the **Category** list box, you can choose one of the following views:

- **General:** see [General settings for Cosmic toolset](#)
- **Debug Info:** see [Customizing Cosmic C compiler debug information settings](#)
- **Optimizations:** see [Customizing Cosmic C compiler optimizations](#)
- **C Language:** see [Customizing Cosmic C compiler language settings](#)
- **Listings:** see [Customizing Cosmic C compiler listings](#)
- **Preprocessor:** see [Customizing Cosmic C compiler preprocessor definitions](#)
- **Input:** see [Customizing Cosmic C compiler input options](#)
- **Output:** see [Customizing Cosmic C compiler output options](#)

General settings for Cosmic toolset

With the category set to **General**, you can access the following standard settings and options:

- [Compiler memory models](#)
- [Debug information](#)
- [Optimizations](#)
- [Compiler messages display](#)
- [C language](#)
- [Listings](#)
- [Preprocessor definitions](#)
- [User-defined options](#)

Compiler memory models

You can choose to compile your application according to one of the predefined memory models that can help you optimize your code's size and performance. These models tell the compiler how the stack is used and where the stack, global variables and pointers are located in the memory.

[Table 31](#) and [Table 32](#) provide a summary of the memory model options for ST7 and STM8 microcontrollers.

Table 31. Cosmic C compiler memory models for ST7

Memory model ⁽¹⁾	Stack type and location	Location of global variables ⁽²⁾	Location of pointers ⁽²⁾
Small memory (+modms)	Simulated in Short range memory	Long range memory	Long range memory
Medium memory (+modmm)	Simulated in Long range memory	Short range memory	Long range memory
Long memory (+modml)	Simulated in Long range memory	Long range memory	Long range memory
Short memory (+modm) (<i>Default</i>)	Simulated in Short range memory	Short range memory	Long range memory
Compact memory (+modc) (<i>ST7 only</i>)	Simulated in Short range memory	Short range memory	Short range memory
Long stack (+modsl)	Physical stack in Long range memory	Long range memory	Long range memory
Short stack (+mods)	Physical stack in Long range memory	Short range memory	Long range memory

1. For complete descriptions of memory models, refer to the **C Cross Compiler User's Guide**

2. Short range means the variable is stored on one byte. Long range means the variable is stored on two bytes.

Table 32. Cosmic C compiler memory models for STM8

Memory model ⁽¹⁾	Stack type and location	Location of global variables ⁽²⁾
Short stack (+modsl0)	Physical stack in Long range memory for code size under 64Kbytes	Short range memory
Long stack (+modsl0)	Physical stack in Long range memory for code size under 64Kbytes	Long range memory
Short stack (+modsl)	Physical stack in Long range memory	Short range memory
Long stack (+modsl)	Physical stack in Long range memory	Long range memory

1. For complete descriptions of memory models, refer to the **C Cross Compiler User's Guide**

2. Short range means the variable is stored on one byte. Long range means the variable is stored on two bytes.

Debug information

The **Debug Info** settings allow you to choose the debug information to generate for your application based on use (for programming or debugging). Including the debug information will make the application file larger, however debug information is not included in the code that is loaded in the microcontroller's memory. This information is used by STVD's debugger, only.

When the category is set to **General**, the standard debug information options are:

- **None:** The resulting file is free of the debug information required for the debugger. This option should only be used once your application is completely debugged and ready for programming.
- **Debugger Compatible (default):** Inserts the +debug option in the command line, which generates the debug information required by STVD for all supported debugging instruments. Also inserts the -pxp option in the command line to disable the use of absolute paths in the debug information. In this case, you provide the paths in the **Debug** tab of the **Project Settings** window (see [Section 4.5.2: Debug settings tab on page 86](#)).
- **On Data Objects Only:** Inserts the -pxd option in the command line, causing the compiler to add the debug information to the object file for data only.
- **On Any Label:** Inserts the -pxx command in the command line, causing the compiler to add debug information for any label defining code or data to the object file.
- **Customize:** Allows you to choose the options you want for generation of debug information (Options are summarized in [Customizing Cosmic C compiler debug information settings on page 99](#)).

Optimizations

The **Optimization** settings allow you to optimize your code once you have finished debugging your application, in order to make the final version more compact and faster. You should not optimize your code when debugging because some optimizations will eliminate or ignore the debug information required by STVD and your debug instrument.

When the category is set to **General**, the standard optimization options for ST7 are:

- **Default:** The following optimizations are employed by default:
 - Uses registers **a** and **x** to hold the first argument of a function call if the function call does not return a structure and is a “char,” “short,” “int,” “pointer to” or an “array of” type function call.
 - Performs operations in 8-bit precision if the operands are 8-bit.
 - Eliminates unreachable code.
 - Chooses the smallest possible jump/branch instructions and eliminates jumps to jumps and jumps over jumps.
 - Performs multiplication by powers of two as faster shift instructions
- **Disable (For Debugging):** Introduces `-no` option in the command line, which disables all optimizations.
- **Customize:** Allows you to choose the options you want for generation of debug information. These options are summarized in [Customizing Cosmic C compiler optimizations on page 101](#).

The standard optimization options for STM8 are:

- Maximize execution speed
- **Disable (For Debugging):** Introduces `-no` option in the command line, which disables all optimizations.
- **Minimize code size:** Introduces `+compact` on the command line.
- **Customize:** Allows you to choose the options you want for generation of debug information. These options are summarized in [Customizing Cosmic C compiler optimizations on page 101](#).

Compiler messages display

The **Compiler Messages Display** field offers the following options:

- **Display compilation errors only**
- **Display errors and warnings (STM8 only)**

C language

The **C language** settings allow you to define the programming language styles to apply while compiling the application.

When the category is set to **General**, the standard C language options are:

- **Default:** Compiler default language options.
- **Enforce Functions Prototyping:** Inserts the `-pp` option in the command line, which enforces prototype declaration for functions. An error message is issued if a function is used and no prototype declaration is found for it.
- **Enforce Strict ANSI Checking:** Adds the `-psa` option in the command line, which enforces strict ANSI checking by rejecting any syntax or semantic extension. This option overrides the enum size optimization (`-pne`, see [Customizing Cosmic C compiler optimizations on page 101](#)).
- **Customize:** Allows you to choose the options you want for generation of debug information. These options are summarized in [Customizing Cosmic C compiler language settings on page 102](#).

Listings

The **Listing** settings allow you to generate a listing file and log errors to a file.

Listings are also used in generating jump tables when you optimize your code (see [Customizing Cosmic C compiler optimizations on page 101](#)).

General, the standard Listing options are:

- **None**: Removes the `-l` option from the command line so no listing is generated.
- **C and Assembly**: Adds the `-l` option to the command line to generate a listing with the filename `application.lis`.
- **Customize**: Allows you to choose the options that you want for generation of debug information. These options are summarized in [Customizing Cosmic C compiler listings on page 104](#).

Preprocessor definitions

This field allows you to enter user specified preprocessor definitions. By changing the category to **Preprocessor Definitions**, you also have access to a field for defining additional include libraries. See [Customizing Cosmic C compiler preprocessor definitions on page 105](#).

User-defined options

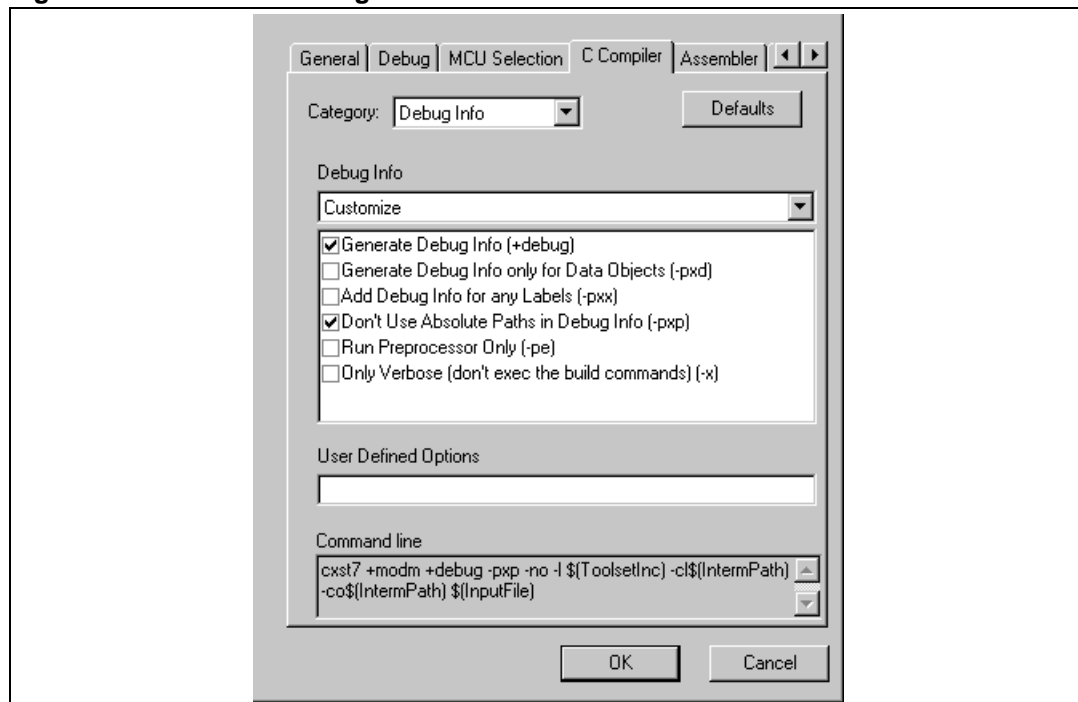
This field allows you to enter the command for an option that you have defined for the Cosmic C compiler. The options that you type in these fields are immediately added to the command line below. For more information on creating user defined options, refer to the **C Cross Compiler User's Guide**.

Customizing Cosmic C compiler debug information settings

To further customize the debug information, select **Customize** in the **Debug Information** list box. Alternatively, from the **Category** list box, select **Debug Info**.

The view in the **C Compiler** tab changes to that shown in [Figure 65](#).

Figure 65. Cosmic C debug information view



In this tab view, you can choose from the optimization options described in [Table 33](#).

Table 33. Cosmic C compiler debug information options

Option ⁽¹⁾	Description
Generate Debug Info (+debug)	Produce complete debugging information that will allow you to take full advantage of STVD's debugging features.
Generate Debug Info only for Data Objects (-pxd)	Add debug information to the object file for data only
Add Debug Info for any Labels (-pxx)	Add debug information for any label defining code or data to the object file.
Don't Use Absolute Paths in Debug Info (-pxp)	Do not include absolute path names from the prefix filenames in the debug information. You must provide the actual location of the files for the debugger in the Debug settings tab .
Run Preprocessor Only (-pe)	Expand the C source code and parse the resulting text only.
Only Verbose (don't exec the build commands) (-x)	Do not execute the passes, but write the commands that would have been performed to STDOUT.

1. For complete descriptions of compiler debug options, refer to the **C Cross Compiler User's Guide**.

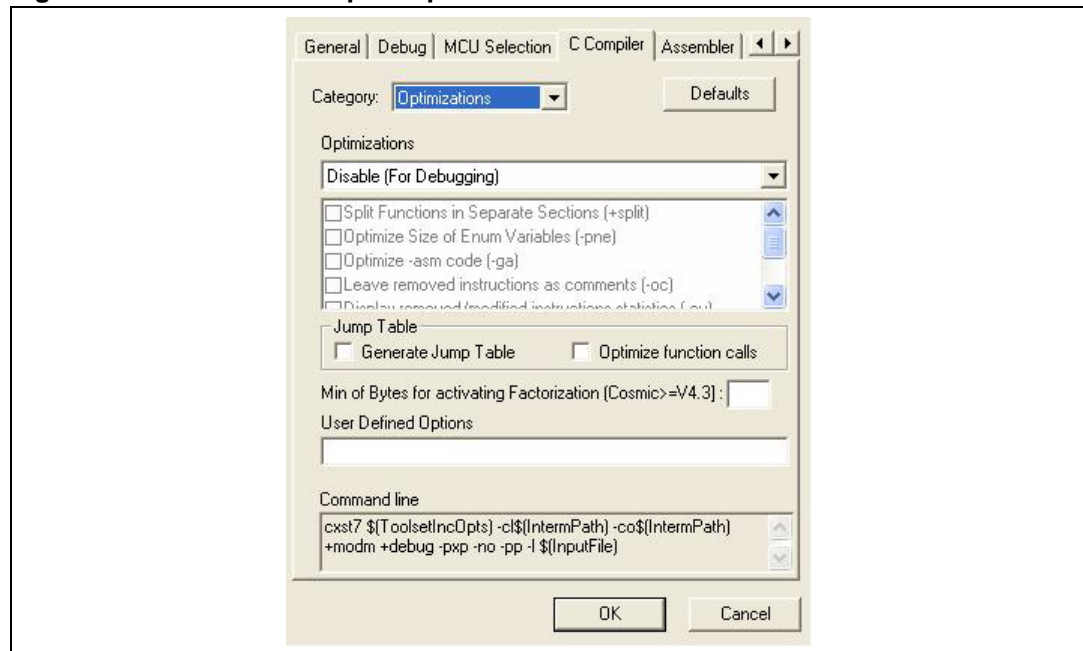
As you check off each option, you can see the resulting command line changes in the **Command line** box at the bottom of the tab.

You can also add a user-defined option to the command line by typing it in the provided field (see [Figure 65](#)). When you type an option in this field it is appears immediately in the command line, below. For more information about creating your own options for the Cosmic C compiler, you can refer to the **C Cross Compiler User's Guide**.

Customizing Cosmic C compiler optimizations

To further customize optimizations, select **Customize** from the list box. Alternatively, from the **Category** list box, select **Optimizations**. The view in the **C Compiler** tab changes to that shown in [Figure 66](#).

Figure 66. Cosmic C compiler optimizations view



In this tab view, you can choose from the optimization options described in [Table 34](#).

Table 34. Cosmic C compiler optimization options

Option ⁽¹⁾	Description
Minimize Code Size (+compact)	Adds the +compact option to the command line, which enables the optimizer factorization feature with a default depth of 7 instructions. This option is not available with the ST7 C compiler versions under 4.4.
Split Functions in Separate Sections (+split)	Produce each C function in a separate section, thus allowing the linker to suppress unused functions if the -k option has been specified on at least one segment in the linker command file.
Verbose Mode When Building Jump Table (-v) (ST7 only)	Display the name of each file that has been processed.
Optimize Size of Enum Variables (-pne)	Do not optimize size of enum variables. By default, the compiler selects the smallest integer type by checking the range of the declared enum members.
Optimize -asm code (-ga)	Optimize assembler code by invoking the absolute assembler.
Leave removed instructions as comments (-oc)	Leaves all removed instructions as comments in the application code.

Table 34. Cosmic C compiler optimization options (continued)

Option ⁽¹⁾	Description
Display removed/modified instruction statistics (-ov)	Write a log of modifications to STDERR. This displays the number of removed instructions followed by the number of modified instructions.
No Constant Propagation (-pcp)	Introduces the -pcp option on the command line, which disables the constant propagation optimization. By default, when a variable is assigned with a constant, any subsequent access to that variable is replaced by the constant itself until the variable is modified or a flow break is encountered. This option is not available with the ST7 C compiler versions under 4.4.

1. For complete descriptions of compiler optimizations, refer to the **C Cross Compiler User's Guide**.

Minimum number of bytes for activation of factorization

This field allows you to enter an integer for the number of bytes required to trigger the factorization feature. Entering an integer adds the -of option to the command line with the integer specifying the number of bytes that triggers factorization. Any value less than 4 disables the factorization. The default value for triggering factorization is 7.

This option is not available with the ST7 C compiler versions under 4.4.

Generate jump table checkbox (for ST7 only)

When checked, this options creates a jump table using the -l option to generate a listing file, then linking the full application to generate the jmptab.s file. This is a necessary step prior to using the Optimize Function Calls (+jmp) option. Be careful not to disable -l via the interface for the listing settings (see [Customizing Cosmic C compiler listings on page 104](#)).

Optimize function calls (for ST7only)

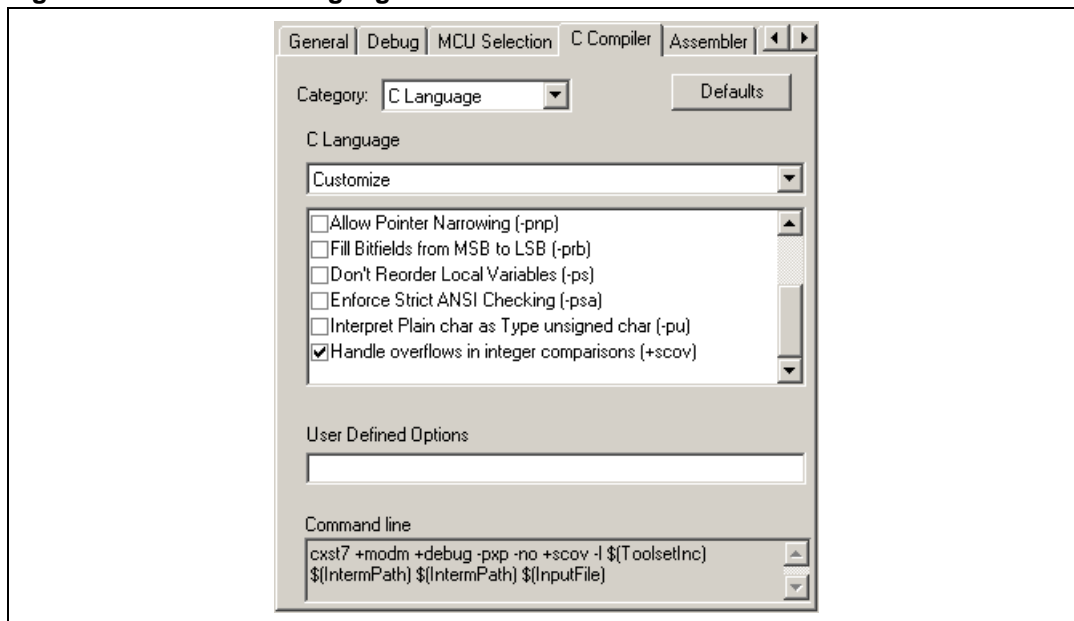
After compiling your application to generate a jump table, this option enables the optimization of function calls to reduce the volume of code. However, it also creates a time overhead at each function call.

When checked, this introduces the +jmp option on the command line. You cannot check this option if you have not already compiled the application to generate a jump table.

Customizing Cosmic C compiler language settings

To further customize optimizations, select **Customize** from the list box. Alternatively, from the **Category** list box, select **C language**. The view in the **C Compiler** tab changes to that shown in [Figure 67](#).

Figure 67. Cosmic C language view



In this tab view, you can choose from the C language options described in [Table 35](#).

Table 35. Cosmic C compiler language options

Option ⁽¹⁾	Description
Don't make public symbols with absolute addresses (-gna)	Do not produce an <code>xdef</code> directive for the <code>equate</code> names created for each C object declared with an absolute address.
Enforce Type Checking (+strict)	Instructs the compiler to enforce stronger type checking.
Enforce Prototyping for Functions (-pp)	Enforces prototype declaration for functions. An error message is issued if a function is used and no prototype declaration is found for it.
Don't replace a const object by its value (-pnc)	Instructs the compiler to store const objects as variables.
Allow Pointer Narrowing (-pnp)	Allow pointer narrowing. By default, the compiler refuses to cast the pointer into any smaller object.
Fill bitfields from MSB to LSB (-prb)	Reverse the bitfield fill order. By default, bitfields are filled from least significant bit (LSB) to most significant bit (MSB). If this option is specified, bitfields are filled from the MSB to the LSB.
Don't reorder local variables (-ps)	Do not reorder local variables. By default, the compiler sorts the local variables of a function and stores the most used variables as close as possible to the frame pointer in order to shorten addressing modes for those variables.
Enforce Strict ANSI Checking (-psa)	Enforce a strict ANSI checking by rejecting any syntax or semantic extension. This disables the enum size optimization (-pne).

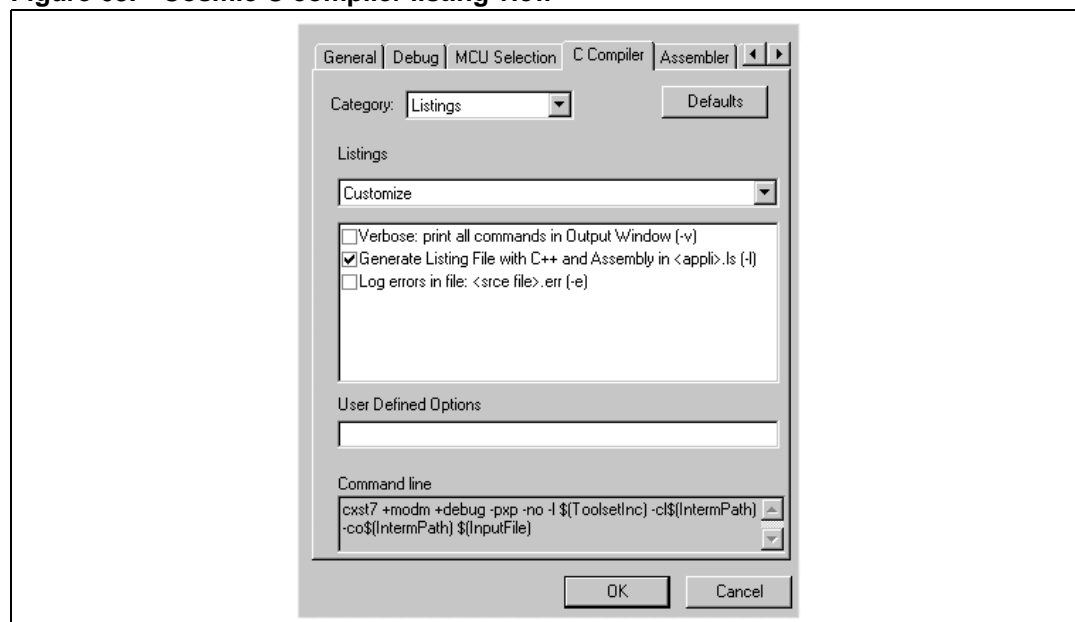
Table 35. Cosmic C compiler language options (continued)

Option ⁽¹⁾	Description
Interpret Plain char as Type unsigned char (-pu)	Take a plain char to be of type unsigned char, not signed char. This also affects strings constants.
Authorize Constant and Volatile Repetition (-pc99)	Adds the -pc99 option to the command line, which enables the repetition of the const and volatile modifiers in the declaration either directly or indirectly in the typedef. This option is not available with the ST7 C compiler versions under 4.4.

1. For complete descriptions of compiler C language options, refer to the **C Cross Compiler User's Guide**.

Customizing Cosmic C compiler listings

To customize compiler listings, select **Customize** in the **Listings** list box. Alternatively, from the **Category** list box, select **Listings**. The tab view changes as shown in [Figure 68](#).

Figure 68. Cosmic C compiler listing view

In this tab view, you choose from the listing options described in [Table 36](#).

Table 36. Cosmic C compiler listing options

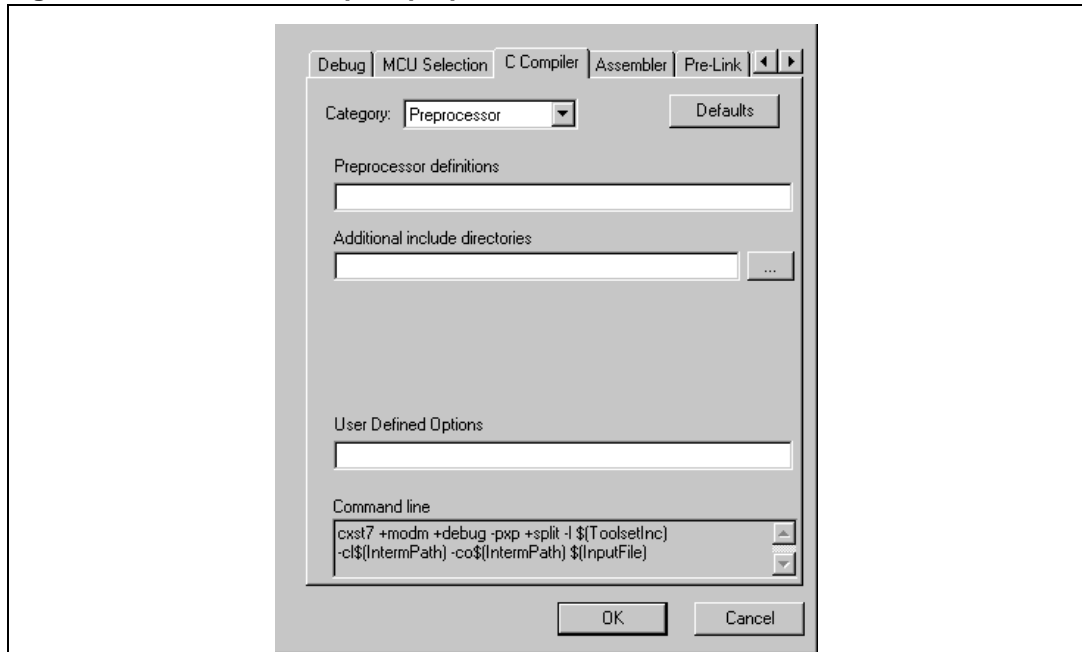
Option ⁽¹⁾	Description
Verbose: print all commands in Output Window (-v)	Each function name is sent to STDERR when <i>cgst7</i> starts processing it.
Generate Listing File with C++ and Assembly (-l)	Merge C source listing with assembly language code. The default listing output is <i>application.ls</i> .
Log errors in file (-e)	Log errors from parser in a file instead of displaying them on the terminal screen. The name of the error file is <i>source_file.err</i> . It is created only if there are errors.

1. For complete descriptions of compiler listing options, refer to the **C Cross Compiler User's Guide**.

Customizing Cosmic C compiler preprocessor definitions

When the **Category** list box is set to **Preprocessor**, the view in the tab changes to that shown in [Figure 69](#). In addition to adding preprocessor commands (**-d**), this view allows you to add include directories (**-i**).

Figure 69. Cosmic C compiler preprocessor view



When entering a **Preprocessor definition** (**-d**), you specify the name of a user-defined preprocessor symbol. The form of the definition is:

```
-dsymbol [=value]
```

If the value is omitted, it is set to 1.

When entering definitions, the **-d** is automatically added to the command line as you type the symbol. When you enter several definitions, separate them with a space and **-d** will automatically be added to the next symbol in the command line. The compiler allows you to specify up to 60 definitions.

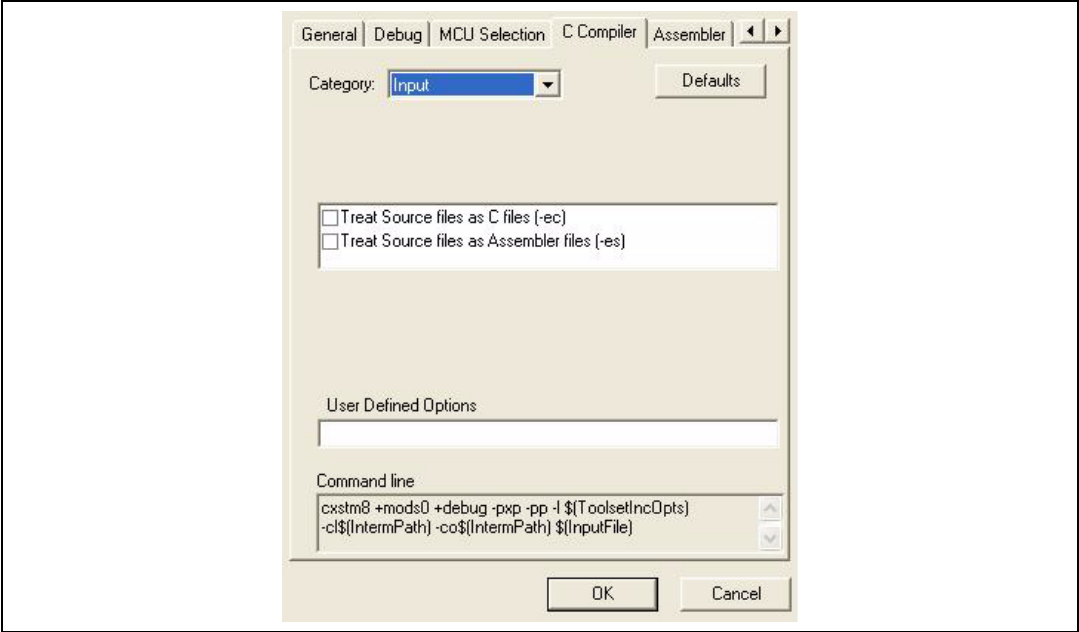
The **Additional include directories** (**-i**) allows you to specify the paths for other directories that you want to include when compiling. You can either type the path name or use the browse button to locate the directory.

When you type the path name for the directory, **-i** is automatically added to the command line. Each path name is the name of a directory and is not ended with a directory separator (****). When you enter several path names, separate them with a semicolon (**;**) and **-i** is added automatically to the next entry.

Customizing Cosmic C compiler input options

When the **Category** list box is set to **Input**, the view in the tab changes to that shown in [Figure 70](#). It allows to specify how input files are to be treated during compilation. This view is not available with the ST7 C compiler versions under 4.4.

Figure 70. Cosmic C compiler input view



The input options are:

- **Treat Source files as C files** (-ec)
- **Treat Source files as Assembler files** (-es)

Customizing Cosmic C compiler output options

When the **Category** list box is set to **Output**, the view in the tab changes to that shown in [Figure 70](#). This view allows to specify the options listed in [Table 36](#). This view is not available with the ST7 C compiler versions under 4.4.

Figure 71. Cosmic C compiler output view for ST7(left) and STM8 (right)

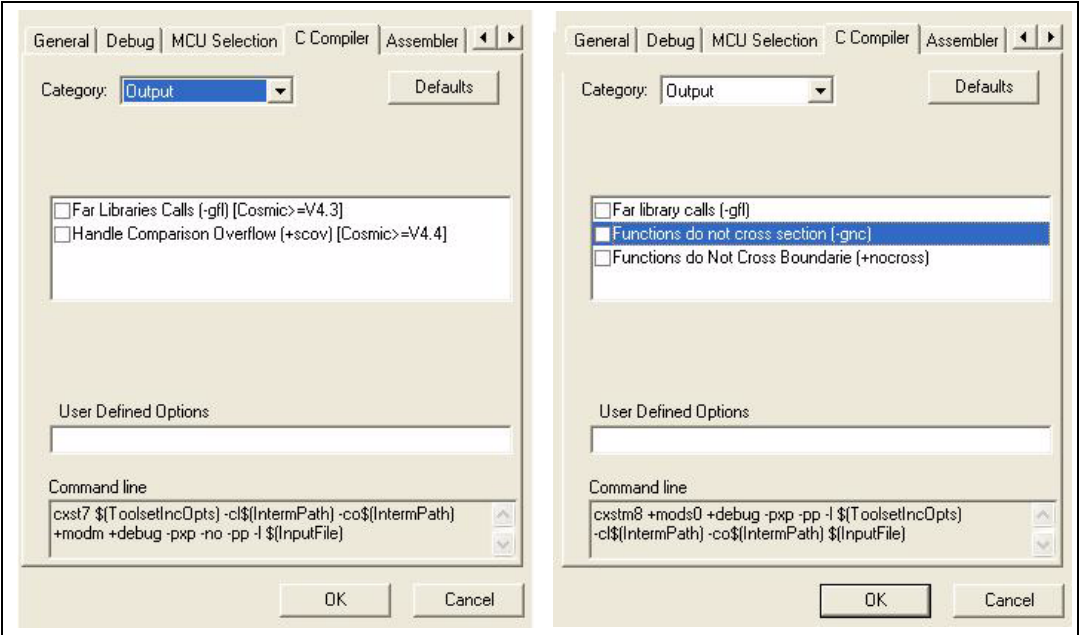


Table 37. Cosmic C compiler output options

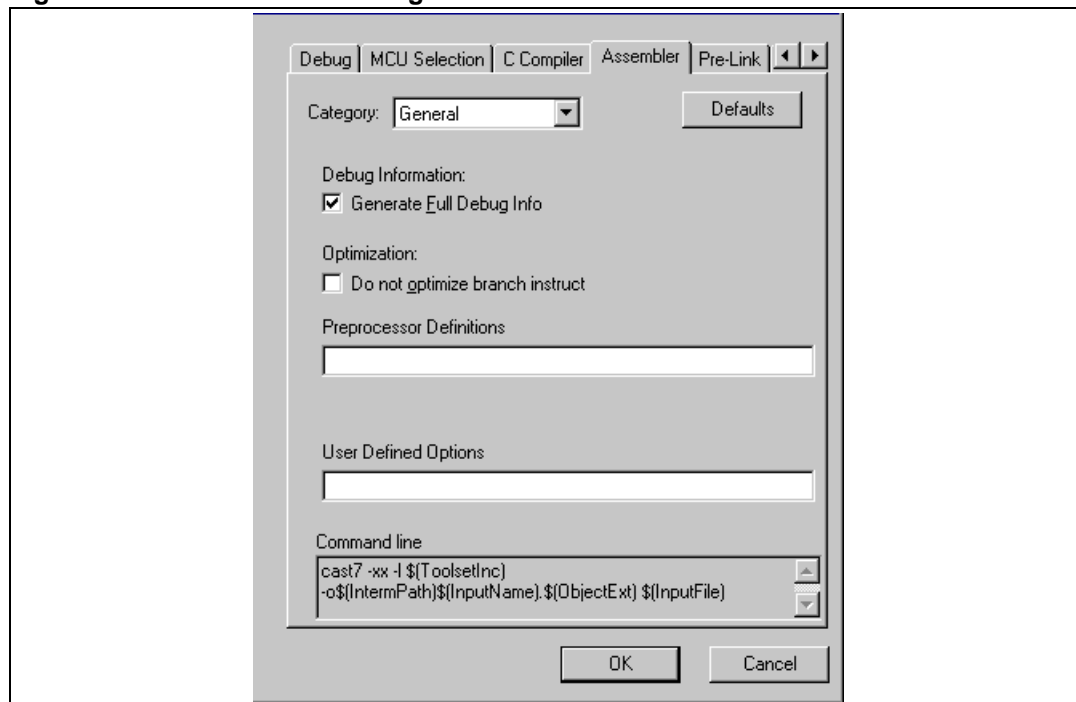
Option ⁽¹⁾	Description
Far Library Calls (-gfl) (STM8 only)	Adds -g to the command line to invoke the code generator with the fl option, which uses callf instructions for machine library calls. This is used in conjunction with memory models for large applications. This option is not available with the ST7 C compiler versions under 4.4.
Handle Comparison Overflow (+scov) (ST7 only)	Adds +scov option, which allows production of code for signed integer comparisons handling overflow situations. This increases overall code size. This option is not available with the ST7 C compiler versions under 4.4.
Functions do not cross section (-gnc) (STM8 only)	Specifies that functions must remain within the same 64Kbyte section.

1. For complete descriptions of compiler optimizations, refer to the **C Cross Compiler User's Guide**.

4.7.2 Cosmic C Assembler tab

The **Assembler** tab (see [Figure 72](#)) provides an interface for setting the options in the command line for the Cosmic C assembler (cast7). The gray **Command line** field displays all the changes to the command line as you make them.

Figure 72. Cosmic Assembler general view



The **Category** list box allows you to access a general view for easy access to standard settings and options. You can also change the category to access views where you can customize settings in greater detail. In this list box you can choose from the following views:

- **General:** We recommend starting in the [General settings for Cosmic Assembler](#) view.
- **Debug:** [Customizing Cosmic Assembler debug information](#)
- **Language:** [Customizing Cosmic Assembler language settings](#)
- **Listing:** [Customizing Cosmic Assembler listing settings](#)

General settings for Cosmic Assembler

When the **Category** list box is set to **General**, you can enable and disable the following standard options:

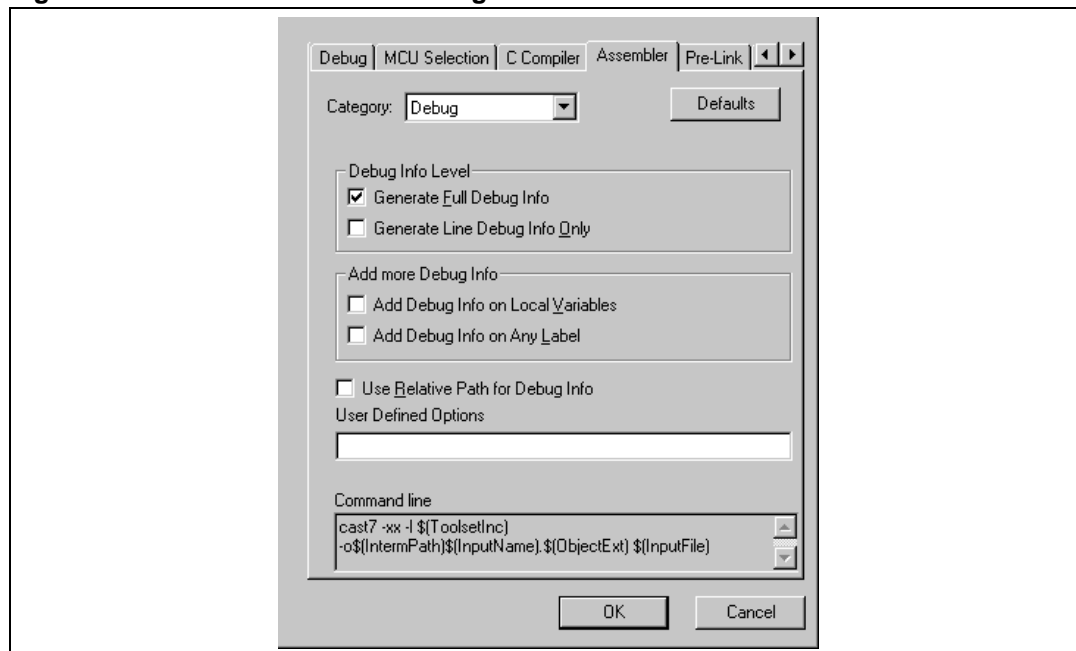
- **Generate Full Debug Info:** Checking this box introduces the `-xx` option in the command line, which enables the generation of debug information. For more debug options, refer to [Customizing Cosmic Assembler debug information on page 108](#).
- **Do not optimize branch instruct:** Checking this box introduces the `-b` option in the command line, which disables the optimization of branch instructions.

This view also provides a [Customizing Cosmic C compiler preprocessor definitions](#) field and a [User-defined options](#) for entering commands that you define.

Customizing Cosmic Assembler debug information

To further customize the debug information options, select **Debug** from the **Category** list box. The view in the **C Compiler** tab changes to that shown in [Figure 73](#).

Figure 73. Cosmic Assembler debug view



From this view you can choose from the options listed in [Table 38](#).

Table 38. Cosmic Assembler debug options

Option ⁽¹⁾	Description
Generate full (-xx)	Generate debug information for any label defining code or data in the object files, that will allow you to take full advantage of STVD's debugging features.
Generate line (-x)	Add only line debug information to the object file. This information provides the debugger the link between a line of source code and the beginning and end address of an instruction in the compiled application.
Add on local variable (-p1)	Add local symbols to the debug symbol table. These are only displayed in the linker map file.
Add on any label (-c)	Lines that are changed or removed are kept in the assembly source as comments.
Use relative path for debug info (-xp)	Filenames in the debug information are not prefixed with an absolute path name. The debugger must be informed about the actual file location (see Section 4.5.2: Debug settings tab on page 86)

1. For complete descriptions of assembler debug options, refer to the **C Cross Compiler User's Guide**.

Customizing Cosmic Assembler language settings

The assembler language settings allow you to set options related to language checking during assembly. To access these options, select **Language** from the **Category** list box. The resulting view provides check boxes to enable or disable the options listed in [Table 39](#).

Table 39. Cosmic Assembler language options

Option ⁽¹⁾	Description
Map all sections as Absolute (-a)	Map all sections, including predefined sections to absolute.
Mark all symbols as Public (-p)	Mark all symbols as public. This option has the same effect as adding an <code>xdef</code> directive for all symbols.
Mark all "equ" defined Symbols as Public (-pe)	Mark all symbols defined with an <code>equ</code> directive as public. This option has the same effect as adding an <code>xdef</code> directive for those symbols.

1. For complete descriptions of assembler language settings, refer to the **C Cross Compiler User's Guide**.

Customizing Cosmic Assembler listing settings

The assembler **Listing** settings allow you to generate listing and error files. The listing contains the source code that was input to the assembler, along with the hexadecimal representation of the corresponding object code and the address for which it was generated.

To access these options select **Listing** from the **Category** list box. The resulting view provides check boxes to enable or disable the options listed in [Table 40](#).

Table 40. Cosmic Assembler listing options

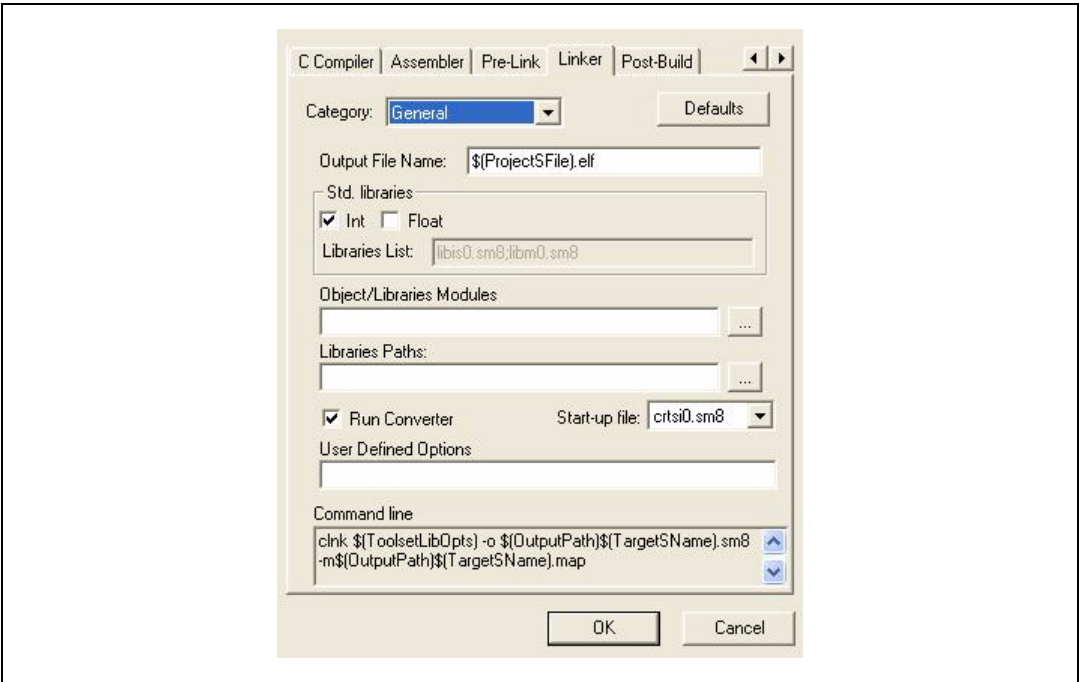
Option ⁽¹⁾	Description
Basic (-l)	Generate a listing file. The name of the file is <i>name_of_input_file</i> . <i>ls</i> , unless you use the +l option to specify the name of the file.
With errors on undefined symbols (-u)	Produces an error message in the listing file for all occurrences of an undefined symbol.
Generate error log file (+e*)	Generates an assembler error log instead of displaying the error messages on screen.

1. For complete descriptions of assembler listing options, refer to the **C Cross Compiler User's Guide**.

4.7.3 Cosmic C linker tab

The **Linker** tab provides an interface for setting the options in the command line for the Cosmic C linker (`clnk`). The gray **Command line** field displays all the changes to the command line as you make them.

Figure 74. Cosmic linker general view



The **Category** list box allows you to access a general view for easy access to standard settings and options. You can also change the category to access views where you can customize settings in greater detail.

In the **Category** list box you can choose from one of the following views:

- **General:** We recommend starting in the [General settings for Cosmic linker](#) view.
- **Input:** [Customizing linker input settings](#)
- **Output:** [Customizing linker output settings](#)

General settings for Cosmic linker

With the category set to **General**, you can access the following standard options:

Output filename option

Allows you to specify the name for the output file. By default, this field contains a macro that will specify the project source in `.elf` format.

Std. libraries

Select whether to use the **Int** or **Float** libraries by checking the appropriate box. If **Int** is checked, the builder inserts the standard integer library in the `.lkf` template.

Note: *There are several standard integer libraries depending on the memory model you have chosen.*

Object/Library modules

In this field, you can type the names for specific library or object modules to include in the build.

Library path names

This field allows you to enter additional libraries to be included by the linker.

You can type the library path name or use the browse button to enter the libraries to include. When entering more than one library, entries are separated by a semicolon (;). When you add a library via this field, the `-l` option followed by the library path name are added to the command line.

Run the converter

The converter (`chex` utility) allows you to translate executable files produced by the linker into either the Motorola S-record or Intel standard hexadecimal format, and output to a specific location.

When you click to place a checkmark in the **Run Converter** checkbox, a command line with the **Write output to file** option (`-o`) is added to the **Commands** field in the **Post-build** tab. A pop-up window opens whenever you activate this option, to tell you that the command line is being added or removed to the **Post-build** tab.

You can customize the command line with options that allow you to override text and data biases or output only a portion of the executable. For more about the supported options refer to **C Cross Compiler User's Guide**.

Startup file

The startup file establishes the runtime environment for C. Typically it provides:

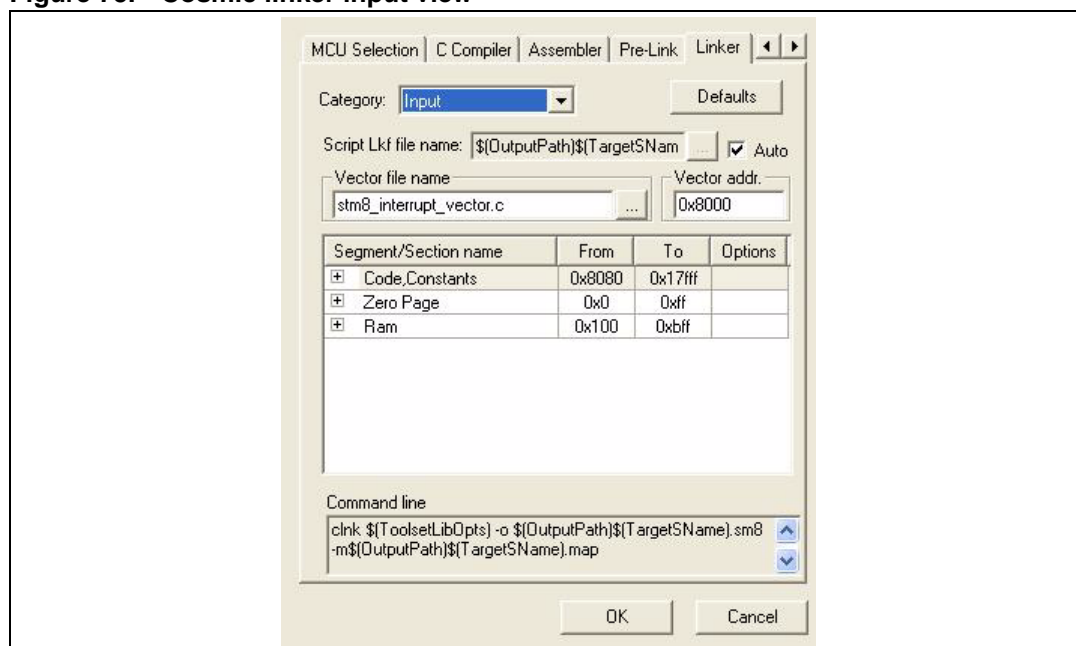
- Initialization of the `.bss` section
- ROM to RAM copy
- Initialization of the stack pointer
- Call-up of the program entry point
- An exit sequence

The **Startup File** list box allows you to choose to use one of the provided startup files. The list of startup files is different for the ST7 and STM8 toolsets. For a complete description of startup files, refer to the **C Cross Compiler User's Guide**.

Customizing linker input settings

The input settings interface for the linker allows you to define input sources for the linker such as libraries and vector files, as well as allowing you to define segments in your microcontroller's memory and assign code sections to them. Access this view by selecting **Input** in the **Category** list box. The view will change to that shown in [Figure 75](#).

Figure 75. Cosmic linker input view



Auto segment declaration and section assignment

There are three ways to manage the Cosmic script link file:

- Automatic
- Semi-automatic
- Custom

When the **Auto** option is checked the script link file is automatically generated to the output path. The filename by default is *TargetSName.lkf*.

When the **Custom** option is selected, you can enter the path and filename that you want for the script link. In this mode the script link file will not be updated if you make changes to your project, such as adding or removing a source file. If you do make such a change you will have to edit the script link manually.

When the **Semi-auto** option is selected, STVD automatically updates the linker file sections that are delimited by specific markers. All fields out of these sections can be manually edited. The markers reserved for STVD are:

```
<BEGIN SEGMENT_CONF>, <END SEGMENT_CONF>
<BEGIN STARTUP_FILE>, <END STARTUP_FILE>.
```



```
<BEGIN OBJECT_FILES>, <END OBJECT_FILES>  
<BEGIN LIBRARY_FILES>, <END LIBRARY_FILES>  
<BEGIN VECTOR_FILE>, <END VECTOR_FILE>  
<BEGIN DEFINED_VARIABLES>, <END DEFINED_VARIABLES>
```

If you want to stop STVD automatic update for one section, simply remove both `BEGIN/END` markers of this section. You may restore the mechanism later by restoring the markers.

In **Automatic mode**, a configurable mapping is displayed in the **Linker** tab. The mapping's first level elements are segments in the target microcontroller's memory. Clicking on the "+" sign for a segment allows you to see the code sections that are assigned to the segment. This mapping is used to automatically generate a linker script file (default name *TargetSName.lkf*) that contains the declarations of memory segments and the code sections assigned for each.

The linker script file is regenerated automatically when you close the **Project Settings** window, or execute a build command. In Automatic mode, any modification that you made externally into the linker file are lost. In Semi-automatic mode, any modification that you made inside STVD-reserved sections are lost.

If you want to customize the linker file but keep some automatic mechanisms (for instance, define a customized segment while keeping the automatic update of object files list when adding or removing files into the project), use the Semi-automatic mode.

Modify the segment mapping and section assignments

You can move code sections from any memory segment except the Zero Page Vars to any other code segment by simply dragging and dropping them.

You can add memory segments and delete or rename the segments that you have created. Default segments cannot be deleted or renamed. However, you can modify their start and end addresses.

Right-clicking on a cell in the mapping opens a contextual menu with the following commands:

- **Add Segment** - Adds a row of empty fields to the mapping. Enter the name of the new segment and press the Enter key. If you do not enter a segment name, but click on enter, the new segment is removed from the mapping. You cannot use the same segment name twice. Naming is case sensitive.
- **Add Section** - Adds a new code section to the mapping. If you do not enter a section name, but click on enter, the new section is removed from the mapping. You cannot use the same section name twice. Naming is case sensitive. The interface only allows you to enter the name of the section (not an address range for it). Upon link, the code sections are assigned to the memory segment in the order in which they appear in the mapping.
- **Change** - Allows you to change the value or entry in the selected cell of the mapping. Address values can be entered in decimal or hexadecimal (use 0x_____ notation) formats. You cannot use the same section or segment name twice in the mapping. You can change the names of the default memory segments, however you cannot change the names of the default code sections.
- **Delete** - Deletes the selected entry. You cannot delete the default memory segments or code sections.

- **Protect** - Protects the selected code section from optimization. Adds the `-k` option to the selected item in the link script file. Code sections that you have applied protection to are displayed in red.

Vector file name and Vector address

By default, the filename is set to `stm8_interrupt_vector.c` for STM8. You can change the name of the interrupt vector file and also the interrupt vector table start address.

The interrupt vector file is regenerated with each build, which means that you must not modify it yourself. Instead, you should disable the automatic management of the vector table by leaving the **vector file name** edit box empty.

The script link file contains a section dedicated for the interrupt vector table. This section is automatically generated in Automatic mode, and in Semi-automatic mode if `<BEGIN VECTOR_FILE>` and `<END VECTOR_FILE>` markers are present. By default, the section will consist of the following lines:

```
#Setting the beginning of the .const section at 0x8000:
+seg .const -b 0x8000
#The vector table is declared in stm8_interrupt_vector.c as a
#constant, so it will start at 0x8000
#address:stm8_interrupt_vector.o
```

STM8 linker `.const` and `.text` section limitations

- The `.const` section must be located in the first 64k segment (mandatory).
- Allowing large function to cross memory section boundaries means that you cannot use the `-gnc` option (because `-gnc` forces the compiler to use short jumps).

Customizing linker output settings

The linker **Output** settings allow you to apply options that define the types and formats or output. Access this view by selecting **Input** in the **Category** list box. The view will change to that shown in [Figure 76](#).

Figure 76. Cosmic linker output view

MCU Selection | C Compiler | Assembler | Pre-Link | **Linker** | < | >

Category: **Output** Defaults

Errors

☐ Generate Error File

Error File Name:

Map File

☒ Generate Map File

Map File Name:

☒ Logical Address Symbols ☒ Symbols sorted by Alphabetical

☐ Physical Address Symbols ☐ Symbols sorted by address

User Defined Options

Command line

OK Cancel

In this view, the options listed in [Table 41](#) are available.

Table 41. Cosmic linker output options

Option ⁽¹⁾	Description
Generate Error File	You can choose to generate an error file by checking this option, and entering the name for the error file in the field provided. By default, this option is not enabled.
Generate Map File	You can generate a map file by checking this option and entering a map file name in the field provided. By default, this option is checked, and a map file is generated to the output path with the extension <code>.map</code> . In this map file, by clicking on the appropriate radio button, you can have: <ul style="list-style-type: none"> – logical address symbols – physical address symbols – symbols sorted alphabetically (<i>STM8 only</i>) – symbols sorted by address (<i>STM8 only</i>)
User Defined Options	In this field you may enter any additional options you want to apply when generating the linker output.

1. For complete descriptions of linker output options, refer to the **C Cross Compiler User's Guide**.

4.8 Customizing build settings for Raisonance C toolset

The **Project Settings** window for building your application with the Raisonance C toolset contains five tabs that are common to all the toolsets (General, MCU selection, Debug, Pre-link and Post-Build) and three tabs that contain options and settings that are specific to Raisonance.

This section provides details about configuring options that are specific to the Raisonance toolset. These options include:

- [Raisonance C compiler tab](#)
- [Raisonance Assembler tab](#)
- [Raisonance C linker tab](#)

Configuring projects settings in the five common tabs is described in [Section 4.5: Configuring project settings on page 84](#).

Provided C header files for Raisonance

Depending on your MCU, you can include a header file provided with STVD that defines the device's peripheral registers. This will save you the time of creating this header file yourself.

Caution: The C header files provided with STVD for ST7 and STM8 microcontrollers are not the same as those provided for the ST7 Library and STM8 Library. ST7 and STM8 library users should use the C header and/or ASM include files specific to that library.

Depending on your installation, the C header files provided with STVD are typically located at:

```
C:\Program Files\STMicroelectronics\st_toolset\include\
```

To build your project using the provided header file for your device, specify its inclusion in your application code. The name of the file to use is *device_name.h*.

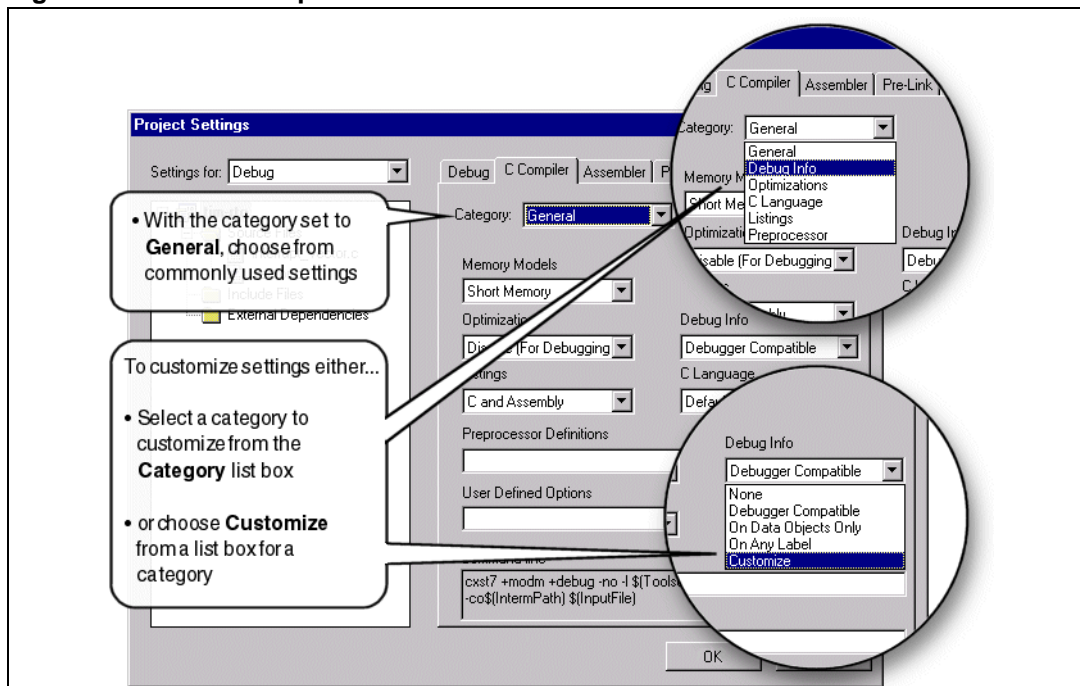
Header files are provided for virtually all the supported devices. If you don't find your device's name in the header files, this may be because the full name is not used. To determine if your device has a header file, you can check the **Peripheral Registers** window (**View > Peripheral Registers**) in the debug context. If peripheral register information is provided in this window, then a header file has been generated for your device. The correct file for your device, is the one with the device name you found in the **Peripheral Registers** window.

Using the Project Settings interface

The tabs (C Compiler, Assembler and Linker) have a common interface that allows you to choose from standard configurations, customize options and view changes to the command line as you apply different options.

When the **Category** list box is set to **General**, the tab provides check boxes and list boxes for easy access to commonly used options and configurations (see [Figure 77](#)).

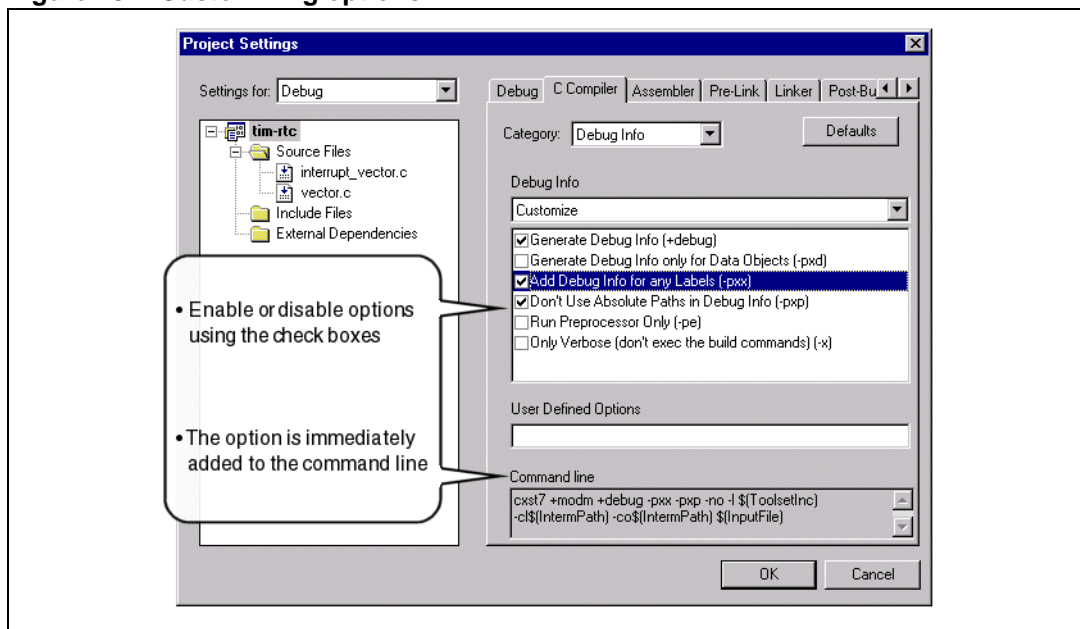
Figure 77. Standard options



To further customize options, select a category in the **Category** list box, or select **Customize** in a list box for a category of settings. The view in the tab will change to present more options (see [Figure 78](#)).

You can enable or disable options by clicking on the check boxes. When you invoke an option, the change to the command line is immediately visible in the gray **Command Line** window at the bottom of the tab.

Figure 78. Customizing options

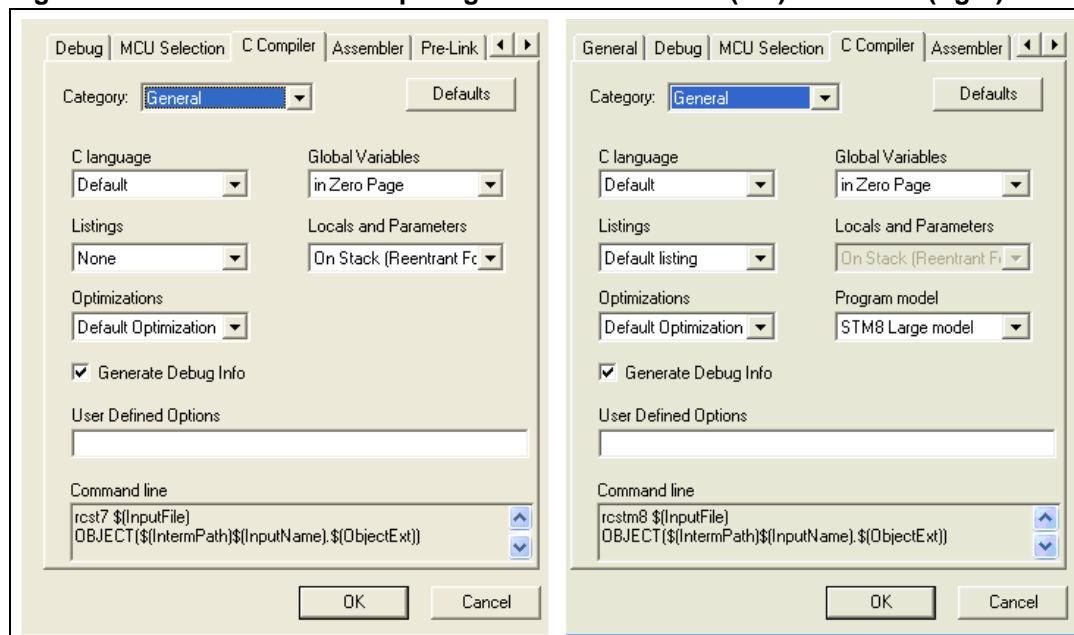


4.8.1 Raisonance C compiler tab

The **C Compiler** tab provides an interface for setting the options in the command line for the Raisonance C compiler for ST7 and STM8. The options available on the **C compiler** tab view correspond to the microcontrollers that are selected on the **MCU Selection** tab view.

The gray **Command line** field displays all the changes to the command line as you modify the settings in the graphical interface.

Figure 79. Raisonance C compiler general view for ST7 (left) and STM8 (right)



The **Category** list box allows you to access a general view that displays standard options and settings. You can also change the category to access views where you can customize settings in greater detail.

In the **Category** list box, you can choose one of the following views:

- **General:** it is recommended to start in the general settings view, see [General settings for the Raisonance toolset](#)
- **Optimizations,** see [Customizing Raisonance C compiler optimizations on page 121](#)
- **C Language,** see [Customizing Raisonance C compiler language settings on page 122](#)
- **Listings,** see [Customizing Raisonance C compiler listings on page 123](#)
- **Preprocessor definitions,** see [Customizing Raisonance C compiler preprocessor definitions on page 124](#)

General settings for the Raisonance toolset

With the category set to **General**, you can access the following standard settings and options, as shown in [Figure 79](#):

- [C language](#)
- [Listings](#)
- [Optimizations](#)
- [Global variables](#)
- [Locals and parameters \(disabled for STM8 microcontrollers\)](#)
- [Generate debug info](#)
- [User-defined options](#)
- [Program model \(STM8 only\)](#)

C language

The C language settings allow you to define the programming language styles to apply while compiling the application.

When the category is set to **General**, the standard C language options are:

- **Default:** Compiler default language options.
- **ANSI mode restricted:** Adds the `MODANSI` option in the command line, which enforces strict ANSI checking by rejecting any syntax or semantic extension.
- **Customize:** Allows you to choose the options you want for generation of debug information. These options are summarized in [Customizing Raisonance C compiler language settings on page 122](#).

Listings

The **Listing** settings allow you to generate a listing file and log errors to a file.

When the category is set to **General**, the standard Listing options are:

- **None:** Adds the `NOPR` option to the command line so no listing is generated.
- **Default listing:** automatically selects the following options:
 - “Show the assembler translation of C statements in the listings file”
 - “Add the list of symbols generated from the source file”
- **Preprocessor listing:** automatically selects the option “Generate a preprocessor listing file”.
- **Customize:** When you select this option, the category displayed changes to **Listings**. This view allows you to choose the options that you want for generation of debug information. These options are described in [Customizing Raisonance C compiler listings on page 123](#).

Optimizations

The **Optimization** settings allow you to optimize your code once you have finished debugging your application, in order to make the final version more compact and faster. You should not optimize your code when debugging because some optimizations will eliminate or ignore the debug information required by STVD and your debug instrument.

When the category is set to **General**, the standard optimization options are:

- **Default optimization:** The following optimizations are employed by default and cannot be disabled:
 - Expressions are reduced to constants whenever possible
 - Internal data and bit access optimization
 - Jump optimization
- **Speed optimization:** When you select this option, the category displayed changes to Optimizations. A list of options for speed optimization is displayed. For further information, refer to [Customizing Raisonance C compiler optimizations on page 121](#).
- **Size optimization:** When you select this option, the category displayed changes to Optimizations. A list of options for size optimization is displayed. For further information, refer to [Customizing Raisonance C compiler optimizations on page 121](#).

Global variables

When the category is set to **General**, you can use the **Global Variables** list box to specify the default location of global variables:

- **In Zero Page:** This is the default option.
- **In Data:** Specifies to place global variables in the first 64Kbytes of memory.

Locals and parameters (disabled for STM8 microcontrollers)

When the category is set to **General**, the options available for the location of local variables and parameters are:

- **In Zero Page:** This is the default option.
- **In Data:** Specifies to place local variables in the first 64Kbytes of memory.
- **On Stack (reentrant Fct):** Specifies to place local variables on stack if you want functions to be reentrant. For more information, see the definition of the AUTO mode in the *Raisonance RCST7 Compiler Manual*.

Generate debug info

Select the **Generate Debug Info** checkbox if you want the compiler to output debug information.

User-defined options

The **User Defined Options** field allows you to enter the command for an option that you have defined for the Raisonance C compiler. The options that you type in this field are immediately added to the command line below. For information on creating user-defined options, refer to the *Raisonance RCST7 Compiler Manual*.

Program model (STM8 only)

When the category is set to **General**, the **Program Model** field offers the following options:

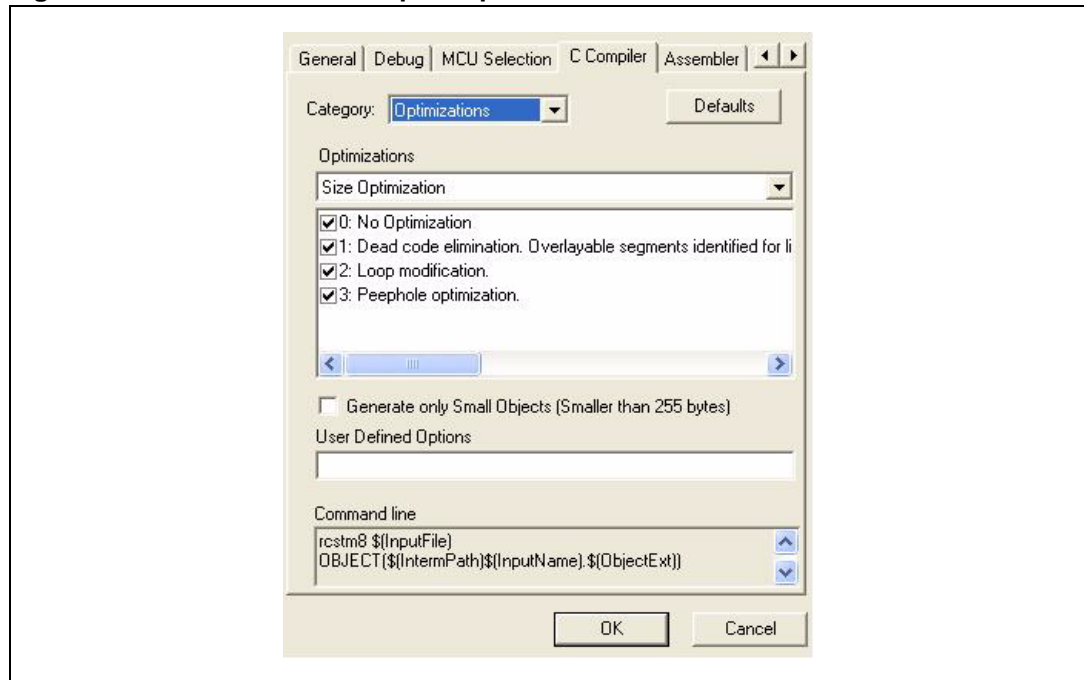
- **STM8 small model:** specifies to generate small model code. Should be used for projects where all the code fits in the zero section, that is, where all functions calls are limited to a 16Kbyte address range.
- **STM8 large model:** specifies to generate large model code. Should be used for projects with code above the zero section, with functions located within the 16MB memory address range.

Customizing Raisonance C compiler optimizations

On the C Compiler tab of the **Project Settings** window, in the **Category** field, select **Optimizations** to display the view shown in [Figure 80](#).

Alternatively, when the category is set to **General**, in the Optimizations field, select either **Speed Optimization** or **Size Optimization**.

Figure 80. Raisonance C compiler optimizations view



In the **Optimizations** field, you can choose:

- **Default Optimization:** you cannot modify the default settings. See description of the optimizations performed when this option is selected in [Optimizations on page 119](#).
- **Speed Optimization**
- **Size Optimization**

The optimization options available for speed optimization and size optimization are the same, but the interface allows you to set different levels of optimization for speed and for size. These options are described in [Table 42](#).

Note that speed optimization options and size optimization options are incremental: if you select option 4 as shown in [Figure 80](#), options 0 (default), 1, 2 and 3 are automatically selected.

Table 42. Raisonance C compiler optimization options

Level	Description ⁽¹⁾
0	No optimization
1	<ul style="list-style-type: none"> – Dead code elimination – Overlayable data is identified for linker optimization – Conditional jump optimization – Complex move operations optimization – Switch optimization – Loop optimization – Local and global sub-expression optimization
2	Loop performance optimization
3	Peephole optimization

1. For complete descriptions of compiler optimizations, refer to the *Raisonance RCST7 Compiler Manual*.

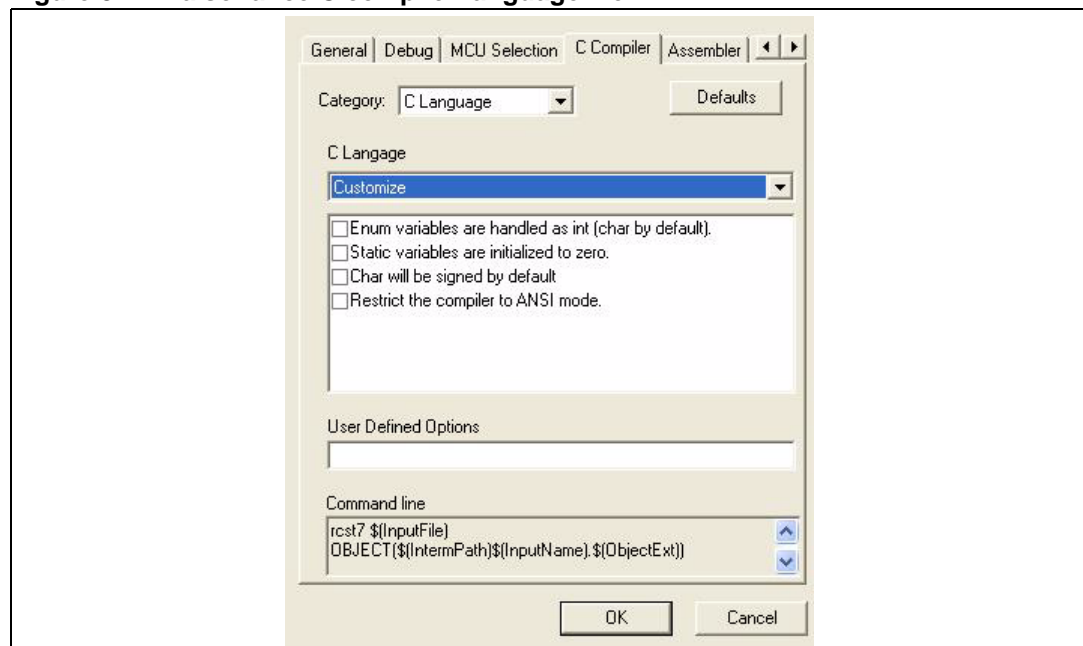
As you select each option, you can see the changes in the command line in the **Command Line** field at the bottom of the tab.

Generate only small objects

Check the **Generate only small objects** checkbox to indicate to the compiler to generate objects smaller than 255 bytes.

Customizing Raisonance C compiler language settings

On the **C Compiler** tab of the **Project Settings** window, in the **Category** field, select **C Language** to display the view shown in [Figure 81](#). Alternatively, when the category is set to **General**, in the C Language field, select **Customize**.

Figure 81. Raisonance C compiler language view

In the **C Language** field, you can choose:

- **Default:** compiler default language options
- **Ansi Mode Restricted:** automatically selects the ANSI-compliant mode
- **Customize:** the options available when you select this mode are described in [Table 43](#).

Table 43. Raisonance C compiler language options

Directive	Description ⁽¹⁾
ENUMTYPE	Defines whether enum variables are handled as char or int (char by default).
INITSTATICVAR/ NOINITSTATICVAR	Defines whether static variables are initialized to zero or left uninitialized.
SIGNEDCHAR/ UNSIGNEDCHAR	Defines whether characters are signed or unsigned (signed by default)
MODANSI	Restricts the compiler to ANSI mode. Strict ANSI checking is enforced, rejecting all syntax and semantic extensions.

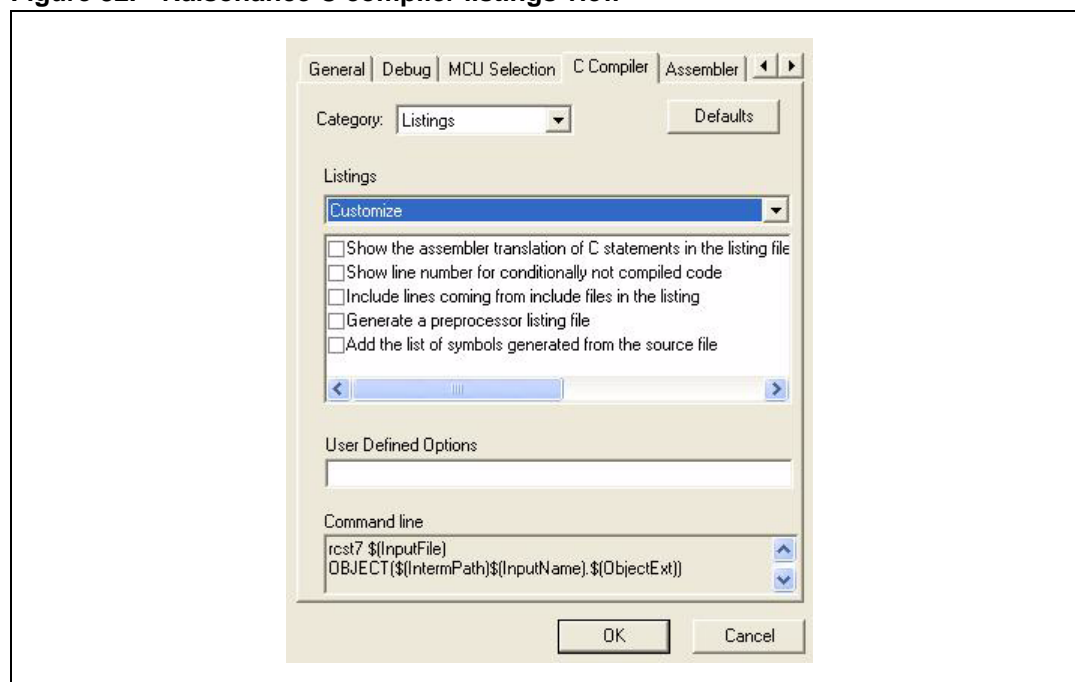
1. For complete descriptions of compiler C language options, refer to the *Raisonance RCST7 Compiler Manual*.

As you select each option, you can see the changes in the command line in the **Command Line** field at the bottom of the tab.

Customizing Raisonance C compiler listings

On the C Compiler tab of the **Project Settings** window, in the **Category** field, select **Listings** to display the view shown in [Figure 82](#). Alternatively, when the category is set to **General**, in the **Listings** field, select **Customize**.

Figure 82. Raisonance C compiler listings view



From the **Listings** field, you can choose:

- **None**: all options are grayed out.
- **Default listing**: automatically selects the default options, see [Listings on page 119](#).
- **Preprocessor listing**: automatically selects the option **Generate a preprocessor listing file**.
- **Customize**: displays a list of options that you can select individually. These options are described in [Table 44](#).

Table 44. Raisonance C compiler listing options

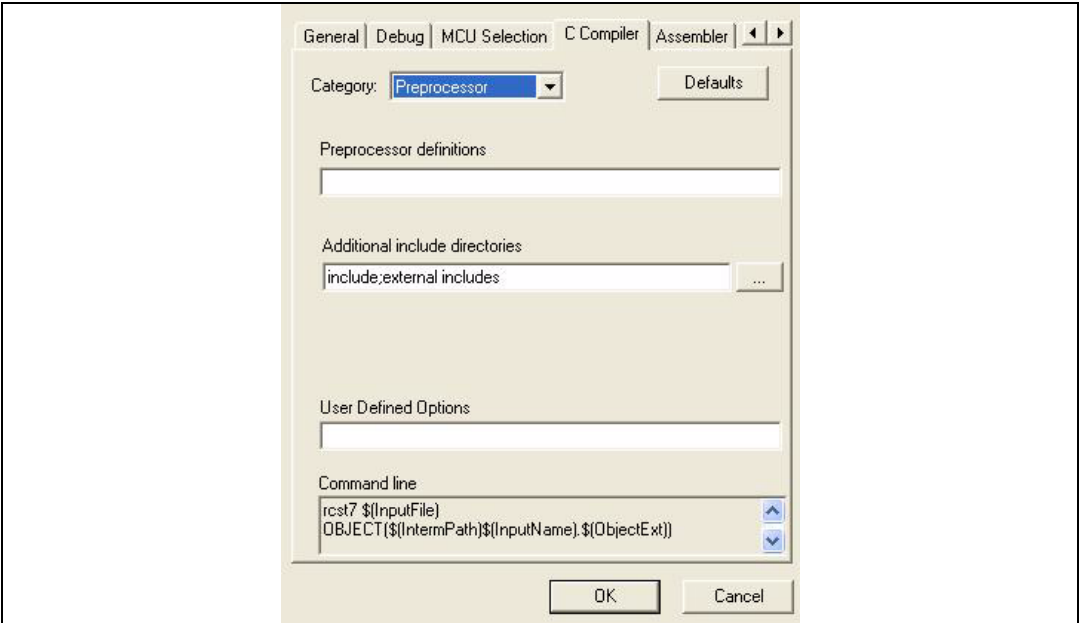
Directive	Option ⁽¹⁾
CODE	Includes the assembler translation of C statements in the listing file.
COND/NOCOND	Specifies whether to include the lines that were not compiled because part of conditional compilation statements in the listing file or not.
LISTINCLUDE/NOLISTINCLUDE	specifies whether to include lines coming from include files in the listing or not.
PRINT/NOPRINT	Specifies whether to generate a preprocessor listing file.
SYMBOLS	Adds the list of symbols generated from the source file.

1. For complete descriptions of compiler C language options, refer to the *Raisonance RCST7 Compiler Manual*.

Customizing Raisonance C compiler preprocessor definitions

On the **C Compiler** tab of the **Project Settings** window, in the **Category** field, select **Preprocessor** to display the view shown in [Figure 83](#).

Figure 83. Raisonance C compiler preprocessor view



This view allows you to add preprocessor commands, and to specify include directories.

Adding preprocessor definitions

When entering a preprocessor definition, you must specify the name of a user-defined preprocessor symbol. The form of the definition is:

```
DF (symbol [=value] )
```

If the value is omitted, it is set to 1.

When entering definitions, the `DF` option is automatically added to the command line as you type the symbol. When you enter several definitions, separate them with a space or a comma.

Specifying include directories

The field **Additional include directories** allows you to specify the paths for other directories that you want to include when compiling. You can type the path name or use the browse button to locate the directory. The form of the definition that is added to the command line is:

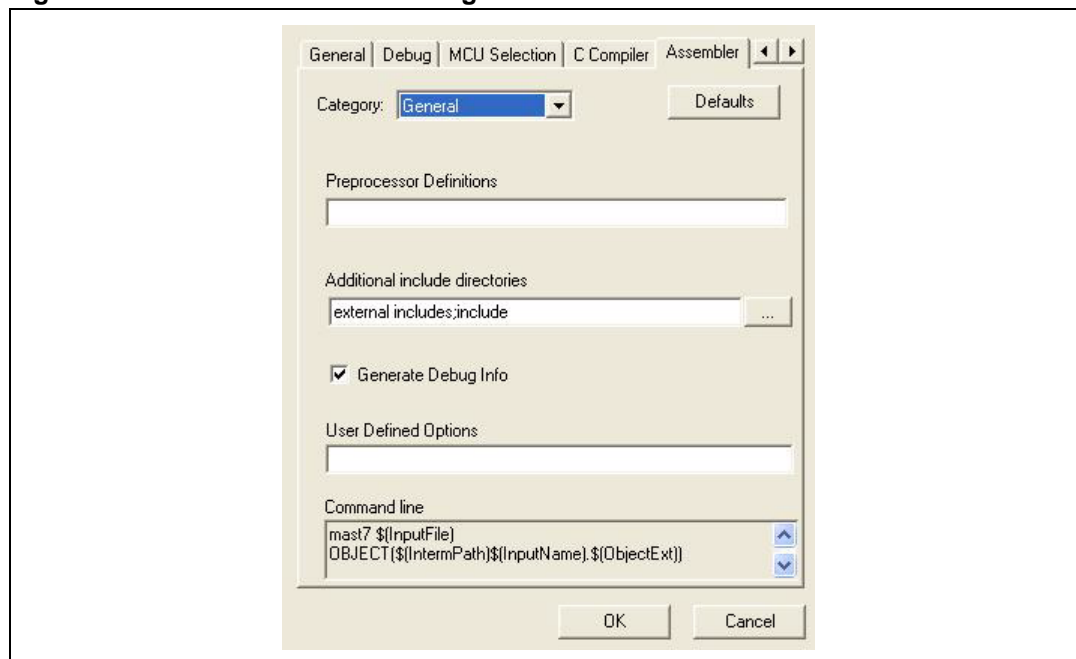
```
PIN (pathname)
```

When you type the path name for the directory, the `PIN` option is automatically added to the command line. Each path name is the name of a directory and is not ended with a directory separator (`\`). When you enter several path names, separate them with a semicolon (`;`) and the `PIN` option is added automatically to the next entry.

4.8.2 Raisonance Assembler tab

The **Assembler** tab of the **Project Settings** window provides an interface for setting the options in the command line for the Raisonance C assembler. The gray **Command line** field displays all the changes to the command line as you make them through the graphical interface.

Figure 84. Raisonance Assembler general view



The category list box gives you access to the following views:

- General, see [General settings for Raisonance Assembler](#)
- Listings, see [Customizing Raisonance Assembler listings](#)

General settings for Raisonance Assembler

When the category is set to **General**, the options available are identical to the preprocessor view of the C Compiler tab, already described in [Customizing Raisonance C compiler preprocessor definitions on page 124](#).

In addition, this view offers a **Generate Debug Info** checkbox, that adds the `SET` option in the command line at the bottom of the tab which enables the generation of debug information.

Customizing Raisonance Assembler listings

When the category is set to **Listings** and the **Generate a Listing file** checkbox is enabled, you have access to the listing file options described in [Table 45](#).

Table 45. Raisonance assembler listing options

Directive	Description ⁽¹⁾
COND/NOCOND	Specifies whether to show unassembled line of conditional constructs.
NOLIST	Specifies to not include the program source text. By default, it is included.
LISTINCLUDE/ NOLISTINCLUDE	Includes the content of include files in the listing.
GEN/NOGEN	Specifies whether to expand assembly instructions of macros.
SB/NOSB	Specifies whether to generate a table of symbols.
XREF/NOXREF	Specifies whether to generate a cross-reference table of symbols.

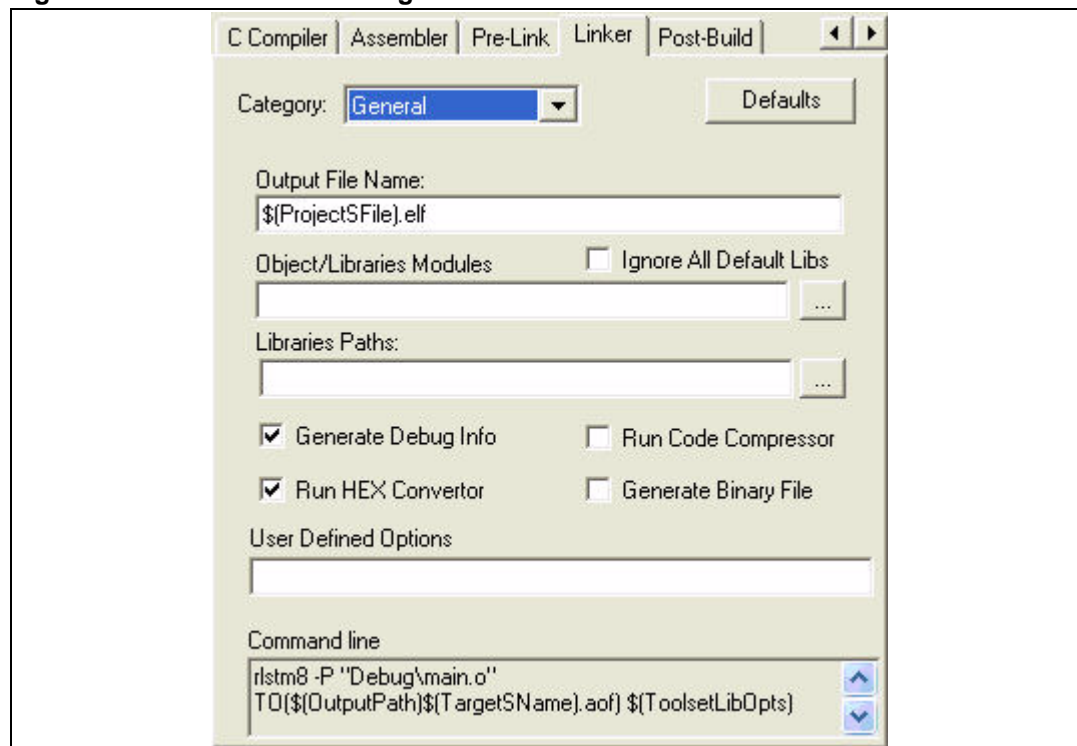
1. For complete descriptions of assembler listing options, refer to the *Raisonance MA-ST-7 Assembler Manual*.

In addition, in this view you can enable the **Log assembly errors in a file (.ERR)** to generate an error log file.

4.8.3 Raisonance C linker tab

The **Linker** tab of the **Project Settings** window provides an interface for setting the options in the command line for the Raisonance C linker. The gray **Command line** field displays all the changes to the command line as you make them through the graphical interface.

Figure 85. Raisonance linker general view



The **Category** list box provides access to the following views:

- **General:** displays standard settings and options, see [General settings for Raisonance linker](#)
- **Input:** provides access to linker input options, see [Customizing Raisonance linker input settings](#)
- **Output:** provides access to linker output options, see [Customizing Raisonance linker output settings](#)

General settings for Raisonance linker

When the category is set to **General**, you can access the following standard options:

- **Output filename**
Allows you to specify the name for the output file. By default, this field contains a macro that specifies the project source in *.elf format.
- **Object/Library modules**
Use this field to enter the names for custom libraries or object modules to include in the build. You do not need to specify standard libraries.
- **Library path names**
Use this field to enter additional libraries to be included by the linker. You can enter the library path name or use the browse button to locate the libraries to include. If you add several libraries, separate each one by a semicolon (;). For each library that you add, the `LIBPATH` option followed by the library path name is added to the **Command line** field at the bottom of the tab.
- **Generate Debug Info**
Enable this checkbox if you want to generate debug information.
- **Run HEX converter**
The converter (`ohst7` utility) allows you to translate executable files produced by the linker into Intel standard hexadecimal format, and output to a specific location.
When you place a checkmark in the **Run HEX Converter** checkbox, a command line is added to the **Commands** field in the **Post-build** tab. A pop-up window opens whenever you activate this option, to indicate that the command line is being added to or removed from the **Post-build** tab.
- **Ignore all default libraries**
Place a checkmark in this checkbox if you do not want the compiler to use the default libraries. In this case, use the **Object/Libraries Modules** field to specify the location of the libraries to use.
- **Run Code Compressor**
When **Run Code Compressor** is selected, the code compressor runs during the link thus allowing some in-lining and code factorization.
- **Generate binary file**
When you place a checkmark in this checkbox, the `ohst7` utility produces a binary file.
- **User-defined options**
The **User Defined Options** field allows you to enter the command for an option that you have defined for the linking stage. The options that you type in this field are immediately added to the command line below. For information on creating user-defined options, refer to the *Raisonance ST7 Linker/Locator Manual*.

Customizing Raisonance linker input settings

On the **Linker** tab of the Project Settings window, in the **Category** field, select **Input** to display the view shown in [Figure 86](#).

Figure 86. Raisonance linker input view

C Compiler | Assembler | Pre-Link | **Linker** | Post-Build

Category: **Input** Defaults

Memory Area	From	To
DATA	0x0	0x17ff
CODE	0x8080	0x27ff
STACK	0x1400	0x17ff

Relocation Directives

User Defined Options

Command line

```
rlst7 -P "Debug\newnew.o", "Debug\main.o"
TO:${OutputPath}${TargetSName}.aof) ${ToolsetLibOpts}
```

OK Cancel

Note that variables are automatically allocated a memory location. For information on memory relocation directives, refer to the *Raisonance ST7 Linker/Locator Manual*.

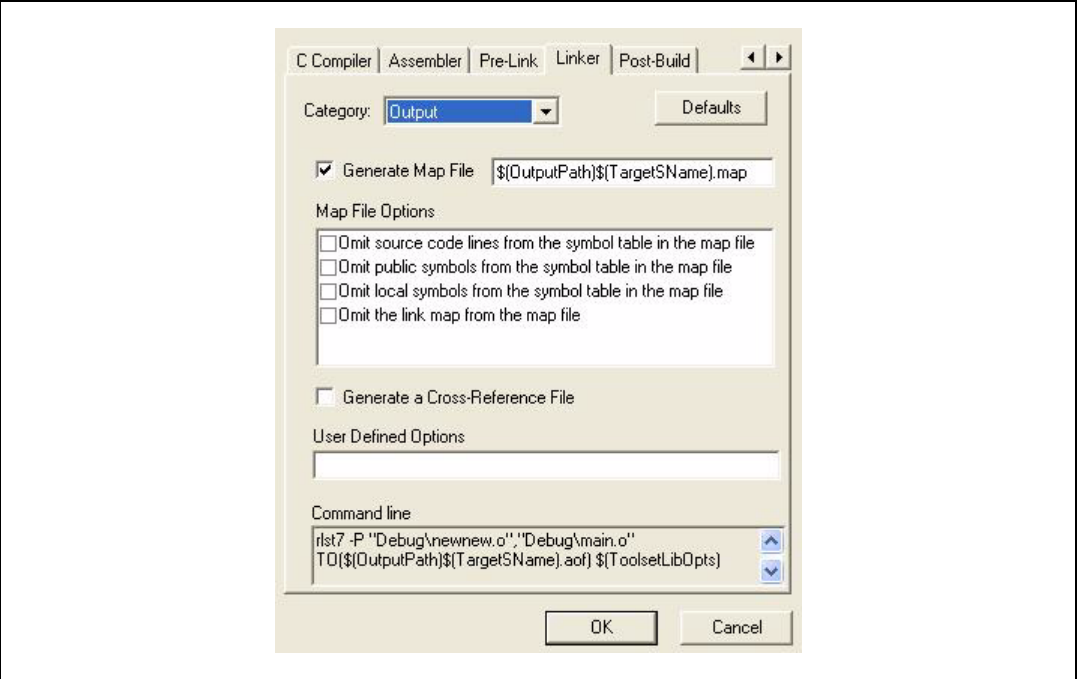
The **Relocation Directives** field allows you to directly enter relocation directives using the syntax defined in the *Raisonance ST7 Linker/Locator Manual*.

The **User Defined Options** field allows you to enter the command for an option that you want to apply when generating the linker input. The options that you type in this field are immediately added to the command line below. For information on creating user-defined options, refer to the *Raisonance ST7 Linker/Locator Manual*.

Customizing Raisonance linker output settings

On the **Linker** tab of the **Project Settings** window, in the **Category** field, select **Output** to display the view shown in [Figure 87](#).

Figure 87. Raisonance linker output view



The following options are available in this view:

- **Generate map file**
You can generate a map file by checking this option and entering a map file name in the field provided. By default, this option is checked, and a map file is generated with the output path name and the extension .map.
The available map file options are described in [Table 46](#). By default, all options are turned on. You can use the list to turn each one off individually.
- **Generate a cross-reference file**
This checkbox instructs the linker to generate a cross-reference file as an aid to debugging.
- **User defined options**
The **User Defined Options** field allows you to enter the command for an option that you want to apply when generating the linker output. The options that you type in this field are immediately added to the command line below. For information on creating user-defined options, refer to the *Raisonance ST7 Linker/Locator Manual*.

Table 46. Raisonance linker output options

Directive	Description ⁽¹⁾
NOLI	Omits source code lines from the symbol table in the map file
NOPU	Omits public symbols from the symbol table in the map file
NOSB	Omits local symbols from the symbol table in the map file
NOMA	Omits the link map from the map file

1. For complete descriptions of linker output options, refer to the *Raisonance ST7 Linker/Locator Manual*.

4.9 Customizing build settings for Metrowerks C toolset

The **Project Settings** window for building your application with the Metrowerks C toolset contains five tabs that are common to all the toolsets (General, MCU selection, Debug, Pre-link and Post-Build) and three tabs that contain options and settings that are specific to Metrowerks.

This section provides details about configuring options that are specific to the Metrowerks toolset. These options include:

- [Metrowerks C Compiler tab](#)
- [Metrowerks Assembler tab](#)
- [Metrowerks linker tab](#)

Note: For more details regarding options for this toolset, refer to the **STMicroelectronics ST7 Compiler** and **STMicroelectronics ST7 Assembler** reference manuals from Metrowerks.

Configuring project settings in the five common tabs is described in [Section 4.5: Configuring project settings on page 84](#).

Provided ST7 C header files for Metrowerks

Depending on your MCU you can include a header file provided with STVD, which defines the device's peripheral registers. This will save you the time of creating this header file yourself.

Caution: The C header files provided with STVD are not the same as those provided for the ST7 Library. ST7 library users should use the C header and/or ASM include files specific to that library.

Depending on your installation, the C header files provided with STVD are typically located at:

```
C:\Program Files\STMicroelectronics\st_toolset\include\
```

To build your project using the provided header file for your device, specify its inclusion in your application code. The name of the file to use is *device_name.h*.

Header files are provided for virtually all the supported devices. If you don't find your device's name in the header files, this may be because the full name is not used. To determine if your device has a header file, you can check the Peripheral Registers window (**View > Peripheral Registers**) in the debug context. If peripheral register information is provided in this window, then a header file has been generated for your device. The correct file for your device, is the one with the device name you found in the Peripheral Registers window.

When including the provided header files, you should be aware of the following:

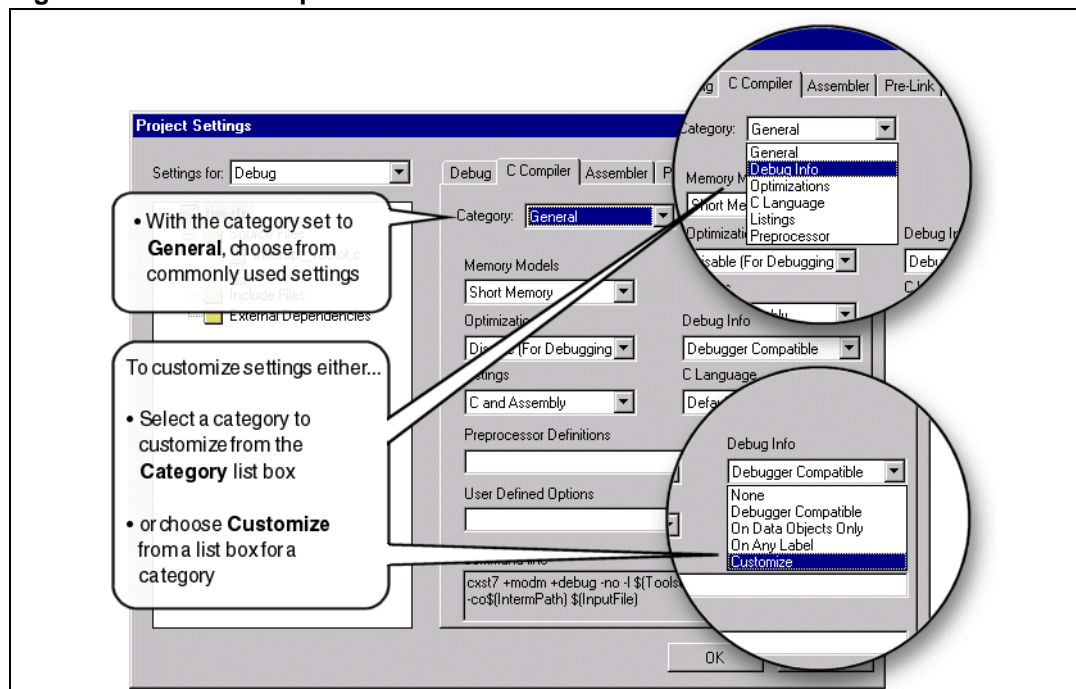
- To specify the inclusion of the header file (**#Include**), you must add the define statement, **#Define __DEFINE_REGISTERS_STVD7_INCLUDE__** in one source file and before any include statement.
- A link error L1818 may be generated due to paginated registers. To override this error, apply **-wmsgsi1818** as a user-defined option. This option will convert the error to a warning message only.

Using the Project Settings interface

The tabs (C Compiler, Assembler and Linker) have a common interface that allows you to choose from standard configurations, customize options and view changes to the command line as you apply different options.

When the **Category** list box is set to **General**, the tab provides check boxes and list boxes for easy access to commonly used options and configurations (see [Figure 88](#)).

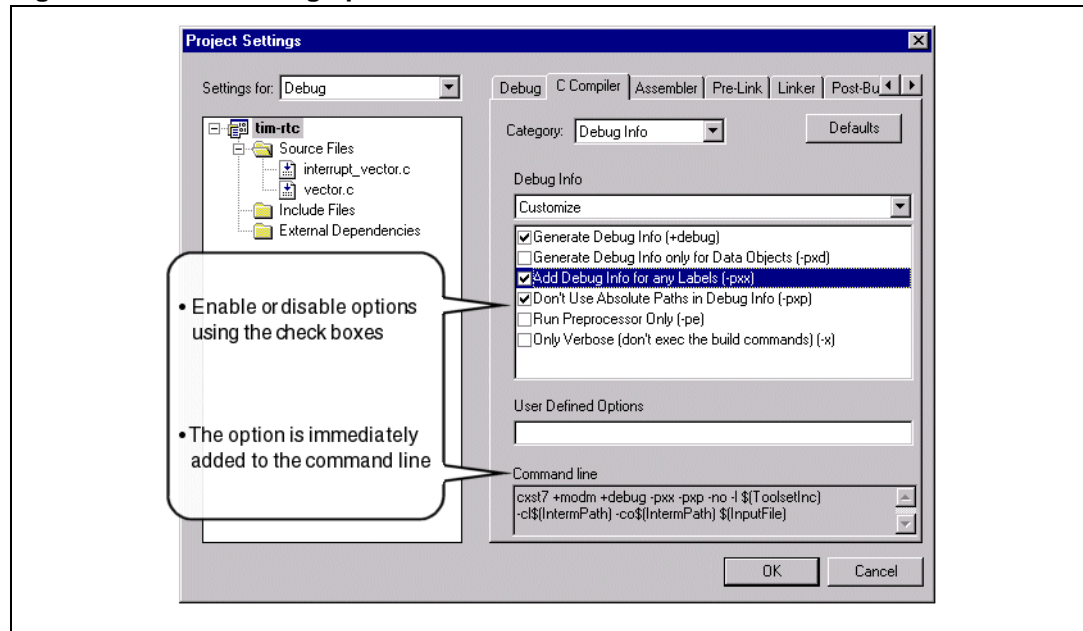
Figure 88. Standard options



To further customize options, select a category in the **Category** list box, or select **Customize** in a list box for a category of settings. The view in the tab will change to present more options (see [Figure 89](#)).

You can enable or disable options by clicking on the check boxes. When you invoke an option, the change to the command line is immediately visible in the gray **Command Line** window at the bottom of the tab.

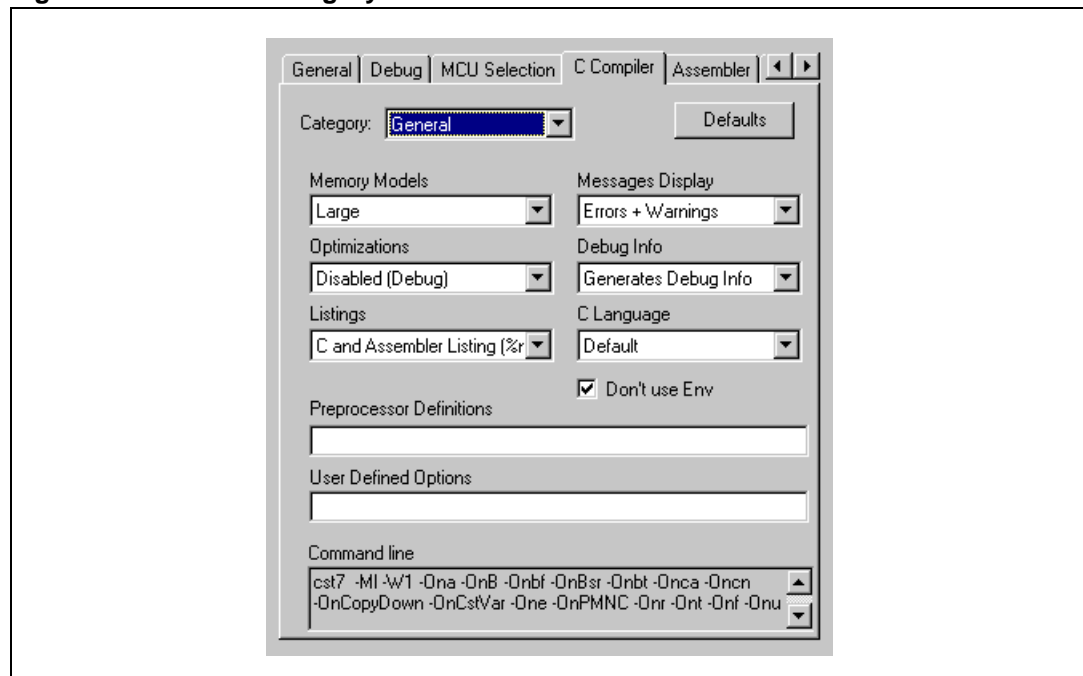
Figure 89. Customizing options



4.9.1 Metrowerks C Compiler tab

The **C Compiler** tab provides an interface for setting the options in the command line for the Metrowerks C compiler. The gray **Command** line field displays all the changes to the command line as you make them.

Figure 90. General category view



The **Category** list box provides a choice of standard options to choose from. For descriptions of standard options and information about further customizing them, refer to the category sections below:

- **General:** We recommend starting in the [General settings for the Metrowerks toolset](#) view.
- **Optimizations:** [Customizing Metrowerks compiler optimizations](#)
- **Input:** [Customizing Metrowerks C compiler input](#)
- **Output:** [Customizing Metrowerks C compiler output](#)
- **Listings:** [Customizing Metrowerks C compiler listings](#)
- **C Language:** [Customizing Metrowerks C compiler language settings](#)

General settings for the Metrowerks toolset

With the category set to **General**, you can access the following standard settings and options:

- [Compiler memory models](#)
- [Message Display](#)
- [Optimizations](#)
- [Debug info](#)
- [Listings](#)
- [C Language](#)
- [“Do not use environment” option \(-NoEnv\)](#)
- [Preprocessor definitions](#)
- [User-defined options](#)

Compiler memory models

From the **General** interface, you can choose to compile your application according to one of three predefined memory models that can help you optimize your code's size and performance (see [Figure 90](#)).

[Table 47](#) below provides a summary of these predefined memory models.

Table 47. Metrowerks C compiler memory models

Memory model ⁽¹⁾	Local variables ⁽²⁾	Global variables ⁽²⁾	Use
Large (-M1)	Direct	Extended or Direct	<ul style="list-style-type: none"> – Default for debugging – Most efficient memory model for typical microcontroller applications.
Small (-Ms)	Direct	Direct	For small applications that only reference global constant data (Strings) through <i>_far</i> pointers.
Extended (-Mx)	Extended	Direct	For large applications with lots of local and global data

1. For complete descriptions of memory model options, refer to the **STMicroelectronics ST7 Compiler** user manual from Metrowerks.

2. Direct: Data must be in the zero page, so that short addressing mode is used for all data access.

Extended: Data can be either in the zero page (short addressing mode), or outside the zero page (extended addressing mode).

Message Display

With the **Category** list box set to **General**, you can choose what types of messages to generate from the following options:

- **Errors + Warnings:** Introduces `-w1` option in the command line, which suppresses the generation of information messages.
- **Only Errors:** Introduces `-w2` option in the command line, which suppresses the generation of error and warning messages.
- **Errors + Warnings + Info:** No option in the command line. Information, error and warning messages are generated.

Optimizations

The **Optimization** settings allow you to optimize your code once you have finished debugging your application, in order to make the final version more compact and faster. You should not optimize your code when debugging because some optimizations will eliminate or ignore the debug information required by STVD and your debug instrument.

When the **Category** list box is set to **General**, the standard optimization options are:

- **Default (Minimize code size):** Introduces `-Os` option in the command line, which directs the Compiler to generate smaller code, even though it may be slower than other possible solutions.
- **Disabled (Debug):** Introduces options in the command line that disable optimizations that are not compatible with debugging.
- **Maximize Execution Speed:** Introduces `-O3` option in the command line, which directs the Compiler to generate fast code, even though the code may be longer than other possible solutions.
- **Customize:** Allows you to choose the optimization options that you want (options are summarized in [Customizing Metrowerks compiler optimizations on page 137](#)). For complete descriptions of these options, refer to the **STMicroelectronics ST7 Compiler** user manual from Metrowerks.

Debug info

The **Debug Info** settings allow you to choose to generate debug information for your application based on use (for programming or debugging). By default, the Metrowerks C compiler generates the debug information required to debug your application with STVD. Including the debug information will make the application file larger, however debug information is not included in the code that is loaded in the microcontroller's memory. This information is used by STVD's debugger only.

When the **Category** list box is set to **General**, the standard debug information options are:

- **Generates debug info:** No option in the command line. This setting causes the compiler to generate the debug information required by STVD
- **No Debug info in Objects:** Introduces `-NoDebugInfo` option in the command line, which suppresses the generation of debug information in the object file(s).
- **Strip Path Info from Objects:** Introduces `-NoPath` option in the command line, which suppresses the generation of path information in the object files

Listings

The **Listing** settings allow you to generate a listing file and log errors to a file.

When the **Category** list box is set to **General**, the standard listing options are:

- **C and Assembler Listing:** Adds `-Lasm` option to the command line. The output is `filename.lst`.
- No Listing
- **List of Include Files:** Adds `-Li` option to the command line. The output is `filename.inc`.
- **Preprocessor Output:** Adds `-Lp` option to the command line, which generates a text file containing the preprocessor output. The output is `filename.pre`.
- **Customize:** Allows you to choose the Listing options you want (options are summarized in [Customizing Metrowerks C compiler listings on page 141](#)).

C Language

The Compiler C language settings allow you to define the programming language options to apply while compiling the application.

When the **Category** list box is set to **General**, the standard language options are:

- **Default:** No option in the command line. The compiler accepts ANSI standard keywords and some non-ANSI compliant keywords, including: `@address`, `far`, `near`, `rom`, `uni`, `_alignof_`, `_va_sizeof_`, `interrupt` and `asm`.
- **Strict ANSI Checking:** Introduces `-Ansi` option in the command line, which restricts the compiler to recognition of ANSI standard keywords.
- **Customize:** Allows you to choose from the C language options that are listed in [Customizing Metrowerks C compiler language settings on page 143](#). For complete descriptions of options, refer to Metrowerks **STMicroelectronics ST7 Compiler** user manual.

“Do not use environment” option (-NoEnv)

This checkbox enables and disables the no environment option (`-NoEnv`) in the startup routine. This option is specified independently for the assembler (see [Section 4.9.2: Metrowerks Assembler tab on page 143](#)) and the linker (see [Section 4.9.3: Metrowerks linker tab on page 148](#)).

When checked in the C compiler tab, the `-NoEnv` option is applied to the compiler command line. This option is checked by default so the compiler uses environment variables specified by options.

Uncheck this option if you want to use an environment file (such as `default.env`) to specify environment variables for the C compiler. The environment file should be located in the current directory.

Preprocessor definitions

This field allows you to enter user-specified macros for the preprocessor. The compiler allows the definition of a macro in the command line. The effect is the same as having a `#define` directive at the beginning of the source file.

When entering definitions, the `-d` option is automatically added to the command line as you type the symbol. When you enter several definitions, separate them with a space and `-d` will automatically be added to the next symbol in the command line.

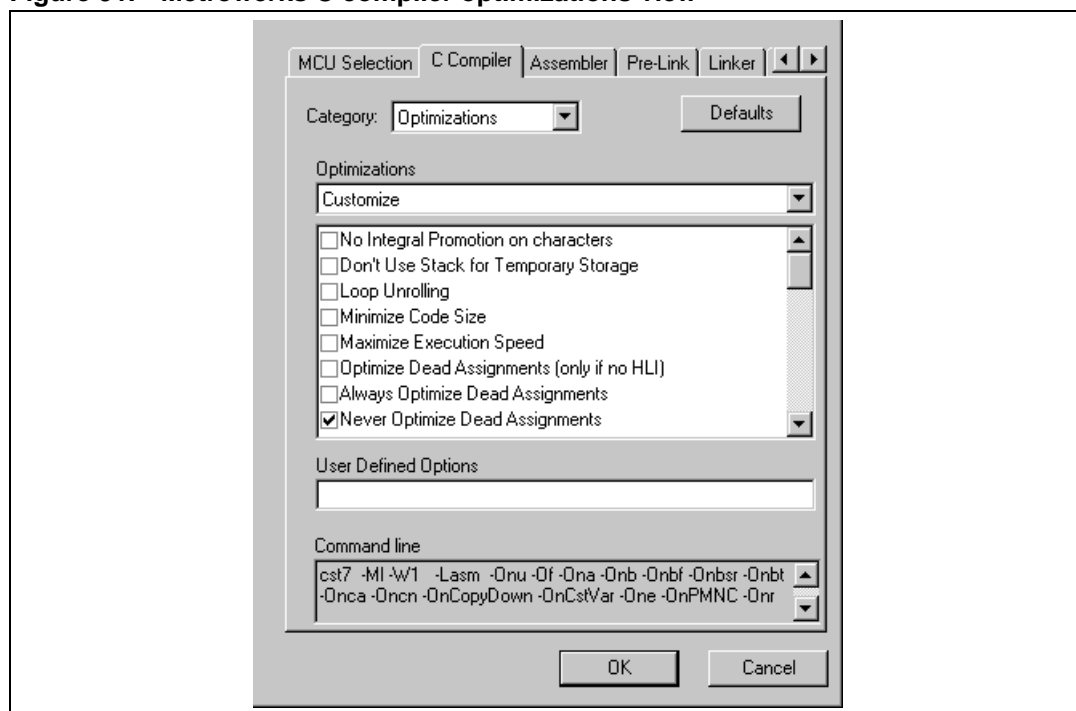
User-defined options

This field allows you to enter the command for an option that you have defined for the C compiler. The options that you type in these fields are immediately added to the command line below. For more information on creating user-defined options, refer to Metrowerks' *STMicroelectronics ST7 Compiler* reference manual.

Customizing Metrowerks compiler optimizations

To further customize optimizations, select **Customize** from the **Optimizations** list box. The view in the **C Compiler** tab changes to that shown in [Figure 91](#).

Figure 91. Metrowerks C compiler optimizations view



In this view, you can choose from the optimization options listed in [Table 48](#).

Table 48. Metrowerks C compiler optimization options

Option ⁽¹⁾	Description
-Cni	No integral promotion on characters
-Cns	Do not use stack for temporary storage
-Cu	Loop unrolling
-Obfv	Optimize bitfields and volatile bitfields
-Obsr	Generate always near calls
-Oc	Common sub-expression elimination
-OdocF	Dynamic option configuration for functions
-Of	Create sub-functions with common code when optimizing code size (-Os)

Table 48. Metrowerks C compiler optimization options (continued)

Option ⁽¹⁾	Description
-Oi=c\$CodeSize	Enables inline expansion
-Oi=c0	Switches on inlining except for Pragma INLINE
-Oi=OFF	Switches off inlining except for Pragma INLINE
-OiLib	Inline library functions
-Ol0	Totally disable loop variables allocation in registers
-Ona	Disable alias checking
-Onb	Disable peephole optimization
-Onbf	Disable optimize bitfields
-OnBsr	Disable far-to-near optimization
-Onbt	Disable ICG level branch tail merging
-Onca	Disable and constant folding
-Oncn	Disable any constant folding in case of a new constant
-OnCopyDown	Don't generate copy down information for zero values
-OnCstVar	Disable constant variable replaced by value
-One	Disable and low level common sub-expression elimination
-Onf	Create sub-functions with common code when optimizing code execution speed (-Ot)
-OnPMNC	Disable code generation for NULL Pointer to Member check
-Onr	Disable optimizing register accesses
-Ont	Disable all tree optimizations
-Onu	Never optimize dead assignments
-Or	Allocate local variables to registers
-Os	Minimize code size
-Ot	Maximize execution speed
-Ou	Always optimize dead assignments
-Rpe	Large return value with temporary elimination
Rpt	Large return value by pointer

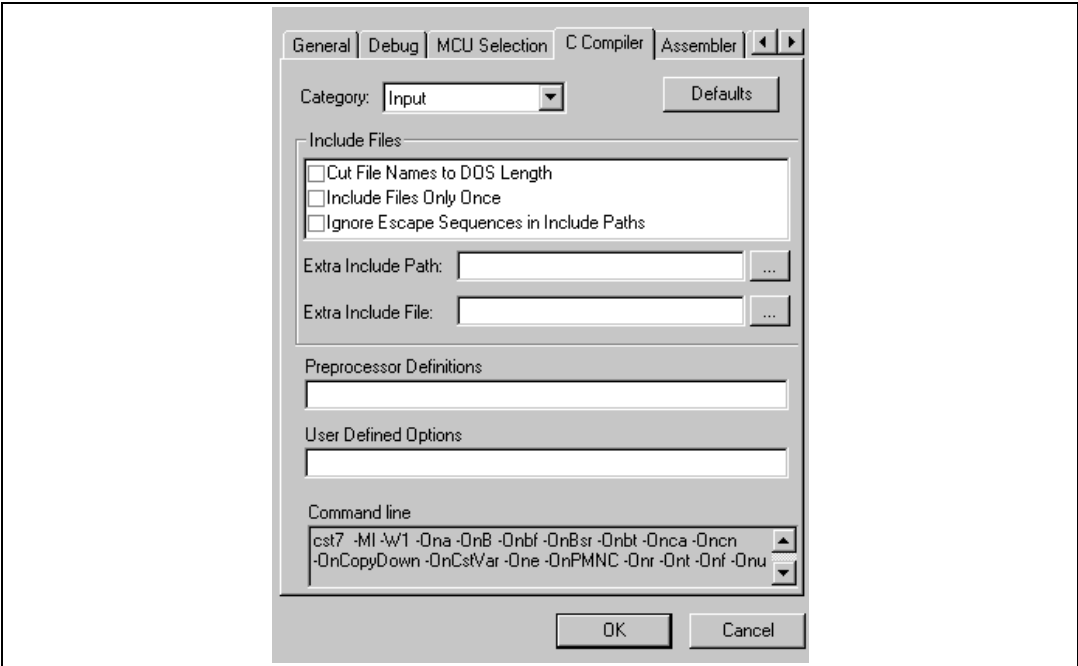
1. For a complete description of optimization options, refer to the **STMicroelectronics ST7 Compiler** user manual from Metrowerks.

Customizing Metrowerks C compiler input

The compiler input settings allow you to specify the options defining the files to be used in compiling your application.

In the C Compiler tab, select **Input** in the **Category** list box. The view in the **C Compiler** tab changes to that shown in [Figure 92](#).

Figure 92. Metrowerks C compiler input view



In this view, you can choose from the optimization options listed in [Table 49](#).

Table 49. Metrowerks C compiler input options

Option ⁽¹⁾	Description
Cut File Names to DOS Length (- !)	This option cuts all file names to a length of 8 characters
Include Files Only Once (- Pio)	This option causes the compiler to ignore include files that have already been read once while compiling.
Ignore Escape Sequences in Include Paths (- Pe)	This option causes the compiler to ignore escape sequences in strings that contain a DOS drive letter followed by a colon and a back slash(:\). This is useful when the code contains macros with file path names.

1. For complete descriptions, refer to the **STMicroelectronics ST7 Compiler** user manual from Metrowerks.

In addition, this view includes fields that allow you to enter the path name for a directory, or a file to include when compiling.

Extra include paths

This field allows you to enter multiple paths for directories to include when compiling the application.

You can type the path name. When you begin typing, the -I for this compiler option is automatically added to the command line. Separate directory entries with a semi-colon (;). When you type the semi-colon to start a new entry, the -I for the new entry is automatically added to the command line.

You can use the browse button to locate a directory. Click on the file you want in the browse window and the command line is automatically modified with the -I option and the path name. To make another entry press the space bar and the browse window opens automatically so that you can enter the next file path name.

Extra include files

This field allows you to enter multiple files to include when compiling the application.

You can type the file's path name. When you begin typing, the `-AddInclude=` for this compiler option is automatically added to the command line. Separate multiple include files with a semi-colon (;). When you type the semi-colon to start a new entry, the `-AddInclude=` for the new entry is automatically added to the command line.

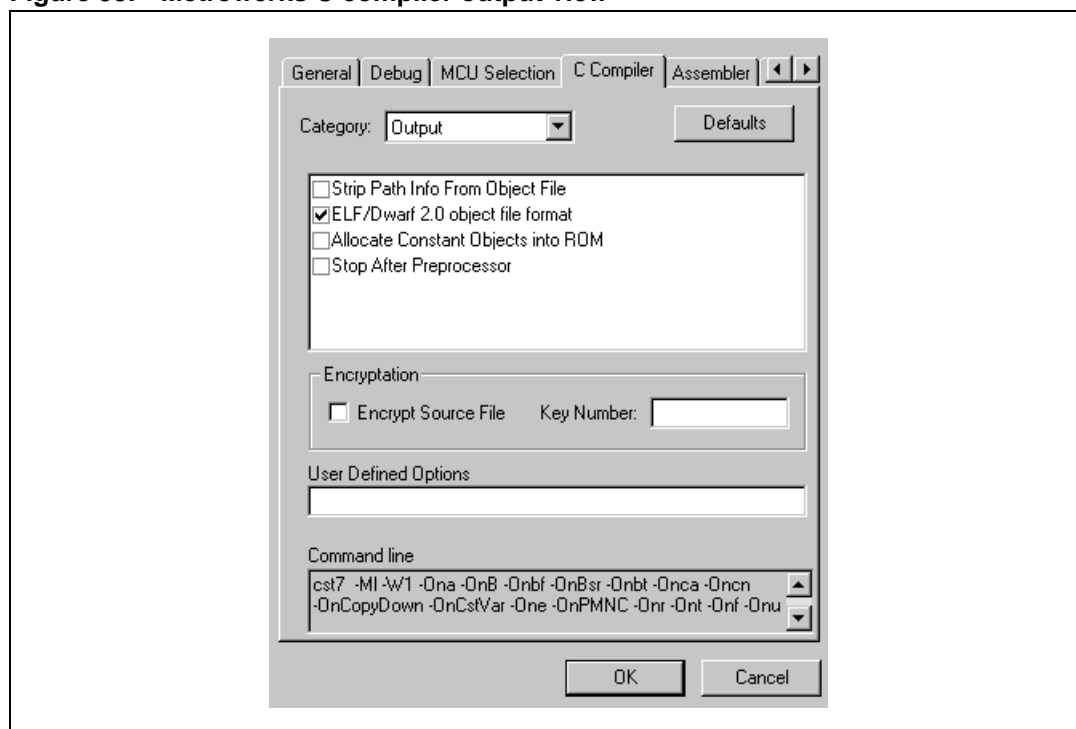
You can use the browse button to locate a file to include. Click on the file you want to include in the browse window and the command line is automatically modified with the `-AddInclude=` option and the path name. To make another entry press the space bar and the browse window opens automatically, so that you can enter the next file path name.

Customizing Metrowerks C compiler output

The compiler output settings allow you to customize the type and options for files that are output from the compiler.

In the C Compiler tab select **Output** in the **Category** list box. The view in the **C Compiler** tab changes to that shown in [Figure 93](#).

Figure 93. Metrowerks C compiler output view



In this view, you can choose from the compiler options listed in [Table 50](#).

Table 50. Metrowerks C compiler output options

Option ⁽¹⁾	Description
Strip Path Info from Object Files (-NoPath)	This option suppresses the generation of path information in the object files.
ELF/Dwarf 2.0 Object File Format (-F2)	Adds the -F2 option to the command line, which causes the compiler to generate object files in ELF/Dwarf 2.0 format. When not checked the compiler generates object files in Hiware object file format (-Fh option in the command line). If you change the output setting to Hiware, STVD warns you that you must specify the PRM file to use in the Linker PRM File settings of the Linker tab (see Configuring the Metrowerks linker PRM file on page 153).
Allocate Constant Objects into ROM (-Cc)	This option causes any variables defined as <code>const</code> to be assigned to a ROM section in the linker PRM file.
Stop after Preprocessor (-LpX)	This option limits compiling to preprocessing so that no object file is generated. Used with -Lp to generate a listing file (see Customizing Metrowerks C compiler listings).

1. For complete descriptions of optimization options, refer to the **STMicroelectronics ST7 Compiler** user manual from Metrowerks.

In addition, this view includes an **Encryption** field that allows you to enable and disable encryption and set a key number for the encryption.

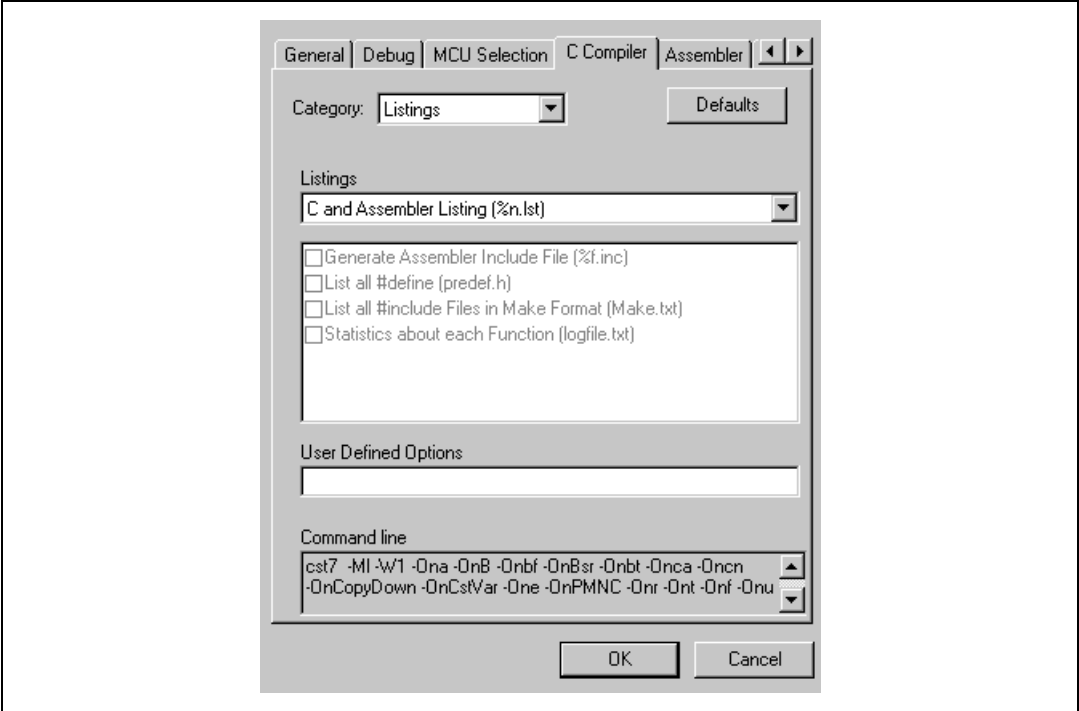
Checking the **Encrypt Source File** checkbox introduces the -Eencrypt option to the command line, which causes the compiler to encrypt the source file using a key code that you enter. The default name for the encrypted file is *filename.file extension*_e (for example, the encrypted file for `application.c` is `application.ce`).

To enter a key code for encryption, type the number in the **Key Number** field. Only numeric characters are accepted. If you enter a non-numeric character in the Key Number field, STVD prompts you with a warning. The -EKey option and the number you entered are automatically added to the command line. 0 is the default if you do not enter a key code. However, using this default is not recommended.

Customizing Metrowerks C compiler listings

If you choose **Customize** in the **Listings** list box, the tab view changes as shown in [Figure 94](#). The same tab view is displayed if you choose **Listing** directly from the **Category** list box.

Figure 94. Metrowerks C compiler listing view



In this view, you choose from the listing options summarized in [Table 51](#).

Table 51. Metrowerks C compiler listing options

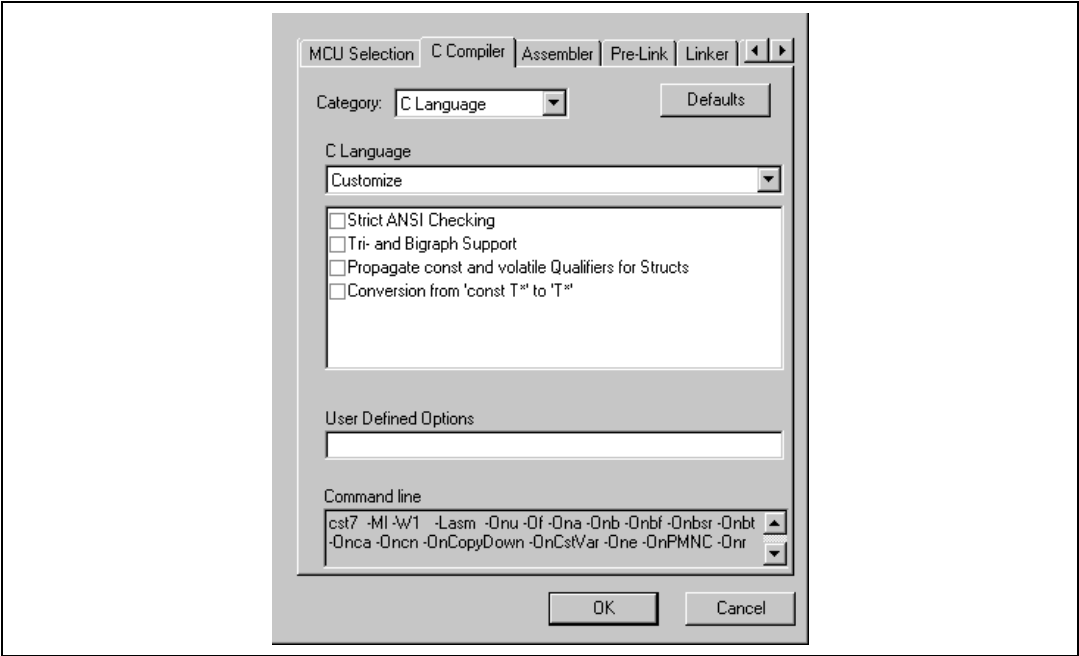
Option ⁽¹⁾	Description
Generate Assembler Include Files (-La)	This option causes the compiler to generate an assembler include file when a pragma CREATE_ASM_LISTING occurs. The default output is <i>filename.pre</i> .
List all #define (-Ldf)	This option allows you to list all #defines in the compiled source files. The output is <i>predef.h</i> .
List all #include Files in Make Format (-Lm)	This option creates a listing of #include files specified by the source to aid you in the creation of a makefile. The output is <i>make.txt</i> .
Statistics about each function (-Ll)	This option generates a file with the following statistics for procedure in the compiled file: code size in bytes, stack usage in bytes and compilation time. The output is <i>logfile.txt</i> .

1. For complete descriptions of optimization options, refer to the **STMicroelectronics ST7 Compiler** user manual from Metrowerks.

Customizing Metrowerks C compiler language settings

In the **C Compiler** tab, select **C Language** in the **Category** list box. The view in the **C Compiler** tab changes to that shown in [Figure 95](#).

Figure 95. Customizing C language view



In this tab view, you can choose from the language options described in [Table 52](#).

Table 52. Metrowerks C compiler language options

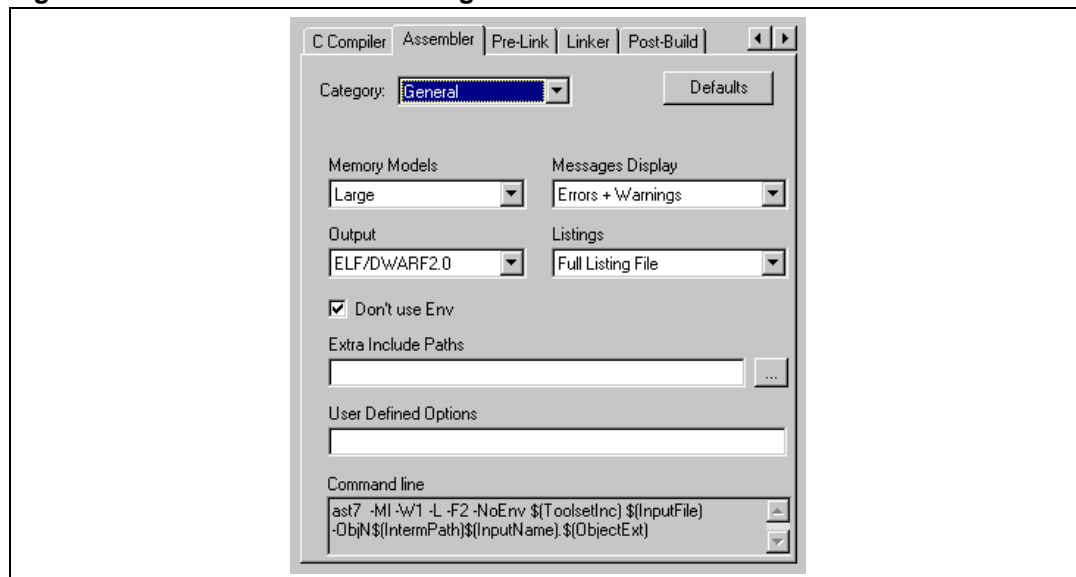
Option ⁽¹⁾	Description
Strict ANSI (-Ansi)	This option restricts the Compiler to recognition of ANSI-standard keywords.
Tri- and Bigraph Support (-ci)	This option enables the interpretation of Trigraph and Bigraph symbols if present in the C source code.
Propagate const and Volatile Qualifiers for Structs (-Cq)	If a structure is labeled constant or constant/volatile, then this option propagates the qualifier to all members of the structure.
Conversion from 'const T*' to 'T*' (-Ec)	This option causes pointers to constants to be treated as pointers to their non-constant equivalents. Not ANSI compliant

1. For complete descriptions of optimization options, refer to the **STMicroelectronics ST7 Compiler** user manual from Metrowerks.

4.9.2 Metrowerks Assembler tab

The **Assembler** tab provides an interface for customizing the command line for the Metrowerks assembler. The gray **Command line** field, displays all the changes to the command line as you make them.

Figure 96. Metrowerks Assembler general view



The **Category** list box allows you to access a general view for easy access to standard settings and options. You can also change the category to access views where you can customize settings in greater detail.

In the **Category** list box, you can choose one on the following views:

- **General:** We recommend starting in the [General settings for Metrowerks Assembler](#) view.
- **Language:** [Customizing Metrowerks Assembler language settings](#)
- **Listings:** [Customizing Metrowerks Assembler listings](#)

General settings for Metrowerks Assembler

With the category set to **General**, you can access the following standard settings and options:

- [Assembler memory models](#)
- [Message display](#)
- [Output](#)
- [Listings](#)
- [“Do not use environment” option \(-NoEnv\)](#)
- [Extra include paths](#)
- [User-defined options](#)

Assembler memory models

From the **General** interface, you can choose to compile your application according to one of three predefined memory models that can help you optimize your code's size and performance. These are the same command line options as for the **Compiler** tab. For a brief description of these options, refer to [Compiler memory models on page 134](#).

Message display

With the category set to **General**, you can choose what types of messages the Assembler will generate. These are the same command line options as those used by the compiler.

For a brief description of these options, refer to [Message Display on page 135](#).

Output

With the category set to **General**, you can choose the format of the object file that the Assembler will generate. You can choose from:

- **ELF/DWARF 2.0:** Adds the `-F2` option to the command line, which causes the assembler to generate object files in ELF/Dwarf 2.0 format.
- **Hiware:** Adds the `-Fh` option to the command line, which causes the assembler to generate object files in Hiware proprietary format. If you change the output setting to Hiware, STVD warns you that you must specify the PRM file to use in the Linker PRM File settings of the Linker tab (see [Configuring the Metrowerks linker PRM file on page 153](#)).

Note: We strongly recommend using the ELF/DWARF 2.0 format to build your application. This gives you access to a greater range of toolset options. The Hiware format continues to be supported for reasons of legacy compatibility.

Listings

The **Listing** settings allow you to generate a listing file and to log errors to a file.

When category is set to **General**, the standard listing options are:

- **Full Listing File:** Inserts the `-L` option in the command line. By default, the full listing contains expanded included files, as well as macro definition, call and expansion lines. The output is `filename.lst`.
- **No Listing**
- **Customize:** Allows you to choose the Listing options that you want (options are summarized in [Customizing Metrowerks C compiler listings on page 141](#)).

“Do not use environment” option (-NoEnv)

This checkbox enables and disables the no environment option (`-NoEnv`) in the startup routine. This option can be specified independently for the C compiler (see [Section 4.9.1: Metrowerks C Compiler tab on page 133](#)) and the linker (see [Section 4.9.3: Metrowerks linker tab on page 148](#)).

When checked in the Assembler tab, the `-NoEnv` option is applied to the assembler command line. This option is checked by default so that the assembler uses environment variables specified by options in the project settings.

Uncheck this option if you want to use an environment file (such as `default.env`) to specify environment variables for the assembler. The environment file should be located in the current directory.

Extra include paths

This field allows you to enter multiple paths for directories to include when assembling the application.

You can type the path name. When you begin typing, the `-I` for this compiler option is automatically added to the command line. Separate directory entries with a semi-colon (;). When you type the semi-colon to start a new entry, the `-I` for the new entry is automatically added to the command line.

You can use the browse button to locate a directory. Click on the file you want in the browse window and the command line is automatically modified with the `-I` option and the path name. To make another entry, press the spacebar and the browse window opens automatically so that you can enter the next file path name.

User-defined options

This field allows you to enter the command for an option that you have defined for the C compiler. The options that you type in these fields are immediately added to the command line below. For more information on creating user defined options, refer to Metrowerks' **STMicroelectronics ST7 Compiler** reference manual.

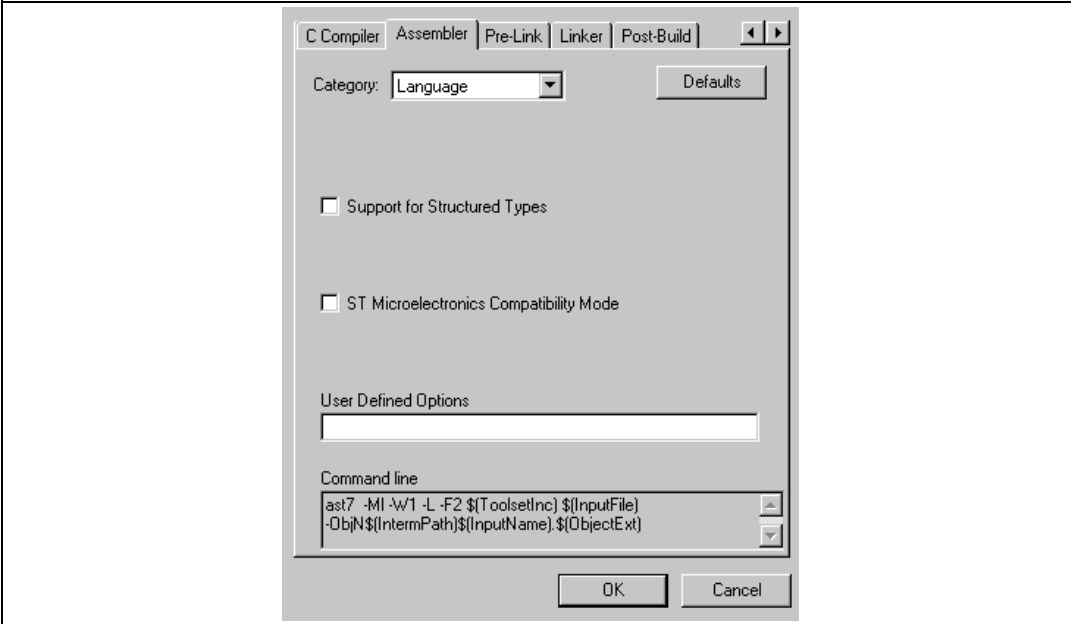
This view also provides a [Customizing Cosmic C compiler preprocessor definitions on page 105](#) and a [User-defined options on page 99](#) field for entering commands that you define.

Customizing Metrowerks Assembler language settings

The Assembler language settings allow you to set options related to language checking during assembly.

In the **Assembler** tab, select **Language** in the **Category** list box. The view in the **Assembler** tab changes to that shown in [Figure 97](#).

Figure 97. Metrowerks Assembler language view



In this view, you can choose from the language options listed in [Table 53](#).

Table 53. Metrowerks assembler language options

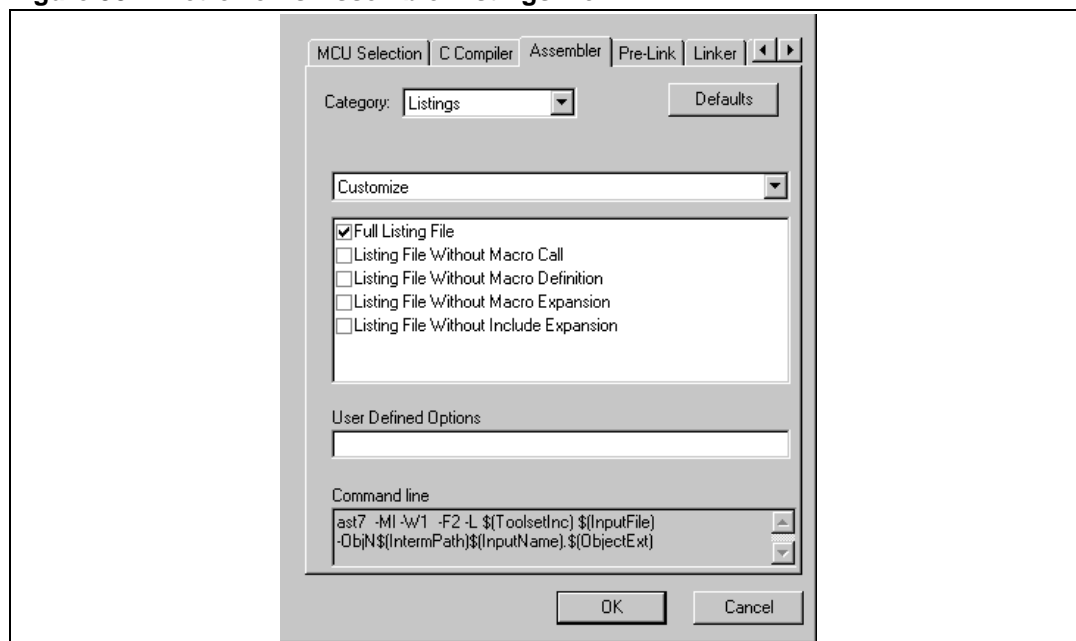
Option ⁽¹⁾	Description
Case Insensitivity on Label name (ELF/DWARF) (-ci)	Switches off case sensitivity for label names. However, this is only used when generating directly the ELF/DWARF 2.0 Absolute file (-FA2 option). Refer to Output on page 145 .
Support for Structured Types (-struct)	Supports the definition and use of structured types for applications containing ANSI C and Assembly modules.
STMicroelectronics Compatibility Mode (-st)	Allows the assembler to process input files containing ST Assembler notation and directives. Refer to the STMicroelectronics ST7 Assembler user manual from Metrowerks for limitations.

1. For complete descriptions of optimization options, refer to the **STMicroelectronics ST7 Compiler** user manual from Metrowerks.

Customizing Metrowerks Assembler listings

If you choose **Customize** in the **Listings** list box, the tab view changes as shown in [Figure 98](#). The same tab view is displayed if you choose **Listings** directly from the **Category** list box.

Figure 98. Metrowerks Assembler listings view



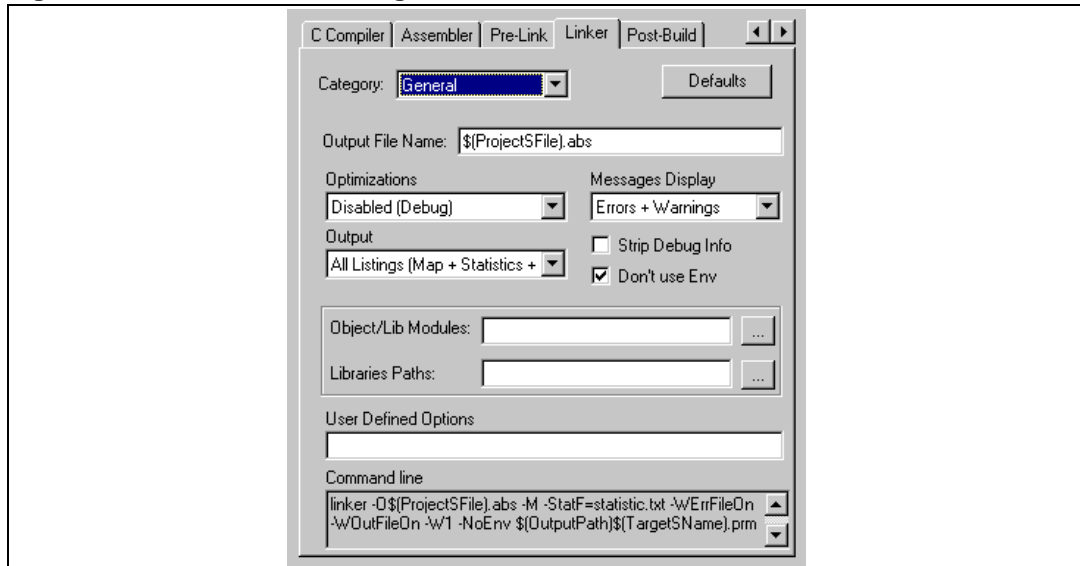
From this interface you can generate listings that include any, or all of the following:

- **Macro call lines** (-Lc)
- **Macro definition lines** (-Ld)
- **Expanded macros** (-Le)
- **Expanded include files** (-Li).

4.9.3 Metrowerks linker tab

The **Linker** tab provides an interface for customizing the command line for the Metrowerks SmartLinker. The gray **Command line** window, displays all the changes to the command line as you make them.

Figure 99. Metrowerks linker general view



The **Category** list box allows you to access a general view for easy access to standard settings and options. You can also change the category to access views where you can customize settings in greater detail.

In the **Category** list box, you can choose one of the following views:

- **General:** We recommend starting in the [General settings for Metrowerks linker](#) view.
- **Optimizations:** [Customizing Metrowerks linker optimization settings](#)
- **Output:** [Customizing Metrowerks linker output settings](#)
- **Linker PRM File:** [Configuring the Metrowerks linker PRM file](#)

General settings for Metrowerks linker

With the category set to **General**, you can access the following standard settings and options:

- [Output filename options](#)
- [Optimizations](#)
- [Message display](#)
- [Output](#)
- [Strip debug information](#)
- ["Do not use environment" option \(-NoEnv\)](#)
- [Object/Lib modules](#)
- [Library paths](#)

Output filename options

Allows you to specify the name for the output file. It appears in the command line preceded by the option `-O`.

Optimizations

The Optimization settings allow you to optimize your code once you have finished debugging your application, in order to make the final version more compact and faster. You should not optimize your code when debugging as some optimizations will eliminate or ignore the debug information required by STVD and your debug instrument.

When category is set to **General**, the standard optimization options are:

- **Priority is to Fill Banks:** This option minimizes the occurrence of partially filled blocks of banks. This option adds the `-DistOptiFillBanks` option in the command line. Used in generation of ELF files only.
- **Priority is to Minimize Code Size:** This option gives priority to minimizing the code size over the memory fill optimization. The `-DistOptiCodeSize` option is added to the command line. Used in generation of ELF files only.
- **Disable (For Debugging):** No optimizations, no option in the command line.
- **Customize:** Allows you to choose the options that you want for optimization (options are summarized in [Customizing Metrowerks linker optimization settings on page 151](#))

Note: The Metrowerks toolset outputs files in ELF format and a Hiware proprietary format. The optimization options above are for ELF format. Hiware optimizations are listed in [Table 54](#).

Message display

This allows you to determine what messages are displayed in the **Build** tab of STVD's **Output** window. The display options include:

- **Only Errors** (`-w2` in the command line)
- **Errors + Warnings** (`-w1`)
- **Errors + Warnings + Info** (`-wmsg`)

Output

The Output settings allow you to define listings and other output for the linker to generate.

When category is set to **General**, the standard output options are:

- **All Listings (Map + Statistics + Errors):** This option generates linker statistics in a text file (`statistic.txt`), in addition to map and error listings. This option adds the `-M` and `-StatF=statisti.txt` options in the command line.
- **No listing:** No listing generated, no option in the command line.
- **Customize:** Allows you to choose the options that you want for optimization (options are summarized in [Customizing Metrowerks linker output settings on page 152](#))

Strip debug information

Check the **Strip Debug Info** checkbox to remove debug information when you are generating the release version of your application. Introduces the `-s` option in the command line.

“Do not use environment” option (-NoEnv)

This checkbox enables and disables the no environment option (-NoEnv) in the startup routine. This option can be specified independently for the C compiler (see [Section 4.9.1: Metrowerks C Compiler tab on page 133](#)) and the assembler (see [Section 4.9.2: Metrowerks Assembler tab on page 143](#)).

When checked in the Linker tab, the -NoEnv option is applied to the linker command line. This option is checked by default so that the linker uses environment variables specified by options in the project settings.

Uncheck this option if you want to use an environment file (such as `default.env`) to specify environment variables for the linker. The environment file should be located in the current directory.

Object/Lib modules

This field allows you to specify additional object or library files without modifying the link parameter file.

You can type the library path name or use the browse button to enter the object and library modules to include. When entering more than one path, the entries are separated by a semicolon (;). When you add an element via this field, the -Add option followed by the filename are added to the command line.

Note: The Metrowerks Linker does not accept spaces in path names.

Library paths

This field allows you to enter the paths for additional objects or libraries to be included by the linker.

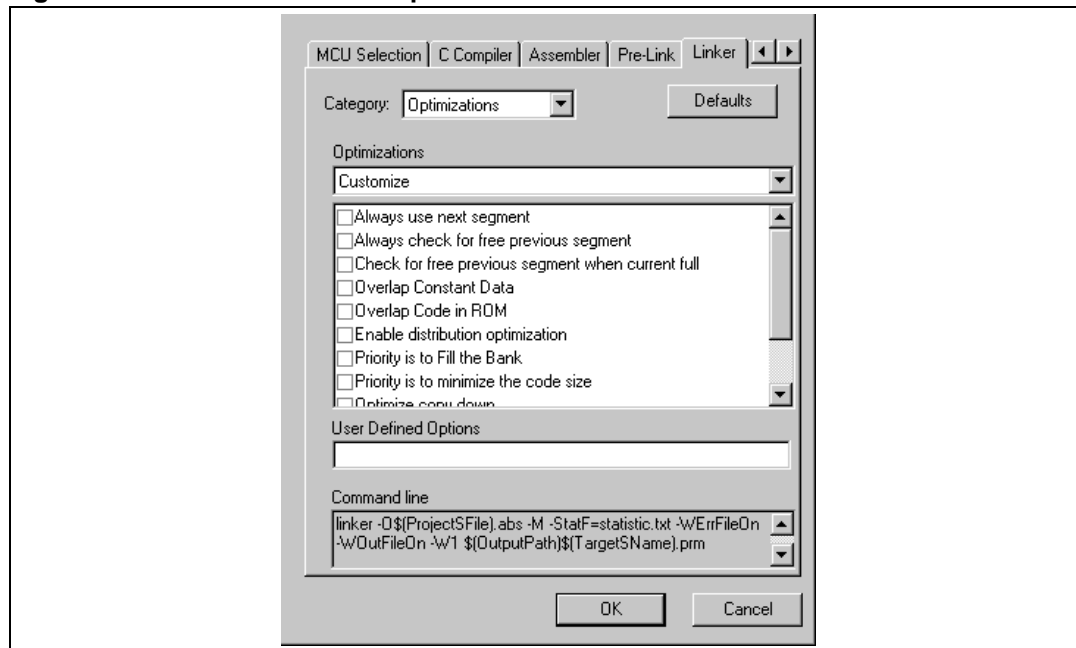
As with Object/Lib Modules, you can type the library path name or use the browse button to enter the objects and libraries to include. When entering more than one path, the entries are separated by a semicolon (;). When you add a library via this field, the -L option followed by the path name are added to the command line.

Note: The Metrowerks Linker does not accept spaces in path names.

Customizing Metrowerks linker optimization settings

To further customize optimizations, select **Customize** from the **Optimizations** list box. The view in the **Linker** tab changes to that shown in [Figure 100](#).

Figure 100. Metrowerks linker optimizations view



In this view, you can choose from the optimization options listed in [Table 54](#).

Table 54. Metrowerks linker optimization options

Option ⁽¹⁾	Description
Always use next segment (-AllocNext)	Used in generation of ELF files only. Method that allocates objects in the order presented.
Always check for free Previous Segment (-AllocFirst)	Used in generation of ELF files only. Method that checks if a partially filled segment will accommodate the object. Allocation does not respect the order of objects.
Check for free Previous Segment when Current Full (-AllocChange)	Used in generation of ELF files only. Method that allocates an object to the first available segment that is large enough to accommodate it.
Overlap Constant Data (-COCC=D)	Used in generation of ELF files only. This option sets the default for optimization to data.
Overlap Code in ROM (-COCC=C)	Used in generation of ELF files only. This option sets the default for optimization to code.
Enable Distribution Optimization (-Dist)	Used in generation of ELF files only. Causes the linker to generate a file with the optimized distribution.
Priority is to fill the bank (-DistOptiFillBanks)	Used in generation of ELF files only. Minimizes the free space in every bank.

Table 54. Metrowerks linker optimization options (continued)

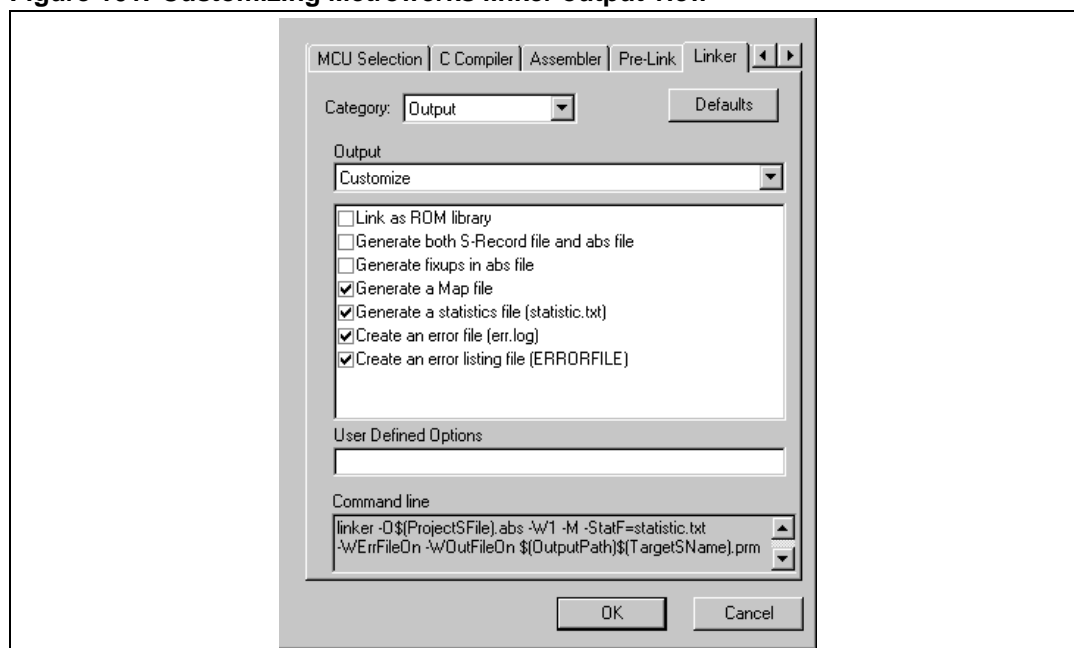
Option ⁽¹⁾	Description
Priority is to minimize Code Size (-DistOptiCodeSize)	Used in generation of ELF files only. Prioritizes optimum code size over fill optimization.
Optimize Copy Down (-OCopyon)	Used in generation of ELF files only. This optimization changes the copy down structure when space is limited.
Hiware: Allocate not referenced overlap variables (-CAllocUnusedOverlap)	This option is used to ensure that all defined objects are allocated when Smart Linking is switched off.
Hiware: Allocate not specified const segments in RAM (-CRam)	Constant data segments that are not specifically allocated to ROM are allocated to RAM.

1. For complete descriptions of optimization options, refer to the **SmartLinker user manual** from Metrowerks.

Customizing Metrowerks linker output settings

To further customize the linker output, select **Customize** in the **Output** list box. The view in the **Linker** tab changes to that shown in [Figure 101](#).

Figure 101. Customizing Metrowerks linker output view



In this view, you can apply the options listed in [Table 55](#).

Table 55. Metrowerks linker output options

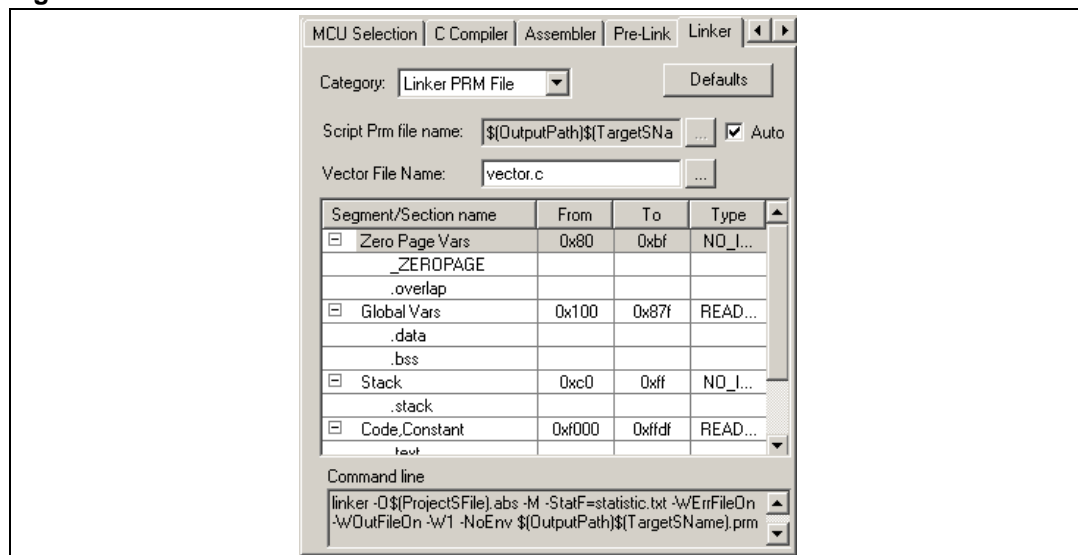
Option ⁽¹⁾	Description
Link as ROM Library (-AsROMlib)	The output objects are burned to ROM memory. Using this option is equivalent to specifying "AS ROM_LIB" in the parameter file.
Generate both S record and ABS file (-B -EnvSRECORD=s19)	Results in the creation of an ABS file and an S record. By default the name for the S record is <filename>.sx.
Generate Fixups in ABS file (-SFixups)	Used in generation of ELF files only. This option is not compatible with debugging.
Generate Map file (-M)	Generates a map file at the end of a successful link session. This option is the equivalent to using MAPFILE ALL in the parameter file.
Generate a Statistics file (-StatF=statistic.txt)	Generates listing of attributes of each allocated object in text file. The default file name is statistic.txt.
Create an error file (err.log) (-WErrFileOn)	Creates an error log called err.log.
Create an Error Listing file (ERRORFILE) (-WOutFileOn)	Creates an error listing file called ERRORFILE.

1. For complete descriptions of optimization options, refer to the SmartLinker user manual from Metrowerks.

Configuring the Metrowerks linker PRM file

The **Linker PRM** file tab provides an interface for selecting the linker's parameter file (PRM) or defining one automatically. Access this view by selecting **Linker PRM File** in the **Category** list box. The view will change to that shown in [Figure 102](#).

Figure 102. Metrowerks linker PRM view



From this interface you can identify a parameter file to use during linking or use the **Auto** option to generate a parameter file according to the settings in the mapping table.

Note: If you specified to output in Hiware format in the **Compiler** or **Assembler** tabs, you must specify a PRM script that is written for this output format in the **Script Prm file name** field. You cannot use the **Auto** option.

When the **Auto** option is not checked, you have access to the **Script Prm file name** field. All other fields are grayed, as they pertain only to automatic generation of the parameter file. In the **Script Prm file name** field, identify the parameter file to use either by typing the path name or using the browse button to locate it.

If you are using your own PRM file (**Auto** option is not checked) you should remove the **LINK command** from the PRM file before building your application. The `-O` option in the **Linker command line** is redundant with the LINK command in a PRM file. This results in a warning during the linking of the application. For this reason, the LINK command is not added to PRM files that are automatically generated by STVD.

For more information about creating parameter files manually, refer to the *Metrowerks SmartLinker User Manual*.

Automatic segment declaration and section assignment

When the **Auto** checkbox is not checked, the Linker relies on script file (*.prm) that you write for information about memory segments and the code sections that are assigned to them.

When the **Auto** checkbox is checked, a configurable mapping is displayed in the Linker tab. The mapping's first level elements are segments in the target microcontroller's memory. Clicking on the "+" sign for a segment allows you to see the code sections that are assigned to the segment. This mapping is used to automatically generate a linker script file (default name *TargetSName.prm*) that contains the declarations of memory segments and the code sections assigned to each.

The automatically generated linker script file is not write protected and you can modify this file in any editor. However, it is regenerated each time you build the application when the **Auto** feature is enabled. If you make changes to the file and **Auto** is enabled, the changes will be overwritten when STVD updates the file based on the mapping in the Linker tab.

To protect your modifications, rename the file. You can then use this file to link your application in the same way that you would use a link script file that you wrote yourself (see [Configuring the Metrowerks linker PRM file on page 153](#)).

Note: Using the ELF/DWARF 2.0 format to build your application will allow you to use both the **Auto PRM file generation** option, or your previous PRM file. If you choose to use the previous PRM file, note that it must correspond to the syntax used with the ELF/DWARF 2.0 format, notably the ELF syntax uses **SECTION** in place of **SEGMENT**. Also ensure that the variables for the **start07** routine are placed in a **No.Init** section.

If you have disabled the automatic generation of the parameter file, it will not be updated if you make changes to your project, such as removing a source file. If you make such a change you will have to edit the parameter file manually.

Modify the segment mapping and section assignments

You can add memory segments and delete or rename the segments that you have created. Default segments cannot be deleted or renamed. However, you can modify their start and end addresses.

You can also modify the memory type for any segment. Right-click on the Type cell for any segment. The pull down menu in the Type cells of the mapping allows you to choose from:

- **READ_ONLY** - ROM memory that cannot be modified during execution.
- **READ_WRITE** - Any memory that can be read and modified during execution.
- **NO_INIT** - Specifies any memory that is not to be initialized upon startup.

*Note: When you apply your memory settings STVD will determine the compatibility of your entries during build. Any incompatibilities will appear as errors in the **Build** tab of the **Output** window.*

Right-clicking on a cell in the mapping opens a contextual menu with the following commands:

- **Add Segment** - Adds a row of empty fields to the mapping. Enter the name of the new segment and press the Enter key. If you do not enter a segment name, but click on enter, the new segment is removed from the mapping. You cannot use the same segment name twice. Naming is case sensitive.
- **Add Section** - Adds a new code section to the mapping. The interface only allows you to enter the name of the section (not an address range for it). You cannot use the same section name twice in the mapping. Upon link, the code sections are assigned to the memory segment (address range) in the order in which they appear in the mapping.
- **Change** - Allows you to change the value or entry in the selected cell of the mapping. Address values can be entered in decimal or hexadecimal (use "0x_____" notation) formats. You can change the names of the default memory segments, however you cannot change the names of the default code sections.
- **Delete** - Deletes the selected entry. You cannot delete the default memory segments or code sections.
- **Add Segment** - Adds a row of empty fields to the mapping. Enter the name of the new segment and press the Enter key. If you do not enter a segment name, but click on enter, the new segment is removed from the mapping. You cannot use the same segment name twice. Naming is case sensitive.
- **Add Section** - Adds a new code section to the mapping. If you do not enter a section name, but click on enter, the new section is removed from the mapping. You cannot use the same section name twice. Naming is case sensitive. The interface only allows you to enter the name of the section (not an address range for it). Upon link, the code sections are assigned to the memory segment in the order in which they appear in the mapping.

Vector file name field

This field allows you to enter the path name for the file that contains the vector definitions for your application. You can either type the file's path name or use the browse button to locate it. STVD will not allow you to continue if this field is empty.

4.10 Configuring folder and file settings

STVD's folder and file settings allow you to configure a subset of project settings that are specific to certain folders and files in your project. Some files, such as header files in the dependencies folder do not have configurable settings.

When the **Project Settings** window is open, access the settings for a file or folder by selecting it in the directory tree on the left side of the window. The view will change to show you the tabs that are available for the file or folder.

If the **Project Settings** window is closed, access settings for a file or folder by right-clicking on it to get the **Contextual** menu. Then select **Settings**, and the **Project Settings** window will open with the tabs that are available for the file or folder you have selected.

From this window, you can configure the settings that are specific to the selected file or folder. Only those tabs containing settings that apply to the selected file/folder are displayed. The settings you make here, override the settings made at the project level for the selected file or folder. This section provides information about settings that are specific to folders and files, including:

- [General settings for files and folders](#)
- [Custom build tab](#)
- [Build commands](#)

Otherwise, these tabs contain the same settings and options that are available at the project level. For detailed descriptions of these tabs and their options refer to the Project Settings section for your toolset. The appropriate sections are summarized in [Table 56](#).

Table 56. Toolchain specific tabs

Toolset	Settings tab	Available for...
ST Assembler/Linker	ST ASM tab	folders, source files
Cosmic C	Cosmic C compiler tab	folders, C source files
	Cosmic C Assembler tab	folders, ASM source files
Raisonance	Raisonance C compiler tab	folders, C source files
	Raisonance Assembler tab	folders, ASM source files
Metrowerks	Metrowerks C Compiler tab	folders, C source files
	Metrowerks Assembler tab	folders, ASM source files

Note: No settings tabs are available for the Dependencies folder.

4.10.1 General settings for files and folders

The **General** tab is common to all toolsets and all files or folders.

If no settings can be configured for the selected file or folder, the **Project Settings** window will open to the **General** tab, which will contain the message **"No settings."**

When no options are checked, the tabs that are specific to your toolset and the selected file or folder will be visible (see [Table 56](#)).

When settings are available you can configure the following:

- **Always use custom build step:** Checking this box allows you to access the **Custom Build** tab, where you can enter your own commands and options for building this file (see [Section 4.10.2: Custom build tab on page 157](#)).
- **Exclude file from build:** Checking this box will exclude this file or folder when you build your project. When you check this box, all other tabs disappear. Only the **General** tab is available.
- **Intermediate directory:** This field allows you to specify the path name for the folder where intermediate files from the compiler or assembler will be stored. If you do not specify an intermediate directory to use for a file or folder, then intermediate files are stored in your project's **Output Directory** by default (see [Section 4.5.1: General settings tab on page 85](#)). You can type the path name of a directory you have created, or use the browse button to locate it.

Note: Specifying an intermediate directory may be necessary if you have two or more source files in different directories with the same name. In this case, the default mechanism for naming intermediate files results in identical filenames. When output to the same default directory, the last intermediate file generated will replace any previously generated file with the same name.

4.10.2 Custom build tab

The custom build feature allows you to specify your own commands (based on your toolset) to build your application. Custom build commands override any standard settings that you may have applied using the standard project settings interface. Only those custom commands which you have specified are performed during the build.

This file provides the following features:

Description field: Here you can enter a description that will appear in the **Build** tab of the **Output** window, to help you identify when your custom build command is being executed and view any related messages or errors.

Commands field: Here you enter the commands to be executed during the build using the syntax and options specific to your toolset.

Directory button: Click on this button for a list of commonly used macros to help you identify directories and path name to use during the build. Select the macro from the list and it is automatically added at the cursor's position in the **Commands** field.

Files button: Click on this button for a list of commonly used macros to help define filenames and extensions to use during the build. Select the macro from the list and it is automatically added at the cursor's position in the **Commands** field.

Dependencies button: Click on this button to open the **User Defined Dependencies** window. In this window, you can specify the dependencies for custom building of the selected source file. To add files as dependencies, click on the file icon and type the path name, or browse to find the file. Change the order of the files in the list by using the up and down arrow buttons. Once you have entered the dependencies click on **OK**.

Note: This option is only available in the **Custom Build** tab for files, not folders.

4.11 Specifying dependencies between projects

Because a project may use an element or object generated during the build of another project, STVD provides a **Dependencies** window that allows you to identify projects that are the source of dependencies for other projects. To define dependencies between two projects, both must be included in the workspace.

To define project dependencies:

1. Select **Project > Dependencies**.
The **Project Dependencies** window opens.
2. Click in the **Select project to modify** field, to view the drop-down list of all other projects in the workspace.
3. Select the project that you want to make dependent from the drop-down list.
The view in the **Dependent on the following project(s)** field will change to show all of the other projects in the workspace that it can depend on.
4. Check the box next to any projects that it depends on. Then click on **OK** to confirm the setting.

STVD will not allow you to make projects mutually dependent. For example, if Project1 is dependent on Project2, you cannot make Project2 dependent on Project1. To avoid this, Project1 does not appear in the **Dependent on the following project(s)** field for Project2, as it is already dependent on Project2.

You can also define dependencies for specific files. This requires using the **Custom Build** option in the file-level **Settings** window. For more information, refer to [Section 4.10: Configuring folder and file settings on page 156](#).

4.12 Build commands

From the **Build** menu you can choose from commands that allow you to build your entire application, build parts of your application or just compile its source files.

You can only access the build commands after you have set up a project, and when you are in the Build context. In the Debug context (**Debug > Start Debugging**), these commands are not available. Therefore, if you make changes to your application source files or project settings during a debugging session, you must end your debugging session (**Debug > Stop Debugging**) to access the build commands and rebuild your application.

When you are ready to build your project, ensure that the project is set to active. To do this, select **Project > Set Active Project** and then click on the project that you want to make active. The checkmark is placed next to the project you select and the project name in the **Workspace** window changes to bold type.

To build a project or compile selected files, go to the **Build** menu. Here you have access to the commands listed in [Table 57](#).

Table 57. Build commands

Command	Description
Compile	Compiles the C or Assembler source file that is in the active Editor window. You can compile a file even though the project it belongs to is not set to active. To compile a file, open the file in the Editor window. If it is already open, click on the Editor window that contains the file you want compile to ensure that it is active. The name of the selected source file will appear next to the Compile command. Click on Compile .
Build	Starts the incremental build of the active project with the current configuration or settings.
Rebuild	Cleans (removes intermediate files) and rebuilds the active project with the current configuration or settings.
Batch build	Builds, cleans or rebuilds all the selected projects in a workspace for the selected configuration(s). To do a batch build, select <i>Build > Batch Build</i> to open the Batch Build window. This window displays all the projects in a workspace and the applicable configurations. Select the projects that you want to build and the configuration to use by clicking on the appropriate checkbox. Once you have selected the projects, you can Build, Clean, or Rebuild them by clicking on the appropriate button.
Clean	Removes all intermediate files that have been generated by previous builds of the active project.

Once you have built your application for the selected microcontroller you are ready to select your debug instrument and start debugging your application, or program your application to your target microcontroller.

5 Basic debugging features

Once you have built your application for debugging, you can choose your debugging instrument and enter the **Debug context**. In this context, STVD provides access to the debugging features that are supported by your debugging hardware or instrument. Because debugging instruments support different features, the commands and views that are available during debugging will vary based on your hardware configuration. For this reason, you must select your debug instrument before you can start debugging. For instructions, see [Section 5.1: Selecting the debug instrument](#).

Once you have selected your debug instrument, you are ready to start debugging. The debugging features described in this section are available for all hardware configurations: Simulator, DVP2, DVP3, EMU2B (HDS2), ICD and EMU3 emulators and **STice**. The following sections tell you about:

- [Section 5.1: Selecting the debug instrument](#)
- [Section 5.2: Configuring your target MCU](#)
- [Section 5.3: Running an application](#)
- [Section 5.4: Editor debug actions](#)
- [Section 5.5: Disassembly window](#)
- [Section 5.6: Online assembler](#)
- [Section 5.7: Memory window](#)
- [Section 5.8: Instruction breakpoints](#)
- [Section 5.9: Data breakpoints](#)
- [Section 5.12: Watch window](#)
- [Section 5.10: Call stack window](#)
- [Section 5.11: Local variables window](#)
- [Section 5.13: Core registers window](#)
- [Section 5.14: MSCI tools window](#)
- [Section 5.15: Symbols browser](#)
- [Section 5.16: Peripheral registers window](#)
- [Section 5.17: Memory trace window](#)
- [Section 5.18: Online commands](#)
- [Section 5.19: Limitations and discrepancies](#)

For other features that are specific to your debug instrument refer to:

- [Section 6: Simulator features on page 202](#) for software simulation without a debug instrument
- [Section 7: In-circuit debugging on page 223](#) for EMU3 with ICC add-on
- [Section 8: DVP and EMU2 \(HDS2\) emulator features on page 237](#) for all DVP and EMU2 emulators
- [Section 10: EMU3 emulator features on page 284](#) — for EMU3 emulators
- [Section 9: STice features on page 258](#) — for **STice** advanced emulation systems

5.1 Selecting the debug instrument

Your choice of target microcontroller and debugging hardware determine what features you will have access to when debugging. In the **Debug Instrument Settings** window you can:

- [Identify your debug instrument](#) and the port it is connected to
- and [Add communication ports](#)

5.1.1 Identify your debug instrument

To identify your debug instrument for STVD:

1. Select **Debug Instrument>Target Settings** from the main menu bar.

Figure 103. Identify debug instrument and connection



2. Select the appropriate option for your debugging hardware in the **Debug Instrument Selection** list box. [Table 58](#) describes the available options.

Table 58. Debug instrument types

Selection in STVD	Applicable development tools
DVP2	If you have an ST7-DVP2 series emulator
DVP3	If you have an ST7-DVP3 series emulator
EMU3	If you have an ST7-EMU3 series emulator
EMU2B (HDS2)	If you have an ST7-EMU2 (also known as HDS2) series emulator.
ICD (ST Micro Connect or DVP3)	If you are using an ST7-ICD (STMC) debugger, the EMU3 emulator with ICC add-on, or the DVP3 emulator for in-circuit debugging.
ICD RLINK	If you are using the Raisonance RLink for in-circuit debugging over ICC.
ICD STICK	If you are using the ST7 STICK for in-circuit debugging over ICC.
SIMULATOR	If you are not using any debugging hardware.

Table 58. Debug instrument types (continued)

Selection in STVD	Applicable development tools
STice	If you are using the STice as a plain emulator.
SWIM RLink	If you are using the Raisonance RLink for in-circuit debugging over SWIM.
SWIM STice	If you are using the STice advanced emulation system in ICD configuration.

- If the selected debug instrument is **SWIM RLink** or **SWIM STice**, then you can optionally enable the **Hot Plug Start Debug** checkbox to start a debug session in hot plug mode.
Note that this option is disabled when you start a debug session from an open workspace in STVD.
When you start a debug session in hot plug mode, a connection to the MCU is established while it is running, allowing you to see the state of the memory. For more information on debugging in hot plug mode, refer to [Section 7.4: In-circuit debugging in hot plug mode \(SWIM only\) on page 233](#).
- In the **Target Port Selection** list box, select the communication port (parallel, USB or Ethernet) that your debugging hardware is connected to.
The option **usb://usb** is the automatic USB search option, whereas **usb://hti1**, **hti2**, ..., **hti128**, are used to indicate a specific USB port. Ethernet connections are listed by the IP address for the device. If the connection that you want to use is not in the list, you can add it by clicking on the **Add** button. (see [Section 5.1.2: Add communication ports](#)).

Note: When Simulator (SIM) is selected as the target, **Port Selection** is not available because Simulator is a software debugging tool.

- Click on **OK**, to confirm your settings and close the **Debug Instrument Settings** window.

5.1.2 Add communication ports

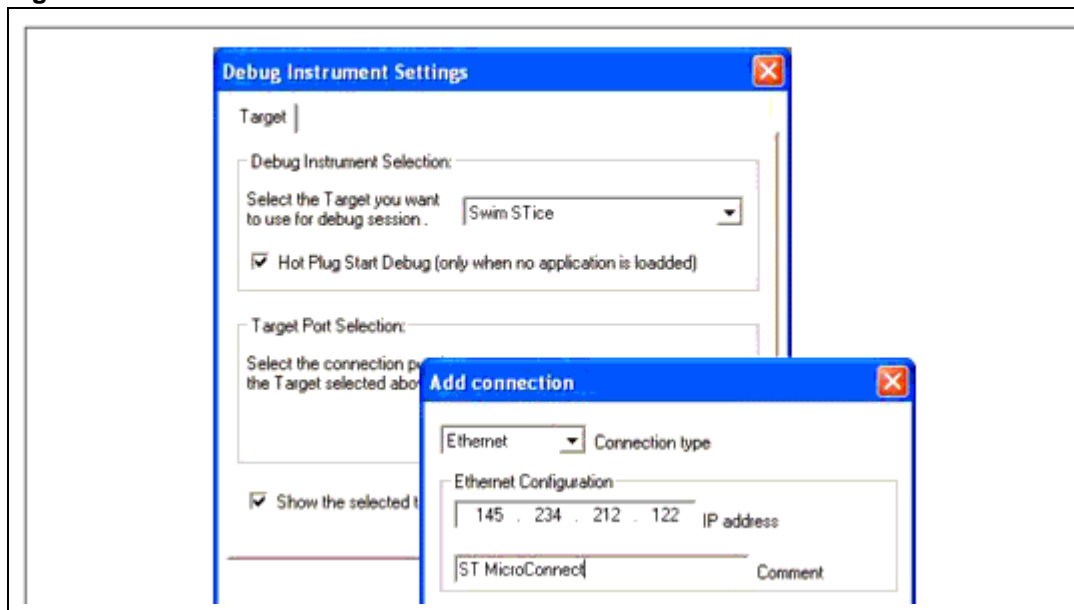
To access the window for adding hardware connections:

- Select **Debug Instrument>Target Settings**, then click on the **Add** button in the **Debug Instrument Settings** window.
This will open the **Add Connection** dialog box where you can specify USB or Ethernet connections to add to the list of options in the **Debug Instrument Settings** window.
- Select the type of connection (USB or Ethernet) that you would like to add from the **Connection Type** list box.
- Enter the necessary information in the appropriate fields.

For Ethernet connections:

- Type the IP address (four sets of three numbers) that is assigned to your debug instrument in the **IP Address** field (see [Figure 104](#)). Information about assigning an IP address to your debug instrument is provided in the Connectivity section of your hardware user manual for the debug instruments that support Ethernet connections.
- Type a description (up to 40 characters) in the **Comments** field.

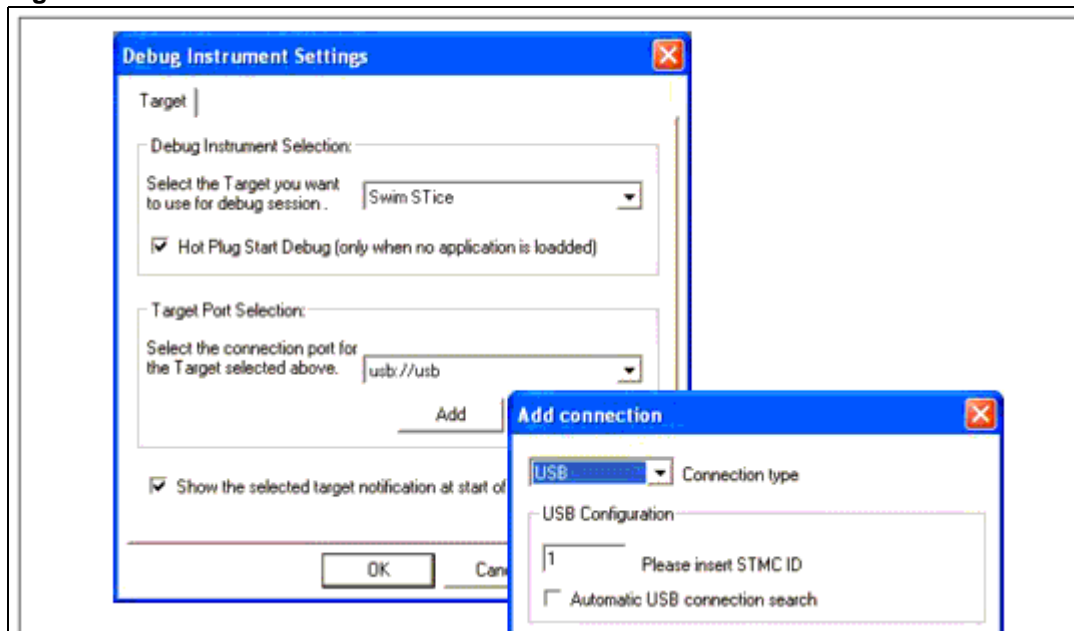
Figure 104. Add an Ethernet connection



For USB connections, either:

- Enter the number for the USB port (an integer between 1 and 128) in the **Please Insert STMC ID** field, to create an option that identifies a specific USB port (see [Figure 105](#)),
- or place a checkmark in the **Automatic USB Connection Search** checkbox to create a connection option that automatically searches all USB ports for your debug instrument (usb://usb).

Figure 105. Add a USB connection



4. Once you have entered the appropriate information, click on **OK**.

This returns you to the **Debug Instrument Settings** window. The new connection option has been added to those in the **Target Port Selection** list box and the new option is selected automatically.

5.2 Configuring your target MCU

From STVD, you can specify the configuration of your microcontroller's memory and the behavior of its peripherals. The **MCU Configuration** window provides interfaces for configuring:

- [Memory map](#)
- [On-chip peripherals](#)

The options presented in the MCU Configuration interface are specific to your microcontroller. For more information about your microcontroller's memory map and supported peripheral features, refer to your microcontroller's datasheet.

5.2.1 Memory map

The default memory settings depend on the microcontroller selected. However, you can configure the memory settings as you want if your application requires non-default settings.

This feature enables you, for example, to temporarily decrease the RAM zone, and increase the size of the ROM (to exceed what is available on the real microcontroller) during the first stages of development. Once your program is functional, you can start to optimize its size by reducing your code and returning these zones to their original size. There are two different actions you can perform on the memory configuration:

- change the type of an entire existing zone
- define a new zone of any type wherever possible

Depending on your target MCU, the available memory types may be: Peripherals, RAM, ROM, Stack, System, EEPROM, Reserved, Vectors, Application, NEM. Some of these zones can have their type and size modified, others cannot be modified. Their definitions and properties are explained in [Table 59](#).

Table 59. Memory types

Type	Description
Peripherals	Microcontroller internal or rebuilt peripherals registers. Their properties are defined as in the microcontroller datasheet. This memory cannot be modified.
RAM	Random-Access-Memory of the microcontroller. This memory type can be modified.
ROM	Read-Only Memory of the microcontroller. Write protected. This memory type can be modified.
Stack	Stack of the microcontroller. This memory type cannot be modified.
System	The emulator uses this space for emulation management. This memory type cannot be modified.
EEPROM	This memory is internal to the microcontroller and is located inside the emulation device. The programming of this zone is done according to an automaton found in the datasheet. This memory type cannot be modified.

Table 59. Memory types (continued)

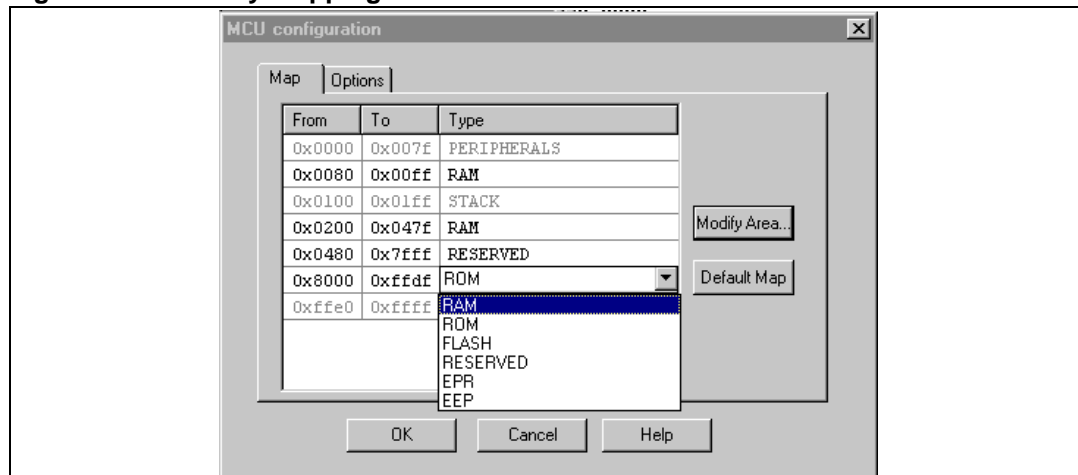
Type	Description
Reserved	This memory zone is reserved as on the microcontroller. It is not allocated to any use and is write protected. This memory type can be modified.
Vectors	This memory zone contains the user interrupt vectors zone. It is write protected. This memory type cannot be modified.
Application	This memory type is microcontroller-specific. You can add memory or peripheral resources on its hardware. It is not available on every emulator. Properties are linked to the user hardware. This memory type can be modified.
NEM (Non-Existent Memory)	This is a memory zone that is inaccessible or masked from use. This memory type can be modified.

For most target MCUs, you may modify the following types of memory zone: **RAM**, **ROM**, **Reserved** and **Application**.

Modifying an entire memory zone

You can modify an entire zone of memory from its default type. Memory zones which cannot be modified are grayed and not accessible.

Figure 106. Memory mapping



The left and center columns of the **Memory Configuration** tab indicate the address range of each memory zone. The right column indicates the memory type of each zone.

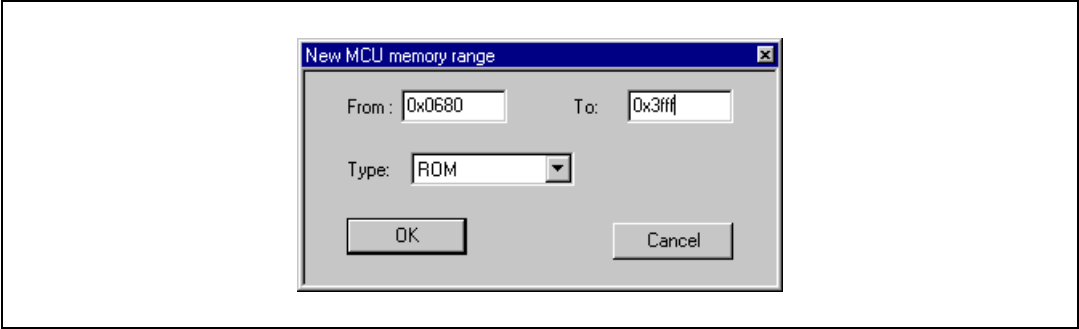
Defining new zones in the memory mapping

1. To designate a new zone in the memory mapping, click on **Modify Area** in the **MCU Configuration** window.
2. In the New MCU Memory Range pop-up, enter the address for the new memory zone, and its type, then click on OK.

The new zone and its type will appear in the map displayed in the MCU Configuration window.

You can restore the original mapping by clicking on **Default**.

Figure 107. Specifying a new memory zone

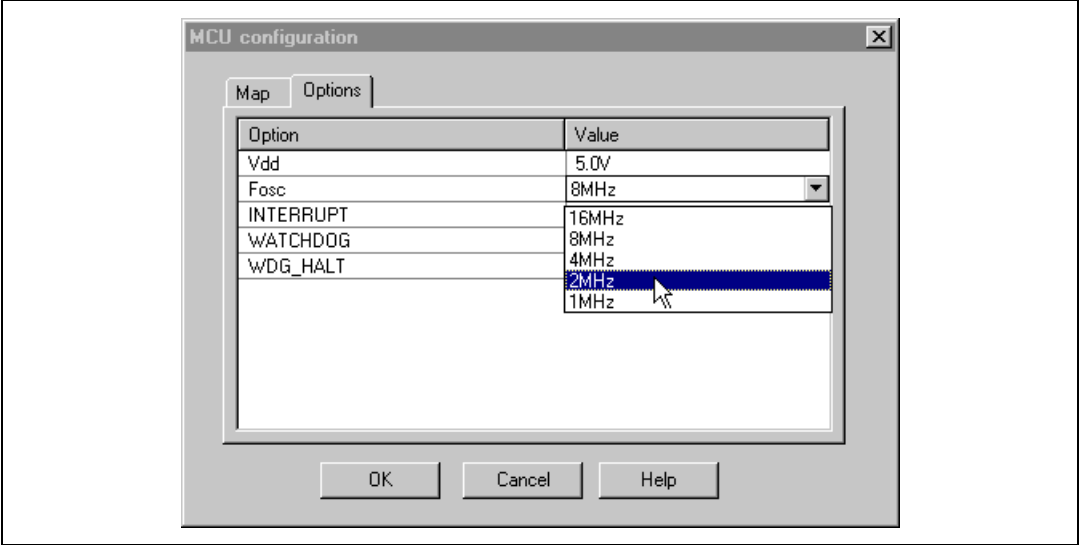


5.2.2 On-chip peripherals

The **Options** tab of the MCU Configuration window, allows you to configure your target device's options and on-chip peripherals. All of the configurable options for your target microcontroller are listed in the **Options** column. Depending on the microcontroller selected, the configurable options and the default settings displayed in the **Option** column will change.

Beside each *option*, a default *value* is displayed. To change the value for an option, double-click in the **Value** field associated with the option. Then select the new value from the pull down list.

Figure 108. On-chip peripherals



- Note:** For more information about the configurable options available on your target hardware device, refer to your microcontroller datasheet.
- Caution:** When in-circuit debugging, if you have the wrong option settings, your microcontroller might fail to start up. For more information, refer to [Section 7.1.4: Configuring option byte settings on page 226](#) under [Section 7: In-circuit debugging on page 223](#).

5.3 Running an application

An application loaded into the STVD may be run using the commands available in the **Debug** menu. In the debugging process, commands in the **Edit** menu and the Editor's contextual menu are also frequently used.

Typically, the source code for the loaded program will be viewed in an Editor window during debugging. Breakpoints may be set and removed, and run commands issued from within the Editor window.

For this reason, most commands associated with the debugging process are available in the **Edit** menu and/or the Editor's contextual menu (to obtain the contextual menu, right-click anywhere within an Editor window). Use of the Editor during debugging is detailed in [Section 5.4: Editor debug actions on page 169](#).

This section provides information about running your application in the debug context, including:

- [Run commands](#)
- [Stepping modes](#)
- [Program and STVD status bar display](#)
- [Monitoring execution in source windows](#)

5.3.1 Run commands

The commands available in the **Debug** menu and on the **Debug** toolbar are listed in [Table 60](#).

Table 60. Debug menu run commands

Command	Description
Start Debugging	Start a debugging session. In order to start a debug session, you must have built your application into an executable, specified a debug instrument that supports the application's target MCU and specified the connection port for the debug instrument.
Stop Debugging	End the debugging session. You cannot build your application or change the target MCU while a debugging session is open.
Go to PC (Program Counter)	The program information displayed in the STVD windows is modified to show the program environment and values as they should be at the current source code statement pointed to by the PC. This command may be used to retrieve the current source code statements after having scrolled in the source file, or after changing source file viewing.
Run	Resets the microcontroller and executes the application.
Restart	Resets the PC to first instruction of the <i>main</i> function for applications in C language. For applications in Assembly, the PC is set to the first instruction of the function labelled <i>main</i> , if one exists.
Chip Reset	Resets the microcontroller. The PC is reset to the beginning of the startup routine.
Continue	Continues execution of the application from the position at which it is halted. When execution has stopped at a breakpoint, it will be continued from the following instruction (as if no break occurred).
Stop	Stops program execution.
Step Into, Step Over, Step Out. These three commands permit enhanced control of single-stepping. Each provides a different single-step operation.	

Table 60. Debug menu run commands (continued)

Command	Description
Step Over:	To avoid single-stepping through functions that are not relevant to the debug operation. When a function call occurs, the command Step Over will cause the program to execute the function, update the program data displayed by STVD and stop at the next instruction line in the main program. This avoids having to single-step line by line through a function call which is not of current interest (for example already debugged).
Step Into:	This command single-steps each time it is employed, so at a function call it will single step line by line through the function which has been called. It is equivalent to, and replaces, a Single Step command.
Step Out:	When used inside a function, the Step Out command runs the executable to the point of return from the function call and then updates program data. In all three cases, the program data is updated to reflect the new PC location after execution of the step command. This corresponds to the values which would exist at that PC location if the program were running normally.
Run to Cursor	This command runs the program from the current Program Counter location to the executable code corresponding to the cursor's current position in the program text. If a breakpoint is encountered before the cursor position, the execution stops at the breakpoint, but if restarted will not stop at the cursor position unless this option is again selected.
Set PC	Set a new value for the Program Counter: move PC from its current location to a new location. This is done without running the program (unlike the Run to Cursor command, described above). Debug actions can then be carried out with the new PC value.

The commands **Go To PC**, **Run to Cursor** and **Set PC** are also available in the Editor contextual menu for ease of access when using the Editor during debugging.

5.3.2 Stepping modes

For the Step Over and Step Into commands, you can choose one of two modes of execution. To change the modes, select **Debug Instrument>Stepping Mode**, the resulting dialog box will allow you to select either:

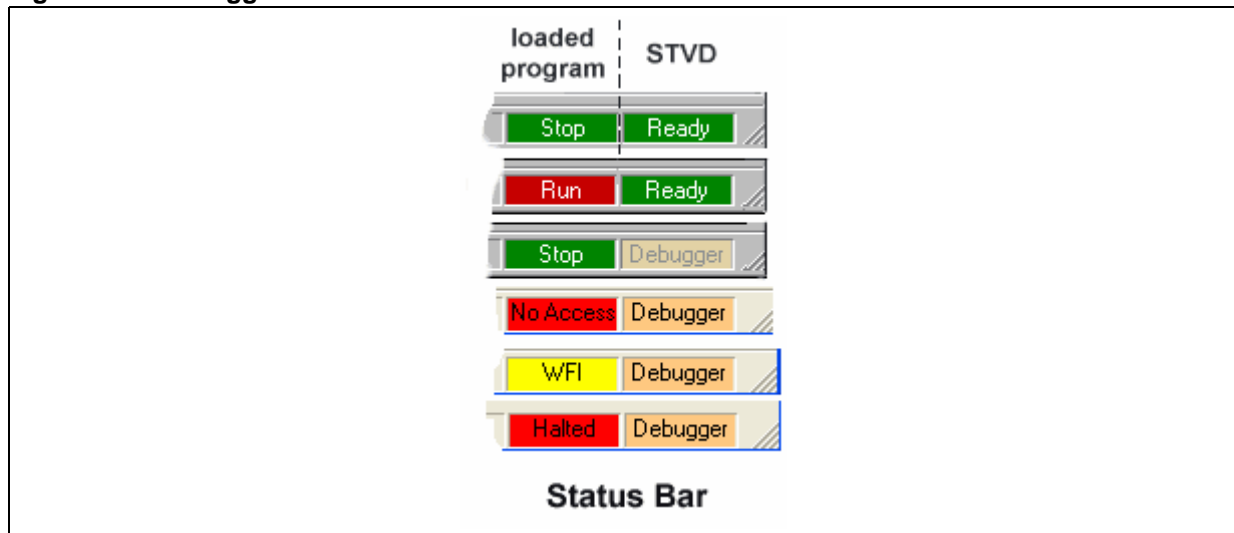
- **Enter Interrupts When Stepping Into:** This is the default mode. In this mode, STVD uses the debug instrument's primitive single step to execute one assembler step at a time. When Stepping Into, STVD will stop at the entry of any interrupts that occur during the stepping process.
- **Don't Enter Interrupts When Stepping Into:** In this mode, to step through a single assembler instruction, STVD sets an instruction breakpoint on the next instruction to be executed if no interrupt occurs. It then runs the application up to this breakpoint. When Stepping Into, STVD will not stop at any interrupts that occur during this stepping process, unless you have placed a breakpoint there prior to executing the step command.

Limitations:

- Cannot handle an instruction that jumps to itself only once. In this case, the instruction is executed again and again. As a result stepping into or over such an instruction (Step Into ASM or Step Over ASM) does not work as expected.

5.3.3 Program and STVD status bar display

Figure 109. Debugger status on status bar



- The left panel of the status bar shows the state of the loaded program.
 - **Run/Stop** indicates whether the loaded program is running or has been stopped.
 - **Halted** the microcontroller is executing the "Halt" instruction.
 - **WFI** the microcontroller is executing the "WFI" instruction.
 - **No Access** (SWIM only) the microcontroller is either executing "Halt", "WFI", "WFE" instructions or protected code. In this state no core resource is accessible for the debugger.
- The right panel of the status bar shows the state of the STVD debugger software.
 - **Ready** indicates the Core Debugger is available and will accept all user commands.
 - **Debugger** indicates the Core Debugger is working. Only the user command **Stop Debug** is accepted by the Core Debugger when working.

5.3.4 Monitoring execution in source windows

Program execution is followed in the *source windows*—meaning the Editor window(s) and the Disassembly window. Whenever execution stops (for example at a breakpoint or after a step command has been completed) the Editor and Disassembly windows are updated automatically so that the new Program Counter (PC) location is visible.

Other STVD windows are also automatically updated to show the current values of variables or register values for example.

5.4 Editor debug actions

To complement its [Editing features](#) the Editor's debug actions provide a rapid graphic method for controlling application execution and viewing the parameters resulting from execution during the debugging phase of application development. Changes in variable values can be traced with ease and related directly to their originating lines in the Assembler or C-language source files.

For debugging, the Editor's active window may contain either a C-language or an Assembly language source file that constitutes a part of the application (and containing debug information).

Debug icons are placed in the [Editor debug margin](#). When the application is run, the Program Counter (PC) indicator is visible in the Editor window, indicating the current program line. You can set breakpoints at selected lines to halt program execution wherever you want.

Run options are accessed from the Debug menu. Program controls include a selection of run and single step actions. The results of these actions, for example the Program Counter position, the real values of program variables at that location, the Stack contents and other debug data, are available in the Editor and the various view windows.

In addition, a Call Stack Frame indicator may be displayed relative to the Program Counter (PC) indicator. The values of program variables at the position marked by the Call Stack Frame indicator are available, whatever the location of the PC.

Editor contextual menu

In addition to editing features, the Editor contextual menu (available by right-clicking with the mouse in the Editor window) gives you direct access to the debugging features described in [Table 61](#).

Table 61. Editor contextual menu commands

Command	Description
Insert Read/Change Data Breakpoint	Refer to Section 5.9: Data breakpoints on page 186 .
QuickWatch	Refer to Section 5.4.2: QuickWatch window on page 174 .
Go to...	Move to the line specified.
Go to Disassembly	Move to the corresponding line in the disassembly window. Refer to Section 5.5: Disassembly window on page 175 .
Go to Next Line With Code	Reposition the text cursor on the next line containing program code or data.
Go to Definition Of “ ”	When the cursor is on a symbol representing a program element, this command moves the cursor to the location in the code where the symbol is defined.
Go to PC (Program Counter)	The program information displayed in the STVD windows is modified to show the program environment and values as they should be for the PC located at the current source code statement. This command may be used to redisplay the current program values at the PC location, after other values have been shown as a result of using one of the other 'Go To' commands.
Insert/Remove Breakpoint, Enable/Disable Breakpoint	Allow you to insert breakpoints, and enable/disable them. These commands affect the line of code where the cursor is located regardless of the cursor's position in the line.

Table 61. Editor contextual menu commands (continued)

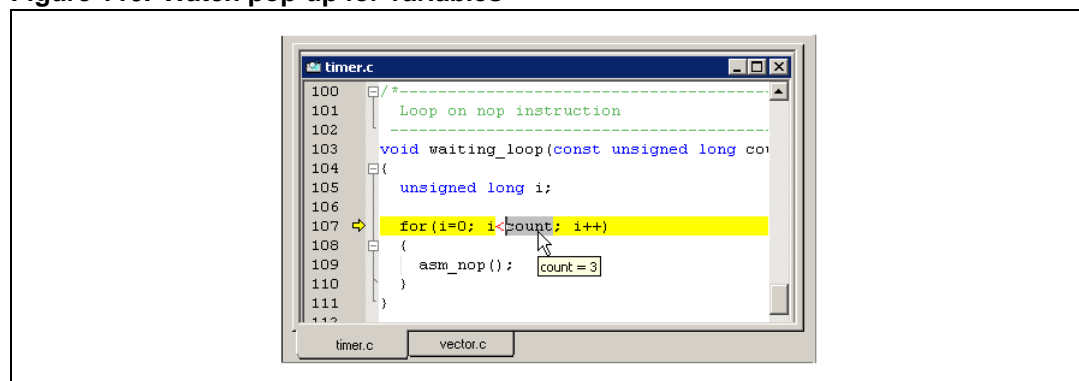
Command	Description
Run to cursor	Runs the program from the current Program Counter location to the executable code corresponding to the cursor's current position in the program text. If a breakpoint is encountered before the cursor position, the execution stops at the breakpoint.
Set next PC (Program Counter)	When the cursor is positioned on a source-code statement in an Editor window, or on a line in the <i>Disassembly window</i> , the command <i>Set Next PC</i> moves the Program Counter from its current location to this new location. This is done without running the program (unlike the <i>Run to Cursor</i> command, described above). Debug actions can then be carried out with the new PC value.

Watch pop-up

When a variable is selected in the active Editor window (the mouse pointer paused on a variable identifier-string delimited by white space or other usual separators, or on any highlighted variable identifier), a tool-tip banner is activated, displaying the current numeric value of the variable at this program location, if known.

This feature can be enabled/disabled from in the **Edit/Debug** tab of the **Options** window (accessible by selecting **Tools>Options** from the main menu).

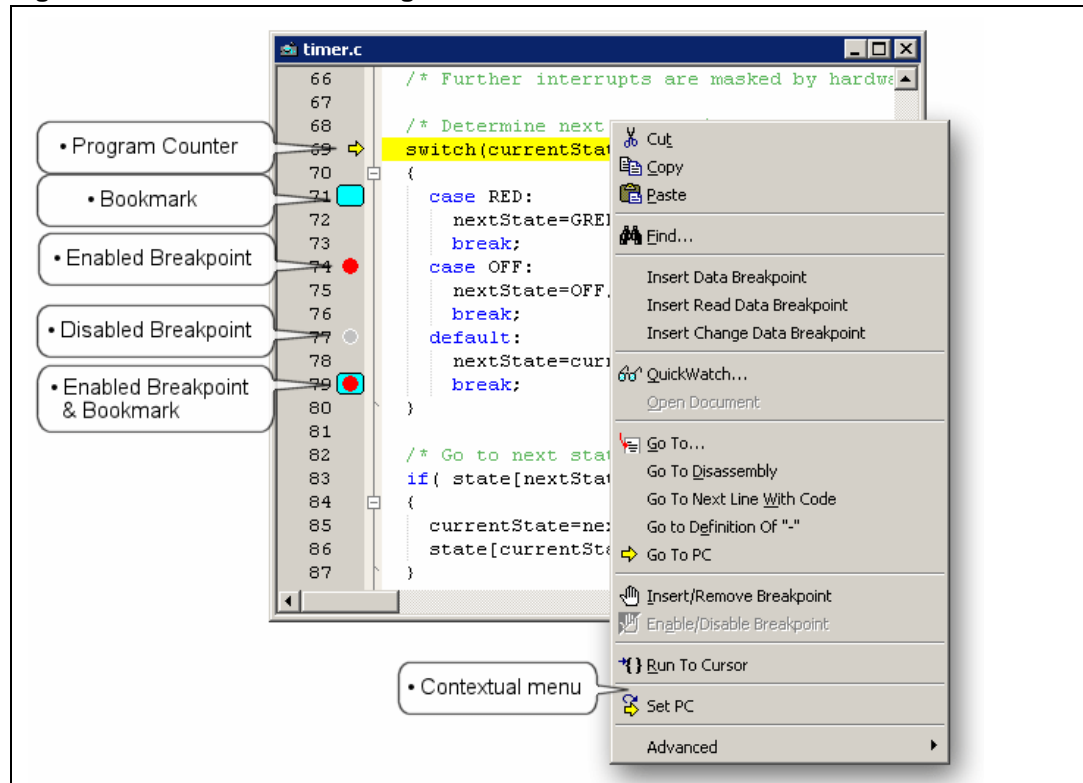
Figure 110. Watch pop-up for variables



5.4.1 Editor debug margin

The left margin of an Editor window is a special area for placing debugging icons.

Figure 111. Editor window margin icons and contextual menu



The illustration shows several margin icons in place (breakpoints—active and disabled, bookmarks and the Program Counter) and shows the Editor's contextual menu open (right-click on mouse to open the contextual menu).

The **Program Counter** location is visible, showing the line at which the program has halted. **Bookmarks** are enabled/disabled in the Edit menu (**Edit>Toggle Bookmark**). Instruction Breakpoints are inserted/disabled/removed in the Editor contextual menu or in the Edit menu (**Edit>Insert/Remove** or **Enable/Disable Breakpoint**).

Debug symbols in other windows

The Editor debug functionality is reciprocal. Breakpoint symbols placed in the Editor window also appear in a corresponding position in other windows.

The Program Counter indicator and the Call Stack indicator are also shown in the **Disassembler** window together with breakpoint symbols. The debug margin of the **Disassembler** window may be used to place breakpoints, which also appear in the corresponding Editor window.

Margin icons

Debugging icons are placed in the Editor's debug margin at the same level as the line of code to which they apply.

Note: A breakpoint and a bookmark may be placed together at the same position.


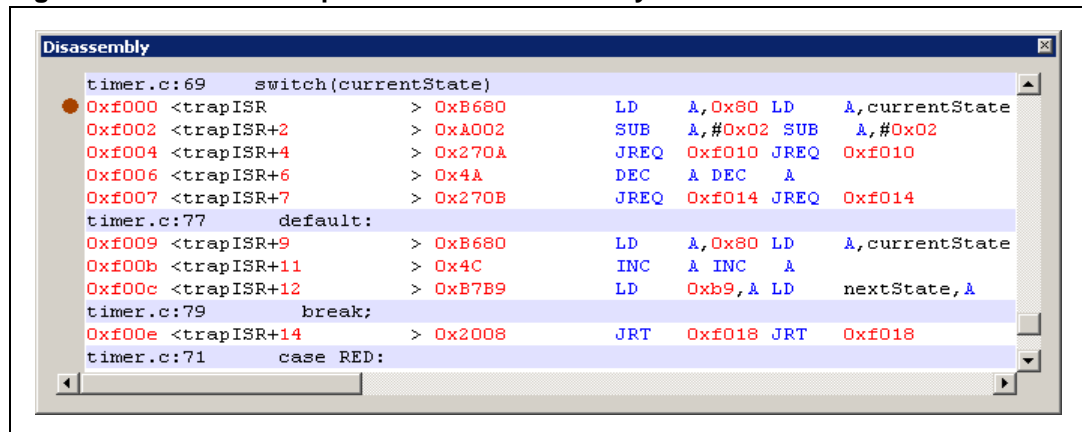
- **Breakpoint icon** : a round red dot symbol. Changes in a breakpoint icon's color indicate the following:
 - A **red** breakpoint dot means that breakpoint is **enabled**.
 - A **grey** breakpoint dot means that breakpoint is **disabled**.
 - A **brown** breakpoint dot (shown in [Figure 112](#)) means that STVD is uncertain about the position of the breakpoint. The position of the brown dot indicates where STVD believes the breakpoint is, but cannot verify its position. This situation can occur if the application you are debugging is missing debug information.

Figure 112. Brown breakpoint icon in Disassembly window



Breakpoints may be inserted, removed, enabled or disabled from the **Edit** menu or toolbar, or by simply clicking on the breakpoint itself—this toggles the breakpoint between the disabled state (dot becomes grey), removed, then back to the enabled state. The Editor's Contextual menu also permits these actions.

If you exit debug mode, the breakpoints you have placed remain visible in the margin of the Editor window. Editing your source code does not perturb the placement of the breakpoint—it follows the source code line at which it has been placed, even if you add or delete other source code lines before and after it.



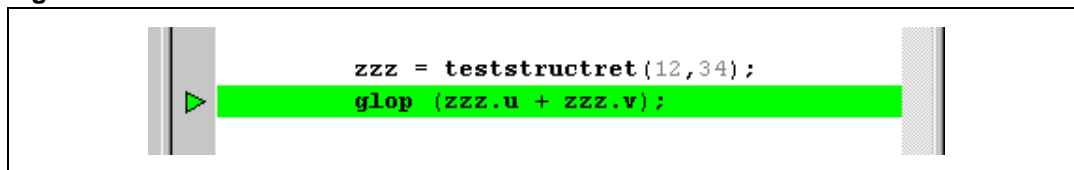
- **Program Counter** : yellow arrow and yellow highlighting. The Program Counter (PC) indicator automatically marks the current position of the program counter. It highlights the line in the source program which is the origin of the next machine instruction to be executed by the processor.
- When a program is run (using the run commands) and halts at an active Breakpoint, the program line corresponding to the Program Counter is highlighted in the Editor window. The program text is adjusted in the window so that the section containing the PC indicator is displayed.
- **Call Stack Frame Indicator** : a green triangular symbol and green highlighting. The **Call Stack** window lists the addresses contained in the Stack; the value of the Program Counter followed by the most recent Call value are at the top of the stack list. If a value in the Call Stack other than the PC is selected (by double-clicking on it), the Call Stack Frame indicator is set to this value. In the Editor window, the corresponding line of the source program is highlighted. (If the **Disassembly** window is open, the selected stack address is also highlighted.)

Figure 113. Call Stack frame indicator




This highlighted line is the source for the instruction found at the chosen call address (this means that the instruction pointed to by the value in the Call Stack, is indicated by the highlighted source line). The Program Counter indicator remains unchanged, as the program is still halted at the PC value.

- **Bookmark icon:** a blue rectangular symbol. Bookmarks are an Editor function provided as an aid to text viewing.

5.4.2 QuickWatch window

The QuickWatch window provides rapid access to the most commonly used Watch functions by displaying the current value of a variable selected in the Editor window. It may be opened in three ways:

- From the Editor contextual menu
- From the main menu, by selecting **Edit>QuickWatch**,
- By clicking on the QuickWatch icon  in the Edit toolbar.

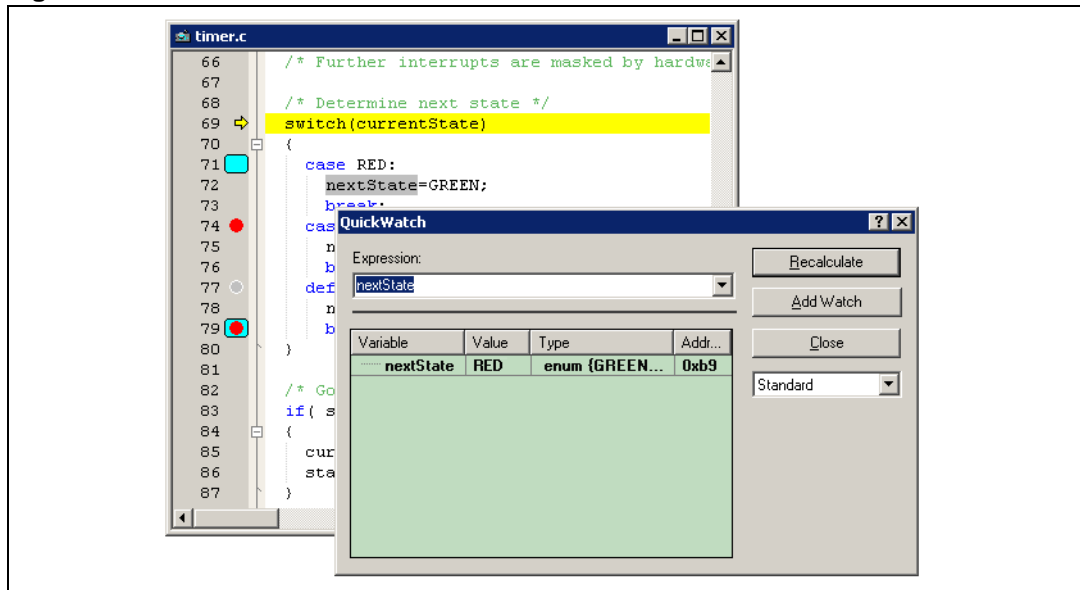
The QuickWatch window is opened in front of the active Editor window. If the text cursor is positioned within an expression string, or the expression highlighted, then the expression appears in the **Expression** field of the QuickWatch window. The main display area of the QuickWatch window contains a list of identifiers and their current values.

Commands in the window allow you to select the display base (**Hexadecimal**, **Decimal** or **Binary**), to recalculate the program values shown in the window, and to add selected variables to the list in the **Watch window**.

Alternatively, it searches for a corresponding variable name when a string is entered directly in the **Expression** field at the top of the QuickWatch window (shown below), and displays the variable name, its current value, variable type and address.

The '+' sign before a variable name in the left hand **Variable** column, indicates that the variable is expandable (see [Section 5.12: Watch window on page 191](#)).

Figure 114. QuickWatch window



QuickWatch functions are:

- **Recalculate:** Recalculates the value of the variable in the current context.
- **Add Watch:** This button adds the QuickWatch variable to the current page of the Watch window.
- **Close:** Closes the QuickWatch window.
- **Display selection menu:** The pull-down menu on the right of the window sets the display option; you can choose between **Standard**, **Decimal**, **Hexadecimal**, **Unsigned**, or **Binary** formats.

5.5 Disassembly window


To open the Disassembly window, either click on the Disassembly window icon  in the View toolbar or, from the main menu, select **View>Disassembly**. Figure 115 shows the program counter in the Disassembly window when execution has stopped at a breakpoint.

Figure 115. Disassembly window

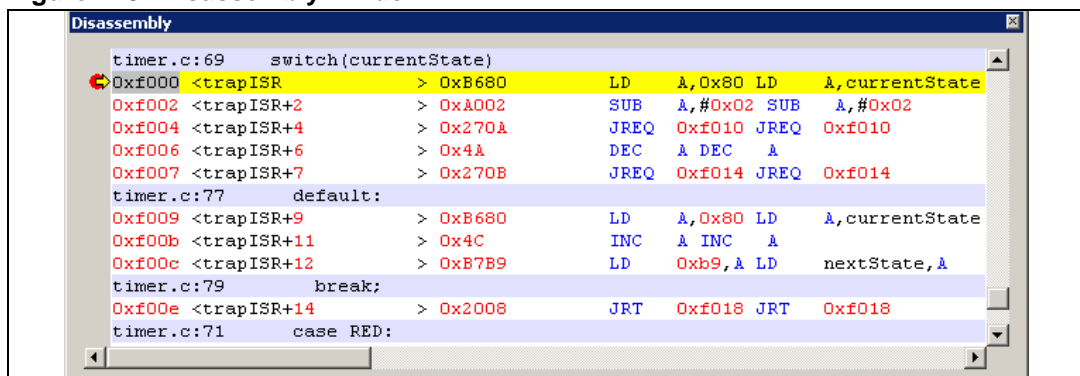
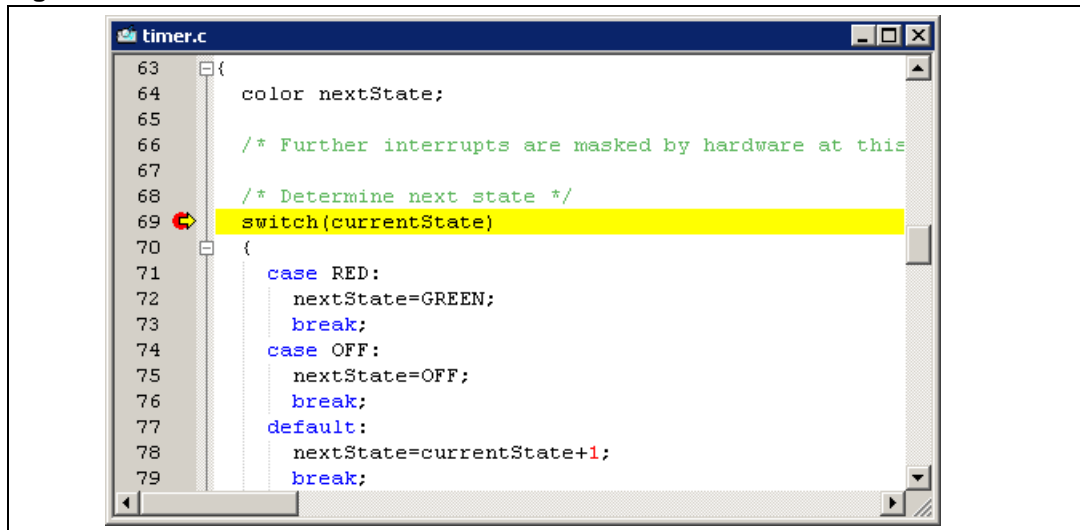


Figure 116 shows the corresponding source view (in the Editor window) for the same Disassembly window.

Figure 116. PC in editor window

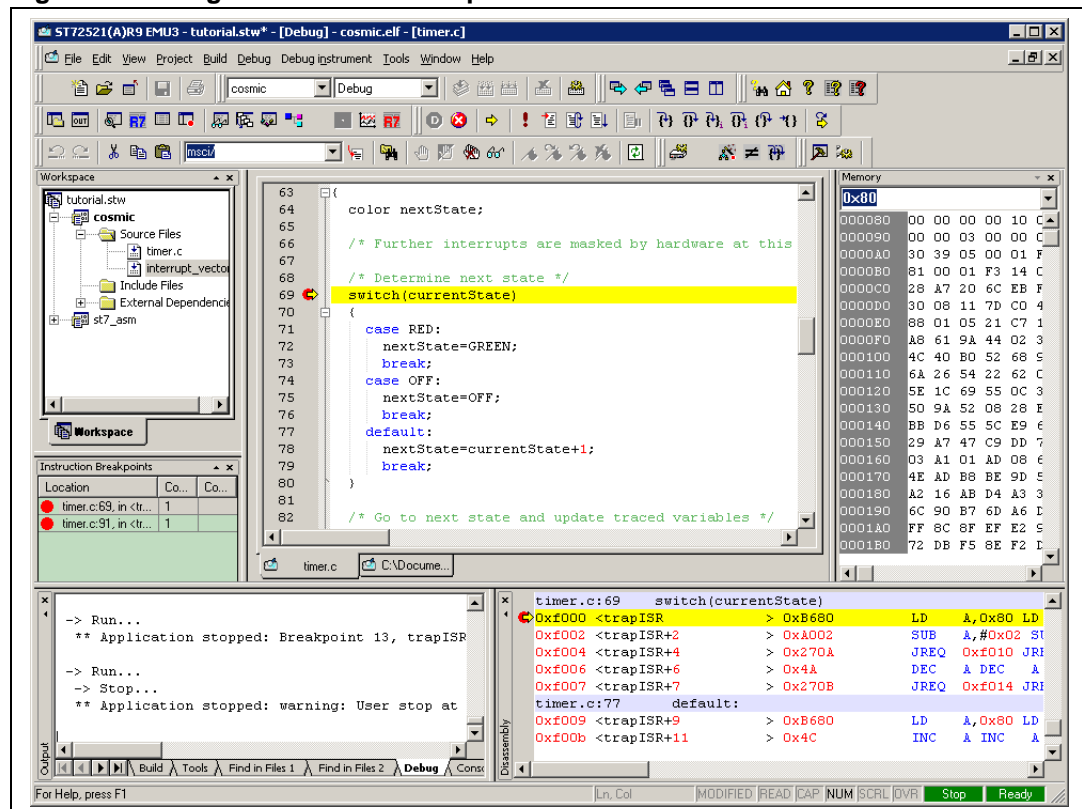


The Disassembly window provides a disassembly of the selected object file. Information given in the Disassembly window includes the **source filename** and **source line number**. These appear on a line by themselves in the Disassembly window, preceding the line of disassembled code.

The line of disassembled source code includes the **physical address**, the corresponding section of **source code**, the **value in memory** at that address, the **assembly mnemonic**, and the **operand(s)**.

For example, when a program halts at a breakpoint, the correct region of disassembled code appears in the Disassembly window as shown in [Figure 117](#).

Figure 117. Program halted at breakpoint

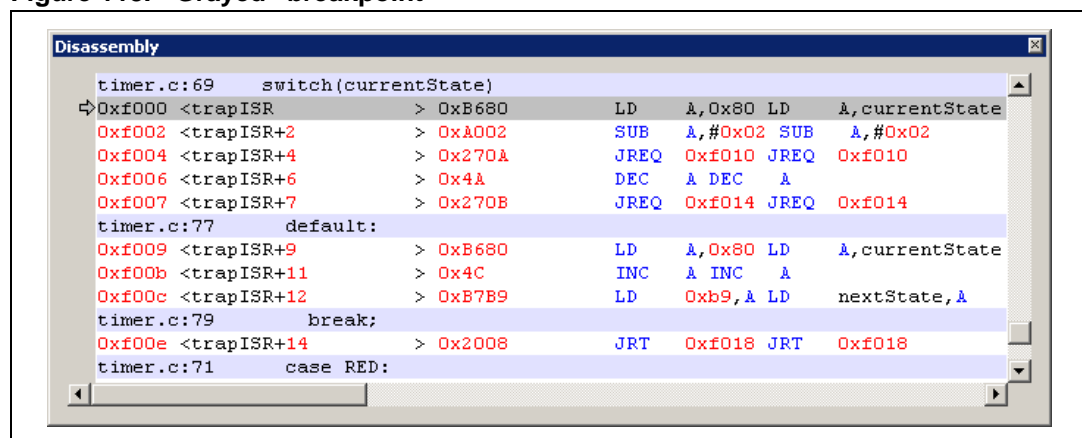


Note: The information appearing in the Disassembly window is the result of direct disassembly of the object file, therefore the location of the program counter in the Disassembly window is not necessarily a reflection of its location in the current Edit window program.

When a program halts at a breakpoint, the correct region of disassembled code appears in the Disassembly window, and the breakpoint appears at the beginning of the appropriate instruction.

However, if you manually modify the value of the PC (using the Set PC command or by changing the PC value in a Registers window), you may find that the PC symbol turns from yellow to grey, as shown in [Figure 118](#).

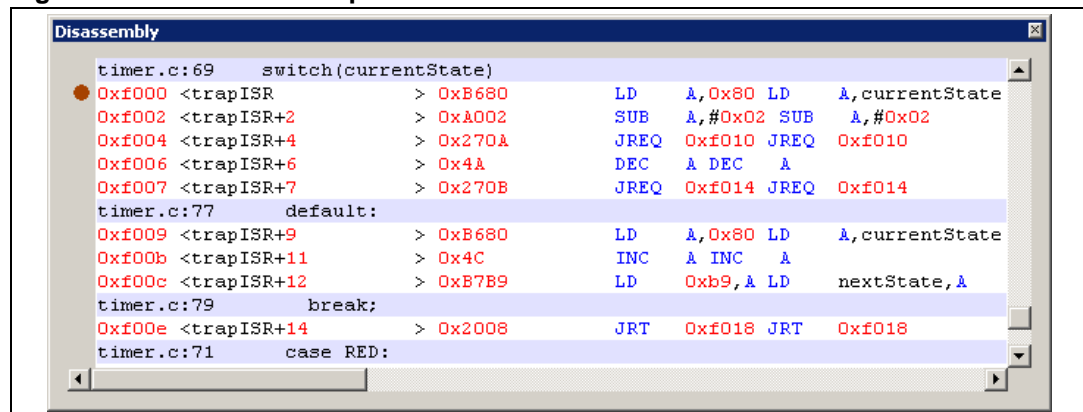
Figure 118. “Grayed” breakpoint



This simply means that the PC is placed in the middle of an instruction, rather than at the beginning of the instruction, and so warns you that the PC location is not exactly where it appears in the Disassembly window.

Similarly, a brown shaded circle, as shown below, indicates that a breakpoint has been placed, but that there is no way to associate this breakpoint with a particular instruction. This can occur if your project does not have any debug information related to it, such as when you load and work on a binary file directly.

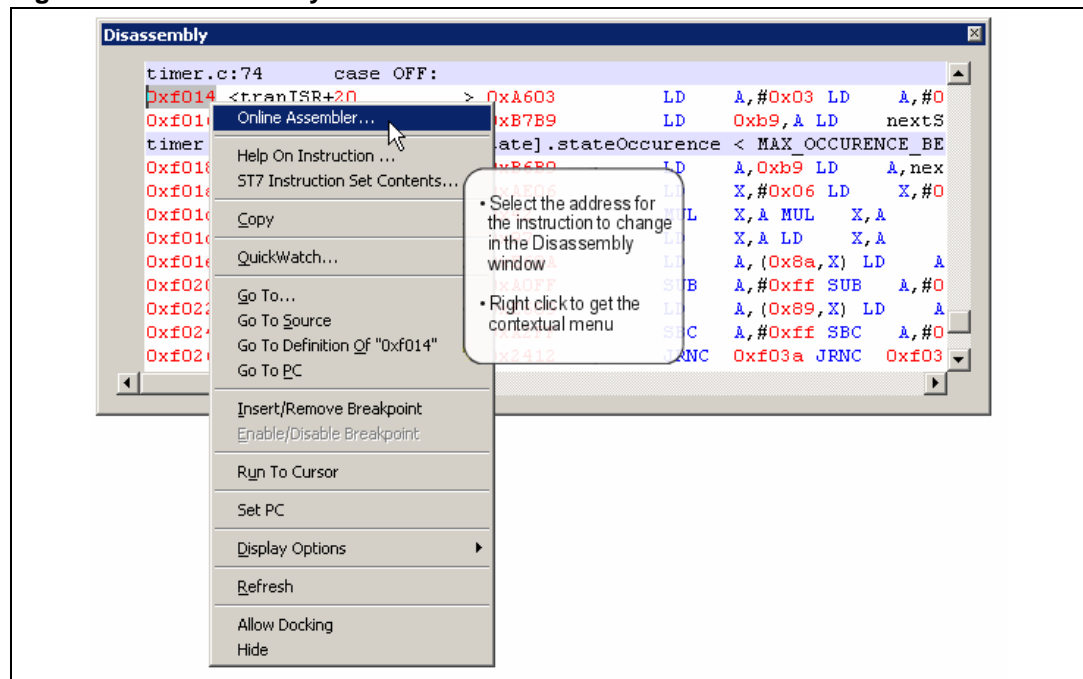
Figure 119. “Brown” breakpoint



5.6 Online assembler

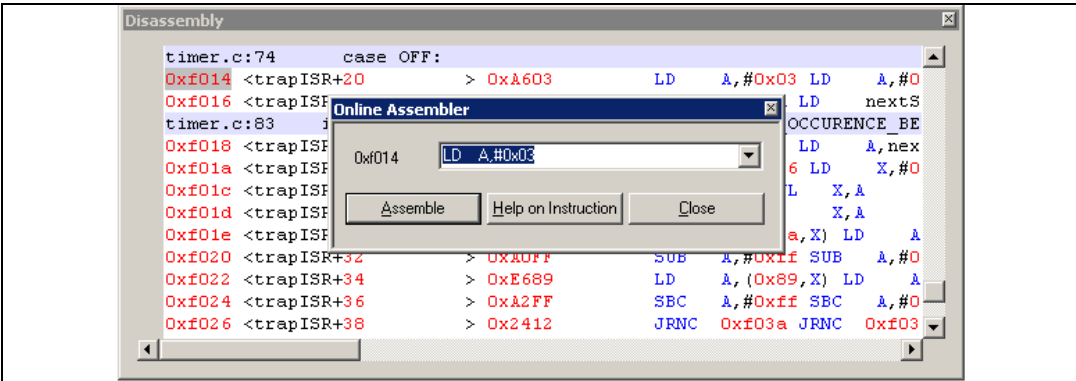
From the Disassembly window you can modify an instruction at a particular address by selecting the address and right-clicking the mouse. A contextual menu (as shown in [Figure 120](#)) will appear, with the option **Online Assembler**.

Figure 120. Disassembly window contextual menu



1. Select the address (so that it is highlighted in blue) at which you want to begin your assembler instruction. Right-click the mouse.
The contextual menu will open.
2. Select the **Online Assembler** option.
A dialog box allowing you to change the assembly instruction at the selected address will open as shown in [Figure 121](#).

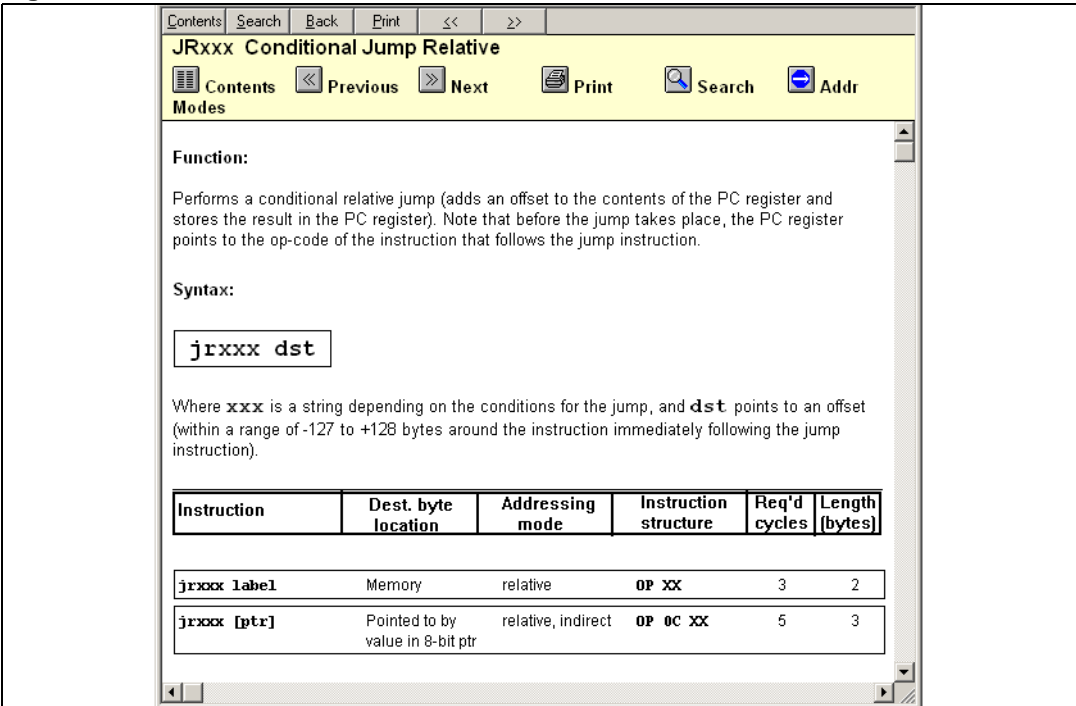
Figure 121. Online assembler dialog box



3. Type in the new assembly instruction you want for the selected address (in the example above, the selected address is 0xf014).
If you want information on an assembler instruction (for example, the correct syntax), click **Help on Instruction**. A contextual help screen will appear that explains the instruction typed in the field.

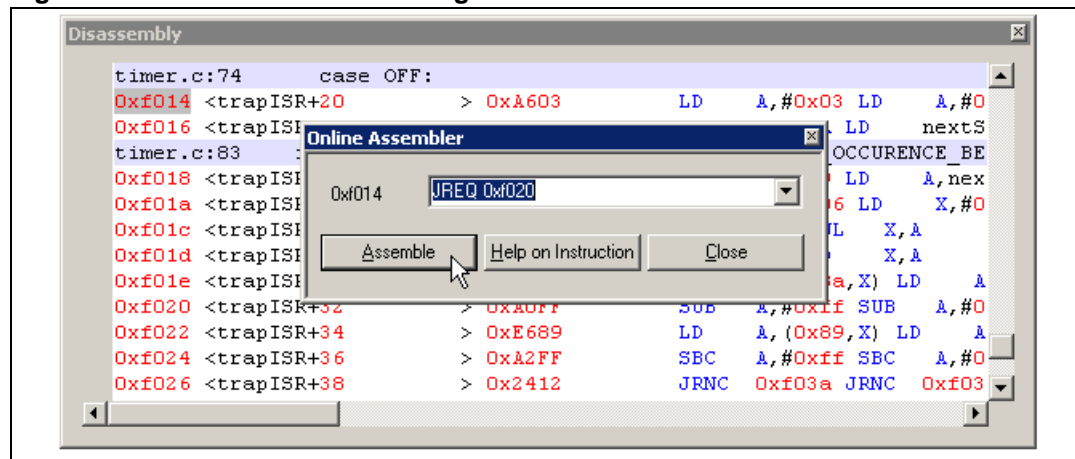
Note: You should follow the same assembly syntax rules that appear in the disassembly window. You may also use symbol names as operands.

Figure 122. Online ST7 instruction set



4. Click **Assemble**. The disassembly window will be refreshed as the new instruction is loaded into the memory, allowing you to modify the assembler instructions online while debugging. (Note that the application will not be permanently modified.)
The Online Assembler dialog box will then show the next address (0xf016, in our example) and by default enter the same instruction in the field.
5. You can continue modifying instructions, address by address, by entering the instruction you want and clicking **Assemble**.


Figure 123. Online assembler dialog box



Note: Because the current PC address is always highlighted in blue in the Disassembly window, you may be mistaken in thinking that you are modifying the instruction at the PC address. However the address for which you are changing the assembler instruction is always shown in the Online Assembler dialog box, so be sure to check that the address in the dialog box is the right one.

6. Click **Close** when you have finished.

5.7 Memory window

To open a **Memory** window, either click on the **Memory** window icon  in the View toolbar or from the main menu select **View>Memory**. Up to 4 Memory windows can be opened concurrently; functionalities and description are exactly the same for each window. This section provides information about:

- [Viewing memory contents](#)
- [Viewing features](#)

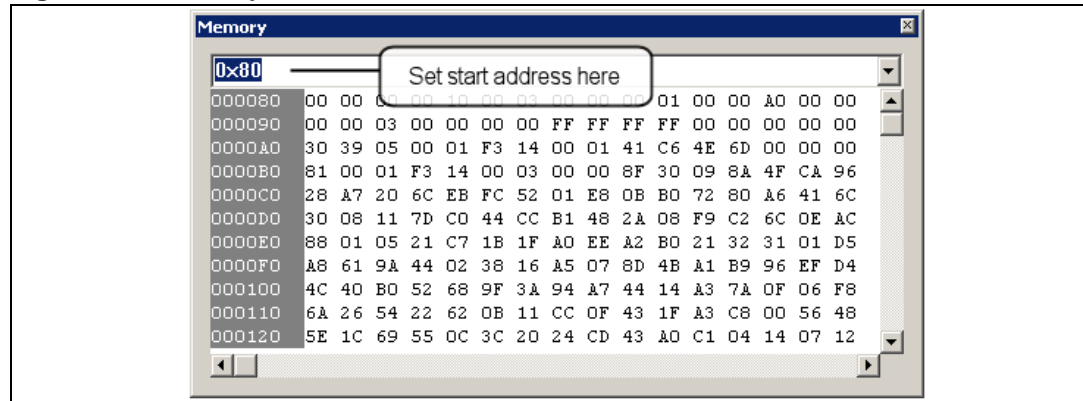
5.7.1 Viewing memory contents

To view an area of memory, enter the starting address into the field at the top of the window and press the **Return** key on your keyboard to display the memory locations starting at that address. An error message will appear if the address is not valid.

At the left of the memory window is displayed the starting address of each block of memory. A block of 16 bytes of memory is displayed as one line of data in the main window area. The content of each byte (16 bits) is represented as a two-digit hexadecimal value. To the right of the window, the same memory location contents are expressed as ASCII characters.

The search and edit options **Find**, **Find Next** and **Replace** may be used in the Memory window, to locate either hexadecimal or ASCII strings.

Figure 124. Memory window

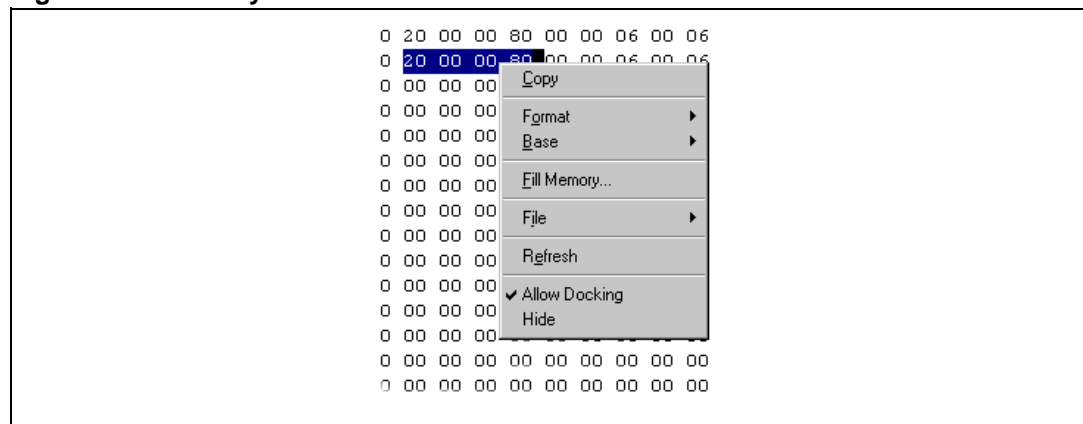


5.7.2 Viewing features

The contents of the Memory window can be refreshed, reformatted, reloaded or modified using the options in the Memory contextual menu. This menu also provides window configuration options (***Allow Docking, Hide, Float in Main Window***).

Right-click on either a selected memory area, or anywhere in the Memory window to open the Memory contextual menu shown in [Figure 125](#).

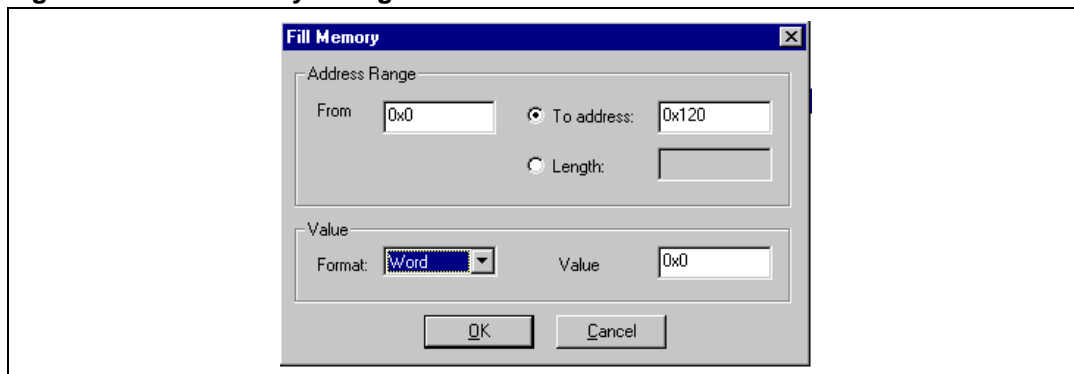
Figure 125. Memory contextual menu



The options available in this contextual menu are:

- **Copy:** Copy selected string to clipboard.
- **Format:** Arrange the memory display as **Byte**, **Word** or **DWord**.
- **Base:** display memory contents using the base **Hex**, **Decimal**, **Binary**, or **Unsigned** (unsigned decimal integer).
- **Refresh:** Reload current data values from the memory region displayed.
- **Fill memory:** Fill a specified region of memory with a specified value. The starting address and final address may be selected, or the starting address and block length. The data format Byte, Word or DWord must be specified, and affects the absolute block length. Display units are those selected in the **Format** option.

Figure 126. Fill Memory dialog box



Note:

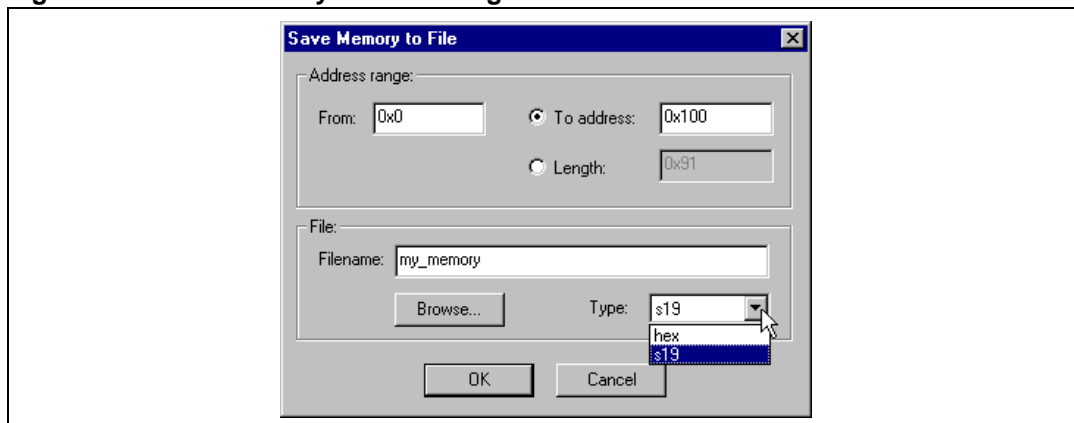
Memory may also be modified directly by typing over the values required. Care should be taken in this operation as the replacement values are written directly to memory.

- **File:** Allows access to two options: **Save layout** and **Restore layout**. Using these options, the contents of the Memory window can be saved to a file, or previously saved layouts can be restored from a file.

To save a memory layout, perform the following steps:

- a) Select **File>Save layout** from the contextual menu. The **Save Memory to File** dialog box opens as shown in Figure 127.

Figure 127. Save Memory to File dialog box

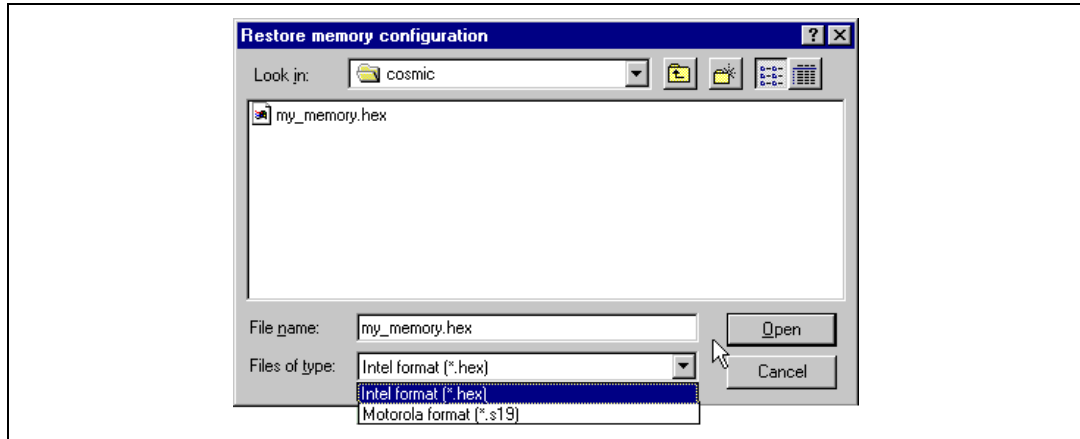


- b) Enter the memory **Address Range** that you want to save to file.
You can either specify the range between two addresses, or simply specify the starting address and the length of the memory range.
- c) Enter the name of the file and the type of the file that you want to save your memory layout to.
Memory contents can be saved as either *.s19 or *.hex files. Alternatively, you may browse for an existing file of either type (*.s19 or *.hex).
- d) Click **OK** to save your memory layout.

To **restore** a previously saved memory layout:

- a) Select **File>Restore layout** from the contextual menu. The **Restore memory configuration** dialog box opens as shown in [Figure 128](#).

Figure 128. Restore memory configuration dialog box



- b) Browse for the memory file that you want to restore. Click **Open**.

5.8 Instruction breakpoints

Instruction breakpoints are markers that stop program execution when a specific instruction is reached. Instruction breakpoints may be set on any C or assembler instruction. If a **condition** and/or a **counter** is set on an instruction breakpoint, program execution is always stopped to evaluate the condition/counter. If the evaluation returns the Boolean value FALSE, execution is resumed after a brief suspension. An instruction breakpoint halts the program *just before* the specified instruction is executed.

Note: When you impose conditions or a counter condition on an instruction breakpoint, the program will **not** run in real-time.

This section provides information about using instruction breakpoints, including:

- [Setting an instruction breakpoint](#)
- [Viewing the instruction breakpoints](#)
- [Setting counters and conditions](#)
- [Showing breakpoints](#)

5.8.1 Setting an instruction breakpoint

Instruction breakpoints are set in the Editor or Disassembler windows by selecting **File>Insert/Remove Breakpoint** from the main menu bar.

The Editor window displays source code and enables a breakpoint to be set on a chosen source-code line. The Disassembly window permits a breakpoint to be set on a line of disassembled code.

You can see the markers for the instruction breakpoint that you have set in the Instruction Breakpoints window (see [Section 5.8.2: Viewing the instruction breakpoints](#)) and in the Debug Margin of the Editor window (see [Section 5.4.1: Editor debug margin on page 172](#)).

5.8.2 Viewing the instruction breakpoints


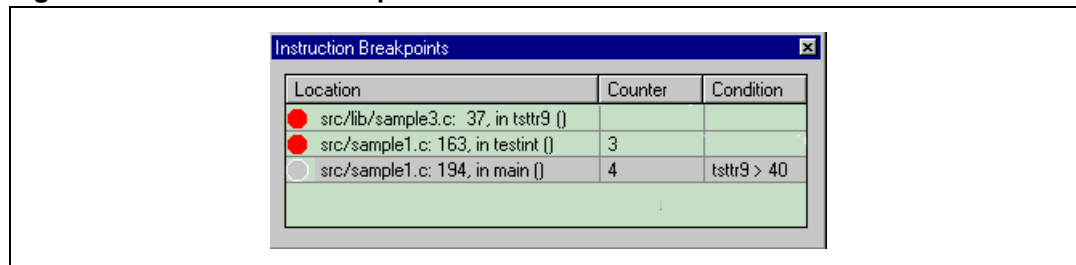
To open the Instruction Breakpoints window, either click on the Instruction Breakpoints window icon  in the View toolbar or from the main menu select **View>Instruction Breakpoints**.

Figure 129. Instruction breakpoints window



When a breakpoint is positioned, its specification appears in the Instruction Breakpoints window. From here, the breakpoint may be disabled/enabled or removed.

In addition, breakpoint counters and breakpoint conditions may be set in this window (double-click in the **Counter** or the **Condition** column to open a box for value entry).

The Instruction Breakpoints window contains three fields:

- **Location:** Information on the breakpoint location, plus an icon indicating the breakpoint is active/inactive.
- **Counter:** A counter may be set in this field, to be decremented each time the Breakpoint Condition is TRUE when the breakpoint position is passed during program execution. (If there is no condition, TRUE is returned at each pass of the breakpoint.) When the counter reaches zero, the program will stop at the **subsequent** pass:
 - With **no condition set, and counter = n**, the program will stop on the (n+1)th iteration.
 - With **condition set and counter = n**, the program will stop after the breakpoint is passed and its condition is found to be true *n* times.
- **Condition:** Breakpoint conditions may be set in this field. A condition is a C expression which returns a Boolean value. For example: `(i == 28) && (j >= 0xC0)`.

Note: **For the EMU2 emulator and DVP versions only:** When using the Counter and Condition features, be aware that STVD will stop the running of the application on the emulator each time it needs to evaluate the counter and/or the condition, which means that the application is not executing in real time, although it appears to be in Run mode. If the evaluation returns the Boolean value FALSE, execution is resumed after a brief suspension.

5.8.3 Setting counters and conditions

If a breakpoint depends on a counter and a condition together, the condition is first taken into account and secondly the counter. **The counter is only decremented when the condition is TRUE.**

For example, consider debugging the following function:

```
void f(int i) {x += i;}
```


If a breakpoint is set at the beginning of **function f**, such that:

- The breakpoint **condition** is set as **i<0**
- The breakpoint **counter** is set to **3**

The counter will **not** be decremented until the condition becomes TRUE.

The **first time** that **i<0** is **TRUE**, the counter will be decremented from 3 to 2

The **third time** that **i<0** is **TRUE**, the counter is decremented from 1 to 0, therefore the counter returns **TRUE** and the breakpoint is activated at the **following** iteration.

The value **Counter = 3** is taken to mean '**ignore condition =TRUE for 3 iterations**'.

In the example given, the executable will stop on the fourth pass in which the argument (**i**) is negative.

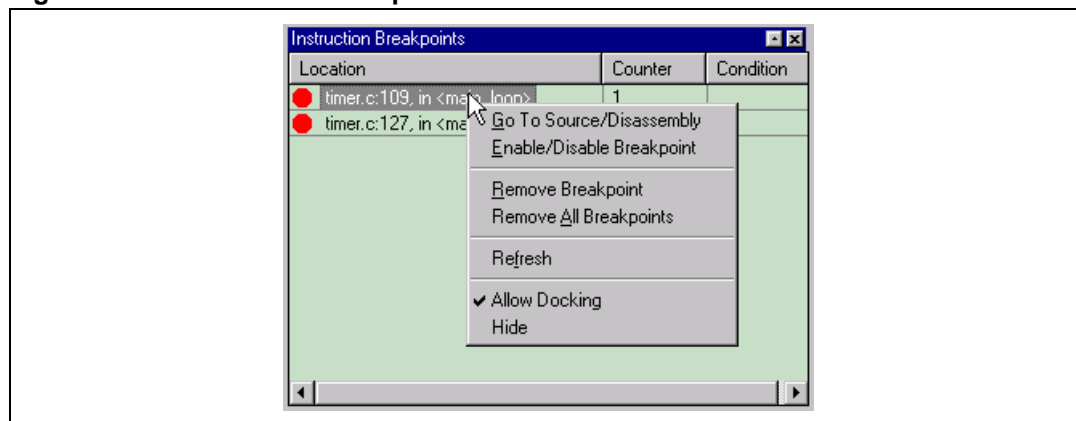
Argument values:

```
f(1);
f(2);
f(3);
f(4);
f(-1);
f(-2);
f(-3);
f(-4); *** first stop is here ***
f(-5)
```

5.8.4 Showing breakpoints

Right-click on an instruction breakpoint listed in this window to open the Instruction Breakpoint contextual menu.

Figure 130. Instructions breakpoints contextual menu



The contextual menu command **Show Breakpoints** will bring up an Editor window showing the section of source file containing the selected breakpoint. This will also appear in the Disassembler window (and in the Disassembler window alone if the source contains no debug information).

5.9 Data breakpoints

Note: Data breakpoints are **not available** on the STVD EMU3 version. However, the equivalent functionality is provided using Advanced Breakpoints (see [Section 10.2: Using advanced breakpoints on page 291](#)).

While instruction breakpoints allow you to stop the running of the application at a specific instruction line, data breakpoints interrupt the run when a data variable is either read or written.

Each data variable has a fixed address, and by placing a data breakpoint on a particular variable, you can effectively interrupt the program when that address is either read or written to.

Note: **For the Simulator and DVP versions of STVD**, there are two types of data breakpoint: a read data breakpoint which stops the program when the flagged variable is read, and a write data breakpoint, which stops the program when the flagged variable is written.

This section provides information about using data breakpoints, including:

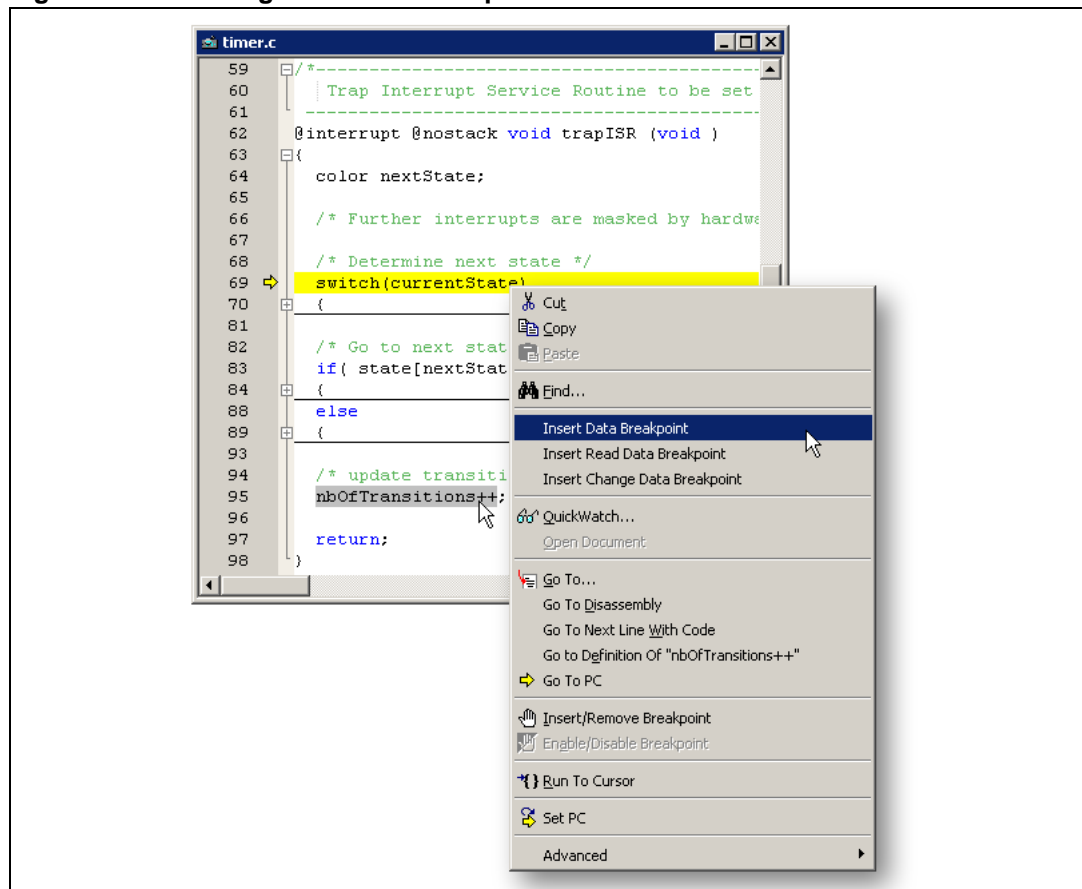
- [Inserting data breakpoints](#)
- [Using the Data Breakpoints window](#)

5.9.1 Inserting data breakpoints

To insert a data breakpoint:

1. In the Editor, select a variable in the source code file. Be sure to select the entire variable name (highlight it with the mouse).
2. While the variable is selected, you may either:
 - Right-click the mouse. A contextual pop-up menu will appear as shown in [Figure 131](#). Select either **Insert Read Data Breakpoint** or **Insert Change Data Breakpoint** from the contextual menu. For EMU2 emulators only **Insert Change Data Breakpoint** is available in the contextual menu.
In the example shown in [Figure 131](#), we are inserting a data breakpoint for the variable `nbOfTransitions++`.
 - or Insert a data breakpoint by dragging and dropping the selected variable into the **Data Breakpoints** window (described in [Section 5.9.2: Using the Data Breakpoints window](#)).

Figure 131. Inserting write data breakpoint



Note: Only read data breakpoints can be inserted this way for the **Simulator and DVP versions**.

5.9.2 Using the Data Breakpoints window

You can track and control your data breakpoints using the **Data Breakpoint** window. To open this window, from the main menu, select **View>Data Breakpoints**.

The data breakpoints window shows:

- A list of the variables for which a breakpoint has been assigned.
- The lettered red dot identifies whether the data breakpoint is a read data breakpoint ("R") or a write data breakpoint, ("W"). For the EMU2 emulator version, where there is no distinction between read and write data breakpoints, all data breakpoints show a read dot with "RW".
- The **Counter** field allows you to enter the number of times that the variable is accessed before the run is halted. In the example shown in [Figure 133](#), the application will be halted once the variable `RR200L` has been read 5 times. In the EMU2 emulator example shown in [Figure 132](#), the application run will be halted once the variable `RR200L` has been read or written to 5 times.
- The **Condition** field allows you to enter a specific value of the variable as the condition upon which the application will be halted. In the example shown in [Figure 133](#), when the value 256 is written to `st0`, the application is halted. In the example shown in [Figure 132](#), if `st0` has a value of 256, the application is halted.

Figure 132. Data breakpoints window (EMU2)

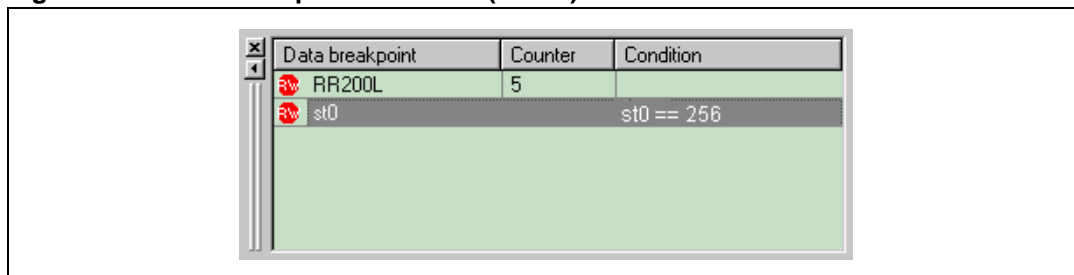
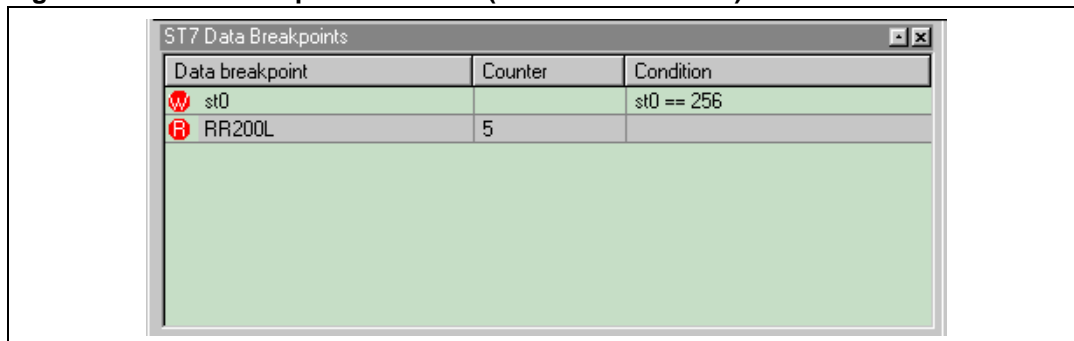


Figure 133. Data breakpoints window (DVP and Simulator)




Below, we see the result when the program is run—the first breakpoint reached is when `st0` equals 256.

Note: *For the EMU2 emulator and DVP only—When using the **Counter** and **Condition** features, be aware that STVD will stop the running of the application on the emulator each time it needs to evaluate the counter and/or the condition, which means that the application is not executing in real time, although it appears to be in Run mode.*

5.10 Call stack window

To open the **Call Stack** window, either click on the **Call Stack** window icon  in the **View** toolbar or from the main menu select **View>Call Stack**.

The **Call Stack** window shows the function calls stored on the microcontroller's **Call Stack** from the most recent function call, traced back to the earliest call. For example, the Program Counter  is always placed at the most recent entry in the **Call Stack** window (#0).

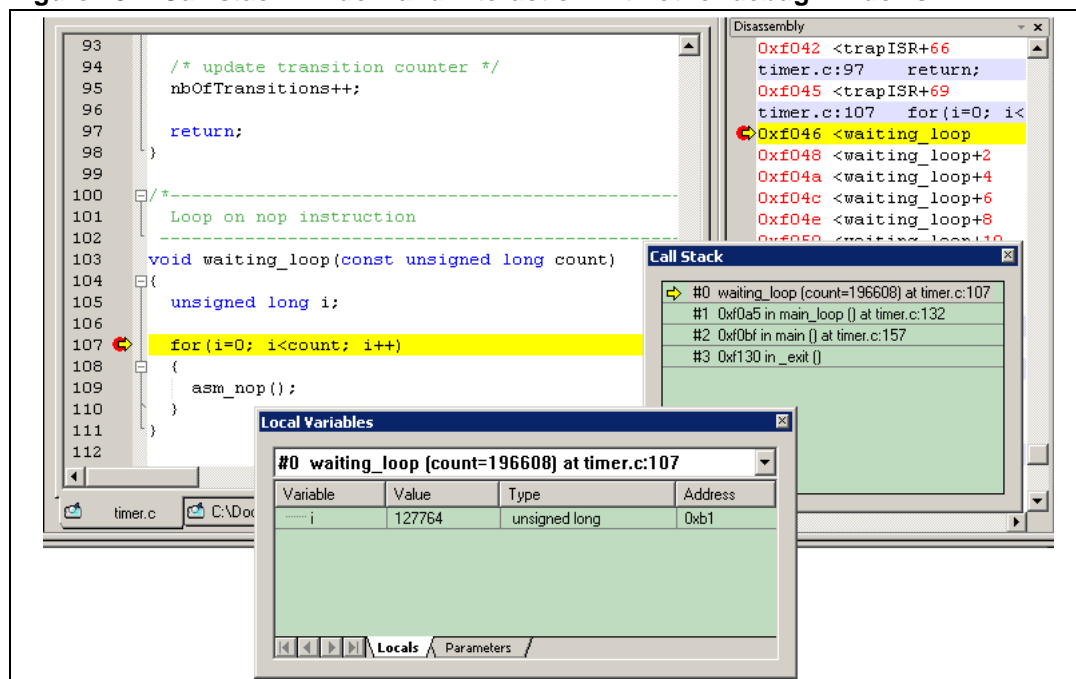
The stack keeps track of all the variables and parameters local to the part of the program where the function call is made at the moment of the function call.

If an entry in the call stack is selected and double-clicked, the source code corresponding to that address appears highlighted in the Editor window. Program variables and parameters are shown in the **Local Variables** window with the values they had when the function call was made.

Being able to move backwards in the stack permits program states corresponding to specific entries in the call stack to be examined in detail without moving the Program Counter.

For example, [Figure 134](#) shows a snapshot of the program at the current Program Counter position. The **Local Variables** window shows the value of the parameter "i", and the location of the PC is shown both in terms of the instruction line in the Editor window, and in terms of the address in the **Disassembly** window.

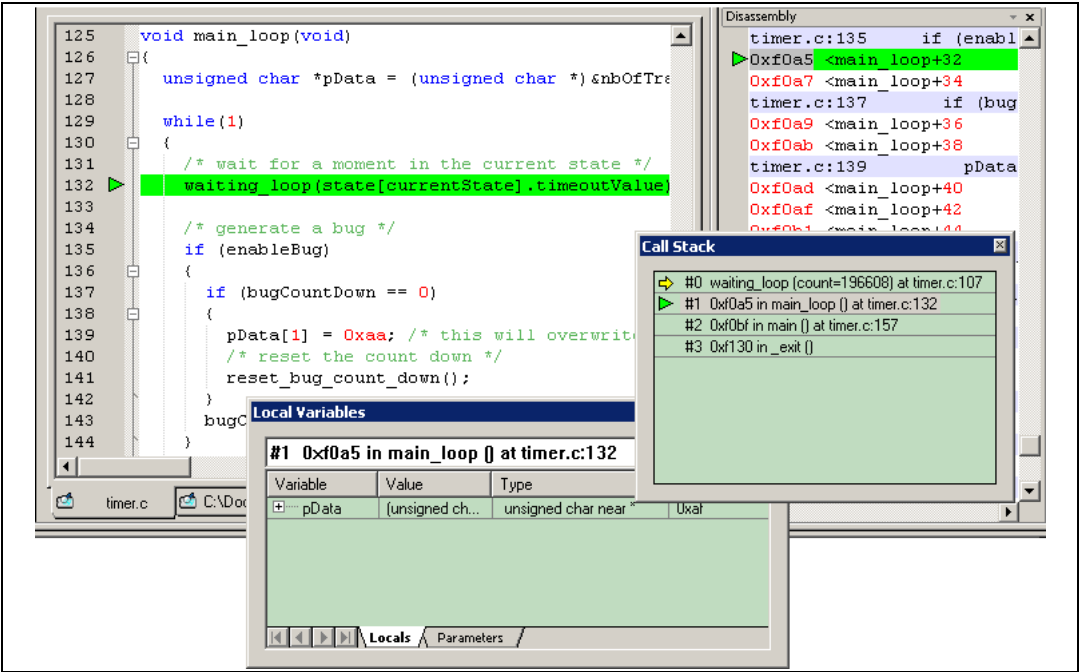
Figure 134. Call stack window and interaction with other debug windows



Now, if you want to step backwards through the call stack, entry by entry, simply double-click on the each entry of interest. Remember that they are numbered backwards in time, meaning that the larger the number, the farther back in the stack you are.


In [Figure 135](#), we decided to step backward in the call stack to look at the function call made just before we reached the current PC position. By double-clicking on entry #1, a Call Stack Indicator will appear in the Editor window to indicate where the call was made in the source code. If the **Disassembly** window is open, a Call Stack Indicator will highlight the address of the instruction after the call. The **Local Variables** window shows the values of the variables as they were at the moment the selected function call was performed.

Figure 135. Stepping backwards in call stack



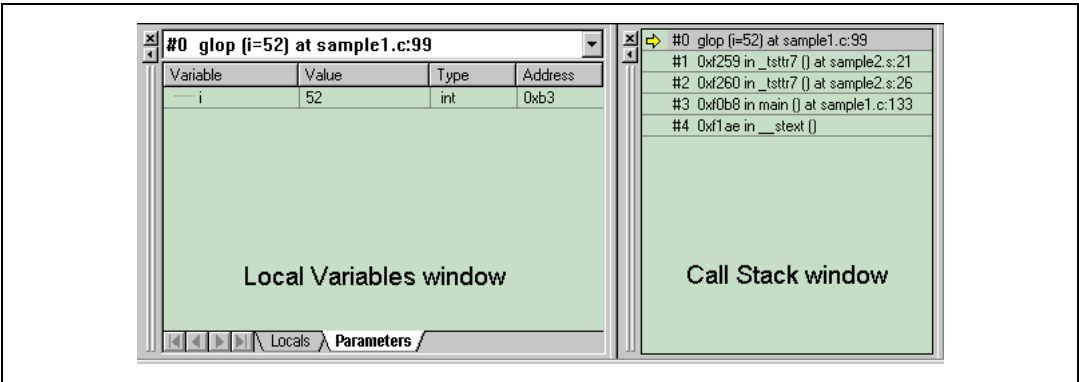
5.11 Local variables window

The **Local Variables** window shows a listing of program variables that are local to the functions that are active at any given time, as well as their values at particular points (such as enabled breakpoints) in the running of the program.

To open the **Local Variables** window, either click on the **Local Variables** window icon  in the View toolbar or, from the main menu, select **View>Local Variables**.

In the example shown below, the value of the variable `i` is equal to zero when the program is halted at a particular breakpoint. By scrolling in the list at the top of the **Local Variables** window, you can display current call stack in the **Call Stack** window, and the list of variables in the current call stack. The current call stack frame can be modified from this list.

Figure 136. Local Variables and Call Stack windows



5.12 Watch window

In addition to the **Watch** window description, this section provides information about:

- [Expanding the display](#)
- [Modifying values](#)
- [Watch contextual menu](#)

To open the Watch window, select **View>Watch**.

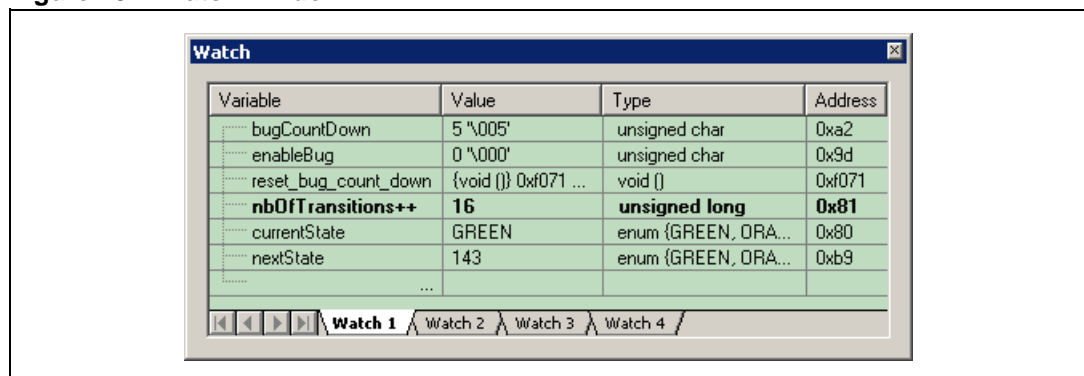
The Watch window displays the current value of selected program variables when the executable is stopped. Enter the variables that you want to monitor in the left-hand column of the window. The information is updated each time the executable stops, or by means of the **Refresh All** option in the Watch contextual menu.

The values displayed are those in the context of the current Program Counter (PC) position. You can change this context by setting the **Call Stack Frame indicator** in the Call Stack window (see [Section 5.10: Call stack window](#)). The Watch window is refreshed to display variable values at the position of the Frame indicator.

You can add program symbols from the source files to the Watch window in several ways:

- The variable name may be typed directly into the left column (double-click on the first empty row in the table to open it for keyboard entry).
- Alternatively, a variable may be selected in the Editor window and dragged and dropped into the Watch window.
- The QuickWatch window also provides the button option **Add to Watch**, to include the QuickWatch variable in the current Watch window.

Figure 137. Watch window



Four pages (accessible by tabs) are provided in the Watch window to permit grouping of variables in particular contexts (for example, *function_x* in **Watch 1**, *function_y* in **Watch 2**).

Expanding the display

A **structure**, **bitfield**, **array** or **pointer** listed in the Watch window can be expanded. The sign '+' before the variable indicates that it is expandable. Click on the + sign to open the next level of display:

- **Structure, bitfield:** The + symbol displays each member of a structure or bitfield.
- **Array:** In an array, the + symbol opens only the first member.
- **Pointer:** The + symbol opens a second line displaying the variable pointed to.

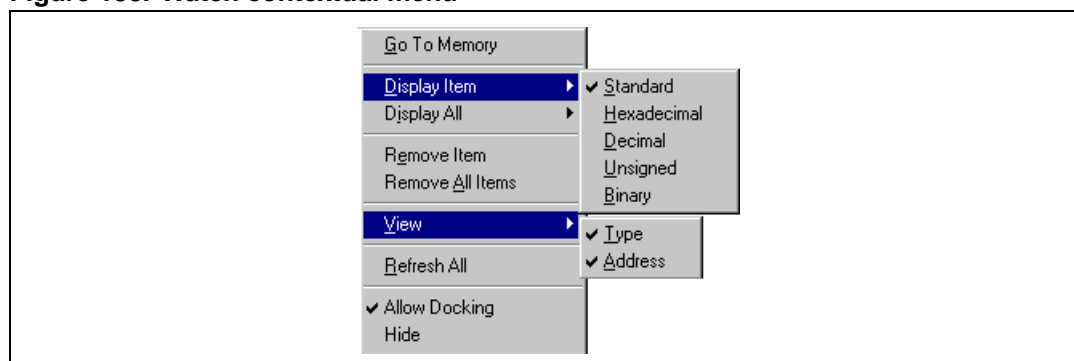
Modifying values

Variable values can be modified directly in the Watch window. Double-click in the right-hand (*Value*) column to open the value for keyboard entry. The new value can be entered in Decimal or Hexadecimal.

Note: Modifying values is only possible in the case of simple numerical values. Hence it is necessary to expand any expandable variable (described above) to the ultimate level where each element is represented by a simple numerical value, before modifying this value.

Watch contextual menu

Figure 138. Watch contextual menu




Right-click within the Watch window to open the Watch contextual menu providing the following options (plus standard docking options):

- **Refresh all:** Refreshes all displayed variables.
- **Display Item:** Sets the display format for the selected item to Standard, Hexadecimal, Decimal, Unsigned, or Binary.
- **Display All:** Sets the display format for all variables to Standard, Hexadecimal, Decimal, Unsigned, or Binary.
- **Remove Item:** Removes the selected (highlighted) variable.
- **Remove all:** Removes all displayed variables.
- **View:** You can choose to show or hide the *Type* and *Address* columns in the Watch window by checking or unchecking (respectively) these two options.

A QuickWatch window is also available via the Editor's contextual menu. This provides rapid contextual access to the most commonly used Watch functions. Values are easily transferred from QuickWatch window to the Watch window as described above.

5.13 Core registers window

To open the **Core Registers** window, either click on the **Core Registers** window icon  in the **View** toolbar or from the main menu select **View>Core Register**.

The **Core Registers** window shows the contents of all registers used by the application, with the values they contain at the current program counter (PC) location. The register window is paged to show the different registers, accessible via the view tabs at the bottom of the window. These provide details of content for the Concurrent Interrupts (IT) and Nested

Interrupts (IT), and for the Simulator version only, the **Simulator Time** registers and the **Simulator Instruction counter**. An example of each is shown in the figures below.

Figure 139. Concurrent IT registers

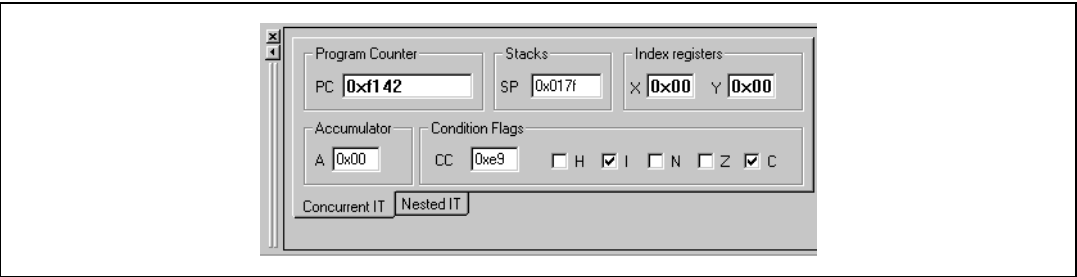


Figure 140. Nested IT registers

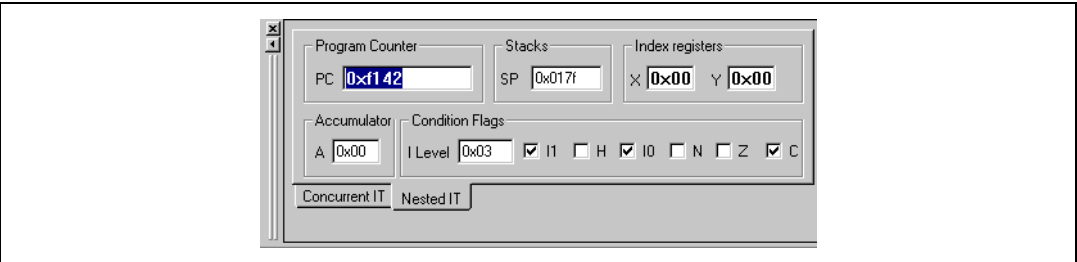


Figure 141. Simulator time registers (Simulator only)

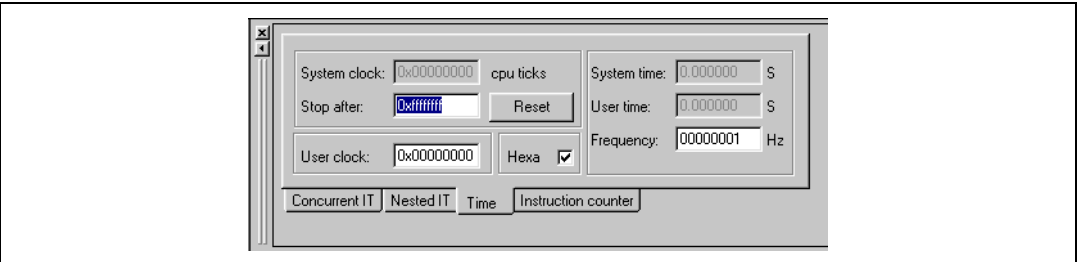
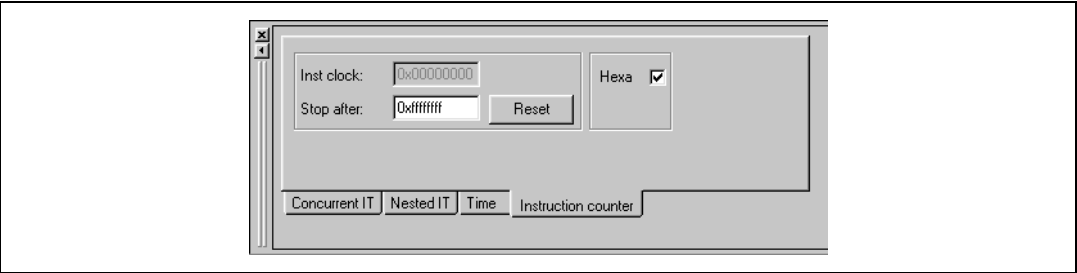


Figure 142. Simulator instruction counter (Simulator only)



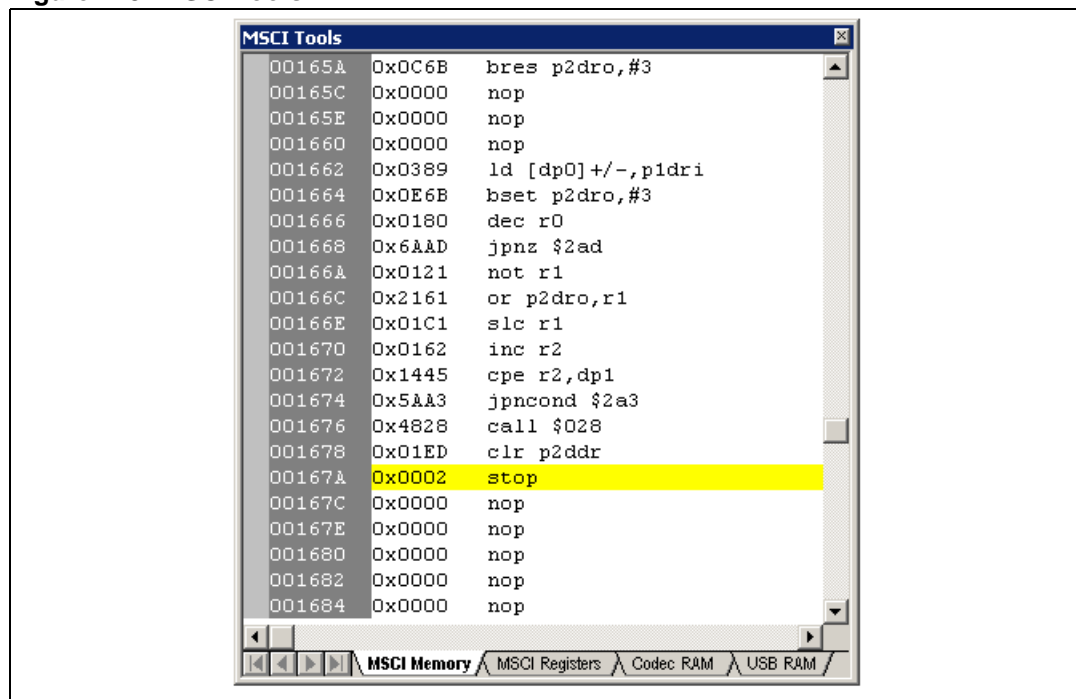
5.14 MSCI tools window

When developing applications that use the ST7 Mass Storage Communication Interface (MSCI), STVD provides special views for **EMU3** and **ICD** that display the contents of the memory space and disassembled code associated with this ST7 system.

Caution: The memory space and registers for the MSCI peripheral are visible in the Memory window and the Core Registers window. However, the viewed values for the MSCI memory and register in these windows may not be accurate. To view the actual values by MSCI routines, always use the the MSCI Tools window.

To access the MSCI Tools window, select **View>MSCI Tools**.

Figure 143. MSCI Tools



The PC showing the state of execution is represented by a yellow highlighted line.

MSCI memory tab

This provides a disassembly view of the MSCI application code contained in the MSCI memory space. When display is set to **Dissassembly** in the contextual menu, the line of disassembled source code includes the **physical address**, the **value in memory** at that address and the **assembly instruction**.

To change the display, right-click to get the contextual menu and select **Dissassembly** or **Hexadecimal**.

In addition, the contextual menu allows you to **Set/Clear breakpoints** and to navigate from one break to another (**Next Break**). When a program halts at a breakpoint, the PC indicates the next MSCI instruction to execute in the **MSCI Memory** tab.

MSCI registers tab

The MSCI Registers tab displays the MSCI internal registers memory and the actual values of MSCI registers during application execution. Values of registers can be modified by selecting a bit and typing a new value. Symbolic display allows you to view the names of the various MSCI registers and their associated values in word format.

To change the display, right-click to get the contextual menu and select **Symbolic** or **Hexadecimal**.

Codec RAM tab

Displays the contents of the RAM for the MSCI's Code Decompressor (CODEC). Memory contents are displayed in hexadecimal in byte format. You can change the values in RAM by highlighting the bit to change and typing the new value.


To view a specific address, right-click to get the contextual menu and select **Go to**. Then enter the address you want to display and click on **OK**.

USB RAM tab

The USB RAM tab displays the contents of the RAM for the MSCI's USB buffer. Memory contents are displayed in hexadecimal in byte format. You can change the values in RAM by highlighting the bit to change and typing the new value.

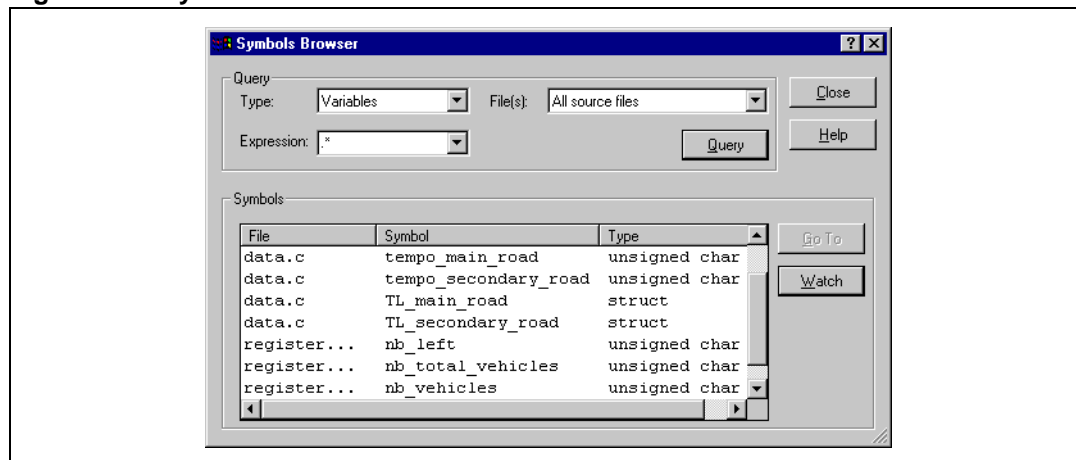
To view a specific address, right-click to get the contextual menu and select **Go to**. Then enter the address you want to display and click on **OK**.

5.15 Symbols browser

To open the **Symbols Browser** window, either click on the **Symbols Browser** window icon  in the **View** toolbar or, from the main menu, select **View>Symbols Browser**.

In the **Symbols Browser** window, you can select program symbols, first by specifying the type of symbol and the type of file in the **Type** and **Files** fields respectively, then clicking on **Query** to instigate a search of the specified files. If you want to further narrow your search, the **Expression** field can be used to define a search string (wildcards are accepted).

Figure 144. Symbols Browser — variables



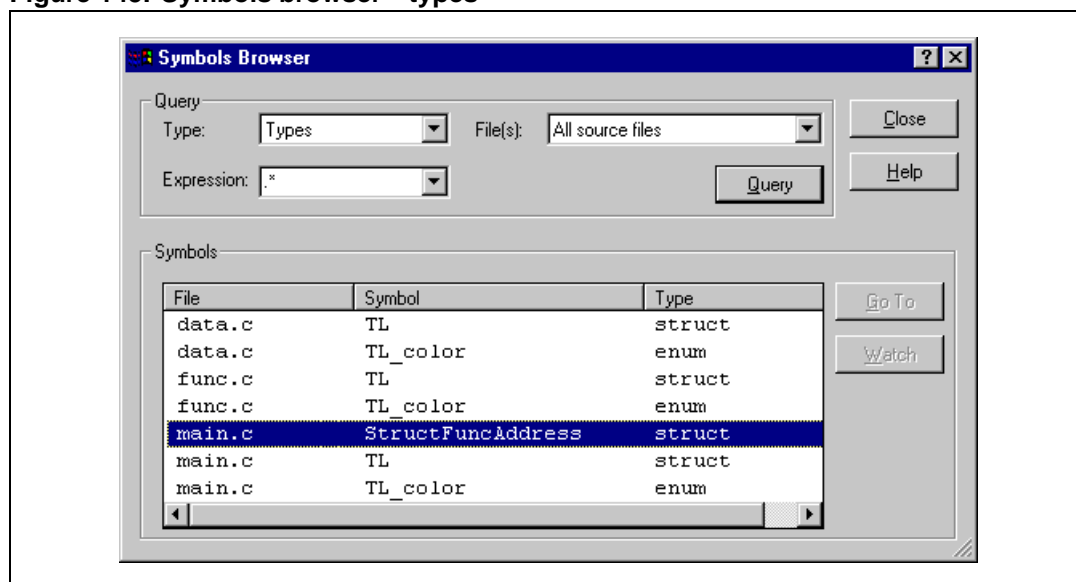
In the resulting listing, there are three fields: **File**, which is the filename in which the symbol was found; **Symbol**, which gives the name of the symbol (either function or variable); and **Type**, which specifies the type of the function or variable.

Selecting a symbol from the list (using the **GoTo** button or by double-clicking on the symbol name in the table) will open an Editor window displaying the corresponding source file at that symbol's location, and will also open the Disassembly window at the symbol's address.


The **Watch** button will copy the found symbol to the watch window.

Note that if you perform a query of all Types (by selecting **Types** in the *Type* drop-down list when defining your query), the result will be a listing of all variable or function types present in each of the specified file(s). However, you will be unable to **Go To** any of the entries in the list, as these entries do not correspond to a particular occurrence of a symbol, but indicate the presence of a given symbol type in a given file. This sort of query allows you to quickly check whether a certain symbol type is present in a particular file or group of files.

Figure 145. Symbols browser—types



5.16 Peripheral registers window

To open the **Peripheral Registers** window, either click on the **Peripheral Registers** window icon  in the View toolbar or, from the main menu, select **View>Peripheral Registers**.

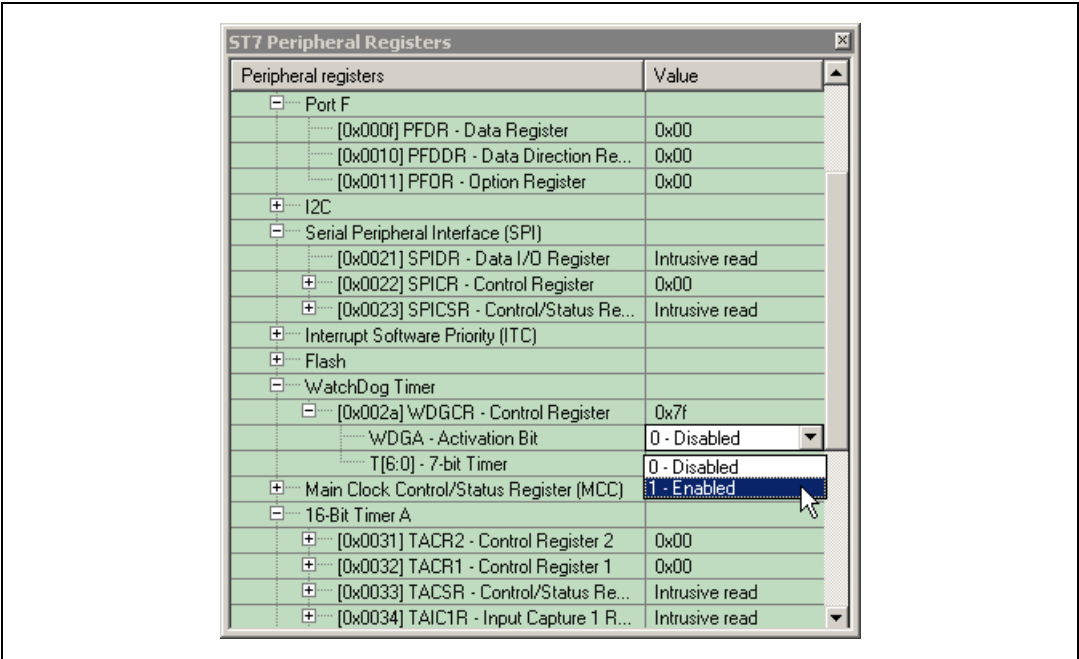
In addition to the description of the **Peripheral Registers** window, this section contains information about:

- [Peripheral registers contextual menu](#)
- [Peripheral registers display options](#)
- [Forced read of status registers](#)

The **Peripheral Registers** window shows a list of the peripherals available on the target MCU, as well as the value of each peripheral's register at a particular value of the program counter (PC).

This allows you to check the value of the peripheral registers at any breakpoint in the debugging of the program.

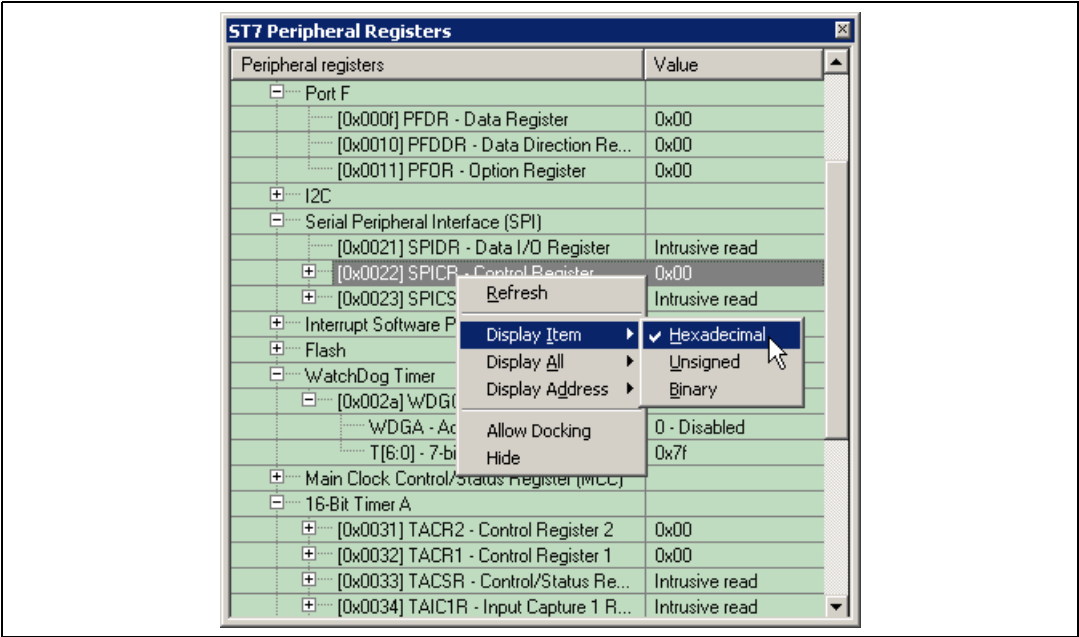
Figure 146. Peripheral registers window



Peripheral registers contextual menu

Using the **Peripheral Registers** contextual menu, you can access options for the window. Right-click the mouse while the cursor is in the window and the contextual menu shown in [Figure 147](#) will appear.

Figure 147. Peripheral registers contextual menu



Peripheral registers display options

Using the display options in the contextual menu, you can customize the display format of information shown in the **Peripheral Registers** window:

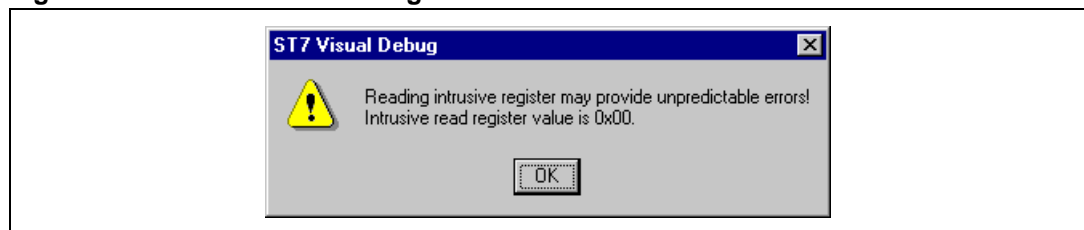
- **Display Item:** You can choose to display the value of the selected Peripheral Register item in either hexadecimal, unsigned integer or binary format.
- **Display All:** You can choose to display all values of all Peripheral Register items in either hexadecimal, unsigned integer or binary format.
- **Display Address:** You can choose to display the address of the peripheral or of each individual register within the peripheral, or both for all items in the Peripheral Registers window.

Forced read of status registers

Values of status register items in the **Peripheral Registers** window are not normally available—beside each item, you will see the message "Intrusive read" indicating that reading the value of these registers could intrude upon the normal running of the program, and give rise to unpredictable results.

However, if you absolutely must know the value of a particular status register item, select the item with the mouse and right-click to bring up the contextual menu, as shown above. You will then have access to an additional option -- the **Forced Read** option. Choosing this option will allow an intrusive read of the status register. A dialog box will appear as shown in [Figure 148](#), warning you of the possibility of unpredictable consequences, and giving the value of the status register.

Figure 148. Forced read warning



Note: The value of the status register is only valid at the moment of the forced read.

5.17 Memory trace window

The **Trace** window allows you to view recorded hardware cycles that have occurred during the execution of your application. In addition, different trace recording modes allow you to control what information is viewed and when.

You can open the **Trace** window either by clicking on  (the Trace window icon) in the View toolbar, or from the main menu by selecting **View>Trace**.

The memory trace is implemented differently for the debug instruments that support this feature. For more information on the trace capabilities, refer to the section specific to your debug instrument:

- [Section 8.3: Trace recording on page 242](#) for DVP/EMU2
- [Section 10.1: Trace recording on page 284](#) for EMU3
- [Section 9.1: Trace recording on page 258](#) for STice

5.18 Online commands

An online command is a single line of input that you can use to control the debugger (GDB) via the **Console** tab of the **Output** window. There is no limit on how long it can be. It starts with a command name, which is followed by arguments whose meaning depends on the command name. For example, the command `step` accepts an argument which is the number of times to step, as in `'step 5'`. You can also use the `step` command with no arguments. Some command names do not allow any arguments.

Online command names may always be truncated if that abbreviation is unambiguous. Other possible command abbreviations are listed in the documentation for individual commands. In some cases, even ambiguous abbreviations are allowed; for example, `s` is specially defined as equivalent to `step` even though there are other commands whose names start with `s`. You can test abbreviations by using them as arguments to the `help` command.

A blank line as input to the debugger (typing just RET) means to repeat the previous command. Certain commands (for example, `run`) will not repeat this way; these are commands whose unintentional repetition might cause trouble and which you are unlikely to want to repeat.

The `list` and `x` commands, when you repeat them with RET, construct new arguments rather than repeating exactly as typed. This permits easy scanning of source or memory.

The following section describe some of the most useful online commands.

5.18.1 The `load` command

```
load filename
```

The `load` command makes a *filename* (an executable) available for debugging on the remote system—by downloading, or dynamic linking, for example. The `load` command also records the *filename* symbol table in GDB, like the `add-symbol-file` command.

The file is loaded at whatever address is specified in the executable. For some object file formats, you can specify the load address when you link the program; for other formats, like `a.out`, the object file format specifies a fixed address.

`load` does not repeat if you press RET again after using it.

5.18.2 The `output_file` command

```
pin -output_file <yes/no>
```

The `output_file` command sends the input/output values on pins to a text file `port.out` in your project directory. This text file reports pin names and their input/output values at intervals of 1 cpu cycle. To disable the output, enter `gdi -output_file no` in the Console tab.

5.18.3 The `stimuli` command

```
pin -stimuli
```

The `-stimuli` command allows you to see the input stimuli commands that are being used by the simulator in the Console tab of the Output window.

5.18.4 The `symbol-file` command

```
symbol-file filename [-readnow ] [ -mapped ]
```

You can override the GDB two-stage strategy for reading symbol tables by using the `-readnow` option with any of the commands that load symbol table information, if you want to be sure GDB has the entire symbol table available.

If memory-mapped files are available on your system through the `mmap` system call, you can use another option, `-mapped`, to make GDB write the symbols for your program to a reusable file. Future GDB debugging sessions map in symbol information from this auxiliary symbol file, rather than spending time reading the symbol table from the executable program. Using the `-mapped` option has the same effect as starting GDB with the `-mapped` command-line option.

You can use both options together, to make sure the auxiliary symbol file has all the symbol information for your program.

The auxiliary symbol file for a program called *myprog* is called `myprog.syms`. Once this file exists (providing it is newer than the corresponding executable), GDB always attempts to use it when you debug *myprog*; no special options or commands are needed.

The `.syms` file is specific to the host machine where you run GDB. It holds an exact image of the internal GDB symbol table. It cannot be shared across multiple host platforms.

5.18.5 The `save_memory` command

```
save_memory from_address count filename [format]
```

Saves the contents of a range of memory to a file.

from_address is the start address of the memory block to save.

count is the size of the memory block to save.

filename is the name of the file to create.

format is the file format extension: `*.srec` for Motorola Srecord format, `*.ihex` for Intel Hexa format (the default format is `*.srec`).

5.18.6 The `help` command

```
help command
```

GDB contains its own online help facility giving definitions of all GDB commands. To access it, at the GDB prompt, enter the command `help` followed by the name of the command you want usage information on.

```
cd directory
```

Set the GDB working directory to directory.

5.19 Limitations and discrepancies

STVD may be subject to limitations or discrepancies (differences between emulation and actual MCU behavior). This section provides information about [General debug limitations](#) that are common to all debug instruments and the simulator. It also provides information on where to find out about limitations and discrepancies that are specific to your debug instrument and your microcontroller.

Limitations are documented in the current **STVD Release Notes**. When possible, workaround procedures are provided to help minimize their impact on your project. The release notes will also inform you of the resolution of known limitations.

Discrepancies between emulation hardware and your MCU's behavior are provided in STVD's [Debugging discrepancies window](#).

General debug limitations

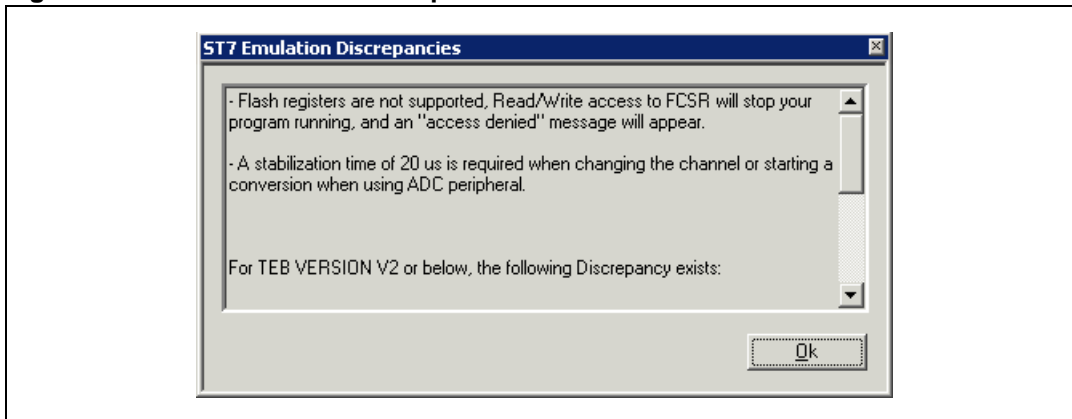
In-Application Programming (IAP) limitation

STVD cannot debug IAP applications (user code that reprograms Flash memory), as the Flash programming algorithm is not emulated nor simulated.

Debugging discrepancies window

You will find a complete list of discrepancies for your emulation hardware and target devices in the **Debugging Discrepancies** window (see [Figure 149](#)). To access this window, start a debugging session, then select **Debug Instrument>Emulation Discrepancies**.

Figure 149. ST7 Emulation Discrepancies window



Discrepancy information for your particular MCU and emulation hardware is uploaded at the same time as the rest of your MCU configuration information.

6 Simulator features

In addition to debugging with emulation hardware and in-circuit debugging devices, STVD also allows you to debug your application while it runs on a software simulation of your target microcontroller. The Simulator is part of STVD and requires no additional emulation or in-circuit debugging hardware.

Once you have set your debug instrument to **Simulator** and entered the debug context (**Debug>Start Debugging**), the **Debug Instrument** and **View** menus change to display the commands that are specific to the Simulator.

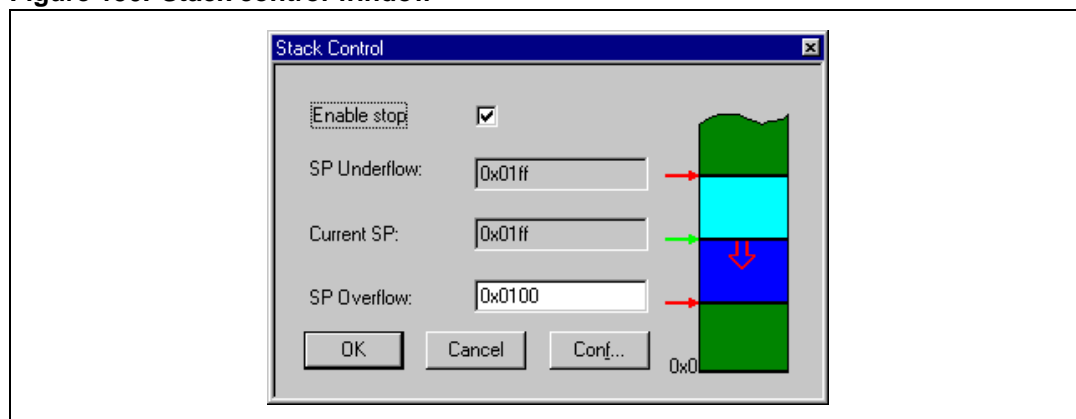
This section explains how to use the debugging features that are specific to the STVD Simulator:

- [Section 6.1: Using the stack control features](#)
- [Section 6.2: Using the input pin stimulator](#)
- [Section 6.3: Using the simulation plotter](#)
- [Section 6.4: Using read/write on-the-fly](#)
- [Section 6.5: Forcing interrupts](#)

6.1 Using the stack control features

The **Stack Control** window allows you to set a specific stack address as the stack overflow point. This means that if, during the course of debugging, the current stack pointer (SP) value is exceeded, the application will be halted (provided that you check the **Enable stop** box in the **Stack Control** window).

Figure 150. Stack control window



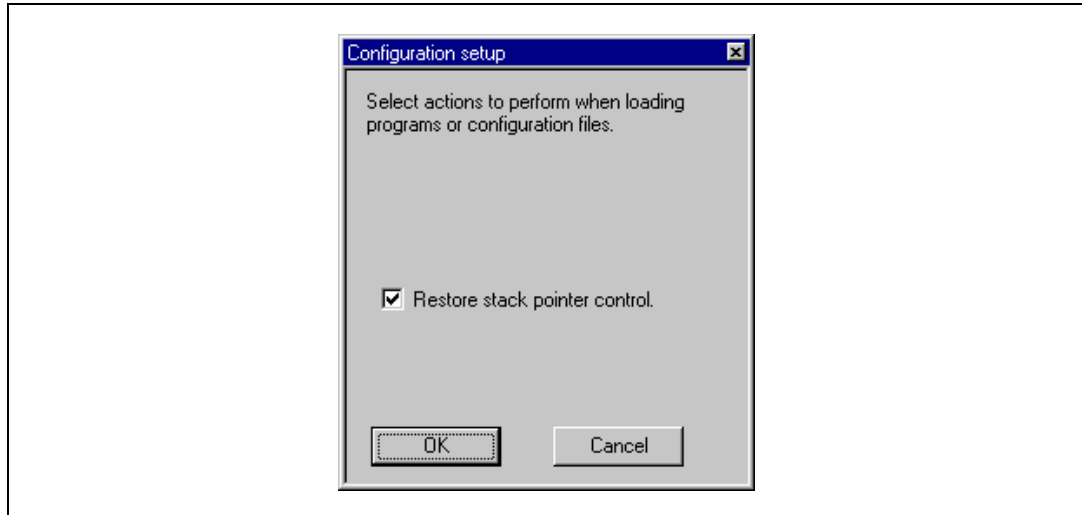
This effectively allows you to specify a breakpoint at the SP address level.

To set the stack control:

1. Open the **Stack Control** window by clicking on  (the Stack Control window icon) on the Tools toolbar, or by selecting, **Tools>Stack Control...** from the main menu.
2. To enable a break in the running of the application when the specified conditions are met, check the **Enable stop** box.
3. Type the overflow address in the **SP Overflow** field.

4. To view/change the configuration settings, click on the **Conf...** button.
The configuration setup window shown in [Figure 151](#) is displayed. Here you can enable or disable the stack pointer control.

Figure 151. Configuration setup window



5. Click on **OK** when you have set up the stack control parameters as you want.

6.2 Using the input pin stimulator

The **input pin stimulator** allows you to simulate an analog or digital signal on a selected input pin of your target microcontroller. This allows you to test the application software's response to an input as if the signal were coming from your application hardware.

During a debugging session, you can generate input signals on-the-fly or with a time delay using the **I/O Stimulation** window. You can also create a script of signal inputs in a text file called a **stimuli file** that you import prior to running your simulation, which tells the simulator when and on what pins to generate input signals.

You can view and define input signals from the **I/O Stimulation** window. This window shows all I/O pins for your selected MCU. In this window, the input pins for your target microcontroller are listed in the **Name** column. The current input signal is displayed in the **Current Value** field. When you define an input signal, the value that you have specified for the signal appears in the **New Value** field.

From this window you can:

- [Trigger an analog or digital signal with a delay](#)
- [Trigger a periodic binary signal with or without a time delay](#)
- [Trigger a binary signal on-the-fly](#)
- [Create and load a stimuli file](#)

When defining the signal, digital signals can have a value of **1** (VDD) or **0** (GND). Analog signals have any value between **0** and **6.5V** TTL. Times for delay and periodicity are specified in CPU cycles.

If you enter an input signal while your application is running, the signal is sent, or the count down for the delay begins once you have defined the signal to send. If you enter an input

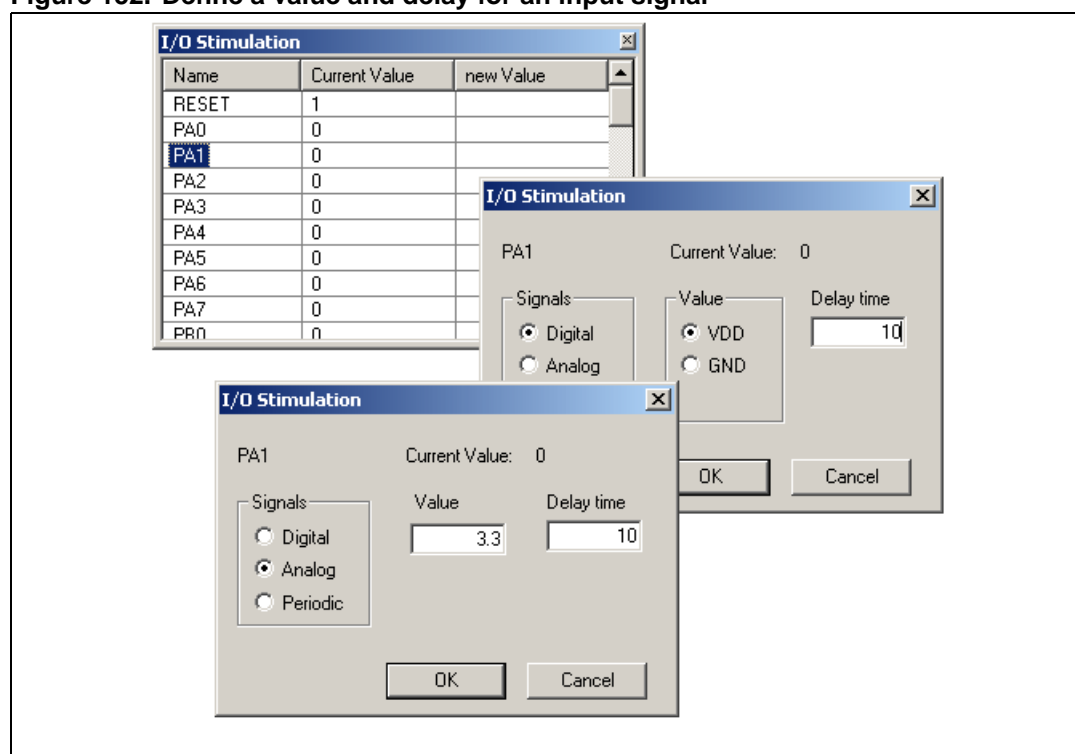
signal while your application is not running, the signal is taken into account, or the countdown for the delay begins as soon as you start running your application (**Debug>Run**).

Signals are reset to their defaults when the application is restarted or when the MCU is reset.

Trigger an analog or digital signal with a delay

1. Select **View>I/O Stimulation** from the main menu bar.
The **I/O Stimulation** window opens, showing all I/O pins for your selected MCU.
2. Click on the pin name to select it.
3. Double-click in the value field corresponding to the pin that you want to simulate a signal on.
The **I/O Stimulation** dialog box opens.
4. Select the signal type: **Digital** or **Analog**.
5. For a digital signal, click on the radio button for **VDD** (1) or **GND** (0).
For an analog signal, enter a value (0 - 6.5V TTL) in the **Analog Value** field.
6. To specify a time delay, enter a value in the **Delay Time** field. Time delays are measured in CPU cycles.
7. Click on **OK**.

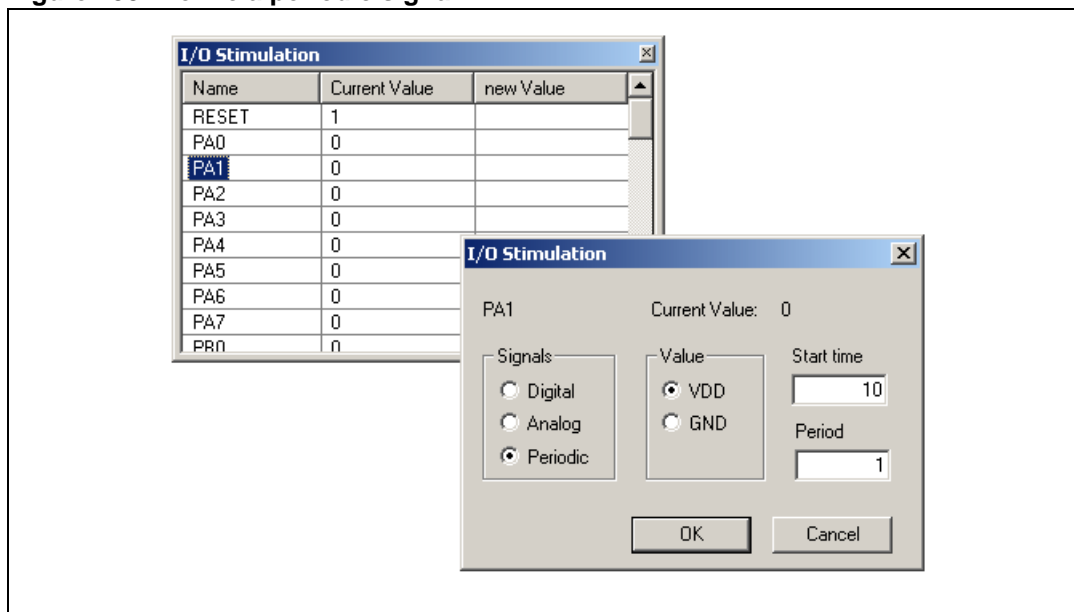
Figure 152. Define a value and delay for an input signal



Trigger a periodic binary signal with or without a time delay

1. Select **View>I/O Stimulation** from the main menu bar.
The **I/O Stimulation** window opens, showing all I/O pins for your selected MCU.
2. Click on the pin name to select it.
3. Double-click in the value field corresponding to the pin that you want to simulate a signal on.
The **I/O Stimulation** dialog box opens.
4. Select **Periodic**.
5. Click on the radio button for **VDD (1)** or **GND (0)**.
6. To specify a time delay, enter a value in the **Delay Time** field.
Time delays are measured in CPU cycles.
7. Enter a value in the **Period** field.
The period is defined in CPU cycles. It defines the duration of the specified signal and the duration between occurrences of the signal.
8. Click on **OK**.

Figure 153. Define a periodic signal



Periodic signal example

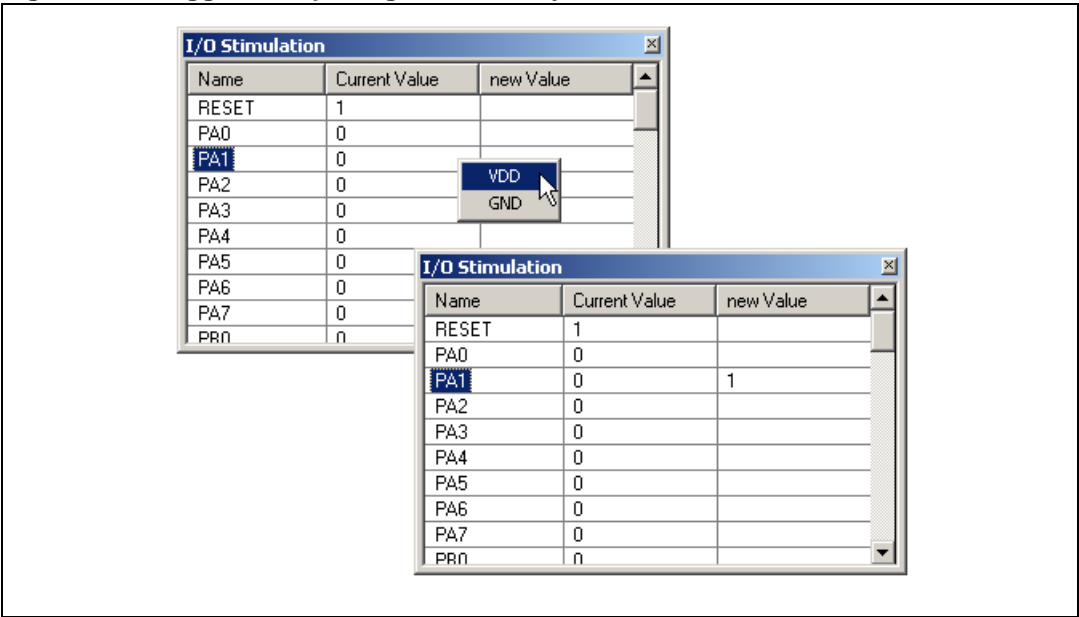
P_{in} = PA1, Value= VDD, Delay= 10, Period= 1

The application runs for 10 CPU cycles before starting the periodic signal. After 10 CPU cycles, PA1 receives a signal at VDD (1) that lasts one CPU cycle then returns to GND (0) for one CPU cycle. This pattern of signals at one CPU cycle continues as long as the application runs or until you define a new signal on PA1.

Trigger a binary signal on-the-fly

1. Select **View>I/O Stimulation** from the main menu bar.
The **I/O Stimulation** window opens, showing all I/O pins for your selected MCU.
 2. Click on the pin name to select it.
 3. Right-click in the value field corresponding to the pin that you want to simulate a signal on.
A pop-up box appears.
 4. Select one of the values (GND = 0, VDD = 1) from the pop-up box.
- Because no delay can be specified, the signal is sent instantly to the input pin you selected.

Figure 154. Trigger an input signal on-the-fly



Create and load a stimuli file

A **stimuli file** is a text file using the **.in** extension that provides the simulator with a script of input signals to generate on pins during the simulation. A stimuli file can be created using any text editor (such as the STVD Editor) and the syntax described in [Table 62](#). It must be saved with the file extension **.in**.

The stimuli file syntax allows you to define digital signals, periodic digital signals or analog signals.

Table 62. Stimuli file syntax

Signal	Description	
Digital input signal		
	Syntax	<code>pin <i>pin_name</i> -i <i>digital_value</i> <i>time</i></code>
	Example	<code>PIN PA1 -i 1 100</code>
	Description	Input the value 1 on PA1, 100 CPU cycles after the start of the simulation. Because the signal is digital, the input value must be 1 or 0 .
Analog input signal		
	Syntax	<code>pin <i>pin_name</i> -a <i>analog_value</i> <i>time</i></code>
	Example	<code>PIN PB1 -a 3.3 1020</code>
	Description	Input the analog value 3.3 on PB1, 1020 CPU cycles after the start of the simulation. Analog values must be between 0 and 6.55 .
Periodic digital input signal		
	Syntax	<code>pin <i>pin_name</i> -c <i>start_digital_value</i> <i>start_time</i> <i>half_period</i> [-cycle <i>cycle_number</i> -end <i>end_time</i>]</code>
	Example	<code>PIN PA1 -c 1 143 20</code>
	Description	Beginning 143 CPU cycles after the start of simulation with an input value of 1, alternate between input of 0 and 1, every 20 CPU cycles.

In the same stimuli file you can script inputs for multiple pins and different types of inputs (for example, you can define both analog and digital inputs on different pins).

Using your stimuli file

To use a stimuli file, simply load it prior to running your simulation by selecting **File>Load Stimuli File**.

During debugging, if you want to see the input stimuli commands that are being used by the simulator, at the command prompt in the **Console** tab of the **Output** window, enter the command:

```
gdi pin -stimuli
```

The text commands for the current stimuli are displayed using the syntax described above.

If you no longer want to use the stimuli file that you have loaded for the simulation, stop using the file by ending the debug session (**Debug>Stop Debugging**), or loading a different stimuli file. Alternatively, on the command line in the Console window, enter the following command:

```
gdi pin -clear
```

6.3 Using the simulation plotter

You can view a graphical display of your simulation in the **Plotter window**. This display shows the values for selected items (such as registers, variables, signals) as they evolve during the simulation. The plotted information for any item that interests you can be printed as it is represented in graphic form by the plotter, or exported in Value Change Dump (VCD)

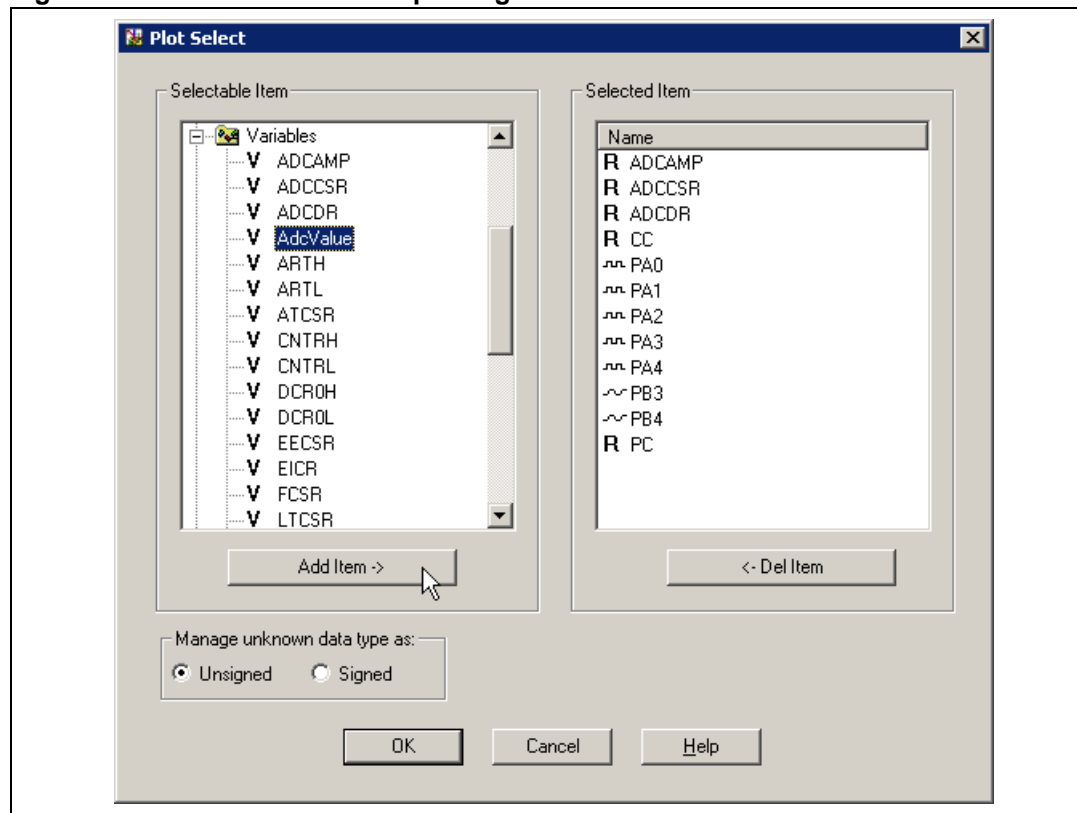
format. Exported items can be quickly and easily imported into later simulations, facilitating comparison between different simulations of the running of your application.

The **Plotter selection window** (*Debug Instrument>Plotter Selection*) provides the interface for choosing the simulated items for plotting.

6.3.1 Plotter selection window

When you are in the debug context, access this window by selecting **Debug Instrument>Plotter Selection**. The **Plotter Selection** window opens (see [Figure 155](#)).

Figure 155. Select elements for plotting



From this interface you can select multiple items that you want to monitor from the **Registers**, **Variables** and **I/O pins** folders in the **Selectable Items** field. To see the full name of a register for an MCU peripheral, use your mouse to place the pointer over the register abbreviation in the list. A pop-up description will appear with the full name of the register.

To add items for plotting:

1. Click on an item to select it.
2. Click on **Add Item**, or double-click on the item that you want to add.
The item is added to the **Selected Item** field. Items are listed in alphabetical order. To delete items, select an item or multiple items (use the SHIFT or CTRL keys) and click on **Del Item**, or drag the item(s) back to the **Selectable Items** field using your mouse.
3. Specify the treatment for unknown data types (**Signed** or **Unsigned**) by clicking on the appropriate radio button.
Your selection is applied to all unknown data types.
4. Click on **OK**, to confirm the list of items for plotting.

The items that you have selected for plotting are saved when you save your workspace, and restored when you open the workspace.

The value changes for the items that you have selected are plotted in the [Plotter window](#) when you run the simulation. The different commands that allow you to execute your application during simulation have different effects on your plot:

- **Run** – restarts the application from initialization and the plot is also restarted. Any plotted information from a previous simulation is lost.
- **Restart** – starts your application from the initialization. Any plotted information from a previous simulation is lost.
- **Continue** – allows you to continue application execution, after a breakpoint for example, and the information plotted prior to the breakpoint is retained in the display.
- **Stepping commands** – the plot continues with each step. The plotted information for preceding steps is retained in the display.
- **Chip reset**– resets the application but does not execute code. The plot is reset, plotted information from the previous simulation is lost.

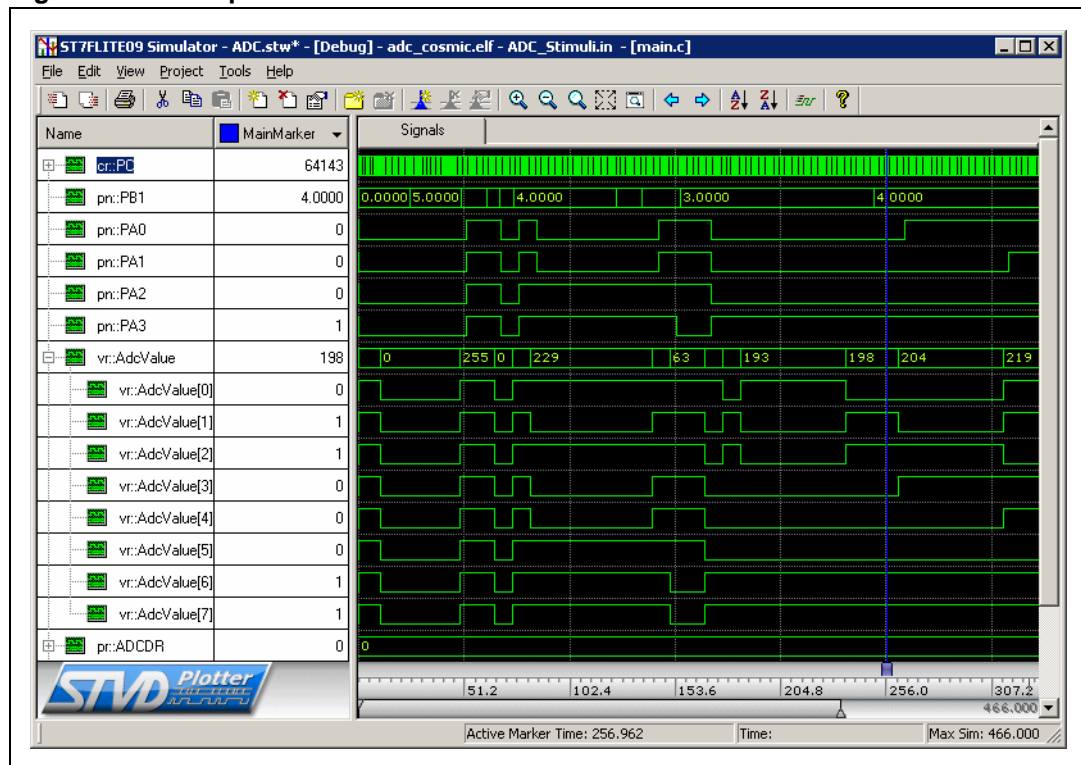
See [Exporting and importing plotted items on page 218](#) for information about saving and reloading the plotted information for a given simulation.

Go to [Plotter window](#) for more information about viewing, navigating and modifying the plot.

6.3.2 Plotter window

To open the **Plotter** window, select **View>Plotter**. While your simulation runs, the Plotter window graphically represents the evolution of values for the items that you have selected in the [Plotter selection window](#) against the application run time.

Figure 156. View plotted elements



The following sections explain in greater detail how to use the plotter window including:

- [Understanding the plotter display](#)
- [Navigating in the plot](#)
- [Adding and removing markers in the display](#)
- [Creating and deleting display tabs and items](#)
- [Modifying item display characteristics](#)
- [Printing a plot](#)
- [Exporting and importing plotted items](#)

Understanding the plotter display

The plotter is composed of the following major elements:

- [Display tabs](#) showing the plotted values for selected and imported items
- [Name column](#) displaying selected items
- [Value column](#) displaying item's value at the selected marker position
- [Time axis](#), with scrolling and integrated time scaling

Display tabs

A main view tab named **Signals** (default name) displays the plotted information from your simulation. When your simulation runs, the items being plotted are updated on this tab and any other tabs that you have added to your project (see [Creating and deleting display tabs and items on page 215](#)).

Table 63. Tab contextual menu

Tab	Description
Add New Tab	Adds a new empty tab (no information is plotted) in the plotter window with a default name.
Remove Active Tab	Removes the active tab from the plotter window. The active tab is the tab whose contents are currently displayed in the Plotter window. Any tab can be removed, however one tab must always remain in the window. It is not possible to delete all of the tabs.
Rename Active Tab	Allows you to rename the active tab. You cannot rename a tab to have the same name as another tab in your project.

In the display area, value changes for selected items are displayed as:

- **Bus** – a bar containing the integer value of an item and/or sub-item.
- **Digital** – a continuous line that indicates the item value as high or low.

The default color for plotted items is green, but the color can be configured to improve visibility of selected items. A darkened plot indicates that the simulation did not provide information to plot the item and the value is unknown. You can modify the display properties of all items to improve the readability of your plot (see [Modifying item display characteristics on page 217](#)).

In addition to the Main Marker (blue), you can add markers to your display. Markers are visible in all tabs. Add, delete and show markers using the commands in the tab contextual menu, which provides access to the commands listed in [Table 64](#).

Table 64. Display area contextual menu

Marker	Description
Add New Marker	Adds a marker, which will be visible in all tabs. Opens a dialog box that allows you to specify the color and name of the marker.
Remove Marker	Removes the marker that you have right-clicked on in the tab. This command cannot be used to remove the blue Main Marker.
Remove All Markers	Removes all the markers from the display except the blue Main Marker. It is impossible to remove this marker.
Show Markers	When checked, the vertical line for each marker is visible in the display.

Name column

This column shows the names of the items that you have selected for plotting, the sub-items that compose them and the names of any items that you have imported. Here you can select items, reorder them, change their display properties (see [Modifying item display characteristics on page 217](#)) and cut/copy and paste items (see [Creating and deleting display tabs and items on page 215](#)).

To **expand** items in order to view the sub-items that they include, click on the “+” next to the item icon. For example, the *AdcValue* variable has been expanded in [Figure 156](#) to view the values of each of the eight bits that it is composed of.

To **change the order** of first-level items, click on them and drag them to a new position. To **sort** them in ascending or descending order, click on a sort button, or select **Tools>Sort>Ascending/Descending** from the menu bar. Sub-items (second-level items)

cannot be sorted or moved. They remain in their initial order with the first level item that contains them.

You can also right-click to get the contextual menu, which provides access to the commands listed in [Table 65](#).

Table 65. Name column contextual menu

Command	Description
Rename	Changes the name of the selected item. You cannot rename an item to have the same name as another item in the tab.
Change Expression	Changes the characteristics of the selected item, including the start time of the plotted item, the time scale and the expression to use to plot the item. See Modifying expressions on page 217 .
Delete	Deletes the selected item from the plot. The deleted item is also removed from the list of selected items in the Section 6.3.1: Plotter selection window on page 208 .
Properties	Changes the display properties of the selected item, including alias, format, height, and color. See Modifying item display characteristics on page 217 .
Default Properties	Sets display properties for the selected item back to the default properties.
Show as	Determines the item display type: <ul style="list-style-type: none"> – Bus – a bar containing the integer value of an item. – Digital – a continuous line that indicates the item value as high or low.

Value column

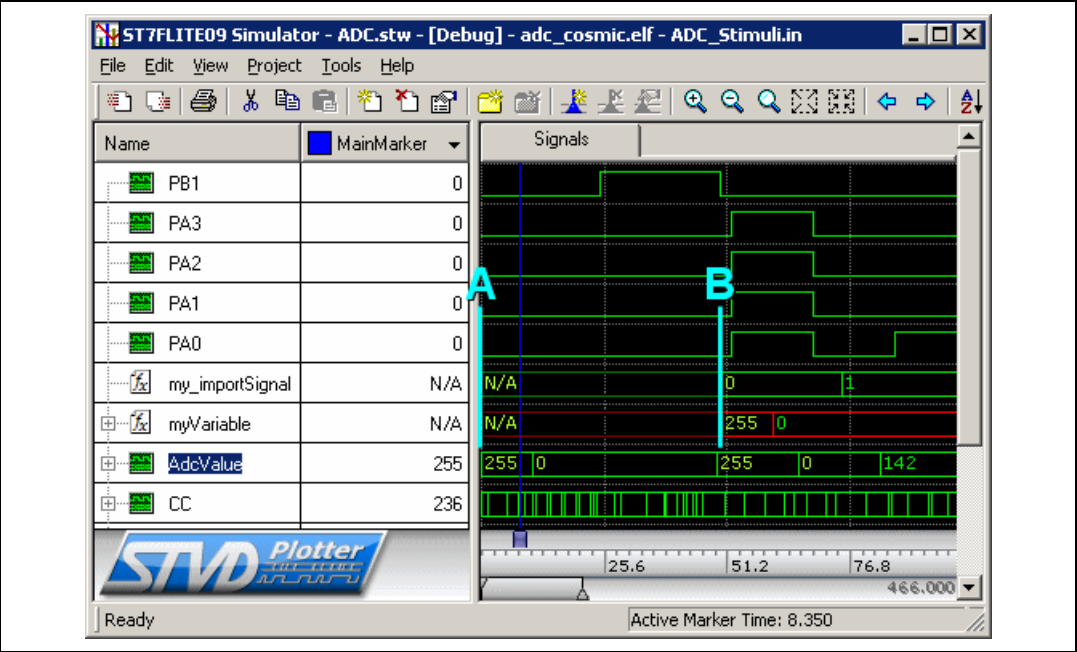
The Value column displays the values of the plotted items at the selected Marker. You can place markers to display the plotted values at any time interval in the plot.

The main marker (blue) and any markers that you create, can be selected in the Value column drop-down menu. To access this menu, click on the heading of the Value column. The name and color of the selected marker are shown in the header.

The value column drop-down menu also provides commands for adding markers, removing markers and viewing/changing marker properties (See [Table 67 on page 215](#)).

In some cases, the plotter does not have information about the values for an item. In this case, the plot color for the item is darkened and the value is shown as **N/A** ([Figure 157](#)). For example, this occurs when the start point (**B**) for plotting an imported or copied item is after the start of the simulation (**A**). In this case, the plotter has no information about the values prior to the time interval **B**. As a result, the plot color is darker and N/A appears in the Value column for the value of the items before time interval **B**.

Figure 157. No information to plot



You can also right-click in the row for an item to get the contextual menu, which allows you to determine the display format for the value of the selected item. For integer values, you can select to display in **Signed** or **Unsigned Decimal**, **Binary**, **Octal** or **Hexadecimal** format. For float values, you can choose to display in **Standard** or **Scientific** format. The current display format is indicated with a check and the default format is indicated in bold.

Time axis

Your items are plotted against a time scale that can be modified, allowing you to zoom in or out and view the plot in more or less detail. Depending on the scaling, you may not be able to view all the value changes if the values change frequently in short time increments.

To change the time scaling of the display, use your mouse to select and drag either of the triangles on the scroll bar, or use the zoom controls on the tool bar (see also [Navigating in the plot](#)).

Navigating in the plot

Zoom controls allow you to zoom in on any part of the plot. You can access these commands in the **Tools** menu, or by using the icons on the tool bar.

Table 66. Zoom commands

Command	Description
Zoom in/out	Enlarges or decreases the display by one step.
Zoom max	Enlarges the display to the maximum plot resolution.

Table 66. Zoom commands

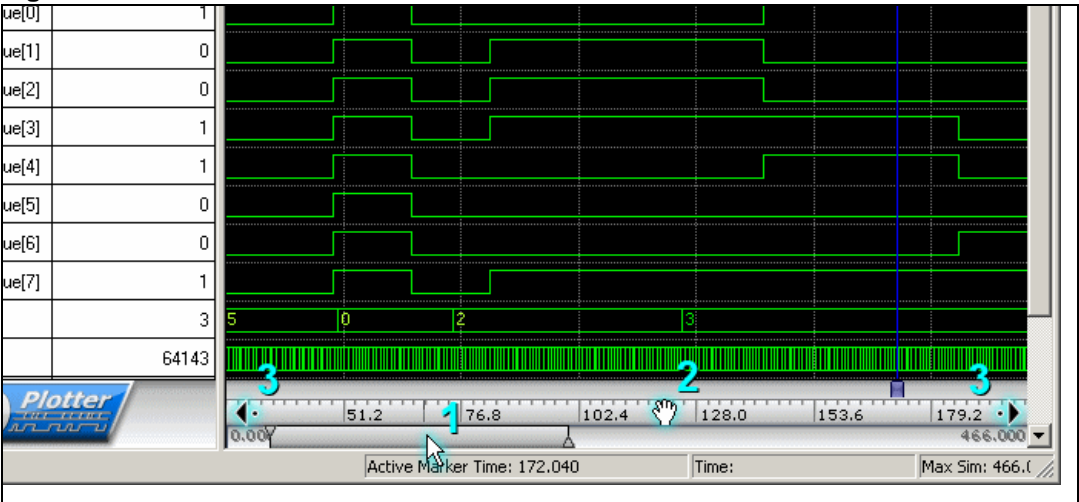
Command	Description
Fit to Window	Decreases the plot to fit into the display tab.
Zoom mode	<p>Zoom in or out on an area that you select. When you select Zoom Mode, the mouse pointer changes to a magnifying glass.</p> <ul style="list-style-type: none"> – Zoom In – (“+” in the magnifying glass) click and drag to define the area to enlarge. Zooms in to view the selected area. – Zoom Out – (“-” in the magnifying glass) Hold down CTRL, then click and drag to define the area. The display decreases when you release the mouse button. The size of the area that you define affects the magnitude of the zoom out (smaller area = smaller zoom out, larger area = larger zoom out). <p>To disable this mode and return to the pointer icon, click a second time on the Zoom Mode Icon on the tool bar.</p>

Scroll features include a scroll bar with integrated scaling control, precision scrolling and fast scroll.

Scroll controls for the time axis include (Figure 158):

1. **Scroll bar** - click and drag the scroll bar to scroll left or right on the time axis. Clicking and dragging on the arrows at either end of the scroll bar changes the time scale of the plot.
2. **Scroll with time scale bar**- place the mouse pointer on the time scale. When a hand appears, click and drag left or right to scroll.
3. **Precision scroll arrows** - move the mouse pointer to the right or left end of the time scale (depending on the direction that you want to scroll) when an arrow icon appears, click and hold to scroll in the selected direction.

Figure 158. Scroll controls



If your plot is long, contains a lot of plotted items and value changes, scrolling may seem slow because the plot is refreshed as you scroll. The **Fast Scroll** setting (**Tools>Fast Scroll**) allows you to scroll rapidly when a large amount of data is plotted. When fast scroll is selected, the value changes in the plot are not refreshed until you stop scrolling.

The **Previous / Next Change** buttons allow you to navigate from one value change to another for the selected item. To use these commands, select an item and click on the Previous or Next change button on the tool bar. The blue Main Marker moves to the next or previous value change for the selected item. The Previous or Next button is not available (it is gray) if there isn't a change in the value to go to.

Adding and removing markers in the display

In addition to the Main Marker, you can specify an unlimited number of your own markers. To help you manage these markers you can specify name, color and position. The tab display area contextual menu, the **Project** menu and the Value column drop-down menu offer the marker controls listed in [Table 67](#).

Table 67. Marker commands

Command	Description
Tab display area contextual menu...	
Add New Marker	Adds a marker, which will be visible in all tabs. In the dialog box, specify the marker Name, Color and Time position in the plot. If you do not specify the Time, by default the marker is placed at the point where you right-clicked in the plot in order to access the contextual menu.
Remove Marker	Removes a marker that you select from a list of markers near the point where you right-clicked in order to access the contextual menu. Select the marker that you want to delete and click on the Remove button.
Remove All Markers	Removes all the markers that you have created. You cannot remove the default Main marker (blue).
Show Markers	When checked, the vertical line for each marker is visible in the display.
Project menu or Value column drop-down menu...	
Add New Marker	Adds a marker, which will be visible in all tabs. In the dialog box, specify the marker Name, Color and Time position in the plot. If you do not specify the Time, by default the marker at the current position of the Main Marker.
Remove Marker	Removes a marker that you select from a list of all markers. A dialog box opens listing the markers that you can remove. The Main Marker cannot be deleted and therefore is not listed. In the dialog box, select the marker you want to delete and click on the Remove button.
Marker Properties	Opens the Marker Properties dialog box, which allows you to select from the markers you have created and modify marker properties including color, name and time.
Remove All Markers	Removes all the markers that you have created. You cannot remove the default Main marker (blue).
Show Markers	When checked, the vertical line for each marker is visible in the display.

Creating and deleting display tabs and items

You can add tabs (my_Signals in [Figure 156](#), for example) and place items in tabs in order to have different views of the same simulation. You can manage tabs using the commands listed in [Table 68](#).

Table 68. Plotter tab management commands

Command	Description
In the Project Menu	
Add New Tab	Adds a new empty tab (no items plotted) in the plotter window with a default name.
Remove Active Tab	Removes the active tab from the plotter window. The active tab is the tab whose contents are currently displayed in the Plotter window. Any tab can be removed, however one tab must always remain in the window. It is not possible to delete all of the tabs.
Rename Active Tab	Allows you to rename the active tab. You cannot rename a tab to have the same name as another tab in your project.
Add New Item	<p>Adds an item to the active display tab from the list of plotted or imported items in all the tabs. To add an item, you must select an Expression to use from the expandable list of plotted items. If you want to use an item that has been exported from another simulation plot, you must import it first, see Exporting and importing plotted items on page 218.</p> <p>In addition, you can specify:</p> <ul style="list-style-type: none"> – Name – Specifies the name that will appear in the name column. Name must start with a character of “_”. – Expandable - Specifies if the item will include sub-items. – Value type – (Signed, Unsigned, Float) Specifies the value type for the plotter display. The default value for the selected expression is Unsigned. – Show as – (Bus, Digital) specifies the display format for the item as a bar containing a value (Bus), or a wave form indicating the value as high or low (Digital). – Time Translation – Specifies in nanoseconds, the point in the simulation plot where the plotter will start drawing the specified item. Default is zero: start at time=0. Entering a value less than zero shifts the start point for the specified item to the left of the start of the simulation (time=0). Entering a value a value greater than zero shifts the start point to the right of the start of the simulation (time=0). – Time Scale – Allows the re-scaling of the expression for an item. The default value is 1: no re-scaling. Entering a value less than 1 compresses the time scale so that the item's value changes occur more rapidly. Entering a value greater than 1 extends the time scale so that the item's value changes occur more slowly.
In the Edit menu	
Cut, Copy and Paste	Allows you to cut or copy a selected item and paste a copy of it into a tab. You can modify the expressions of items that have been copied in this manner. See Modifying expressions on page 217 .
Delete Selected Item	Removes the item that you have selected in the item column from the display in the active tab. If an item is no longer present in at least one of the display tabs, then it is removed from the list of selected items to plot in the Plotter selection window .
Delete All Items of Group	Removes all items from the active tab from the display. If items are no longer present in at least one of the display tabs, then they are removed from the list of selected items to plot in the Plotter selection window .

Modifying item display characteristics

To improve the readability of your plot, you can modify the display characteristics including the item name, color, and size. If you have copied an item to another tab, the changes to the display properties of this item are only applied to it on the active display tab.

Table 69. Item display controls

Command	Description
Edit menu	
Rename	Changes the name of the selected item. You cannot rename an item to have the same name as another item in the tab. Name must start with a character or underscore (_).
Properties	Changes the display properties of the selected item, including: <ul style="list-style-type: none"> – Name – Specifies the display name found in the Name column. Name must start with a character or underscore (_). This can also be changed by selecting Edit>Rename. – Visualization format – Specifies the format for the display of values in the plot and the Value column as Signed or Unsigned Decimal, Binary, Octal, Hexadecimal, or ASCII. – Height – Sets the height of the bar or wave form. – Color – Defines the color of the bar or wave form. – Apply to all check box allows you to apply the properties to all the items in the plot.
Default properties	Sets display properties for the selected item back to the default properties.
View menu	
Signals>Show as	Determines the item display type: <ul style="list-style-type: none"> – Bus – a bar containing the integer or float value of an item. – Digital – a continuous line that indicates the item value as high or low.

Modifying expressions

The plotter's graphical representations of items and values are based on "expressions", which are text descriptions of the information to be plotted. This use of expressions makes it possible to create new items based on existing expressions (see [Creating and deleting display tabs and items on page 215](#)) and export expressions to a text file for import into a later simulation in STVD or into another software (see [Exporting and importing plotted items on page 218](#)).

The time translation (start point), time scale and expression can be changed for copied or imported items. It is not possible to modify the expressions for items that are updated during simulation or for sub-items.

To change the expression for the selected item, select **Edit>Change Expression**. In the **Change Expression** dialog box, you can modify the expression for an item by specifying:

- **Time Translation** – specifies in nanoseconds, the point in the simulation plot where the plotter will start drawing the specified item. Default is zero: start at time=0. Entering a value less than zero shifts the start point for the specified item to the left of the start of

the simulation (time=0). Entering a value a value greater than zero shifts the start point to the right of the start of the simulation (time=0).

- **Time Scale** – allows the re-scaling of the expression for an item. The default value is 1: no re-scaling. Entering a value less than 1 compresses the time scale so that the item's value changes occur more rapidly. Entering a value greater than 1 extends the time scale so that the item's value changes occur more slowly.
- **New Expression** – specifies an expression to use for the selected item from the list of expressions used in the current plot. If you want to use an expression from another simulation plot, you must first import the expression into your current plot (see [Exporting and importing plotted items](#)).

Exporting and importing plotted items

The plotter display is a graphical representation of value changes for a set of items. Value changes for items can be saved to a text file in **value change dump** (VCD) format (*.vcd) for use in later STVD simulation plots. VCD files can be opened and edited in a text editor. They can also be exported to and imported from other software that use this format.

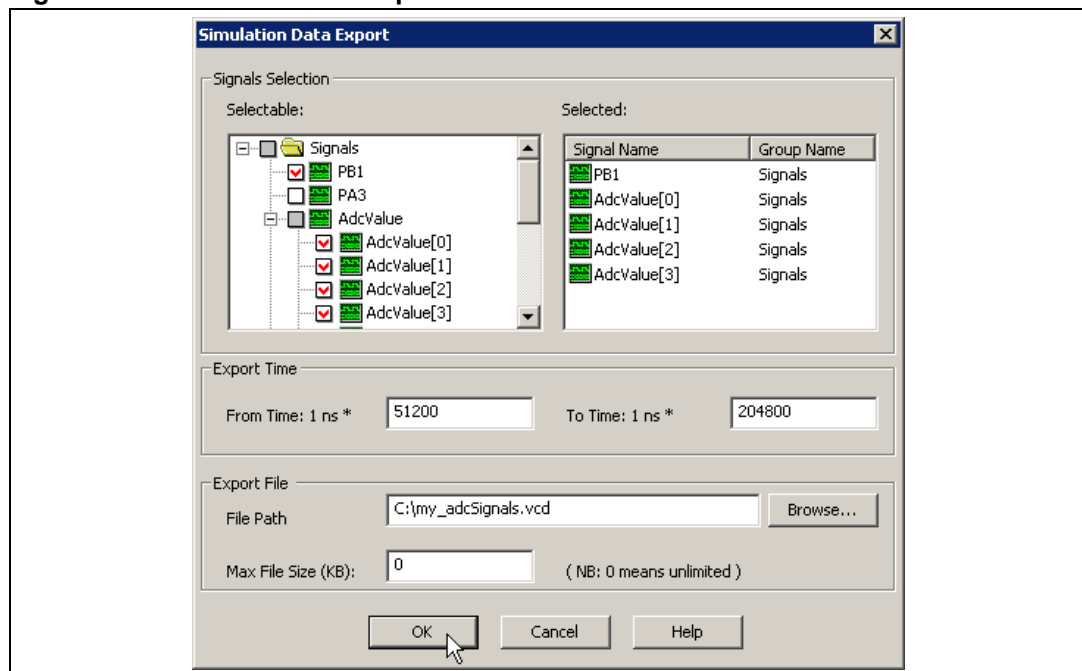
You can easily select only the items that you want to export and import.

To **Export** plotted information for selected items:

1. Select **File>Export** to open the **Simulation Data Export** dialog box.

The **Selectable** field, displays an expandable list of all the items in the current plot. The different display tabs are represented by folders.

Figure 159. Select items for export in VCD format



2. Select the items you want to export. To select all the items in a display tab, place a checkmark next to the corresponding folder. Otherwise, place a checkmark next to each item or sub-item that you want to export.
The items that you select are placed in the **Selected** field. The checkbox next to a folder or item is blue if any items or sub-items that they contain are selected for exporting.
3. Enter the time range that you want to export (*optional*). By default, the start time is 0, and the end time is the end of the current plot.
4. Select the file that you want to export to.
5. Enter a size limit (*optional*). The default setting is 0 for "no size limit."
6. Click on **OK**.

To **Import** plot information for selected items:

1. Select **File>Import** to open the **Simulation Data Export** dialog box.
2. In the **Open** dialog box, select the file that you want to import from and click on **Open**.
3. In the **Select Signals to Import** dialog box, select the items that you want to import from the list.

You may need to specify a Time Translation (start point where the plotter with start drawing the imported item in the current plot) and/or change the time scaling if the time scale is a unit other than *ns*. To do this:

- a) Place a checkmark in the **Define X Translation and/or Scale** checkbox.
- b) Then enter:

Time Translation (in *ns*) specifying the point in the simulation plot where the plotter will start drawing the specified item. Default is zero. Entering a value less than zero shifts the start point for the specified item to the left of the start of the simulation (time=0). Entering a value a value greater than zero shifts the start point to the right of the start of the simulation (time=0).

and/or the **Time Scale**. The default value is 1: no re-scaling of the original expression. Entering a value less than 1 compresses the time scale so that the item's value changes occur more rapidly. Entering a value greater than 1 extends the time scale so that the item's value changes occur more slowly.

4. Click on **OK**.

The imported items are added to the default tab, **Signals**.

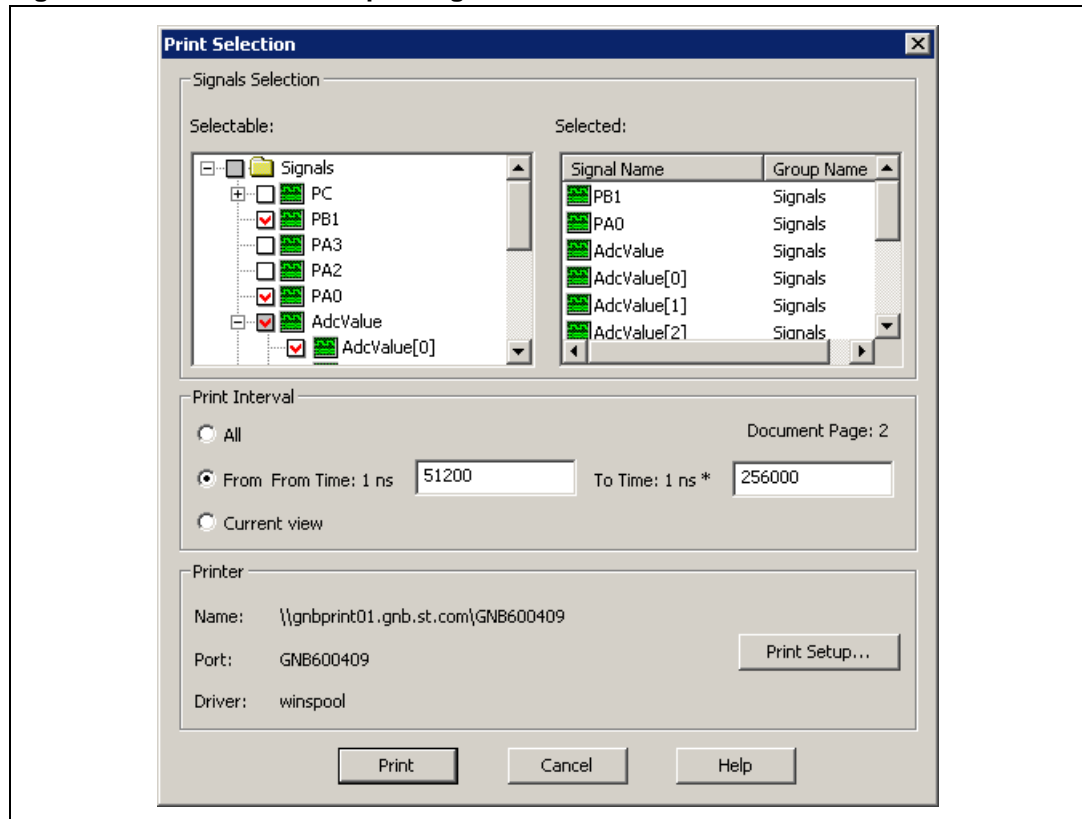
Printing a plot

You can easily select only the items that you want to print, even if the items are on different display tabs.

To **Print** plotted information for selected items:

1. Select **File>Print** to open the **Print Selection** dialog box.
The **Selectable** field, displays an expandable list of all the items in the current plot. The different display tabs are represented by folders.

Figure 160. Select items for printing

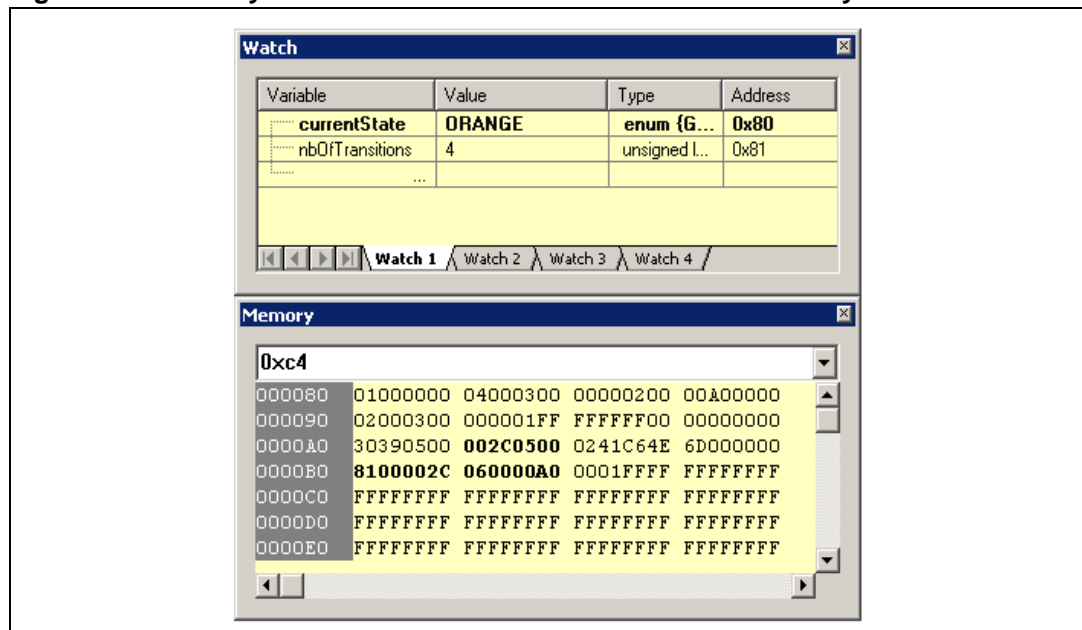


2. Select the items that you want to print. To select all the items in a display tab, place a checkmark next to the corresponding folder. Otherwise, place a checkmark next to each item or sub-item that you want to print.
The items that you select are placed in the **Selected** field. The checkbox next to a folder or item is blue if any items or sub-items that they contain are selected for exporting.
3. Enter the time range that you want to print. Three options are available:
 - **All**: prints the selected items for the full interval of the plotted simulation.
 - **From time ____ to time ____**: prints the selected items within the interval that you specify. Times are specified in nanoseconds.
 - **Current view**: prints the selected items within the interval that is displayed in the current view.
4. Select your printer.
5. Click on **Print**.

6.4 Using read/write on-the-fly

The **Read/Write on the Fly** option permits you to non-intrusively read or write a value in the RAM memory of the simulated microcontroller while the application is executing.

When this option is selected in either the **Memory** window or the **Watch** window, all of the memory entries shown in the window are constantly refreshed while the program is running. In these windows, the background turns yellow if the **Read/Write on the Fly** option is active.

Figure 161. Memory and Watch windows with Read/Write on the fly

Note that when you scroll in the **Memory** Window, the newly visible memory entries turn bold, to show that they are being updated, then change back to normal, indicating that they are current and valid.

When the **Read/Write on the Fly** option is active, the only time that the values shown in the **Memory** window may not be valid is during the direct editing of a memory value during the running of a program. In this event, the edited value appears in red, indicating that it has not yet been taken into account by the program, it then turns to bold black, to indicate that the program is adjusting accordingly, and then finally, all entries are in normal weight black, indicating that the window is valid and updated.

6.5 Forcing interrupts

In the Simulator, you can generate interrupt requests using pseudo pins named IRQx, where x is the interrupt number. This is done by changing the value of the IRQx pseudo pin, either from the from the **I/O Stimulation** window or in the stimuli file.

This feature is useful because it allows you to debug code for peripherals that are not supported by the Simulator. And in the case of supported peripherals, it allows you to trigger an interrupt whenever you want, without waiting for the interrupt conditions to be met.

The prerequisites for using this feature are:

- Interrupts must be globally enabled in the condition code register.
- The interrupt handler corresponding to the IRQ number must be defined, and its address correctly set in the interrupt vector.

Note that if there is no interrupt handler at the address pointed by the interrupt vector corresponding to the interrupt request number, the program will crash.

You can use any kind of signal (digital, analog or periodic), with a delay if you want, to generate an interrupt. Refer to [Section 6.2: Using the input pin stimulator on page 203](#) or

Create and load a stimuli file on page 206 to trigger a signal on an interrupt request pseudo pin.

The IRQx pseudo pin corresponds to the interrupt request number x. Refer to the interrupt mapping table in the datasheet for your simulated microcontroller, to find the interrupt request number for the interrupt that you want to generate. As an example, [Table 70](#) provides the interrupt mapping for the ST7LITE1xB family.

Table 70. Interrupt mapping

N°	Source block	Description	Register label	Priority order	Exit from HALT or AWUFH	Address vector
	RESET	Reset	N/A	<div>Highest priority</div> <div></div> <div>Lowest priority</div>	yes	FFFEh-FFFFh
	TRAP	Software interrupt			no	FFFCh-FFFDh
0	AWU	Auto wake up interrupt	AWUCSR		yes	FFFAh-FFFBh
1	ei0	External interrupt 0	N/A		yes	FFF8h-FFF9h
2	ei1	External interrupt 1				FFF6h-FFF7h
3	ei2	External interrupt 2				FFF4h-FFF5h
4	ei3	External Interrupt 3				FFF2h-FFF3h
5	LITE TIMER	LITE TIMER RTC2 interrupt	LTCSR2		no	FFF0h-FFF1h
6	Comparator	Comparator interrupt	CMPCR		no	FFEEh-FFEFh
7	SI	AVD interrupt	SICSR		no	FFEC h-FFEDh
8	AT TIMER	AT TIMER output compare interrupt or input capture interrupt	PWMxCSR or ATCSR		no	FFEAh-FFEBh
9		AT TIMER overflow interrupt	ATCSR		yes	FFE8h-FFE9h
10	LITE TIMER	LITE TIMER input capture interrupt	LTCSR		no	FFE6h-FFE7h
11		LITE TIMER RTC1 interrupt	LTCSR		yes	FFE4h-FFE5h
12	SPI	SPI peripheral interrupts	SPICSR		yes	FFE2h-FFE3h
13	AT TIMER	AT TIMER overflow interrupt	ATCSR2		no	FFE0h-FFE1h

For example, for ST7LITE1B to generate an auto-reload timer counter 2 overflow interrupt, you must change the value of pin IRQ13. If the interrupts are globally enabled, the program will jump to the counter 2 overflow interrupt handler after the value on the interrupt pin has changed (after execution of the current instruction, if the change is requested without delay).

7 In-circuit debugging

In-circuit debugging (ICD) allows you to debug an application in its final environment by communicating with the microcontroller. There are two protocols for in-circuit debugging:

- ICC (in-circuit communication) for ST7 microcontroller families
- SWIM (single wire interface module) for STM8 microcontroller families

With the necessary connection hardware, these protocols enable STVD to program and read your microcontroller's Flash memory, and control the running of your application on your microcontroller. Once you have configured and programmed your microcontroller, you can debug your application using the microcontroller's on-chip debug modules.

Available in-circuit debugging features depend on your MCU's memory type (HDFlash, XFlash, RAM or ROM) and its debugging resources, notably the number of debug modules. In this section, you will find information about:

- [Section 7.1: Connecting to and configuring the microcontroller](#)
- [Section 7.2: Using breakpoints](#)
- [Section 7.3: Creating a break on trigger input \(TRIGIN\)](#)
- [Section 7.4: In-circuit debugging in hot plug mode \(SWIM only\)](#)
- [Section 7.5: In-circuit debugging limitations](#)

Overview of in-circuit debugging with ICC

ST7 microcontrollers include a dedicated hardware cell (the debug module) for in-circuit debugging, and also a software monitor that manages ICC communications.

When you enter a debugging session, STVD resets your microcontroller in **ICC mode**. The entry into ICC Mode starts the **ICC monitor** — an on-chip firmware that is located in the system memory and manages the ICC protocol. STVD, interfacing via the ICC monitor, programs your microcontroller and allows you to configure it by setting the values of the option bytes.

To debug your application, STVD then resets your microcontroller in **User mode**. However, your microcontroller continues to run the ICC monitor, thereby allowing you to:

- control the running of your application
- set breakpoints by using **TRAP instructions** or **debug modules**
- read and write variables, memory contents and **core registers** when the application is stopped

Overview of in-circuit debugging with SWIM

STM8 microcontrollers include a debug module for in-circuit debugging, and also a SWIM hardware cell that manages communications with the host.

When you enter a debugging session, the SWIM entry sequence activates the communication channel on a single pin. You can then:

- control execution of your application through the debug module (step, break, abort)
- set software breakpoints by using a dedicated instruction
- use the advanced breakpoints offered by the debug module

In-circuit debugging with SWIM offers the following advantages:

- Non-intrusive access to the CPU bus: no dedicated interrupt vector, no software monitor, read/write access on-the-fly to RAM and peripheral registers, and to the Flash memory when the application is stopped.
- Possibility to freeze peripherals when the application is stopped (depending on your MCU, refer to datasheet).
- Possibility of starting a debug session in hot plug mode (without resetting the microcontroller).

7.1 Connecting to and configuring the microcontroller

To connect to your MCU for ICD, you must:

1. Make sure that you have the correct hardware configuration.
2. Correctly identify your MCU in the **ICD MCU configuration** window in STVD.
3. Choose to establish the connection with the MCU while ignoring the current option byte values, or to take them into account.

7.1.1 Connecting the hardware for in-circuit debugging

Hardware connection for ICC protocol

For STVD to connect to the ST7 microcontroller on your application board, you must install an ICC connector on your application board, which relays the necessary signals to your microcontroller. You must then connect the ICC cable between your application board and one of the following devices:

- The ICC add-on of an ST Micro Connect box (ST7-EMU3 emulator)
- The ICC connector of an ST7-DVP3 emulator
- A Raisonance RLink in-circuit debugger/programmer

For more information about setting up this hardware connection, refer to your hardware user manual:

- *ST7-EMU3 Emulator User Manual* (for ST Micro Connect with ICC add-on)
- *ST7-DVP3 Emulator User Manual*

Hardware connection for SWIM protocol

For STVD to connect to the STM8 microcontroller on your application board, you must install a SWIM cable which relays the signals between your microcontroller on your application board and your hardware (STice box, STLink or RLink).

The SWIM cable is connected at one end to your application board through an ERNI connector, and at the other end, to the STice box via the SWIM connector board, or directly to the STLink or RLink.

For more information about setting up this hardware connection for STice, refer to the *STice advanced emulation system for ST microcontrollers User Manual* (UM0406).

7.1.2 Selecting your MCU

The **ICD MCU configuration** dialog box provides the interface for selecting your MCU and for configuring your MCU's option bytes.

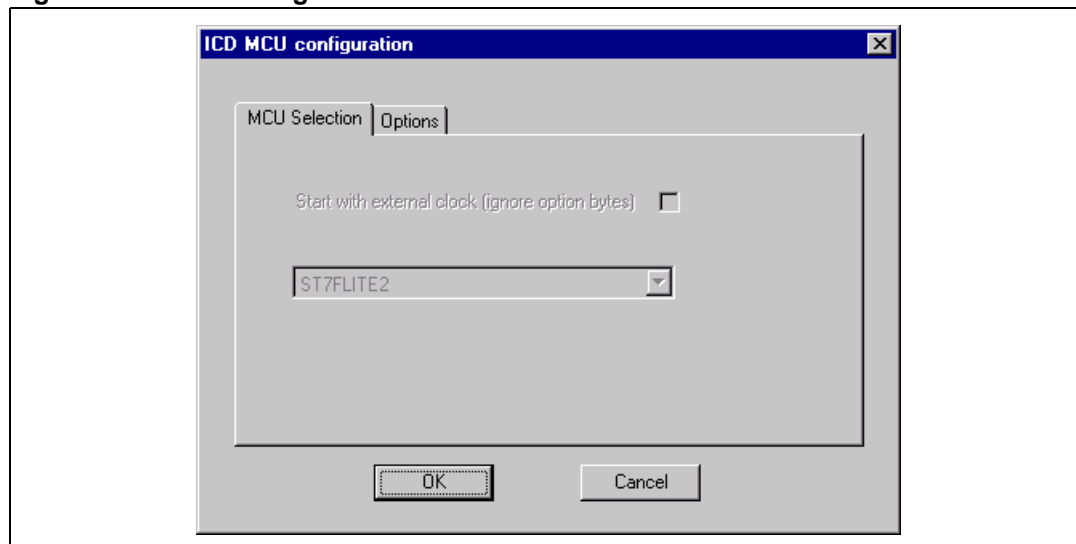
If you have an open workspace, when you start a debugging session, this dialog box is displayed only if STVD *cannot* connect to the microcontroller defined in your project settings. If the connection is established, the dialog box is not displayed, but you can access it by selecting **Tools>MCU Configuration**.

If you do not have an open workspace, when you start a debugging session by selecting **Debug>Start Debugging**, this dialog box is automatically displayed.

The **ICD MCU configuration** window (shown in [Figure 162](#)), offers two tabs:

- **MCU Selection** tab
- **Options** tab

Figure 162. MCU configuration window



MCU selection tab

When you have a project loaded in your STVD workspace, the MCU selection is determined by your project's settings (see [Section 4.5: Configuring project settings on page 84](#)). In this case, when you open the ICD configuration window, the MCU for your project is displayed in the MCU selection field as shown in [Figure 162](#). The field is gray and you cannot access the list of MCUs.

If you do not have a project loaded, you have access to the list of microcontrollers that support in-circuit debugging. Select **Debug>Start Debugging** without creating or loading a workspace or project. Once the session has started, select **Debug Instrument>MCU Configuration** from the main menu bar.

In the **ICD MCU configuration** window, you must select a microcontroller that corresponds to the one present on your application board. When you highlight a microcontroller in the list,

STVD attempts to connect to the microcontroller on your application board. If the connection fails, an error message is displayed, and you must change your MCU selection.

Caution: The MCU selection in the **ICD MCU configuration** window must match the MCU selected in your project settings. If it doesn't, you must open the **Project Settings** window, select the appropriate MCU, and rebuild your application.

Note: When the dialog box opens for the first time, only the MCU selection tab is available.

Options tab

This interface lists the option bytes and their current values, and allows you to configure the values from a list of available settings for your MCU (see [Section 7.1.4: Configuring option byte settings](#)).

7.1.3 Ignoring option bytes (ICC only)

When establishing the initial connection with your microcontroller, option bytes may present a problem if some settings are not consistent with your application. Inconsistent settings can result in an MCU failure. For example, the clock source option byte must identify the correct clock source. If the option byte is set to identify an external clock that is not in your application, then your MCU won't start for lack of a clock signal.

As a security, to avoid this kind of failure, STVD can start your microcontroller using the external clock signal provided via the ICC add-on. This allows STVD to establish communication with your microcontroller without taking into account the current values of the MCU's option bytes.

To do this, check the **Start with external clock (ignore option bytes)** checkbox in the **ICD MCU configuration** dialog box (see [Figure 162](#)). Once STVD has established communications with your MCU, you can confirm and reconfigure the option byte settings in the **Options** tab of the **ICD MCU Configuration** dialog box, if necessary.

- Note:*
- 1 In order to provide an external clock signal to start your MCU, the MCU's clock pin must be relayed to the OSC_CLK pin of the HE10 connector on your application board. For information about this connection, refer to your hardware user manual.
 - 2 If this option is not supported by your microcontroller, it is grayed out as shown in [Figure 162](#), and you cannot check the check box.

7.1.4 Configuring option byte settings

Option bytes give you control of a range of ST microcontroller features such as low voltage detection, oscillator source and range, watchdog behavior and memory read/write protection. Option byte features vary from one microprocessor to another. For more information about these features for your target MCU, refer to the corresponding datasheet.

Note: You can only configure option byte settings once STVD has established the connection to your microcontroller.

Option bytes and ICD compatibility

Some option byte settings are not compatible with in-circuit debugging and, therefore, must be configured to be consistent with ICD requirements. Option byte settings that are not compatible with ICD are listed in [Table 71](#).

Table 71. ICD-incompatible option byte settings

Product memory type	STM8 MCU / SWIM protocol	ST7 MCU / ICC protocol
All memory types	Global readout protection enabled Hardware watchdog enabled Flash memory write protection enabled	Hardware watchdog (WDG_HW) enabled
XFlash, HDFlash		Flash memory read-out protection (FMP_R) enabled
XFlash		– Flash memory write protection (FMP_W) enabled – Sector 0 (SEC [1:0]) other than minimum

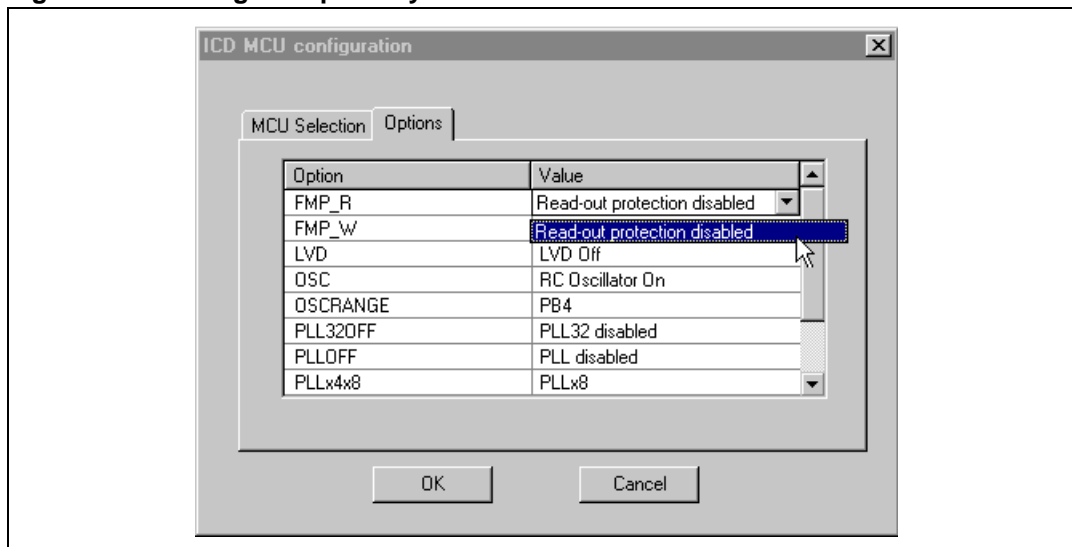
Caution: For XFlash devices, if write protection is enabled, the XFlash memory behaves like ROM memory. You will not be able to debug your application and it is impossible to rewrite the XFlash memory.

Handling incompatible option byte settings

When STVD establishes communication with your target MCU, it first reads the option byte values from your MCU and detects settings that are incompatible with ICD. Detection of incompatible option byte settings results in an error message. If this occurs:

1. Click on **OK** in the error message window.
The **ICD MCU configuration** window opens automatically, so you can reconfigure the option bytes.
2. Select the **Options** tab to view the current option byte settings (see [Figure 163](#)).
The current settings (including ICD-incompatible settings) are in the **Value** field corresponding to each option byte.
3. Click in the **Value** field to see possible settings for any option byte. Select a new setting for any ICD-incompatible settings, then click on **OK** to accept the configuration and close the window (see [Figure 163](#)). If you press **Cancel** at this point, any changes you have made are ignored.

Figure 163. Setting the option byte values



Note: After changing an option byte setting that is not compatible with ICD, the original setting is no longer visible in the **Value** field. Only settings that are compatible with ICD can be viewed and selected. You will have to use a programming tool if you need to change option byte settings to an ICD-incompatible setting.

7.2 Using breakpoints

When doing in-circuit debugging with STVD, you can use advanced breakpoints (see [Section 7.2.3: Setting advanced breakpoints](#)), in addition to simple instruction breakpoints. Advanced breakpoints stop the application when the conditions that you specify are met whereas instruction breakpoints simply stop the application when it reaches a specified line of code.

Breakpoints are generated in two ways:

- **Software breakpoints** rely on trap instructions to create instruction breakpoints with ICC, and on a dedicated instruction with SWIM
- **Hardware breakpoints** rely on the MCU's debug module(s) for instruction and advanced breakpoints

The number of instruction or advanced breakpoints allowed at the same time depends upon the type of microcontroller you are emulating, the number of debug modules it has and the type of memory used (see [Example](#)).

7.2.1 Software breakpoints

By default, when you set an instruction breakpoint, STVD uses software breakpoints. An unlimited number of software breakpoints can be used in address ranges corresponding to memory that is writable byte-by-byte (such as RAM and XFlash memory).

However, software breakpoints cannot be used in memory areas that are not writable byte-by-byte (such as HDFlash, XFlash sector 0 and ROM memory areas). Therefore, instruction breakpoints in those memory areas rely on the MCU's debug modules and hardware breakpoint technology.

Note: ST7 MCUs with XFlash memory do not allow software breakpoints in memory sector 0. This sector cannot be written to when the microcontroller is in USER mode. In some cases, the size of sector 0 can be configured. If this is the case, STVD requires sector 0 to be programmed to its minimum size.

With the ICC protocol, software breakpoints are created by introducing a TRAP instruction, which causes the application to stop running. For this reason, STVD cannot be used for in-circuit debugging of ST7 applications that contain TRAP instructions in the code.

With the SWIM protocol, there is a dedicated software break instruction. This instruction stalls the core, and the peripherals that you previously specified on the **Options** tab in the MCU configuration window are frozen by the debug module.

For in-circuit debugging with SWIM, there is no restriction on using TRAP instructions in your code. However, you must not use software breakpoints on the BOOT area (write protected pages), and it is not possible to debug the protected code area (readout protected pages).

7.2.2 Hardware breakpoints

All microcontrollers that support in-circuit debugging contain debug modules. The number of debug modules determines how many hardware breakpoints you can place at any given time. Each debug module allows you to place two instruction breakpoints, or one advanced breakpoint.

Allocating hardware breakpoint resources

If you have placed an instruction breakpoint (as explained in [Section 5.8.1: Setting an instruction breakpoint on page 183](#)), which uses up part of a debug module, it appears in the **Instruction Breakpoints** field to the left of the **Advanced Breakpoints** window shown in [Figure 165](#). From this field, you can disable the instruction breakpoints, in order to place an advanced breakpoint.

In addition, the stepping commands **Step Over**, **Step Out** and **Run to Cursor**, require the use of a hardware-based instruction breakpoint when using an MCU with ROM or HDFlash memory. If you do not have enough hardware breakpoints available when using these commands, STVD will have to temporarily disable breakpoints to execute these commands. STVD will warn you that it is disabling your instruction, or advanced breakpoints in order to perform the command. Click on **OK** to continue. Once the command has been executed, STVD automatically re-enables your breakpoints.

Example

Imagine that you are debugging your application on an ST7 microcontroller that has the following characteristics:

- 1024 bytes of RAM memory in the address range 0080h to 047Fh,
- 32 Kbytes of program memory in the address range 8000h to FFDFh,
- 3 debug modules.

You will be able to place the following breakpoints:

- an unlimited number of software instruction breakpoints in the RAM memory address range (0080h to 047Fh)
- up to six hardware instruction breakpoints in the program memory address range (8000h to FFDFh), **or** up to 3 hardware advanced breakpoints in *any* address range.

7.2.3 Setting advanced breakpoints

Depending on your target microcontroller and its debug modules, you can set advanced breakpoints to stop the execution of your application when specific conditions are met.


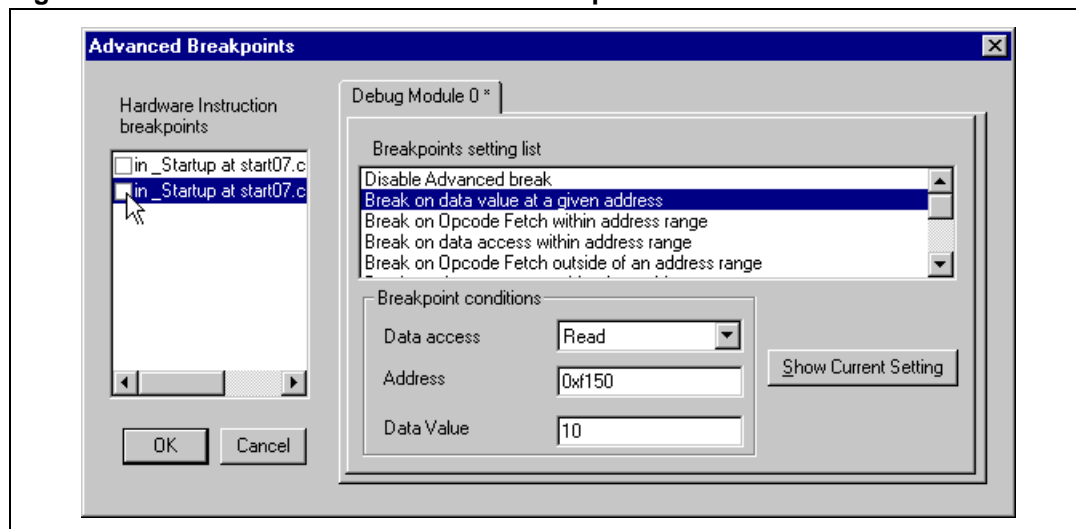
To set an advanced breakpoint, open the **Advanced Breakpoints** window by selecting **Debug Instrument > Advanced Breakpoints**, or click on the Advanced Breakpoints icon -  - in the Emulator toolbar.

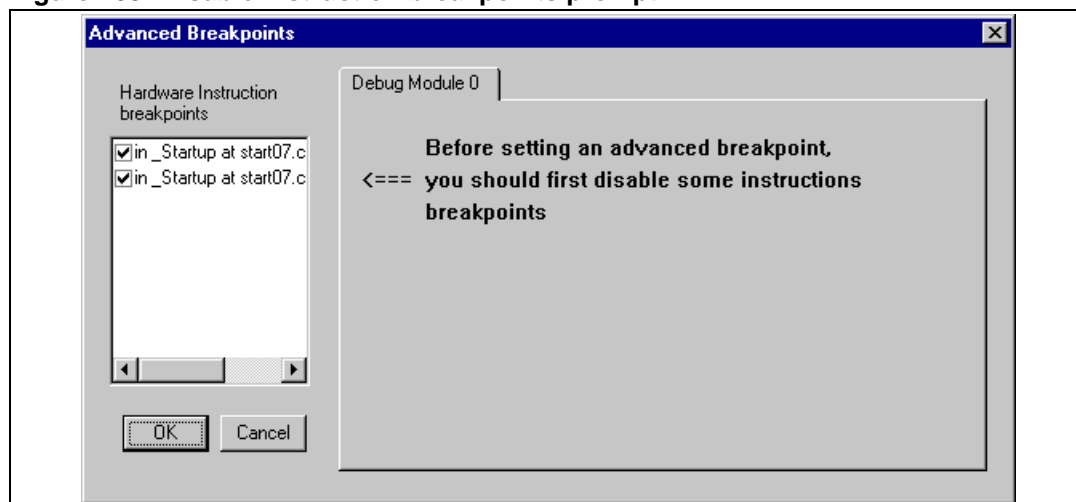
Figure 164. ST7-ICD emulator advanced breakpoints window



In the **Advanced Breakpoints** window, there is a tab for each debug module present on your target microcontroller. [Figure 164](#) shows the window for an ST7 MCU with one debug module. The **Hardware Instructions Breakpoints** list on the left side of the window lists any instruction breakpoints that are already placed and limits the number of advanced breakpoints you can use.

When you open the **Advanced Breakpoints** window, if you have set instruction breakpoints that use all of your MCU's debug modules, STVD prompts you to disable these breakpoints before you can activate an advanced breakpoint, as shown in [Figure 165](#). The breakpoints that you can disable are listed in the **Hardware Instruction breakpoints** field.

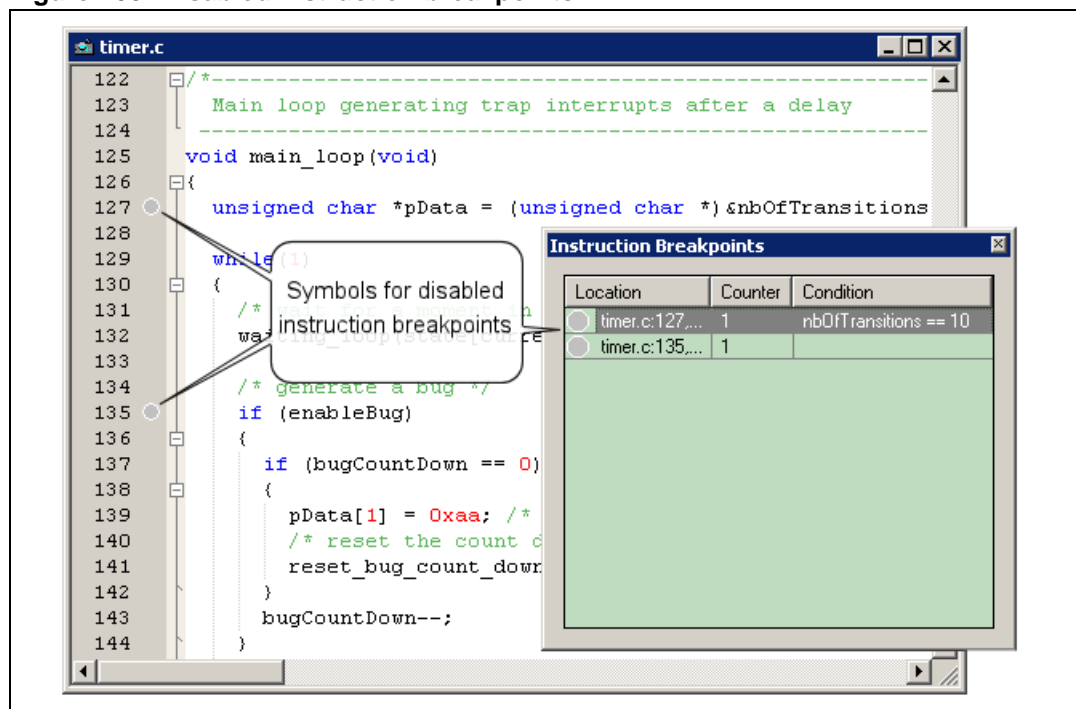
Figure 165. Disable instruction breakpoints prompt



Disable instruction breakpoints by clicking on the checkboxes next to them. Once you have disabled enough instruction breakpoints to free up a debug module, the advanced breakpoint settings will become available, as shown in [Figure 164](#).

Choose an advanced breakpoint from the list and define its settings in the **Breakpoint conditions** field. Breakpoint settings are described in [Section 7.2.4: Advanced breakpoint options](#). Click on **OK** to activate the advanced breakpoint (see [Figure 164](#)). Once activated, the symbols for any instruction breakpoints you have disabled will automatically change to the disabled state (see [Figure 166](#)).

Figure 166. Disabled instruction breakpoints



The instruction breakpoints remain disabled until you reactivate them manually. For more information, refer to [Section 5.8: Instruction breakpoints on page 183](#).

7.2.4 Advanced breakpoint options

In each debug module tab, you can program the following breakpoints settings:

- **Disable Advanced break.** Advanced break function is off.
- **Break on data value at a given address.** You must specify:
 - the type of data access (Write, Read, Read/Write)
 - the address
 - the data value
- **Break on Opcode Fetch within address range.** This option allows you to break on the occurrence of any Opcode Fetch within a specified address range. You must specify:
 - the address at the beginning of the range
 - the address at the end of the range.
- **Break on data access within address range.** This option allows you to break on the occurrence of a data access within a specified address range. You must specify:
 - the type of data access (Write, Read, Read/Write)
 - the address at the beginning of the range
 - the address at the end of the range.
- **Break on Opcode Fetch outside of an address range.** This option allows you to break on an Opcode Fetch that occurs *outside* of a specified address range. You must specify:
 - the address at the beginning of the range
 - the address at the end of the range
- **Break on data access outside of an address range.** This option allows you to break on a specified data access type outside of a specified address range. You must specify:
 - the type of data access (Write, Read, Read/Write)
 - the address at the beginning of the range
 - the address at the end of the range.
- **Break on two sequential Opcode Fetch's.** This option allows you to flag two Opcode Fetch instructions at two addresses. A break will occur when the two flagged Opcode Fetch's have occurred in sequence; the occurrence of the first Opcode Fetch must precede the occurrence of the second Opcode Fetch.
- **Break on data access at one of two addresses.** This option allows you to break on the occurrence of a data access at either of two addresses. You must specify:
 - the type of data access (Write, Read, Read/Write)
 - the first of the two addresses
 - the second of the two addresses
- **Break on Opcode Fetch OR data access at given addresses.** This option allows you to break on either the occurrence of an Opcode Fetch at a specified address, or the

occurrence of a data access at a specified address. (The addresses specified for each instance can be different.) You must specify:

- the type of data access (Write, Read, Read/Write)
- the address where the Opcode Fetch must occur
- the address where the data access must occur
- **Break on conditional stack write OR Opcode Fetch at given address.** This option allows you to break on the occurrence of either a write operation to the stack below a specified address, or an Opcode Fetch at a specified address. You must specify:
 - the maximum stack address below which the write operation must occur
 - the address where the Opcode Fetch must occur
- **Break on conditional stack write OR data access at given address.** This option allows you to break on the occurrence of either a write operation to the stack below a specified address, or a data access at a specified address. You must specify:
 - the type of data access (Write, Read, Read/Write)
 - the address where the data access must occur
 - the maximum stack address below which the write operation must occur

7.3 Creating a break on trigger input (TRIGIN)

You can connect an input signal via the IN trigger connector on the front panel of the ST Micro Connect box. The input signal must be between 0 and 5 volts. The input signal causes a break in the application **only** upon reception of a rising edge signal at TTL level.

The input trigger signal is only supported for in-circuit debugging over ICC with the ST Micro Connect box.

7.4 In-circuit debugging in hot plug mode (SWIM only)

In-circuit debugging in hot plug mode is intended to allow you to take control over the microcontroller on your application board while your application is running, without resetting the microcontroller.

To start a debug session in hot plug mode, you must choose this option when you select your debug instrument. This option is available when you do not have a loaded workspace, and only with the **SWIM RLink**, **SWIM STLink** and **SWIM STice** choices for the debug instrument.

To start a debug session in hot plug mode, follow these steps:

1. Power up the STice box/RLink/STLink, depending on the hardware that you are using.
2. Connect the SWIM cable between the STice box (or RLink or STLink) and the SWIM connector on your application board.
3. Select **Debug Instrument>Target Settings** from the main menu bar.
The **Debug Instrument Settings** window is displayed.
4. From the **Debug Instrument Selection** list box select **SWIM RLink**, **SWIM STLink** or **SWIM STice** depending on your hardware.
5. Enable the **Hot Plug Start Debug** checkbox.
6. Select the communication port (parallel, USB or Ethernet) that your hardware is connected to.
7. Click on **OK** to dismiss the **Debug Instrument Settings** window.
8. Check your MCU selection.
See [Section 7.1.2: Selecting your MCU on page 225](#).
9. From the **Debug** menu, select **Start Debugging** to start a debugging session.

STVD starts running the application, without debugging information, while offering the following debugging features: abort, step, advanced breakpoints, memory dump, register window, read/write on-the-fly on RAM and peripherals.

In hot plug mode, you cannot set software breaks, flash program your microcontroller or perform symbolic debugging.

The constraints on the application are:

- The SWIM_DBG pin must be reserved for debugging (it cannot be used for I/O)
- On the STice, there is no pull-up on the reset pin, therefore you must ensure that there is a pull-up on reset in the application. Pull-up on the reset pin is provided in the RLink/STLink hardware.

If your application includes a number of periodic resets (such as watchdog timers), you might note a difficulty in establishing the connection to the microcontroller. Standard debugging mode is more appropriate for this kind of application.

7.5 In-circuit debugging limitations

In-circuit debugging limitations that are specific to a microcontroller are listed in the **Discrepancies** window. You can access it by selecting **Debug Instrument>Emulation Discrepancies**.

The general limitations described in this section apply to all target microcontrollers when in-circuit debugging.

Stack limitation (ICC only)

For in-circuit debugging, bytes must be reserved on the stack in addition to the memory reserved for the application. When evaluating the stack size required for your application, add 5 bytes. Add 5 more bytes (total of 10 bytes) for target MCUs that support software breakpoints.

TRAP instruction limitation (ICC only)

The TRAP interrupt vector is reserved for the ICC monitor during debugging and TRAP instructions must not be used in the application.

Peripheral limitation (ICC only)

On some old devices, when the execution of your application is stopped from STVD while in-circuit debugging, the peripherals (notably, the timers) are not frozen.

Reset limitation

The reset is not real-time.

In ICC the microcontroller executes the ICC monitor after a reset until the host debugger sends a “continue” command.

In SWIM the microcontroller remains stalled until the host debugger sends a “continue” command.

External reset limitation

Hardware on the application board must not generate external resets when the application is stopped. The effect is different depending on the debugging protocol.

When debugging over ICC, an external reset results in the loss of synchronization between STVD and the application. If this occurs, select **Debug>Stop Debugging** and then **Debug>Start Debugging** to restart your in-circuit debugging session.

When debugging over SWIM, an external reset may cause a SWIM communication error (timeout). Communication is re-established when the reset is released.

Address 0x80-0x81 overwrite on reset (ICC only)

When starting the user application (**Debug>Run**) after a chip **reset**, the ICC monitor automatically overwrites any values stored at 0x80-0x81 with values corresponding to the microcontroller's die identifier.

HDFlash devices Flash memory limitation (ICC only)

On devices with HDFlash memory, the **ICC Monitor** must be copied to a sector of Flash memory because the system memory does not contain the advanced ICC monitor that is required for in-circuit debugging. This zone of Flash memory will not be available for your application. If your application attempts to write to this memory zone you will receive an error message indicating that the application has tried to write to a reserved memory zone. The affected devices and memory zone are listed in [Table 72](#).

Table 72. Flash memory limitation on HDFlash devices

Affected microcontrollers	Address range of reserved memory zone
ST7FLCD1	FF00 - FFDF
ST7FMC2	FF00 - FFDF
ST72F325	FF00 - FFDF

Table 72. Flash memory limitation on HDFlash devices (continued)

Affected microcontrollers	Address range of reserved memory zone
ST7FLITES2/5	FF12-FFDF
ST7FLITE02/5/9	FF12-FFDF

SWIM debugger limitation

While the microcontroller is executing either a "Halt", "WFI" or "WFE" instruction, or protected code, no core resource is accessible for the debugger. As a result, the debugger loses any control over the core while it remains in this state (especially, no possibility to stop it or make it leave this state), and read/write on the fly is no longer possible during this period.

8 DVP and EMU2 (HDS2) emulator features

This section explains how to use the features that are specific to the ST7 DVP or EMU2 emulator, including:

- [Section 8.1: Working with output triggers](#)
- [Section 8.2: Using hardware events](#)
- [Section 8.3: Trace recording](#)
- [Section 8.4: Using hardware testing](#)
- [Section 8.5: Logical analyser \(EMU2 emulators only\)](#)
- [Section 8.6: Stack control window \(DVP emulators\)](#)
- [Section 8.7: Trigger/trace settings \(DVP emulators\)](#)

8.1 Working with output triggers

Triggers are output signals from the EMU2 Emulator or Development Kit (DVP) that can be connected to an external resource.

Two triggers, OUT1 and OUT2 are available on the ST7 EMU2 emulator. These signals can be used to synchronize an external measurement instrument, such as an oscilloscope. When a user-defined OUT1 or OUT2 event occurs, an impulse (TTL level) is emitted at the Trigger OUT1 or OUT2 outlets on the emulator front panel, where a one-clock cycle pulse is emitted.

For the Development Kits, when a user-defined event occurs, an impulse (TTL level) is emitted or the level of a signal is changed at a special outlet (TRIGOUT pin) on the DVP evaluation board. A special dialog box lets you choose the waveform mode for the external signal.

You can set Trigger events so that signals are sent under the following circumstances:

- when a variable or constant is accessed,
- when a data memory address or range of addresses are accessed,
- when a program memory address or range of addresses are accessed, or its contents are executed.

When, and under what circumstances, a signal is output from a Trigger can be controlled using Hardware Events, described in [Section 8.2: Using hardware events](#).

8.2 Using hardware events

A hardware event is used to control when you want to filter the trace or to trigger output signals from your EMU2 Emulator or Development Kit.

The EMU2 Emulator version and the DVP versions are slightly different in the way they use hardware events:


- **For the EMU2 emulators**, hardware events are used to control the trigger outputs (OUT1 or OUT2).
- **For the DVP2 and DVP3 emulators**, there are three types of hardware events that can be used to control either the trace recording or the trigger output:
 - Event On (EVT_ON): The address where the event begins.
 - Event Off (EVT_OFF): The address where the event ends.
 - Event Hit (EVT_HIT): The event is active for the cycle in which one particular address is accessed.
- **For the DVP1 emulators**, there are only two types of hardware event, and these always control the output trigger:
 - FORCE_HIGH: Forces the output trigger signal to one.
 - FORCE_LOW: Forces the output trigger signal to zero.

This section provides hardware event information that is common to DVP and EMU2 emulators, including:

- [The hardware event window and contextual menu](#)
- [Adding a hardware event](#)

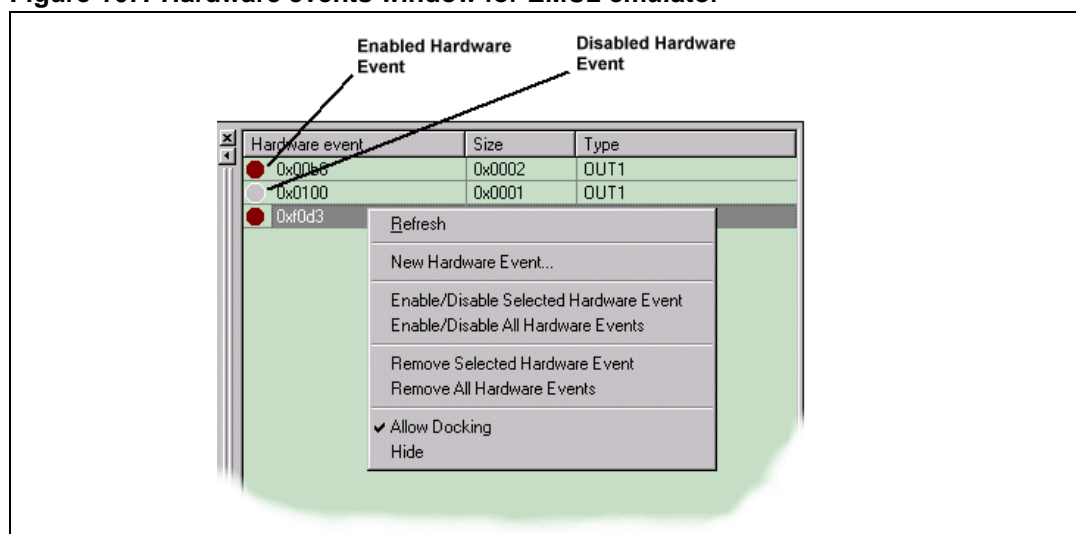
For more information about how to control the triggers and trace on your DVP emulator, refer to [Section 8.7 on page 255](#).

8.2.1 The hardware event window and contextual menu

1. Open the **Hardware Events** window either by clicking on  (the Hardware Events window icon) in the View toolbar, or from the main menu by selecting **View>Hardware Events**.
2. With the mouse pointer in the **Hardware Events** window, right-click the mouse to bring up the Hardware Events contextual menu.

[Figure 167](#) shows the **Hardware Events** window as it appears for the STVD EMU2 Emulator version.

Figure 167. Hardware events window for EMU2 emulator



From this contextual menu, you can:

- **Refresh:** Updates the window.
- **New Hardware Event:** Adds a new hardware event.
- **Enable/Disable Selected Hardware Event:** If you select a specific hardware event, and then right-click to obtain the contextual menu, choosing this option disables the event. A disabled event is signaled by a gray circle in the **Hardware event** column.
- **Enable/Disable All Hardware Events:** A toggle switch to enable or disable all hardware events.
- **Remove Selected Hardware Event:** If you select a specific hardware event, and then right-click to obtain the contextual menu, choosing this option removes that event.
- **Remove All Hardware Events:** Erases all hardware events.

8.2.2 Adding a hardware event

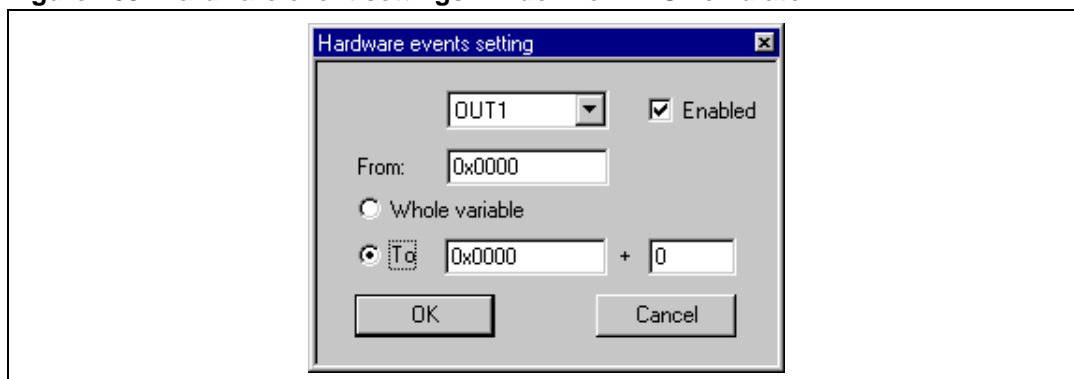
You can add a hardware event from the **Hardware Events** window or from the Editor window (if you want to set a hardware event on a specific symbol in your application program).

To add a hardware event from the Hardware Events window

1. Right-click the mouse while the mouse pointer is anywhere in the **Hardware Events** window.
2. Choose **New Hardware Event** from the contextual menu.

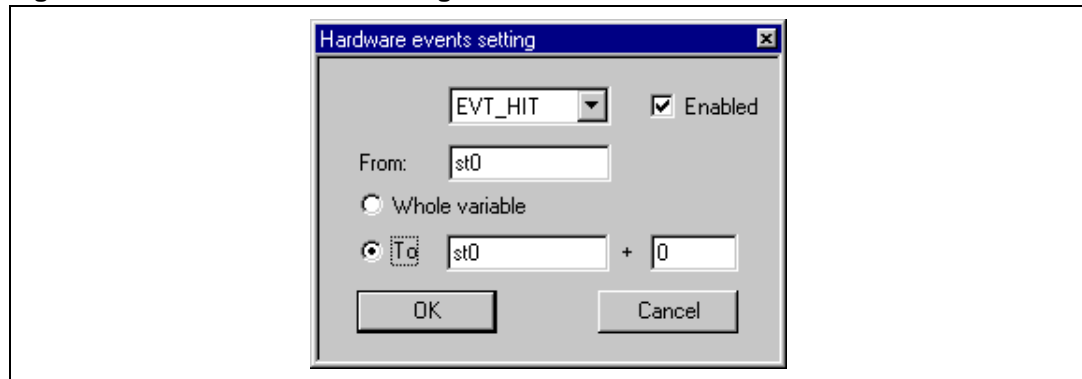
The **Hardware event settings** dialog box opens as shown in [Figure 168](#) or [Figure 169](#).

Figure 168. Hardware event settings window for EMU2 emulator



If you are using an **EMU2 Emulator**, in the **Hardware event setting** dialog box, choose the trigger output that you want the signal to be sent to (either **OUT1** or **OUT2**) and check the **Enabled** box.

Figure 169. Hardware event settings window for DVP2



If you are using a **Development Kit**, in the **Hardware event setting** dialog box, choose the kind of hardware event you want to set (either **EVT_ON**, **EVT_OFF** or **EVT_HIT** for DVP2 and DVP3, or **FORCE_HIGH** or **FORCE_LOW** for DVP1) and check the **Enabled** box. For more information, see [Section 8.7 on page 255](#).

3. In the **From** field, type the begin address of the memory range on which the event is set, or the name of the symbol on which address you want the event to be set.
4. Select **Whole variable** if you want the hardware event to be activated on the whole variable range which is directly linked to the symbol type specified in the **From** field (for example, on whole fields of data symbols for a C structure data type).
5. Select **To** to enter the end address of the memory range on which the event is set. This can be expressed either as a symbolic name or as an address. By default, the symbol name entered is the same as the symbol name in the **From** field. In the **+** field you may also enter an offset if required.
6. When you have finished setting the hardware event, click **OK**.

To add a hardware event on a specific symbol from the Editor window

1. Open the **Hardware Events** window.
2. Open the relevant source file in the Editor.
3. Select the symbol for which you want to add a hardware event by highlighting it with the mouse (it is highlighted in blue).
4. Either:
 - Drag and drop the highlighted symbol into the **Hardware Events** window.
 By default, the size is the whole variable, and the type is **OUT1** (for the EMU2 Emulator version) or **EVT_ON** (for the DVP version), but you may modify these parameters from within the **Hardware Events** window if you want.

Figure 170. Hardware events window (EMU2 emulators)

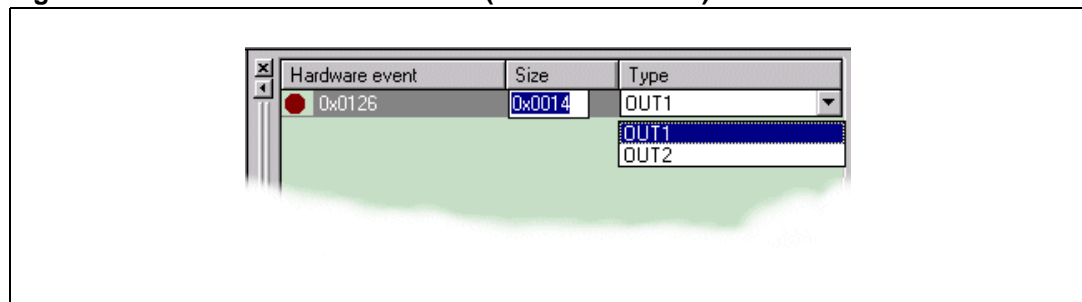
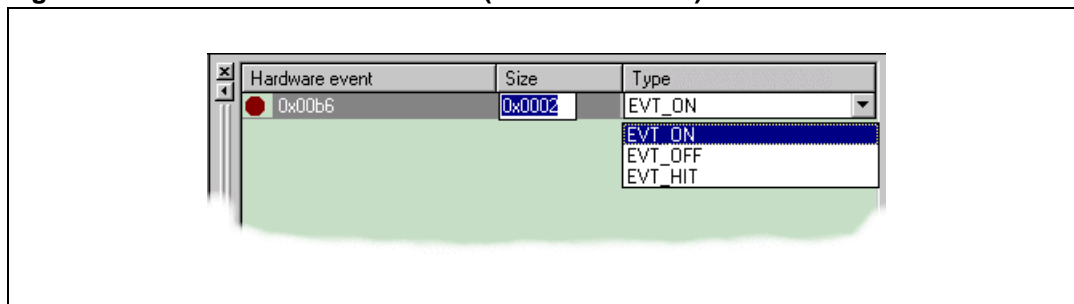


Figure 171. Hardware events window (DVP2 emulators)



- Right-click the mouse. The Editor contextual menu opens. Select **Add Hardware Event**. The **Hardware events setting** dialog box opens.

Figure 172. Hardware events settings dialog box (EMU2 emulators)

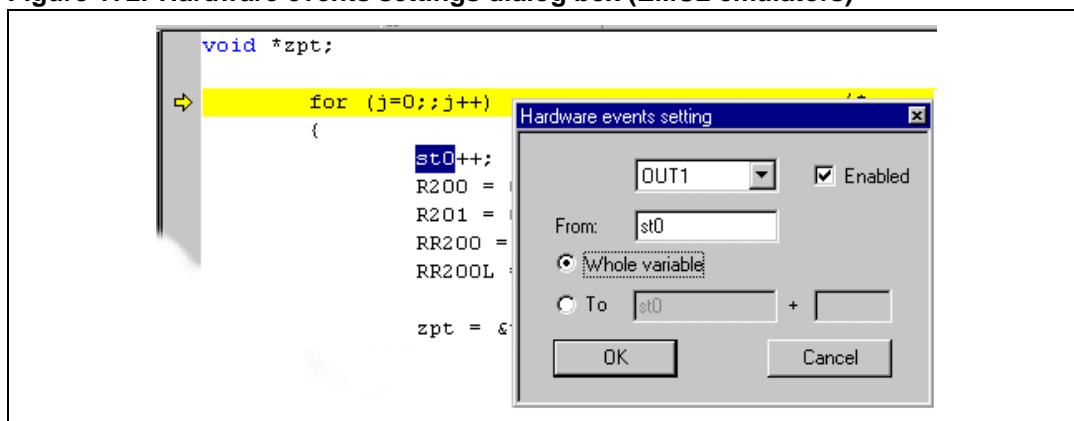
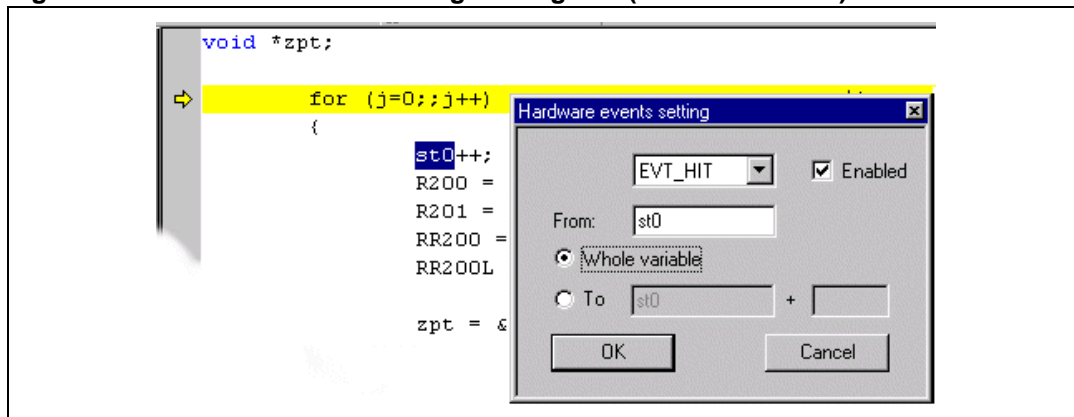


Figure 173. Hardware events settings dialog box (DVP2 emulators)



You can change the event type using the drop down list.

The name of the symbol you highlighted in the editor appears in the **From** field (in our example, *st0*). Proceed as described in [To add a hardware event from the Hardware Events window](#).

To set a hardware event on a line of code


1. Open the **Hardware Events** window.
2. Open the relevant source file in the Editor and place the cursor in the line of code.
3. Right-click the mouse.
The Editor contextual menu opens.
4. Select **Go to Disassembly**.
The **Disassembly** window opens and the code address of the source code line is selected.
5. Drag and drop the code address from the **Disassembly** window to the **Hardware Event** window.
When you open the **Hardware Events** window, the hardware event that you have just added appears. The symbol or code line, as well as the size (the range over which the event is defined) is shown in the hardware event window in terms of the corresponding memory address.

Note: You cannot modify the address of existing hardware events, however you can modify their size or their type.

8.3 Trace recording

The ST7-DVP3, EMU2, and DVP2 series emulators have a trace buffer for recording hardware cycles. The trace buffer has a limited physical size (512 cycles for DVP3, 1024 cycles for EMU2 and 256 cycles for DVP2). STVD's **Trace** window allows you to view recorded hardware cycles that have occurred during the execution of your application. In addition, different trace recording modes allow you to control what information is viewed and when.

Note: The **Trace** window is not available for use with the DVP1 (first generation MDT1/MDT2 Development Kits).

You can open the **Trace** window either by clicking on  (the Trace window icon) in the View toolbar, or from the main menu by selecting **View>Trace**.

Trace recording is activated from the [Trace contextual menu](#). When activated, the **Run**, **Continue** and **Step** commands prompt the trace buffer to save trace information until a breakpoint is reached.

You can either view all of the trace buffer contents or filter those that you want to view (see [Line filter](#)).

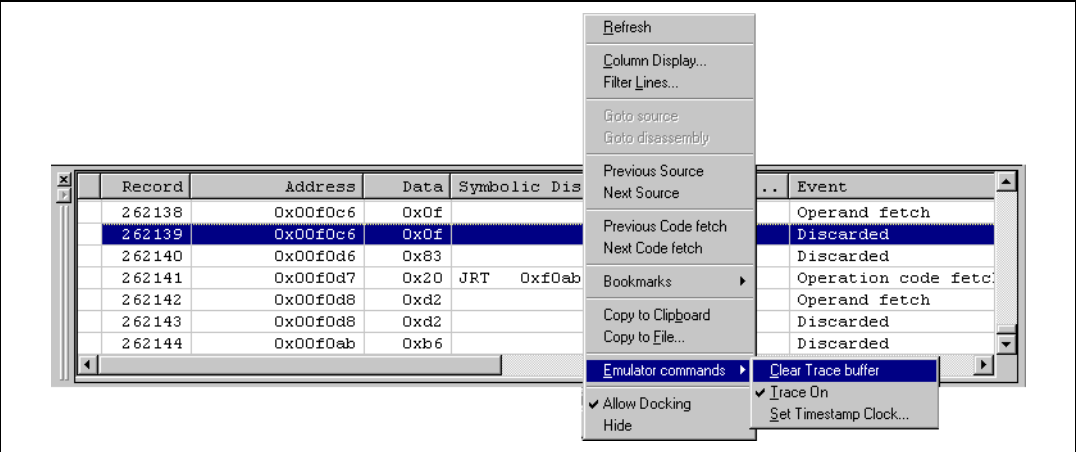
You can also use the [Logical analyser \(EMU2 emulators only\)](#) to define conditions whereby you can filter those cycles you want to record in the trace buffer. For example:

- you can stop or start trace buffer recording after a specific event, or,
- you can cause each access made to the specified area to be recorded a specific number of times.

8.3.1 Trace contextual menu

The Trace window contextual menu contains commands for trace operations plus several trace window configuration options. Right-click anywhere within the **Trace** window to open the Trace contextual menu.

Figure 174. Trace contextual menu



Show/hide columns

Columns may be disabled if they not required and/or do not display any trace information for the particular trace situation. To activate/disable a column, select **Column Display** in the Trace contextual menu. This opens a list of all the columns available in the trace record.

Figure 175 shows the **Columns** dialog box for the DVP version.

Figure 175. Columns dialog box for DVP emulators

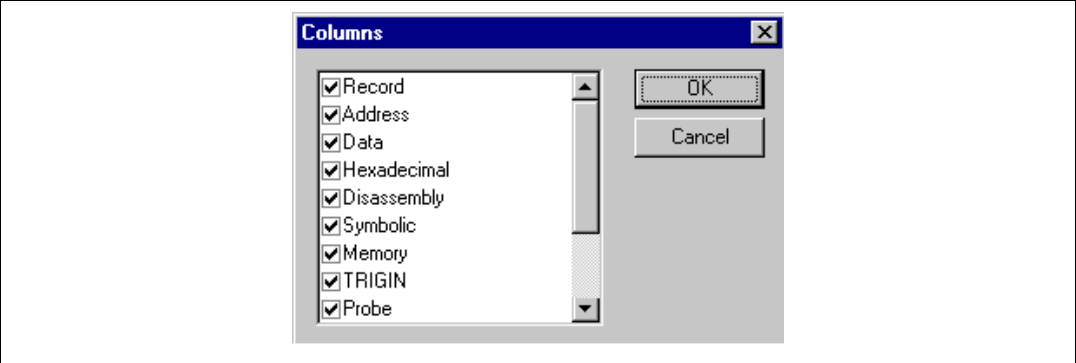
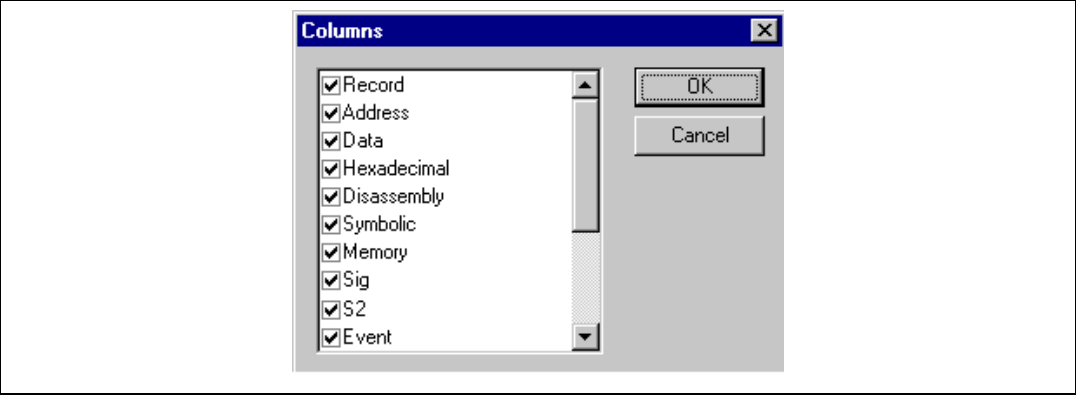


Figure 176 shows the **Columns** dialog box for the EMU2 version.

Figure 176. Columns dialog box for EMU2 emulators



Move columns

Columns may be also shifted right or left for convenience of use. Pick up the column header with the left mouse button and drag to the location required.

Line filter

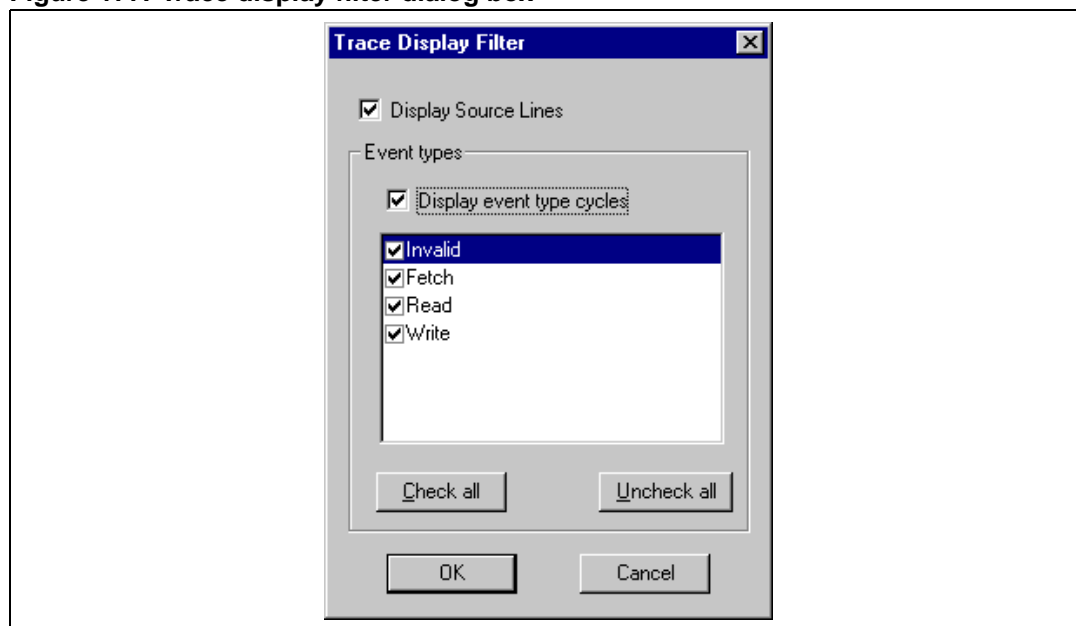
Trace records may be filtered using the **Trace Display Filter** dialog box (see [Figure 177](#)). To open this window, select **Filter Lines** from the Trace window contextual menu.

The line filter is used to restrict the trace display to the operation that you are interested in. Only the selected trace entries appear in the **Trace** window (all others are hidden). The record number and timestamp are not recalculated when lines are removed from the display.

Display Source Lines: This option includes the source line in the Trace information. The **Address** column contains the name of the source file and the source line content is displayed in the **Disassembly** window.

Display event type cycles: Includes the recorded cycles in the trace information. When enabled, the microprocessor actions selected in the **Trace Display Filter** dialog box are included. When disabled, only the source lines are shown in the **Trace** window.

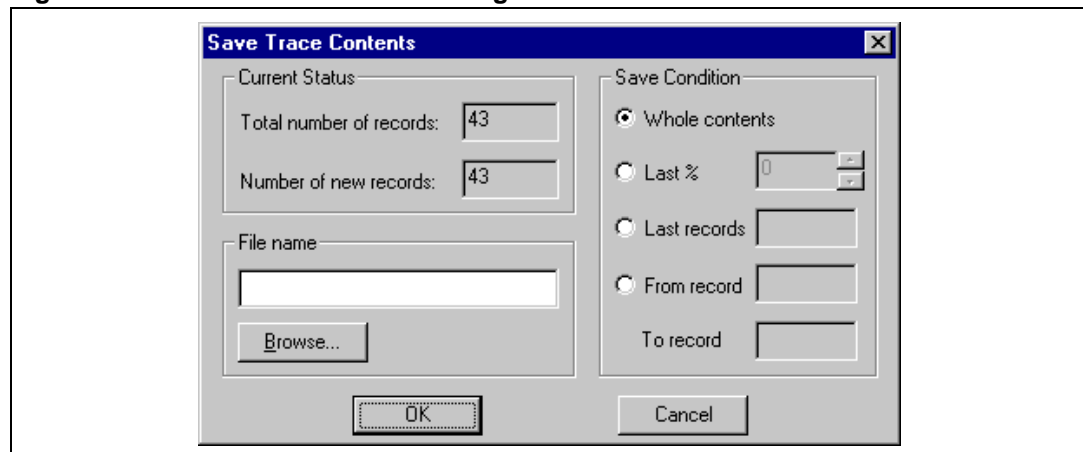
Figure 177. Trace display filter dialog box



Saving trace to file

The trace record may be copied to the clipboard or copied to file via options in the Trace contextual menu. The command **Copy to File** opens the **Save Trace Contents** dialog box ([Figure 178](#)), with details of saved trace files and user options for save parameters.

Figure 178. Save Trace Contents dialog box



8.3.2 Viewing trace contents

The **Trace** window presents a table of nine or ten fields (plus the Symbol Bar) which together form a single trace record.

At the left is the **Symbol Bar** (which has no column heading) followed in default order by **Record / Address / Data / Hexadecimal / Disassembly / Symbolic / Memory**, then the hardware-specific categories.

For **EMU2** emulator: **Sig / S2 / Event**.

Figure 179. EMU2 emulator trace window

	Record	Address	Data	Hexadecimal	Disassembly	Symbolic	Memory	Sig	S2	Event
→	1	0x00f2fd	0x9c	0x9C	RSP	RSP	Fetch	1111	0	
		start07.c	84		RSP					
	2	0x00f2fe	0xcd				Invalid	1111	0	
→	3	0x00f2fe	0xcd	0xCDF22C	CALL 0xf22c	CALL Init	Fetch	1111	0	
		start07.c	88		Init ();					
	4	0x00f2ff	0xf2				Read	1111	0	
	5	0x00f300	0x2c				Read	1111	0	
	6	0x00f22c	0xc6				Invalid	1111	0	
	7	0x0001ff	0x01				Write	1111	0	
	8	0x0001fe	0xf3				Write	1111	0	
	9	0x00f22c	0xc6	0xC6F011	LD A,0xf011	LD A. startupData+13	Fetch	1111	0	



For **DVP2**: **TRIGIN / Probe**.

Figure 180. DVP2/DVP3 trace window

	Record	Address	Data	Hexadecimal	Disassembly	Symbolic	Memory	TRIGIN	Probe [2,1,0]
→		start07.c	84		RSP				
	1	0x00f2fd	0x9c	0x9C	RSP	RSP	Fetch	0	011
	2	0x00f2fe	0xcd				Invalid	0	011
→		start07.c	88		Init ();				
	3	0x00f2fe	0xcd	0xCDF22C	CALL 0xf22c	CALL Init	Fetch	0	011
	4	0x00f2ff	0xf2				Read	0	011
	5	0x00f300	0x2c				Read	0	011

[Figure 179](#) and [Figure 180](#) show all of the trace columns for each STVD version. You can preselect which of these columns are displayed (see [Show/hide columns](#)).

The trace fields contain the following information:

- **Symbol Bar:** This column contains icons such as source line markers  and/or bookmarks .

Note: *When a source line marker occurs, the fields described hereafter are used to display source line information such as the name of the source file, the line of source code, and the instruction call.*

- **Record:** Trace record numbering starts at 1, which corresponds to the earliest cycle to be recorded, and ends at the latest cycle recorded (for the EMU2 emulator, the maximum is generally 1024 cycles, for the DVP2 the maximum is 256, and for DVP3 the maximum is 512).

Take, for example, the case of the EMU2 emulator's trace. If you have an application that runs for 2048 cycles before there is a break, and the maximum size of the trace buffer is 1024 cycles, cycles 1024 to 2048 are recorded in the trace buffer, but they will be numbered 1 to 1024 in the **Record** column.

- **Address:** The memory location accessed.
- **Data:** The hexadecimal value on the data bus
- **Hexadecimal:** The instruction in hexadecimal format, if this is a Fetch instruction cycle.
- **Disassembly:** The instruction in assembly language mnemonics, if this is a Fetch instruction cycle.
- **Symbolic:** The instruction in assembly language mnemonics with symbolic operands, if this is a Fetch instruction cycle.
- **Memory:** The type of memory access (either read/write/fetch or invalid).
- **Sig:** The value of the EMU2 emulator front panel input signals (Analyser probe).
- **S2:** The value of the EMU2 emulator S2 input signal (see emulator documentation).
- **Event:** The name of the Logical Analyser event that has been matched on this cycle (Event 1, Event 2, Event 3 or blank for none).
- **TRIGIN:** The value of the TRIGIN signal (input pin located on the DVP).
- **Probe:** The values of the signals output from the probe pins located on the DVP.

8.4 Using hardware testing

The hardware test function enables you to check that the emulator is still working online and has not suffered a hardware problem. If problems occur during debugging (such as bad debugger responses and unexpected behavior), you should check for hardware problems using the Hardware Test function, and if any are detected, contact your STMicroelectronics sales representative.

Caution: Be careful when performing a hardware test while an application is open. The opened application may be corrupted by the hardware testing process. If you find that your application has been corrupted, simply close the application, and reopen it.


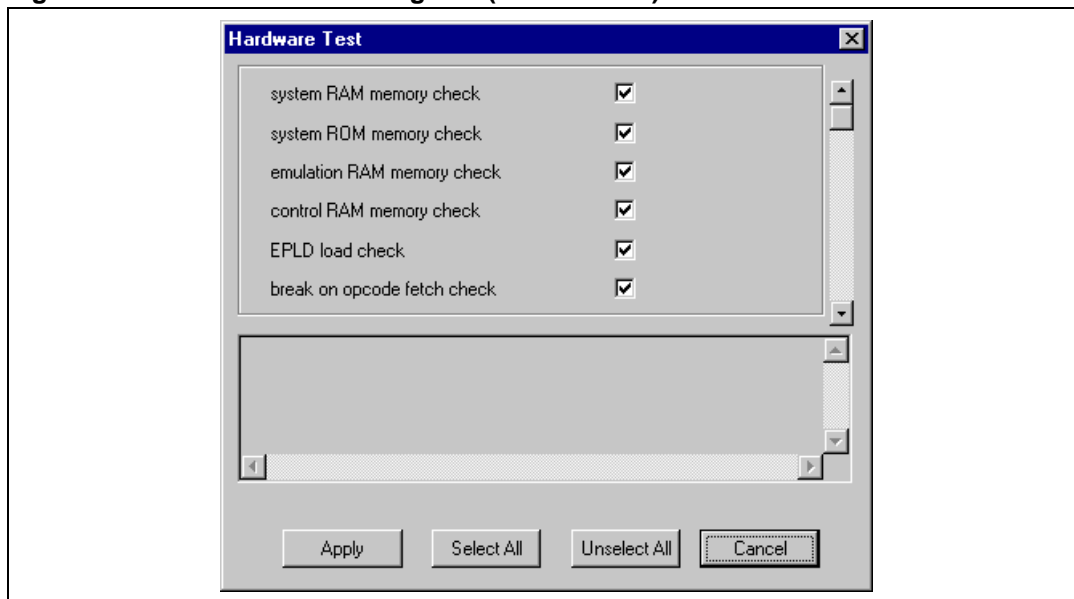
The **Hardware Test** dialog box is accessible either by clicking on  (the Hardware Test icon) in the Emulator toolbar, or from the main menu by selecting **Debug Instrument>Hardware Test**.

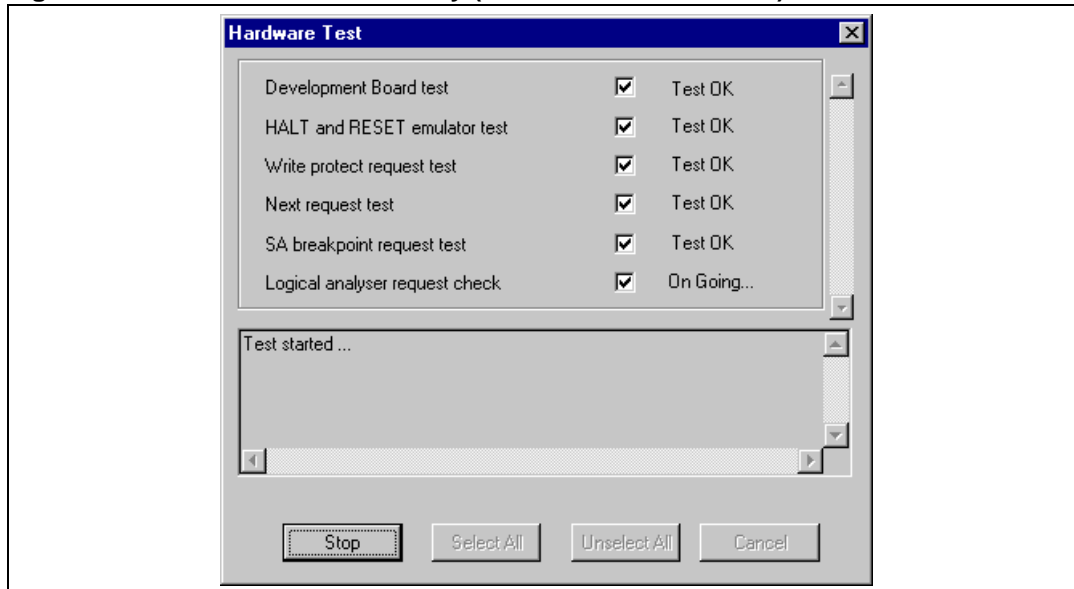
Figure 181. Hardware Test dialog box (DVP version)



The **Hardware Test** dialog box shows a list of different tests that can be performed. Check the box for each test that you want to perform (they are all checked by default) and click **Apply** to start the hardware test.

The hardware tests are performed one by one, and the results summarized in the dialog boxes. For example, the hardware test for the EMU2 Emulator appears as shown in [Figure 182](#).


Figure 182. Hardware test underway (EMU2 emulator version)



Note:

Some MCU options and memory mappings will stop the launching of a Hardware Test. To run the hardware test, you will have to reset the MCU configuration to the default settings. Do this by closing your debug session (Debug>Stop Debugging) and changing the MCU selection in the MCU tab of the Project Settings window (Project>Settings). Once you have changed the MCU selection you can restart your debug session and run the hardware test.

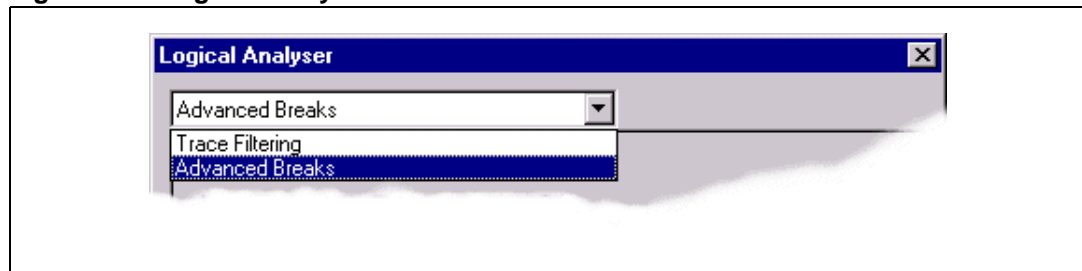
8.5 Logical analyser (EMU2 emulators only)

You can access the **Logical Analyser** window either by clicking on  (the **Logical Analyser** icon) in the Tools toolbar, or from the main menu by selecting **Tools>Logical Analyser**.

The **Logical Analyser** allows additional control over advanced breakpoints or the trace buffer. In the drop-down list at the top of the window, choose between:

- **Advanced Breaks** to set conditional breakpoints in the execution of the application,
- **Trace Filtering** to conditionally filter the cycle events in the trace buffer that you want to view.

Figure 183. Logical Analyser mode selection



Because the **Logical Analyser** uses separate hardware to “spy” on the running of the program on the emulator, the inclusion of advanced breakpoints does not interfere with the real-time running of the program. However, there is a short delay between the time the “spy” identifies a breakpoint condition has occurred and the moment the program execution actually stops. Typically, a few additional instruction lines are executed during this delay.

This behavior contrasts with that of **Instruction Breakpoints** and **Data Breakpoints**, where, because breakpoint conditions (set using the Counter and Condition options) are evaluated internally as if the evaluation were part of the program execution, the use of these breakpoints means that the program cannot run in real-time. However, Instruction and Data breakpoints cause the program to stop **exactly** at the instruction line that satisfies the break condition.

This section provides information about:

- [Defining logical analyser events](#)
- [Advanced breaks using the logical analyser](#)
- [Trace filtering using the logical analyser](#)

8.5.1 Defining logical analyser events

Using the Logical Analyser, you can define conditions to control either the filling of the trace buffer or the occurrence of a breakpoint. Conditions are defined using a combination of events. The Logical Analyser enables you to define up to three events, named **Event 1**, **Event 2** and **Event 3**.

Defining an event is equivalent to flagging either a specific address or symbol and/or a specific data value on the data bus, and/or a specific binary signal value. Each event is defined from the **Logical Analyser** window as described below. The debugger compares the hardware values you specify (such as the accessing of the memory address 0x0ff or the sending of the signal “1” to a probe pin AL2) to what the current hardware values are in the emulator, and when they correspond, the event is flagged as having occurred.

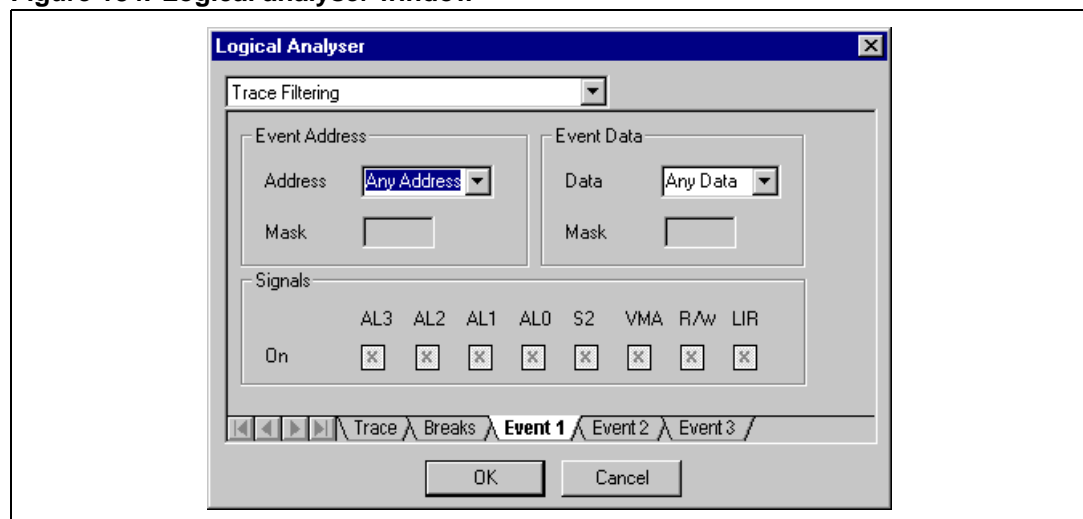
If the condition you define with the Logical Analyser consists of a single event, when this event is met, the Logical Analyser either imposes a breakpoint, or stops recording the trace (depending on which functionality you activated). However, if your condition was made up of several events, the Logical Analyser notes that the first event occurred, and keeps checking for the occurrence of the other events. When all of the events that make up the condition have occurred as specified, the breakpoint condition is filled and the Logical Analyser performs its task (i.e. imposing a breakpoint).

The sequence of events must follow a certain order: Event 1 followed by Event 2 and finally Event 3. However, the events can be combined in such a way that your condition involves the occurrence of an event followed by a specified number of cycles, and/or several occurrences of the same event (in other words, you can specify the number of times that an event must occur before the condition is met).

To define an event:

1. Open the **Logical Analyser** window by selecting **Tools>Logical Analyser** from the main menu.
Each Event number has its own tab in the window.
2. Click on the tab of the Event you want to define.

Figure 184. Logical analyser window



3. If you want to define the event by flagging an address or a symbol access in the memory space, enter the address or the symbol name in the **Event Address** fields. Specify whether you want the access to be Read or Write by choosing R or W in the **R/W** checkbox located under the **Signals** heading. You may request that the access is a valid access by choosing "1" in the **VMA** checkbox.
4. If you want to define the event as the occurrence of a specific value on the data bus, enter the value in the **Event Data** fields. If you specified an address or symbol in the **Event Address** fields, the definition of the event is the reading and/or writing (you define which in the **R/W** signal checkbox) of a specific value to that address.
5. Finally, the event you define can be simply the detection of a specific signal. Under the **Signals** heading is a list of signals each having a tristate checkbox activated by clicking on the signal's box with the mouse. Some signals are pin signals, and others give information on the memory operation being undertaken. "X" means that the signal value is unspecified and its value has no impact on the event. Each signal can also be

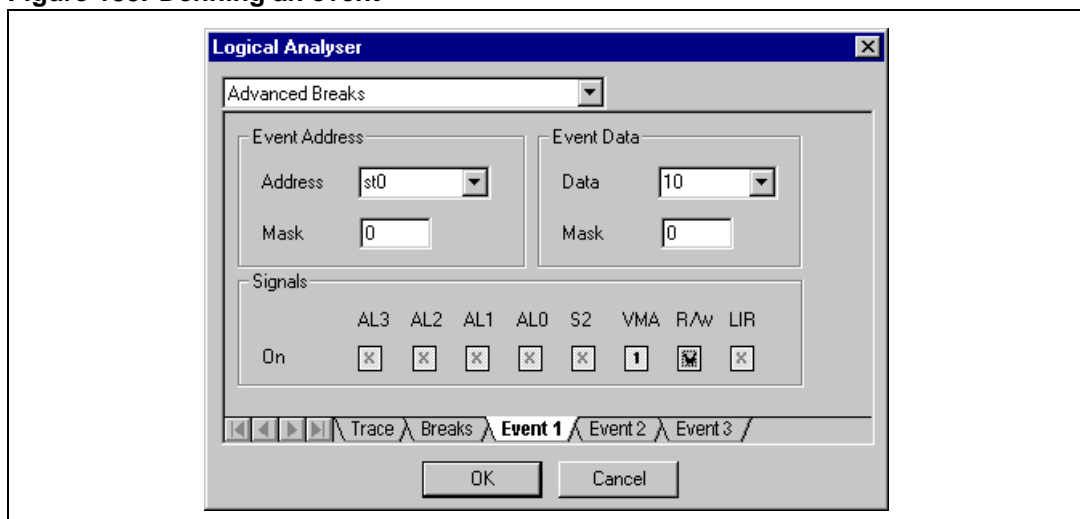
set to 0 or 1 except in the case of the Read/Write signal, where for clarity's sake, you can choose between "R" or "W".

You may flag values on the following pins/signals:

- Four Analyser probe pins (located on the probe port on the front of the EMU2 emulator): AL0, AL1, AL2 and AL3.
 - A supplementary input/output pin, S2, located next to the probe cables on the front of the EMU2 emulator.
 - The VMA signal indicates whether a memory cycle is valid or invalid. For example, during processing, cycles are often used as computation time without significant values on addresses and data buses. In these circumstances, the VMA signal for that address is "0". However, when a validated value is being read from or written to an address, the VMA signal is "1".
 - The R/W signal indicates whether a memory address is being read from (R) or written to (W).
 - The LIR signal indicates whether there is a fetch cycle occurring ("1") or not ("0").
6. When you have finished defining the event(s), you must choose either the **Advanced Breaks** or **Trace Filtering** mode (described in detail below). Click **OK**.

This action closes the **Logical Analyser** window, saving your event definition(s).

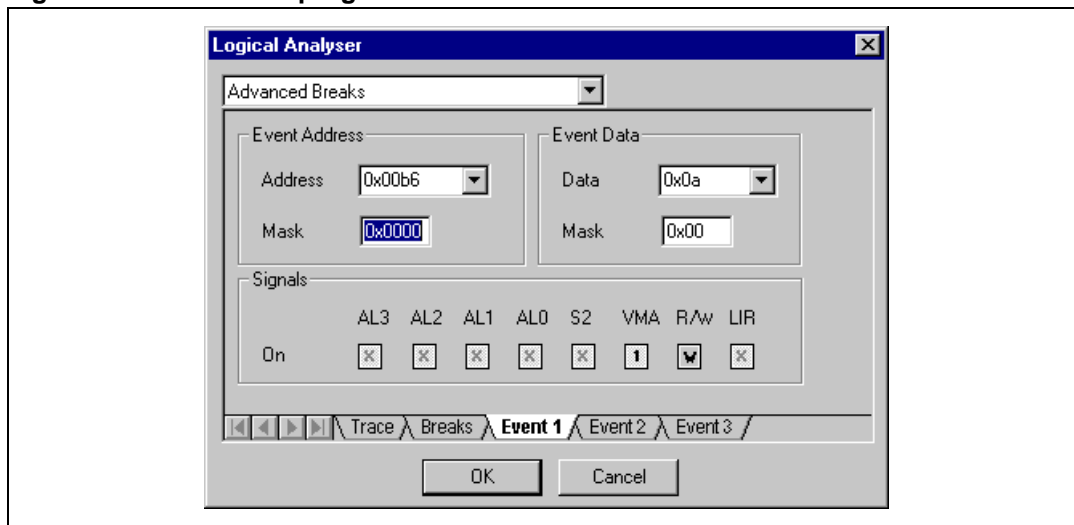
Figure 185. Defining an event



As an example, to set an event in the execution of the program when the value of the symbol st0 is written as 10, the **Event** tab would appear as in [Figure 185](#).

Note: If you exit the **Logical Analyser** window by clicking **OK**, and reopen the Logical Analyser, you will find that the symbol value that you entered in the **Event** tab has been translated to its hexadecimal bus address. This has no impact on the event definition itself, and occurs because events defined in the Logical Analyser are based on the hardware data. As soon as you enter a symbol (which is a programmed entity rather than a hardware address), it is immediately translated in terms of where the symbol occurs in the memory address. For example, the event we defined in [Figure 185](#) appears as shown in [Figure 186](#).

Figure 186. Event 1 as programmed



8.5.2 Advanced breaks using the logical analyser

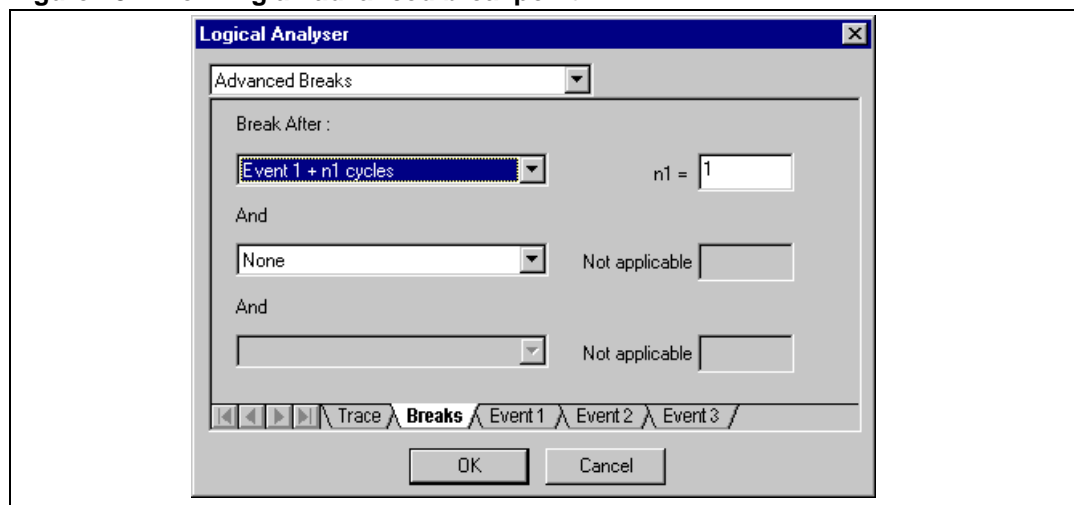
You can use Events as conditions by which to insert a break in the running of an application, by choosing the Advanced Break functionality from the drop-down list at the top of the window.

*Note: A break specified using the logical analyser is different from inserting an **instruction breakpoint** or a **data breakpoint**, because the break in the execution of the application is triggered by the Analyser hardware. Consequently, you may find that there is a delay between the time the hardware condition specified by the event(s) is flagged and the time that the application is actually stopped. In general, you will probably find that a few more lines of the program have been executed before the program is halted.*

To define an advanced break:

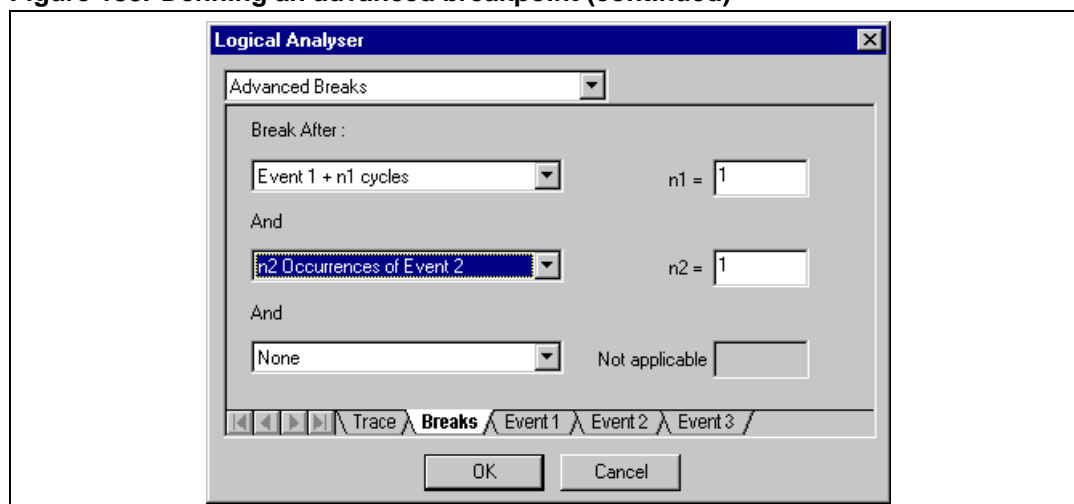
1. Open the **Logical Analyser** window.
2. Choose **Advanced Break** from the drop-down list at the top of the window.
3. In the **Break After** field, you can choose between three options:
 - **No Break:** No breaks are specified.
 - **Event 1 + n1 cycles:** The break occurs after **Event 1** has occurred and **n1** cycles have subsequently elapsed. This requires you to **define Event 1** and to enter a specified number of cycles in the **n1** field.
 - **n2 Occurrences of Event 2:** The break occurs after **Event 2** has occurred **n2** times. This requires you to **define Event 2** and to enter a specified number of occurrences in the **n2** field.

Figure 187. Defining an advanced breakpoint



You can continue to apply break conditions using up to three events by way of the **And** fields. In the example shown in [Figure 188](#), after **Event 1** plus **n1** cycles have occurred, followed by **n2** occurrences of **Event 2**, a break in the execution of the application occurs.

Figure 188. Defining an advanced breakpoint (continued)



8.5.3 Trace filtering using the logical analyser

Choose the Trace Filtering functionality of the logical analyser by selecting this option from the drop-down list at the top of the window.

As described above, the logical analyser enables you to define up to three events, named **Event 1**, **Event 2** and **Event 3**. Event 1 and Event 3 cause trace buffer recording to be stopped after a specified number of cycles. Event 2 causes each access made to a specified area associated with the event to be recorded a specified number of times.

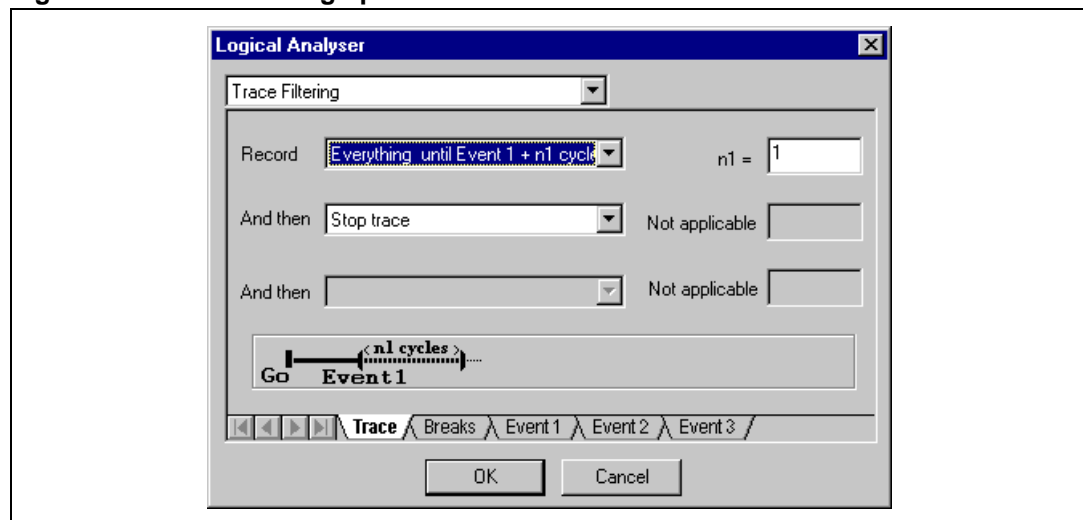
This enables you, for example, to count the occurrence of a complex event, or trace all accesses to an address or an address range. If several conditions are set together (for example, Event 1 and Event 2), recording for an event condition is only triggered when all

previous event condition recordings are completed (for example, Event 2 condition recording only starts when Event 1 condition recording is completed).

To define conditions for trace buffer recording:

1. Open the **Logical Analyser** window.
2. Choose **Trace Filtering** from the drop-down list at the top of the window.
3. In the **Record** field, choose one of the three options (as shown in [Figure 189](#)):

Figure 189. Trace filtering options



- Record **Everything until Event 1 + n1 cycles**: The condition being defined is that the trace is recorded until **Event 1** occurs and then **n1** cycles elapse. This requires you to **define Event 1** and to enter a specified number of cycles in the **n1** field.
- Record **n2 occurrences of Event 2**: The trace does not begin recording until **Event 2** occurs, and then it records everything until **Event 2** has occurred **n2** times. This requires you to **define Event 2** and to enter a specified number of occurrences in the **n2** field.
- **Permanent recording**: The trace buffer records unconditionally.

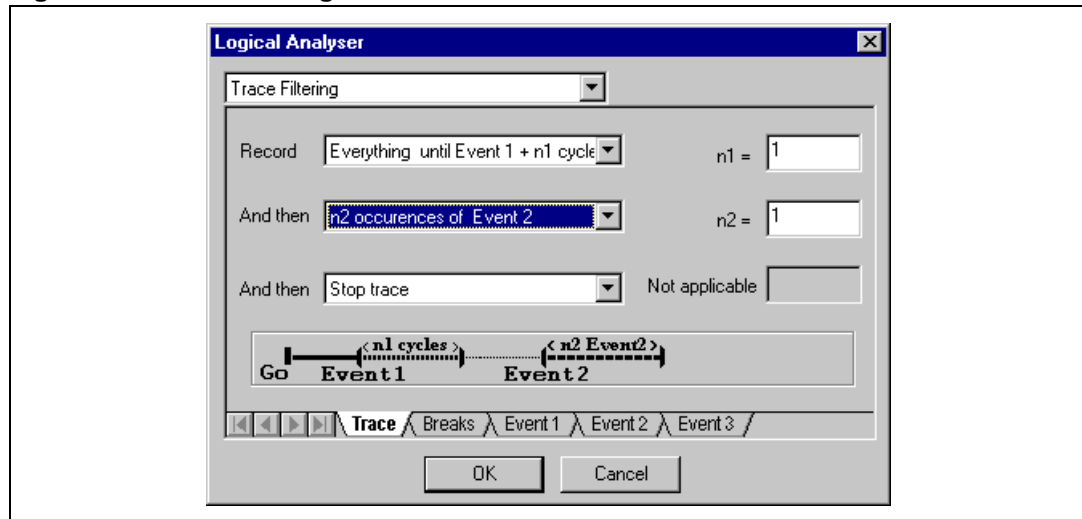
Note:

For each case, a small schematic shows how long the trace is recorded. In the example shown above, the trace starts to be recorded at Go (symbolized by the bold line) when the application is run. It continues to be recorded until Event 1 occurs, and continues during the execution of n1 cycles. Once the n1 cycles have elapsed, trace recording is halted.

4. If necessary, continue to apply trace filtering conditions using up to three events by way of the **And then** fields.

In the example shown in [Figure 190](#), the trace recording starts from Go and continues until **Event 1** plus **n1** cycles have occurred. Then recording stops (as indicated by the finely dotted line), and only restarts when **Event 2** occurs, and continues recording event cycles (bold dotted line) until **Event 2** has occurred **n2** times.

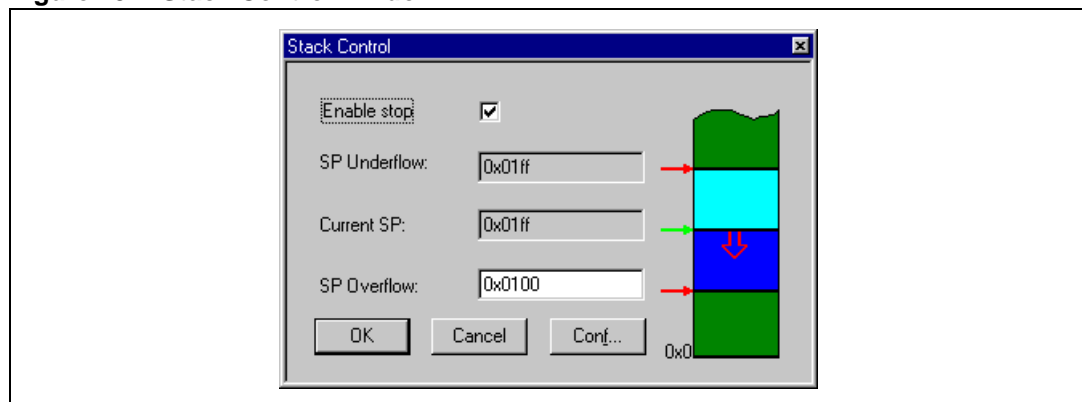
Figure 190. Trace filtering event schematics



8.6 Stack control window (DVP emulators)

The **Stack Control** window command allows you to set a specific stack address as either the stack overflow point or the stack underflow point. This means that if, during the course of debugging, the current SP value exceeds or falls beneath this address, the application is halted (provided that you check the **Enable stop** box in the **Stack Control** window).

Figure 191. Stack Control window



This effectively allows you to specify a breakpoint at the SP address level.

Setting the stack control


1. Open the Stack Control window either by clicking on  (the Stack Control Window icon) on the Tools toolbar, or by selecting *Tools>Stack Control...* from the main menu.
2. To enable a break in the running of the application when the specified conditions are met, check the **Enable stop** box.
3. If you want to specify a new stack overflow point, type the overflow address in the **SP Overflow** field.
4. To view/change the configuration settings, click on the **Conf...** button. Here you can enable or disable the stack pointer control:


Figure 192. Configuration setup window



5. Click on **OK** when you have set up the Stack Control parameters as you want.

Note: If you do not want your Stack Control information to be saved in the workspace file, you must modify the default **Configuration Setup options** by clicking on the **Conf...** button.

8.7 Trigger/trace settings (DVP emulators)

You can access the **Trigger/Trace Settings** window either by clicking on  (the Trigger/Trace Settings icon) in the Tools toolbar, or from the main menu by selecting **Tools>Trigger/Trace Settings**.

The **Trigger/Trace Settings** window allows you to set the following modes:

- **TRIGIN modes**

In the **Trigger/Trace Settings** window, you may choose the option to **Break on TRIGIN**. The TRIGIN port is an input signal port on the DVP board to which you can connect an external input signal. The **Break on TRIGIN** option imposes a break in the running of the program upon reception of a signal rising edge at the TRIGIN pin.

- **Trace overflow break mode (DVP2 and DVP3 only)**

When the **Break on Trace Overflow** option is enabled, the program is stopped as soon as the trace buffer is full (256 records for DVP2 and 512 records for DVP3), otherwise, the program is not stopped, and only the 256/512 last recorded triggers are kept in the trace buffer.

- **Hardware event mode**

For DVP2 and DVP3, there are two hardware event modes: a mode that only influences the trigger output signal (trace filtering disabled), and a mode where the trace is filtered using the same hardware events that the trigger acts on.

For DVP1, hardware events only affect the output trigger signals - there is no trace function available for DVP1. You can either send an output trigger signal in pulse mode, or you can send a trigger signal in window mode using two event settings:

FORCE_HIGH or FORCE_LOW. For more information, refer to [Working with output triggers \(DVP\)](#).

8.7.1 Working with output triggers (DVP)

Triggers are output signals that can be connected to an external resource. On the DVP, there is one output signal pin available called TRIGOUT. This signal can be used to synchronize

an external measurement instrument, such as an oscilloscope. When a user-defined hardware event occurs, an impulse (TTL level) is emitted or the level of the signal is changed, depending on the type of hardware event.

For information on how to define hardware events, see [Section 8.2: Using hardware events on page 237](#).

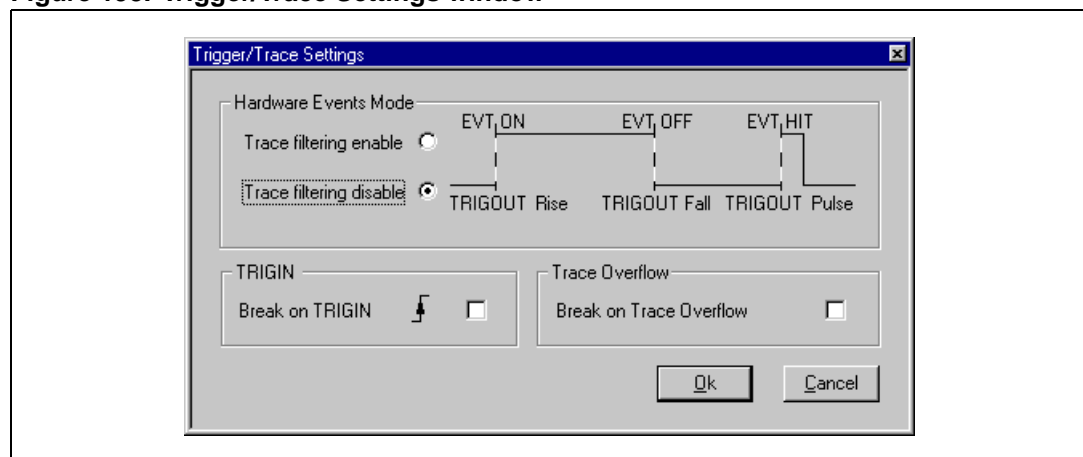
Controlling the output trigger on the DVP2 and DVP3

There are three types of hardware events for the DVP2 and DVP3 and they have the following effect on the trigger signal:

- **EVT_ON** hardware events make the TRIGOUT signal rise from 0 to 1,
- **EVT_OFF** hardware events make the TRIGOUT signal fall from 1 to 0,
- **EVT_HIT** hardware events cause a signal pulse to be emitted. The pulse consists of a positive signal rise from 0 to 1 for the duration of one cycle, after which, the signal falls to 0.

If you only want your hardware events to affect the output triggers, in the **Trigger/Trace Settings** window, click on the **Trace filtering disable** option as shown in [Figure 193](#).

Figure 193. Trigger/Trace Settings window



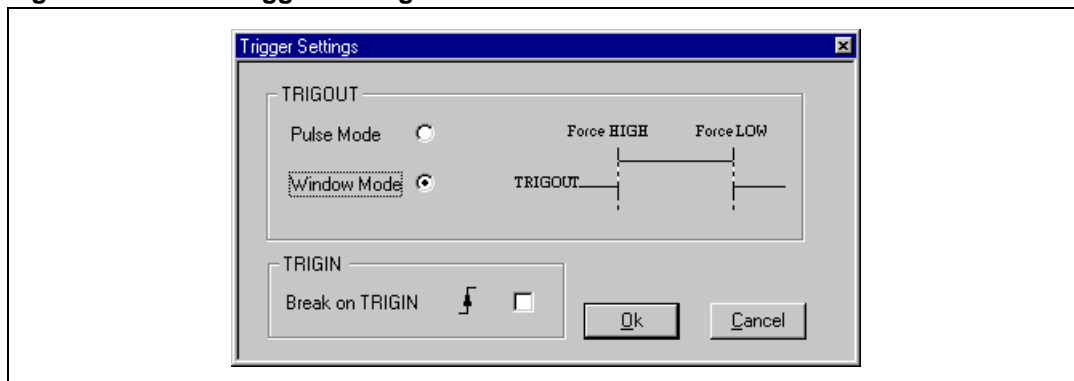
Controlling the DVP1 output trigger

You can set the output trigger to function in either Pulse mode (by selecting the **Pulse Mode** option in the **Trigger Settings** window shown in [Figure 194](#)) or in Window mode. Pulse mode emits a single pulsed signal. Window mode lets you control the output signal using hardware events.

There are two hardware event settings for DVP1:

- **FORCE_HIGH** hardware events make the TRIGOUT signal rise from 0 to 1.
- **FORCE_LOW** hardware events make the TRIGOUT signal fall from 1 to 0.

Figure 194. DVP1 trigger settings



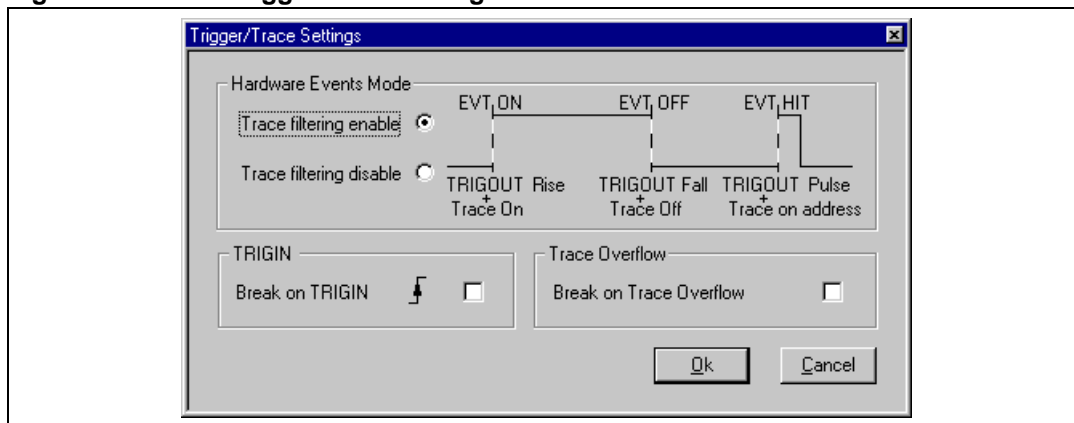
Working with trace filtering (DVP2 and DVP3 only)

In the **Trigger/Trace Settings** window, if you click the **Trace filtering enable** option, the trace is filtered using the same hardware events that affect the trigger output, so that the trace recording mirrors the trigger output:

- **EVT_ON** hardware events start the trace recording. If the program encounters more than one EVT_ON during program execution, the number of EVT_ON events are counted.
- **EVT_OFF** hardware events stop the trace recording. However, the number of EVT_OFF hardware events must equal the number of EVT_ON hardware events before the trace recording is stopped (each EVT_OFF decrements the EVT_ON counter).
- **EVT_HIT** hardware events cause the trace to be recorded for one address cycle. (The address corresponds to that defined in the hardware event).

The schematic drawing in the window shows how the trace and trigger act synchronously (see [Figure 195](#)).

Figure 195. DVP2 trigger/trace settings



9 STice features

The STice advanced emulation tool used in conjunction with STVD supports all of the features supported by the EMU3 emulator, except advanced breakpoints and performance analysis. Instead of performance analysis, it offers support for a comprehensive set of coverage and profiling features.

In the STice features, there is a trace recording section dedicated to STM8 which is not supported by EMU3.

For other emulator features, refer to the corresponding part in [Section 10: EMU3 emulator features](#):

- [Section 10.7: Read/write on the fly](#)
- [Section 10.8: Performing automatic firmware updates](#)

9.1 Trace recording

The STice advanced emulation system can hold 131072 hardware cycle records in a physical memory module called the trace buffer. STVD's **Trace** window allows you to view recorded hardware cycles that have occurred during the execution of your application.

You can open the **Trace** window either by clicking on  (the **Trace** window icon) in the View toolbar, or from the main menu by selecting **View > Trace**.

Trace recording is activated from the [Trace contextual menu](#). When activated, the **Run**, **Continue** and **Step** commands prompt the trace buffer to save trace information until a breakpoint is reached.

Only information obtained up until the occurrence of the last breakpoint is visible in the **Trace** window.

The following sections provide information about:

- [Trace buffer fields](#)
- The commands in the [Trace contextual menu](#)
- [Emulator commands](#)

9.1.1 Trace buffer fields

The **Trace** window presents a table of 19 fields (plus the Symbol Bar) which together form a single trace record.

At the left is the **Symbol Bar** (which has no column heading) followed in default order by **Record** / **PC** / **Instr event** (instruction event) / **Irq** (interrupt request) / **Hexa code** (hexadecimal code) / **Disassembly** / **Symb Disass** (symbolic disassembly) / **R addr** (read address) / **R value** (read value) / **R event** (read event) / **W addr** (write address) / **W value** (write value) / **W event** (write event) / **Time** (timestamp) / **Time event** (timestamp event) / **Trace event** / **Trig** (trigger input) / **AI** (analyzer input) / **BEM events**.

Each is described individually hereafter.

Figure 196. STice trace window

Record	PC	Instr event	I.	Hexa code	Disassembly	S	R addr	R...	R event	W addr	W...	W event
103	0x0080f4	Decoding start		0x5501030102	MOV 0x0102,0x0103	M						
→	main.asm			52	mov var4,var5							
104	0x0080f9	Decoding start		0x5501040103	MOV 0x0103,0x0104	M	0x000103	0x66	Data read	0x000102	0x66	Data
105	0x0080f9	Processing					0x000104		Read stall			
→	main.asm			54	ld a,var1							
106	0x0080fe	Decoding start		0xC60100	LD A,0x0100	L	0x000104	0x66	Data read	0x000103	0x66	Data
107	0x0080fe	Processing					0x000100		Read stall			
→	main.asm			56	jrne loop							
108	0x008101	Decoding start		0x2601	JRNE 0x8104	J	0x000100	0x01	Data read			
109	0x008104	Processing										
→	main.asm			60	nop							
110	0x008104	Decoding start		0x9D	NOP	N						
111	0x008105	Discarded										

Figure 196 shows most of the trace columns. You can preselect which of these columns are displayed (see [Column display: Columns may be disabled if they not required and/or do not display any trace information for the particular trace situation. To activate/disable a column, select Column Display in the Trace contextual menu. This opens a list of all the columns available in the trace record.](#)).

- **Symbol Bar:** This column contains icons such as source line markers (→) and/or bookmarks.

Note: When a source line marker occurs, the fields described hereafter are used to display source line information such as the name of the source file, the line of source code, and the instruction call.

- **Record:** Trace record numbering starts at 1, which corresponds to the earliest cycle to be recorded, and ends at the latest cycle recorded (up to a maximum of 131072 cycles). This means that if you have an application that runs for 150000 cycles before there is a break, and you record the trace continuously, cycles 18929 to 150000 stay recorded in the trace buffer, but they are numbered 1 to 131072 in the **Record** column. Record numbers are not recalculated when you remove events from the display by filtering (see [Filter lines: Trace records can be filtered using the Trace Display Filter dialog box \(Figure 198\).](#) To open this window, select **Filter Lines** from the **Trace window** contextual menu. The line filter is used to restrict the trace display to the operation that you are interested in. Only the selected trace entries appear in the **Trace window** (all others are hidden). The record number and timestamp are not recalculated when lines are removed from the display).
- **PC:** It is the 24-bit address of the instruction being decoded.
- **Instr event (Instruction event):** It is the type of instruction event: Decoding start, Interrupt handler, Interrupted, Discarded, Processing.

In addition, some hardware cycles that are automatically recorded to trace have a special identifier in this field that allows you to filter them out if you want, such as Extra record. These hardware cycles exist but do not correspond to either a specific instruction or operand. The **Instr event** column serves chiefly to allow you to filter out the hardware cycles of interest using the **Filter Lines** option in the **Trace** contextual menu, described in [Filter lines: Trace records can be filtered using the Trace Display Filter dialog box \(Figure 198\).](#) To open this window, select **Filter Lines** from the **Trace window** contextual menu. The line filter is used to restrict the trace display to the operation that you are interested in. Only the selected trace entries appear in the **Trace**

window (all others are hidden). The record number and timestamp are not recalculated when lines are removed from the display.

- **Decoding start:** Indicates that the displayed instruction starts being decoded.
- **Interrupt handler:** is displayed at the start of an interrupt handler. At the same time, the interrupt name is displayed in the **Irq** column.
- **Interrupted:** is displayed when an instruction is interrupted.
- **Discarded:** is displayed for the last cycle recorded before the program stops (breakpoint, execution step by step or user abort).
- **Extra record:** is displayed when a record has been added to indicate a time stamp event while the trace is off. This information is necessary to know if the time stamp difference between two records is meaningful.
- **Processing:** is displayed in all other cases, indicating that the instruction is being processed. There is no information on instruction fetch or instruction execution.
- **Irq** (interrupt request): If an interrupt is starting, the interrupt name is displayed: **reset**, **trap** or **irq n** where n is the number of the interrupt request.
- **Hexa code** (hexadecimal code): The code of the instruction in hexadecimal format, if this is an instruction decoding start cycle.
- **Disassembly:** The instruction in assembly language mnemonics, if this is an instruction decoding start cycle.
- **Symb disass** (symbolic disassembly): The instruction in assembly language mnemonics with symbolic operands, if this is an instruction decoding start cycle.
- **R addr** (Read address): It is the 24-bit address of the read data.
- **R value** (Read value): The read data is the contents of the data read bus or the DMA read bus. The data read bus is a 32-bit data bus but the data is usually 8-bit wide. However when accessing a pointer in EEPROM, the read data can be 16-bit or 24-bit

wide. As a result, the read value may be 1, 2 or 3-bytes long. These data bytes are constants or part of constants accessed by the program.

To simplify the data display, when data is read, the value of the data is shown only when it is valid, that is one cycle after the data request was made (if there is no stall).

- **R event** (Read event): It is the type of read event. The following values can be displayed:
 - Data read
 - Stack read
 - Read stall
 - DMA read

This field can be used to filter the disassembled trace records which are displayed.

- **W addr** (Write address): It is the 24-bit address of the data being written.
- **W value** (Write value): It is the contents of the 8-bit data write bus or the DMA write bus. These data bytes are variables or part of variables written by the program.
- **W event** (Write event): It is the type of write event. The following values can be displayed:
 - Data write
 - Stack write
 - Write stall
 - DMA write

This field can be used to filter the trace records which are displayed.

- **Time** (Timestamp): The value in nanoseconds of the time marker at this event. The time stamp is not recalculated when you remove events from the display by filtering (see *Filter lines: Trace records can be filtered using the Trace Display Filter dialog box (Figure 198). To open this window, select Filter Lines from the Trace window contextual menu. The line filter is used to restrict the trace display to the operation that you are interested in. Only the selected trace entries appear in the Trace window (all others are hidden). The record number and timestamp are not recalculated when lines are removed from the display.*).
- **Time event** (timestamp event): Signals the occurrence of events that affect the validity of the value in the Time column, in particular:
 - **Restart**: An automatic restart of the 38-bit timestamp counter. This means that the 38-bit Timestamp counter has reached its maximum and has started counting from 0 again, and the fact has been registered by the trace. Note that a trace recording is forced each time a timestamp counter restart occurs. This allows you to be able to calculate the absolute amount of time passed between the beginning of the run and any given trace record. However, after a Restart message is shown, remember that the value in the timestamp column is only relative to the last Restart message, not absolute. To obtain the true value of time passed, it is necessary to take into account all the restart events between two records. The frequency of the time stamp counter is 100 Mhz. The counter overflows every $2^{38}/10^8$ that is around 46 minutes (45 minutes and 49 seconds).
 - **Discontinuity**: Interruption of the timestamp counter's ability to write to the trace, due to the emulator's processor being in energy-saving mode, such as after a halt instruction. While the timestamp counter continues, there is no way to force the trace to record any restart messages that occur when the counter restarts. Because of this, there is no way to determine the time elapsed between two

events when a Discontinuity message occurs between them. However, the relative time elapsed between two events occurring before the discontinuity, or two events occurring after the discontinuity can still be calculated and is valid.

- **Trace event:** Start indicates that this is the first trace record after the trace has been switched on or after the program has stopped. It is not the case when the Trace display has been filtered using the **Filter Lines** option, because the record in the buffer is unaffected.
- **Trig** (trigger input): External trigger input.
- **AI** (analyzer input): Input from the analyzer input connector on the STice probe.
- **BEM** (advanced breakpoints) events: Currently, there are no advanced breakpoints and this field is empty.

9.1.2 Trace contextual menu

The **Trace** window contextual menu contains commands for trace operations plus several trace window configuration options.

Right-click anywhere within the **Trace** window to open the Trace contextual menu and access the following options and commands (see [Figure 197](#)).

Figure 197. Trace contextual menu

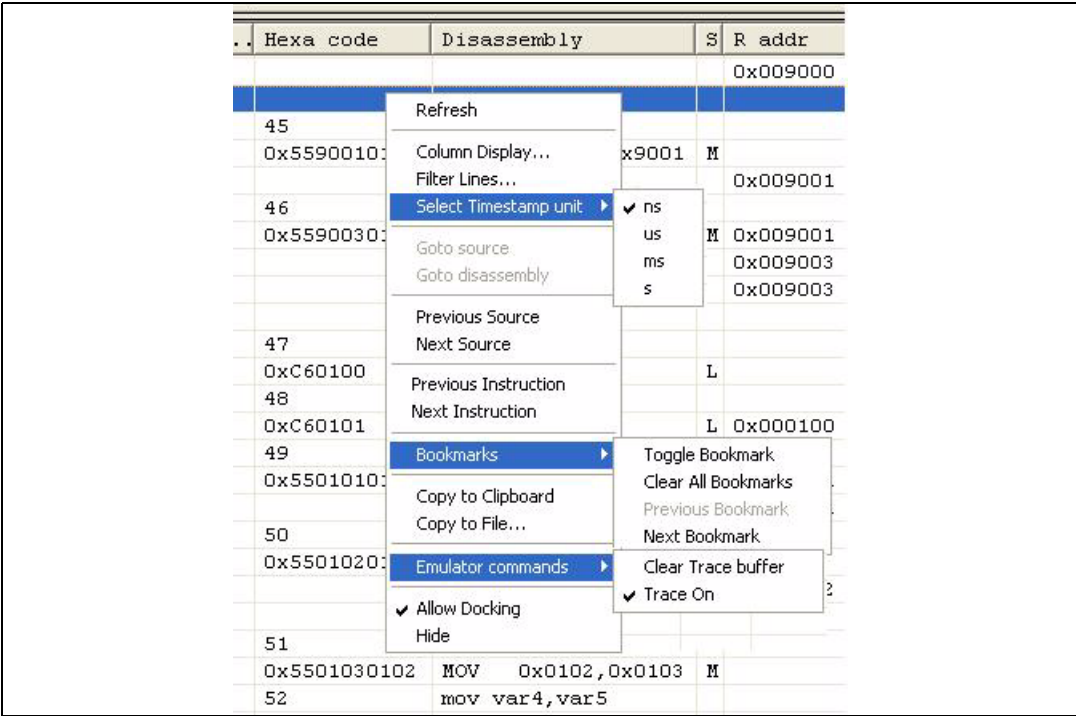
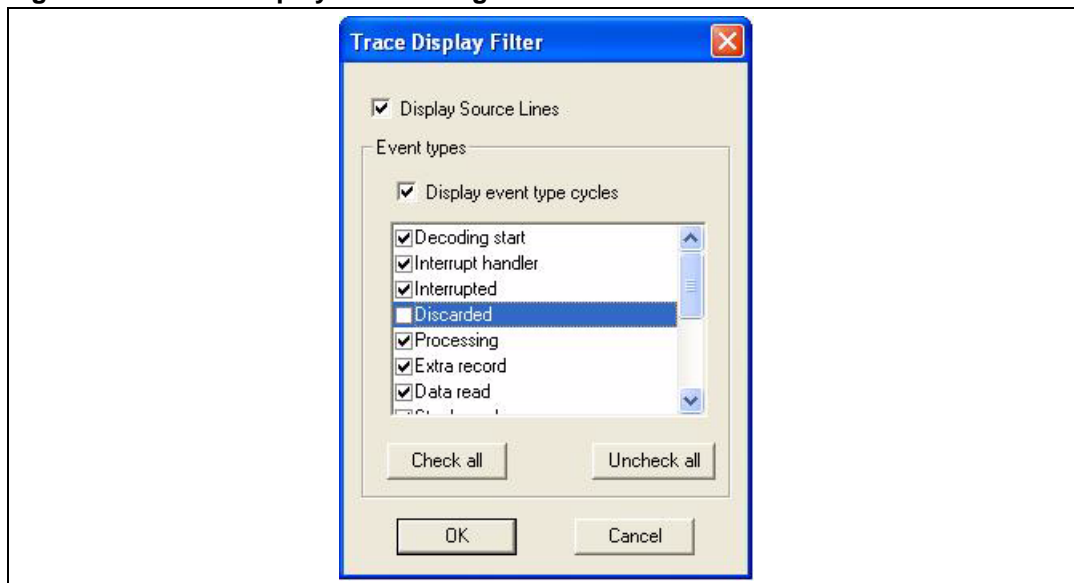


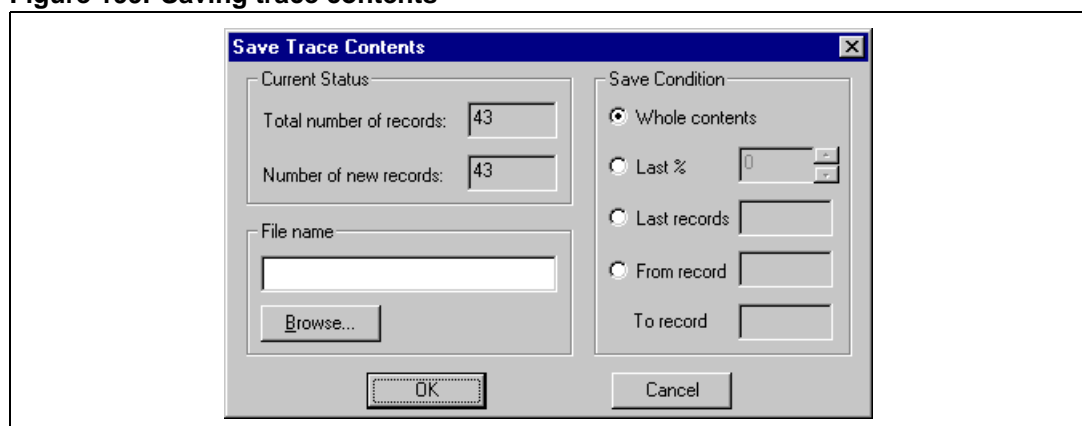
Figure 198. Trace Display Filter dialog box



- **Filter lines:** Trace records can be filtered using the **Trace Display Filter** dialog box (Figure 198). To open this window, select **Filter Lines** from the Trace window contextual menu. The line filter is used to restrict the trace display to the operation that you are interested in. Only the selected trace entries appear in the **Trace** window (all others are hidden). The record number and timestamp are not recalculated when lines are removed from the display.
 - **Display Source Lines:** This option includes the source line in the Trace information. The **Address** column contains the name of the source file and the source line content is displayed in the **Disassembly** window.
 - **Display event type cycles:** Includes the recorded cycles in the trace information. When enabled, the microprocessor actions selected in the **Trace Display Filter** dialog box are included. When disabled, only the source lines are shown in the **Trace** window.
- Note that some of these events can occur in parallel. There is always an event in the instruction event column. There may be an event in the read event column and/or in the write event column. As a result, a cycle is displayed if at least one of its three possible events (instruction event, read event if present, write event if present) is selected.
- In the above example, the **Discarded** event type is unchecked, so that no trace record with a "Discarded" event type is displayed. The last recorded cycle after the emulator's execution of a debugging break action (breakpoint, stepping, user abort) is flagged as discarded. By filtering these cycles out, you can see the true trace of instruction execution. However you may miss some data read or write operations that occur in parallel with the discarded instruction. Note also that breakpoints have an impact on the trace as the normal flow of execution is stopped.
- **Selecting the Timestamp unit:** By clicking on the **Select Timestamp unit** option, you can choose to have the Timestamp value shown in nanoseconds (ns), microseconds (μs), milliseconds (ms) or seconds (s).

- **Goto Source, Goto Disassembly:** You may use these commands in the contextual menu to jump to either an editor window or the **Disassembly** window under the following conditions:
 - If you highlight an entry in the trace where the address is a line of source code, you can use the **Goto Source** command to jump to that line of code in the Editor window.
 - If you highlight a trace record where the event is an opcode fetch, you can use the **Goto Disassembly** command to jump to the appropriate address in the **Disassembly** window.
- **Previous Source, Next Source commands:** If you highlight an entry in the trace where the address is a line of source code, you can use the **Previous Source** and **Next Source** commands in the contextual menu to jump to, respectively, the previous or next source code entries in the trace recording.
- **Previous Instruction, Next Instruction commands:** If you highlight a trace record of a decoding start event, you can use the **Previous Instruction** and **Next Instruction** commands in the contextual menu to jump to the previous or next decoding start records in the trace recording.
- **Bookmarks:** Clicking on the **Bookmarks** commands lets you access the following commands:
 - **Toggle Bookmark:** Allows you to place or remove a bookmark at any trace recording entry.
 - **Clear All Bookmarks:** Clears all bookmarks in the **Trace** window.
 - **Previous Bookmark:** Allows you to jump to the previous bookmark in the **Trace** window.
 - **Next Bookmark:** Allows you to jump to the next bookmark in the **Trace** window.
- **Copying and saving the trace contents:** The trace record may be copied to the clipboard or copied to file via options in the Trace contextual menu.
 - The **Copy to Clipboard** command copies the highlighted trace record(s) to the clipboard.
 - The command **Copy to File** opens the **Save Trace Contents** dialog box, with details of saved trace files and user options for save parameters.

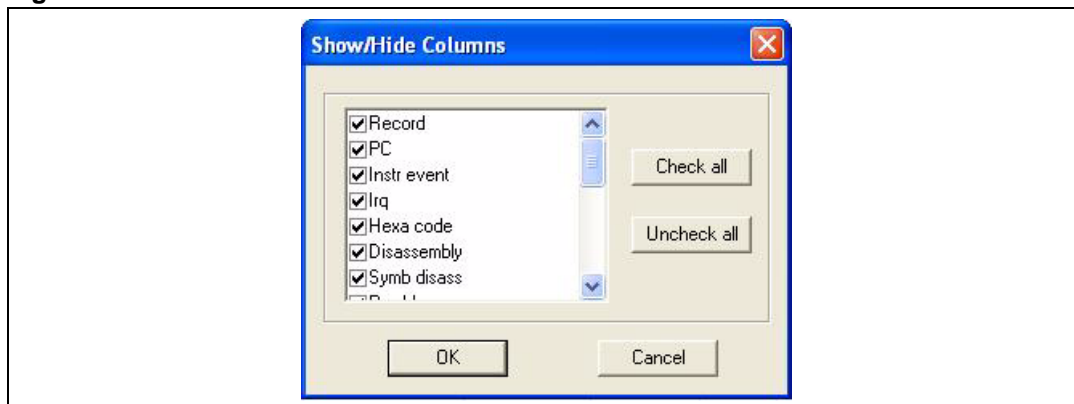
Figure 199. Saving trace contents



- **Column display:** Columns may be disabled if they not required and/or do not display any trace information for the particular trace situation. To activate/disable a column,

select **Column Display** in the Trace contextual menu. This opens a list of all the columns available in the trace record.

Figure 200. Show/hide columns



Columns may also be shifted right or left for convenience of use. Pick up the column header with the left mouse button and drag to the location required.

The **Trace** window in [Figure 201](#) has been set up to show only the **Record**, **PC**, **Instr event**, **Hexa code**, **Disassembly**, **R addr**, **R value**, **R event**, **W addr**, **W value** and **W event** columns.

Figure 201. Customized trace window

	Record	PC	Instr event	Hexa code	Disassembly	R addr	R value	R event	W addr	W value	W event
→	91	0x0080da	Processing								
		main.asm		46	mov var3,rom4	0x009001		Read stall			
	92	0x0080df	Decoding start	0x5590030102	MOV 0x0102,0x9003	0x009001	0x01	Data read	0x000101	0x01	Data write
	93	0x0080df	Processing			0x009003		Read stall			
	94	0x0080e4	Processing			0x009003	0x01	Data read	0x000102	0x01	Data write
	95	0x0080e4	Processing								
→		main.asm		47	ld a,var1						
	96	0x0080e4	Decoding start	0xc60100	LD A,0x0100						
→		main.asm		48	ld a,var2						
	97	0x0080e7	Decoding start	0xc60101	LD A,0x0101	0x000100	0xaa	Data read			
→		main.asm		49	mov var1,var2						
	98	0x0080ea	Decoding start	0x5501010100	MOV 0x0100,0x0101	0x000101	0x01	Data read			
	99	0x0080ef	Processing			0x000101	0x01	Data read	0x000100	0x01	Data write
→		main.asm		50	mov var2,var3						
	100	0x0080ef	Decoding start	0x5501020101	MOV 0x0101,0x0102						
	101	0x0080f4	Processing			0x000102	0x01	Data read	0x000101	0x01	Data write
	102	0x0080f4	Processing								
→		main.asm		51	mov var3,var4						
	103	0x0080f4	Decoding start	0x5501030102	MOV 0x0102,0x0103						
→		main.asm		52	mov var4,var5						
	104	0x0080f9	Decoding start	0x5501040103	MOV 0x0103,0x0104	0x000103	0x66	Data read	0x000102	0x66	Data write
	105	0x0080f9	Processing			0x000104		Read stall			
→		main.asm		54	ld a,var1						
	106	0x0080fe	Decoding start	0xc60100	LD A,0x0100	0x000104	0x66	Data read	0x000103	0x66	Data write
	107	0x0080fe	Processing			0x000100		Read stall			
→		main.asm		56	jrne loop						

9.1.3 Emulator commands

This option in the **Trace** contextual menu gives access to the following commands:

- **Clear Trace buffer:** This option erases all information currently contained in the Trace buffer.
- **Trace On:** When the trace function is on, records are numbered sequentially and stored in memory. The **Trace** window is automatically updated when the program is stopped. This option turns the Trace function on—re-clicking the option toggles the trace off. Note that you may also activate the Trace using the **Advanced Breakpoints**

window (via the BEM)—but doing so disables the Trace On/Off command in the Trace Contextual menu.

9.2 Coverage and profiling

Coverage and profiling provides insight into what portions of code are executed, what areas of memory are accessed, what functions are executed most often and how much time is spent in each function, thus helping you to:

- Obtain reliable data to decide the division of tasks between hardware and software, and select the appropriate MCU during the planning phase.
- Improve critical code and detect unexpected events in the coding and debugging phase.
- Improve time performance, reduce consumption, code size, and RAM size during the optimization phase.
- Detect holes in the test plan and obtain internal metrics during the validation phase.

A major advantage of running coverage and profiling on your application is that it does not require any code modification and has no impact on performance.

This section first explains the concepts of coverage and profiling, and provides detailed information on using the STVD graphical user interface to configure these features, to run a coverage and profiling session, and to understand and use the results:

- [Section 9.2: Coverage and profiling](#)
- [Section 9.2.1: Configuring the coverage and profiling settings](#)
- [Section 9.2.2: Running a coverage and profiling session](#)
- [Section 9.2.3: Reading coverage and profiling results](#)
- [Section 9.2.4: Typical examples of use](#)

The **code coverage** feature helps you identify which parts of code and data are used when the microcontroller runs your application. The STVD graphical user interface distinguishes between:

- Code coverage: indicates whether an instruction in the source code has been executed
- Data coverage: indicates whether a memory area for a variable or data has been accessed (for read or write)

The **code profiling** feature helps you identify which parts of code or data are used most often or least often during execution of your application. The STVD graphical user interface distinguishes between:

- Occurrence profiling: indicates the number of times an instruction is executed or data is accessed
- Time profiling: indicates the instructions on which the processor spends the most time

The coverage and profiling feature in STVD uses counters to gather information from the application execution. There are two types of counter:

- Occurrence counters for instructions and for variables (no unit)
- Time counters (in nano-seconds)

There are three counter sizes:

- Small 15-bit occurrence counters with a maximum value of 32767
- Large 30-bit occurrence counters with a maximum value of 1,073,741,823
- 36-bit time counters enabling to record 22.9 minutes of execution with a 50 MHz time base

Data occurrence counters are enabled when you select **Data coverage and occurrence profiling**.

Code occurrence and time counters are enabled when you select **Code coverage and profiling** (see [Section 9.2.1: Configuring the coverage and profiling settings](#), step 2.)

The counters record information during a coverage and profiling session as long as your application is running. If you start execution of your application after having started your profiling session, the counters remain at zero until the application starts running. When the application execution stops at a breakpoint, or at the end of the program, the counters are frozen. (They are unfrozen when the application resumes. When you stop the application, they are frozen definitively for the current session.)

When an occurrence counter reaches its maximum value, it remains at that value for the rest of the profiling session. In the results window, it is displayed with a greater than symbol (for example, > 32767). This is the default behavior. However, because the results of occurrence profiling are less meaningful if many occurrence counters reach the maximum value, you have the option to stop the application when any one of the counters reaches the maximum value (see [Section 9.2.1: Configuring the coverage and profiling settings](#), step 4.)

9.2.1 Configuring the coverage and profiling settings

Before you configure the coverage and profiling settings, you must ensure that:

- Your STice advanced emulation system is properly connected to your PC and to your application board.
- You have an open debugging session in STVD (see [Section 5.3 on page 167](#)).
- Your application is stopped, either because you have not started execution, or because execution is stopped at a breakpoint.

To configure coverage and profiling settings:

1. In the main window, from the **Debug Instrument** menu, select **Profiling Settings**.
The **Coverage and Profiling Settings** window is displayed, as shown in [Figure 202](#) and [Figure 203](#).
2. In the **General** section, select the mode:
 - **Data coverage and occurrence profiling**
Indicates whether data has been accessed, and the number of times data is accessed. See [Section 9.2.4 on page 276](#) for information on reading the results obtained when you run a profiling session in this mode.
 - **Code coverage and profiling (default)**
Provides time profiling information, and the number of times an instruction is executed. Indicates how much processor time is spent on each instruction. See [Section on page 272](#) for information on reading the results obtained in this mode.

Note: Coverage and profiling is incompatible with the trace analysis feature described in [Section 9.1: Trace recording](#).

Figure 202. Data coverage and occurrence profiling settings window

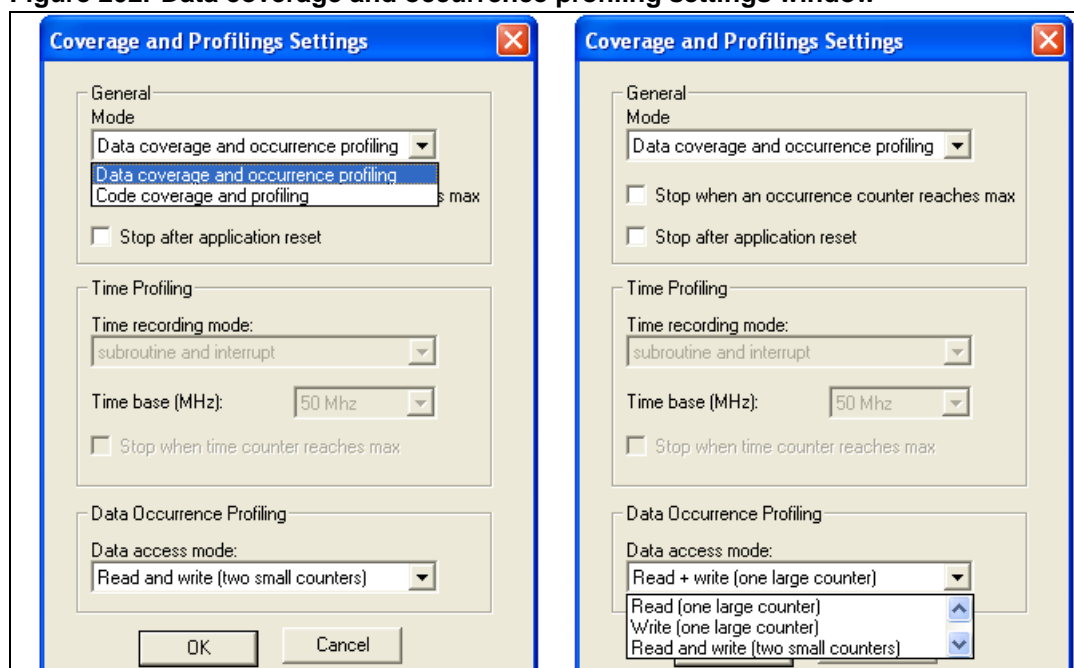
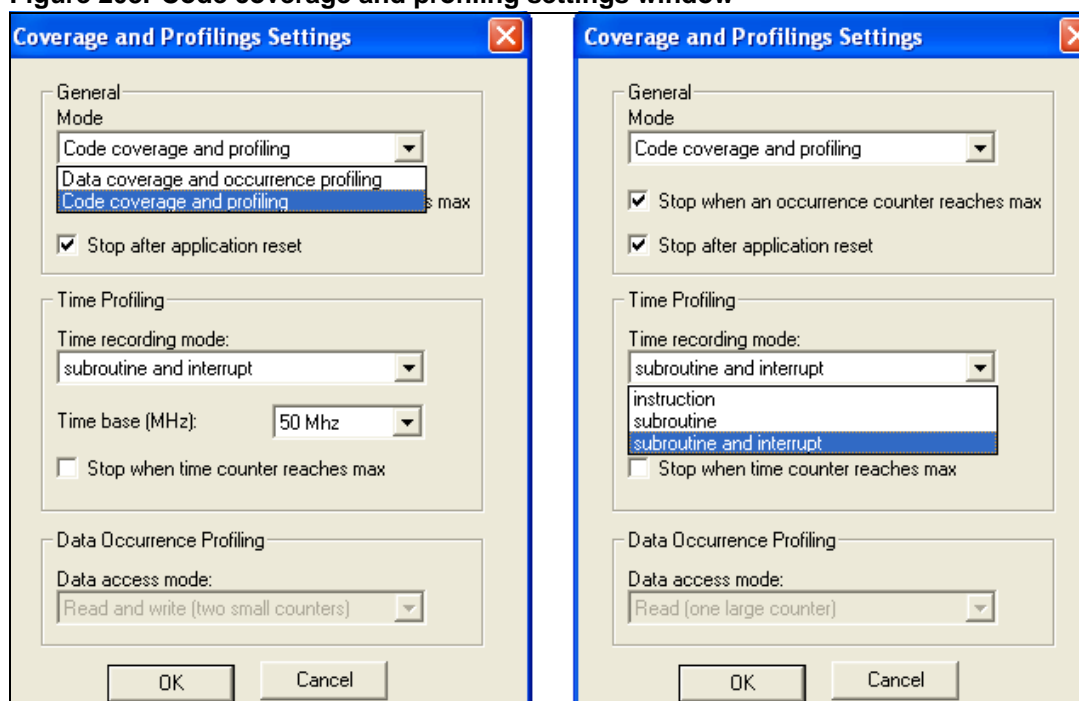


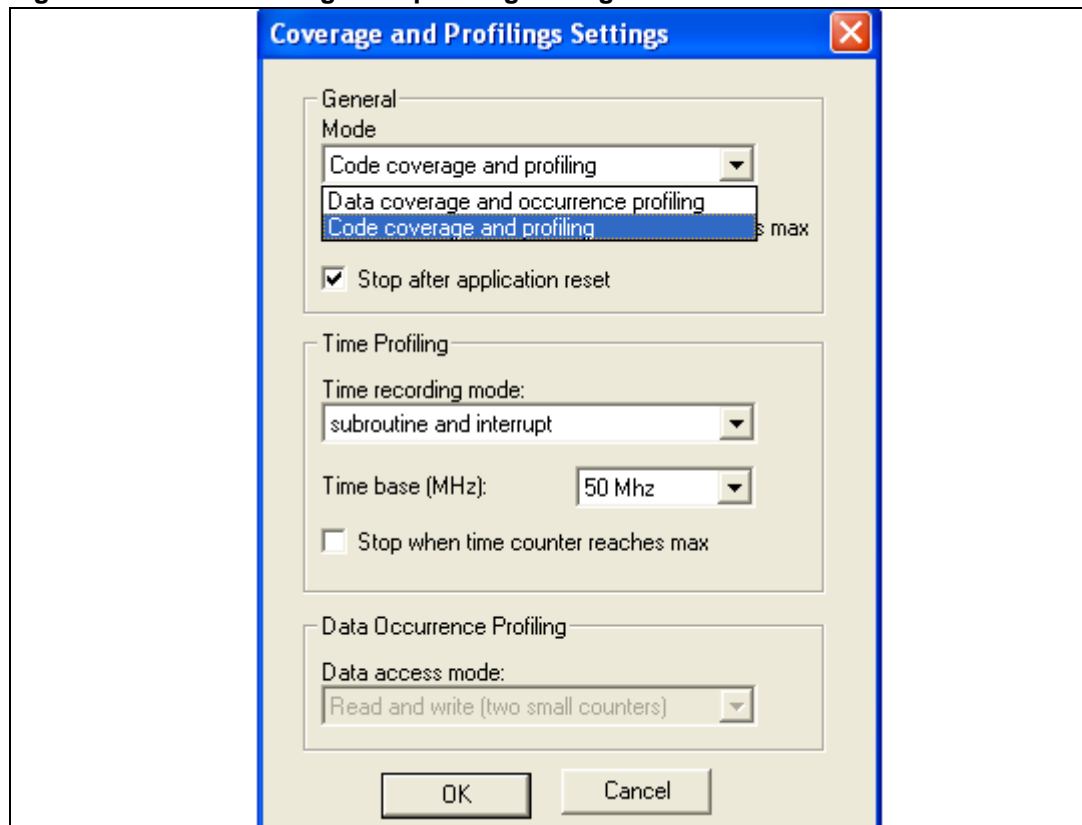
Figure 203. Code coverage and profiling settings window



3. If you have selected **Code coverage and profiling** in the **Time Profiling** section, select the **Time recording mode**:
- **Instruction:** Measures the time from the beginning of an instruction to the beginning of the next instruction, regardless of the time spent in subroutines. In the profiling results, all instructions are presented at the same level.
 - **Subroutine and interrupt (default):** Measures the time from the beginning of an instruction to the beginning of the next instruction, including the time spent in subroutine calls and in interrupt handler routines when an external interrupt stops the subroutine flow.
 - **Subroutine:** Measures the time from the beginning of an instruction to the beginning of the next instruction, including the time spent in subroutine calls. If an interrupt occurs during subroutine execution, the duration of the interrupt is not added to the duration of the call instruction.

In the two subroutine modes, the call instructions are handled in a special way. The number of occurrences of a call instruction only takes into account complete calls where the subroutine is executed from start to end. In the same way, the time spent at a subroutine call only takes into account complete calls. For example if a function "f" is called only once and the program is stopped within "f", the number of occurrences of "call f" will be 0. Its time will also be 0.

Figure 204. Code coverage and profiling settings window



You can select the frequency of the profiler time counter from the following values:

- 12.5 MHz
- 25 MHz
- 50 MHz (default)
- 100 MHz

It is advised to use a profiler frequency which is at least 3 times higher than the frequency of the microcontroller.

To have more precision, increase the frequency of the profiler time counter. Conversely, to have a longer recording time of program execution, decrease the frequency of the profiler time counter.

Note: This section is disabled if you have selected **Data coverage and occurrence profiling** in the previous step.

4. Decide whether you want to stop the application execution and profiling session when a counter reaches the maximum value or when the application is reset. When all options are enabled, the profiling session stops when any one of the conditions is met.
 - Place a checkmark in the **Stop when an occurrence counter reaches max** checkbox in the **General** section, to stop the execution when any one of the occurrence counters reaches the maximum value. Instruction execution occurrences are recorded in a large counter. You can configure the data read/write occurrence counters as described in step 5. Once the application is stopped, the user has to click on **Profiling Session Stop** to stop the session. Note that if you

have chosen two small counters to record read and write operations separately, the maximum value may be reached very quickly.

- Place a checkmark in the **Stop after application reset** checkbox in the **General** section, to stop the execution when a reset has occurred (due to watchdog, for example). In fact this option should always be checked unless your application should not be stopped, otherwise the time profiling results in the two subroutine modes will become erroneous. Once the application is stopped, the user must click on **Profiling Session Stop** to stop the session.
- Place a checkmark in the **Stop when time counter reaches max** checkbox in the **Time profiling** section, to stop the execution when the global profiler time counter reaches the maximum value. All time counters are recorded in 36-bit large counters. In fact this option should always be checked unless your application shouldn't be stopped as otherwise the time profiling results will become erroneous. Once the application is stopped, the user must click on **Profiling Session Stop** to stop the session.

Note: This option is disabled if you selected Data coverage and occurrence profiling.

5. In the **Data Occurrence Profiling** section, select the data access mode:

- **Read and write (two small counters) - default**
Records the number of read and write operations in two separate 15-bit counters. The maximum value each counter can reach is 32767.
- **Read (one large counter)**
Records the number of read operations only in one 30-bit counter. The maximum value the counter can reach is 1,073,741,823.
- **Write (one large counter)**
Records the number of write operations only in one 30-bit counter. The maximum value the counter can reach is 1,073,741,823.
- **Read + write (one large counter)**
Records the number of read and write operations in the same 30-bit counter. The maximum value the counter can reach is 1,073,741,823.

Note: This section is disabled if you selected Code coverage and profiling.

The occurrence counter for code execution is a large 30-bit counter.

6. Click OK. The settings that you configured will be used the next time you start a coverage and profiling session.

9.2.2 Running a coverage and profiling session

To run coverage and profiling on your application, you must first ensure that:

- Your STice advanced emulation system is properly connected to your PC and to your application board.
- You have an open debugging session in STVD (see [Section 5.3: Running an application on page 167](#)).
- Your application is stopped, either because you have not started execution, or because execution is stopped at a breakpoint.

If you have not configured the coverage and profiling settings, either the ones from the last session or the default ones will be used.

To start and stop a coverage and profiling session:

1. In the main window, from the **Debug Instrument** menu, select **Profiling Session Start**.
2. Start execution of your application or continue execution if it is stopped at a breakpoint.
3. Continue application execution until it reaches the next breakpoint, or until the end of the program. You can have as many breakpoints as you like, and they should be placed anywhere that you wish to examine further in the program.
4. In the main window, from the **Debug Instrument** menu, select **Profiling Session Stop**.

The results are displayed in the **Coverage and Profiling Analysis** window.

Note: In the lapse of time between starting the profiling session and starting your application, the coverage and profiling counters remain at zero. Each time application execution stops at a breakpoint, or at the end of the program, the counters are frozen.

The results displayed are dependent on the coverage and profiling settings. Configuration settings are described in [Section 9.2.1](#). For information on interpreting the results, refer to [Section 9.2.3: Reading coverage and profiling results](#).

9.2.3 Reading coverage and profiling results

When you run a coverage and profiling session during the execution of your application, the test results depend on the profiling mode you select:

- [Code coverage and profiling on page 272](#)
- [Data coverage and occurrence profiling on page 274](#)

The STVD graphical user interface presents the coverage and profiling analysis results as shown in [Figure 205](#), and [Figure 207](#).

Code coverage and profiling

To obtain the coverage and profiling results such as those shown in the **Coverage and Profiling Analysis** window in [Figure 205](#), you must run a profiling session with the option **Code coverage and profiling** selected in the **Mode** field (see [Section 9.2.1](#), step 2.) For information on running a profiling session, see [Section 9.2.2](#).

When you stop the profiling session, the **Coverage and Profiling Analysis** window is displayed. It contains two tabs:

- A [Functions/Instructions View](#)
- A [Source View](#), which is displayed optionally

Functions/Instructions View

On the **Functions/Instructions view**, the test results are displayed by function or instruction, with the following fields for each function or instruction:

- **Function**
Indicates the name of the function in the application source files; function names expand to show source lines and instructions.
- **Location**
Indicates the start address in memory in hexadecimal format.
- **Time (ns)**
Indicates the total time spent executing code in the selected component (function, or C line or Assembler instruction) in nanoseconds.

- **Contextual percentage**

Indicates the time spent executing code in the function as a percentage of the total processing time for the containing object in the hierarchy of elements in your program. It may be a percentage of C lines, or functions, or the total.

- **Percentage of total**

For a function, indicates the lines that are executed as a percentage of the total number of lines in the function. A value of zero indicates that the line was not executed.

For a line, indicates the instructions that are executed as a percentage of the total instructions in the line. A value of zero indicates that the instruction was not executed.

- **Average time (ns)**

Indicates the average time spent executing the selected component (function, or C line or assembler instruction) in nanoseconds.

- **Interrupt (Yes/No)**

Indicates whether an interrupt occurred during execution of the instruction, line, or function.

- **Percentage covered**

For a function, indicates the lines that are executed as a percentage of the total number of lines in the function. A value of zero indicates that the line was not executed.

For a line, indicates the instructions that are executed as a percentage of the total instructions in the line. A value of zero indicates that the instruction was not executed.

- **Calls**

Indicates the number of external calls to the function, line of code or instruction (occurrences).

- **Source file**

Indicates the source file (C or Assembler) that contains the function.

Figure 205 shows the coverage and profiling test results organized by function. You can obtain greater detail for each function, by clicking on the plus (+) sign to expand it to display the list of lines and instructions within the function as shown in *Figure 205*.

Figure 205. Code coverage and profiling analysis: functions/instructions view

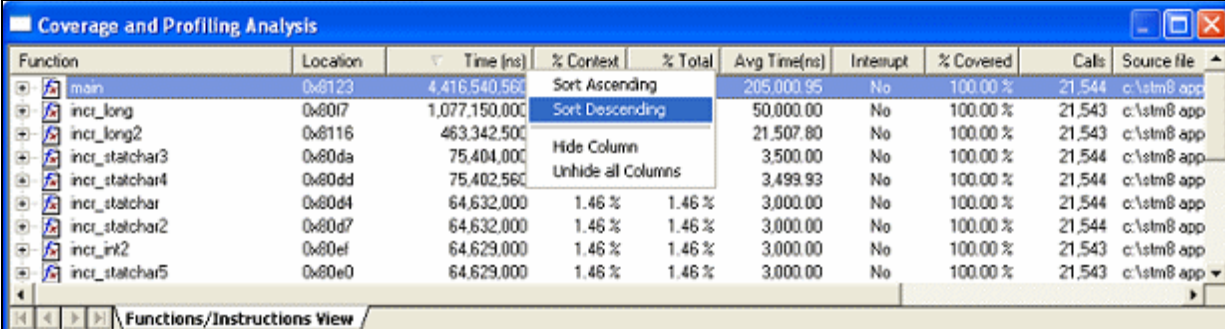
Coverage and Profiling Analysis									
Function	Location	Time (ns)	% Context	% Total	Avg Time(ns)	Interrupt	% Covered	Calls	Source file
incr_int	0x80ed	25,087,660	1.23 %	1.23 %	1,260.00	No	100.00 %	19,911	c:\stm8 applis\...
incr_int2	0x80ef	29,866,500	1.46 %	1.46 %	1,500.00	No	100.00 %	19,911	c:\stm8 applis\...
incr_int3	0x80f1	24,688,400	1.21 %	1.21 %	1,240.00	No	100.00 %	19,910	c:\stm8 applis\...
incr_int4	0x80f3	24,688,400	1.21 %	1.21 %	1,240.00	No	100.00 %	19,910	c:\stm8 applis\...
incr_int5	0x80f5	24,688,400	1.21 %	1.21 %	1,240.00	No	100.00 %	19,910	c:\stm8 applis\...
incr_long	0x80f7	497,750,000	24.39 %	24.39 %	25,000.00	No	100.00 %	19,910	c:\stm8 applis\...
69 long incr_long(long l)	0x80f7	4,778,400	0.96 %	0.23 %	240.00	No		19,910	
72 long l1 = l + 1;	0x80f9	323,736,600	65.04 %	15.86 %	16,260.00	No		19,910	
LDW X,SP	0x80f9	5,176,600	1.60 %	0.25 %	260.00	No		19,910	
ADDW X,#0x0007	0x80fa	9,955,000	3.08 %	0.49 %	500.00	No		19,910	
CALL c_iter	0x80fd	114,283,400	35.30 %	5.60 %	5,740.00	No		19,910	
LD A,#0x01	0x8100	5,176,600	1.60 %	0.25 %	260.00	No		19,910	
CALL c_jadc	0x8102	59,730,000	18.45 %	2.93 %	3,000.00	No		19,910	
LDW X,SP	0x8105	4,778,400	1.48 %	0.23 %	240.00	No		19,910	
ADDW X,#0x0001	0x8106	9,955,000	3.08 %	0.49 %	500.00	No		19,910	
CALL c_tcl	0x8109	114,681,600	35.42 %	5.62 %	5,760.00	No		19,910	
73 return l1;	0x810c	169,235,000	34.00 %	8.29 %	8,500.00	No		19,910	
incr_long2	0x8116	213,910,900	10.48 %	10.48 %	10,743.89	No	100.00 %	19,910	c:\stm8 applis\...
incr_statchar	0x80d4	29,866,500	1.46 %	1.46 %	1,500.00	No	100.00 %	19,911	c:\stm8 applis\...

It is possible to sort the columns according to alphabetical order for **Function**, **Interrupt** and **Source file** columns, and in numerical order for the other columns. The simplest way is to click on the header of the column you wish to sort. For example if you click on **Time (ns)**, the lines will be sorted in ascending order according to the execution time. If you click again on **Time (ns)**, the lines will be sorted in descending order. In this manner it is quite easy to spot the functions which take a long time to be executed. By default, the functions are sorted in ascending alphabetical order according to their names.

Another way to sort the lines is to use the contextual menu which appears when you right-click on the column header.

This contextual menu can also be used to remove a column: click on **Hide Column**. To add it again, just click on **Unhide all Columns**.

Figure 206. Sort results



Function	Location	Time (ns)	% Context	% Total	Avg Time(ns)	Interrupt	% Covered	Calls	Source file
main	0x8123	4,416,540,567	Sort Ascending		205,000.95	No	100.00 %	21,544	c:\stm8 app
incr_long	0x8017	1,077,150,000	Sort Descending		50,000.00	No	100.00 %	21,543	c:\stm8 app
incr_long2	0x8116	463,342,500			21,507.80	No	100.00 %	21,543	c:\stm8 app
incr_statchar3	0x80da	75,404,000	Hide Column		3,500.00	No	100.00 %	21,544	c:\stm8 app
incr_statchar4	0x80dd	75,402,560	Unhide all Columns		3,499.93	No	100.00 %	21,544	c:\stm8 app
incr_statchar	0x80d4	64,632,000	1.46 %	1.46 %	3,000.00	No	100.00 %	21,544	c:\stm8 app
incr_statchar2	0x80d7	64,632,000	1.46 %	1.46 %	3,000.00	No	100.00 %	21,544	c:\stm8 app
incr_int2	0x80ef	64,629,000	1.46 %	1.46 %	3,000.00	No	100.00 %	21,543	c:\stm8 app
incr_statchar5	0x80e0	64,629,000	1.46 %	1.46 %	3,000.00	No	100.00 %	21,543	c:\stm8 app

Data coverage and occurrence profiling

When you run a coverage and profiling session with the option **Data coverage and occurrence profiling** selected in the **Mode** field (see [Section 9.2.1](#), step 2.), you obtain **coverage and profiling** information on the data.

For information on configuring a profiling session, see [Section 9.2.1](#).

The results of the analysis are displayed in the **Coverage and Profiling Analysis** window, in the [Data View](#) tab, see description [on page 274](#).

Data View

On the **Data View**, two kinds of line are displayed depending on the size of the data: single byte data or multiple byte data. If the variable spans more than one byte, a summarized view is shown with the address range and the maximum number(s) of accesses over the range. The analysis results are displayed with the following fields for each variable or data item:

- **Memory location**

If the data occupies more than one byte, it indicates the address range in memory where the data is stored (in hexadecimal format) otherwise if the data is just a byte, it indicates the address of this byte.

A multiple byte variable can be expanded to show each address of the variable address ranges by clicking on the plus (+) sign.

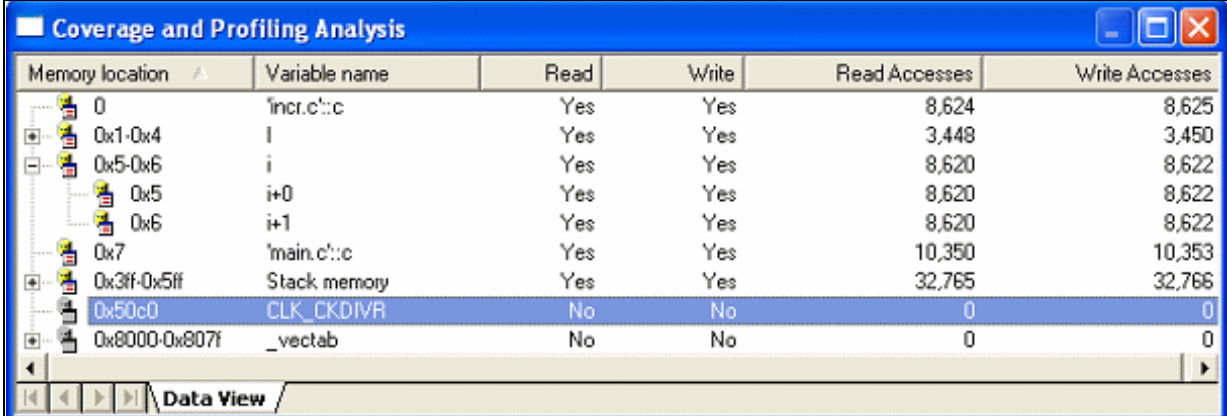
- **Variable name**

Indicates the name of the variable in the C or assembler source, possibly suffixed with the '+' sign followed by the offset relative to the start address of the variable, if the current line results from a multiple byte variable expansion.

- **Read (Yes/No)**
Indicates whether the variable was read or not.
- **Write (Yes/No)**
Indicates whether the variable was written to or not.
- **Read accesses**
Indicates the maximum number of byte accesses from the application for reading the variable or just the number of read accesses for a single byte.
- **Write accesses**
Indicates the maximum number of byte accesses from the application for writing to the variable or just the number of write accesses for a single byte.
- **Reads or Write accesses**
Indicates the number of accesses from the application to read or write the variable for a single data byte, otherwise indicates the maximum number of byte accesses.
Depending on the data access mode, you may get one of these 4 cases:
 - Read access only
 - Write access only
 - Read or Write access
 - Read and Write access

Figure 207 shows coverage and profiling analysis results organized by memory location and variable. Right-click anywhere in the data results window to display the contextual menu.

Figure 207. Data coverage and profiling analysis: data view



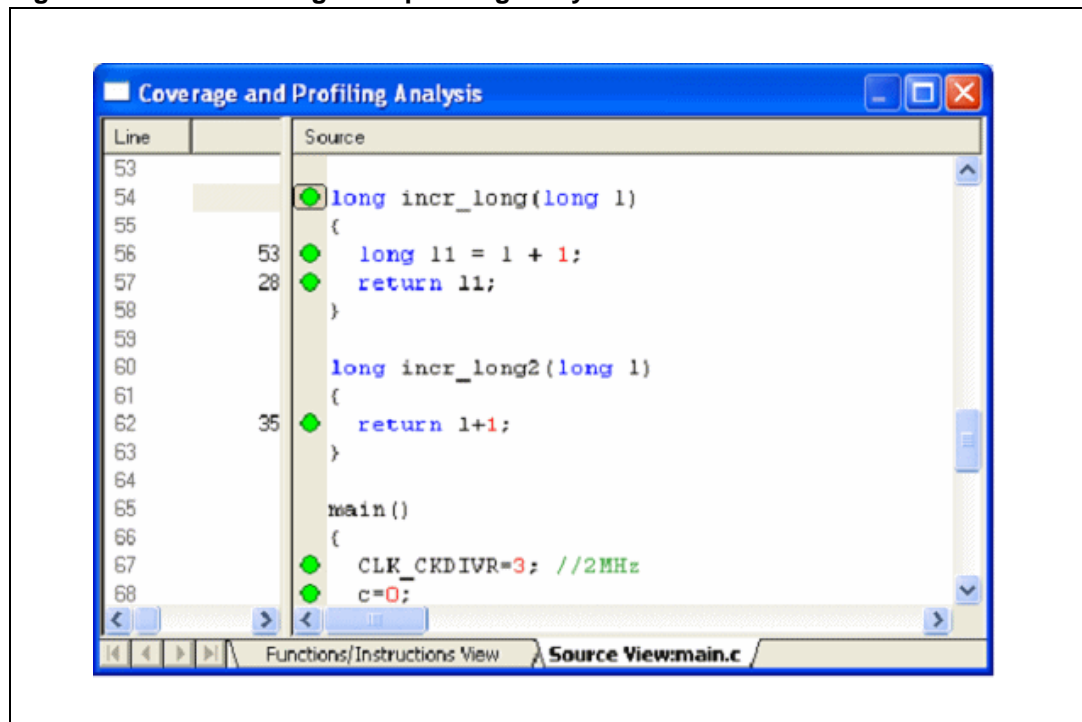
Memory location	Variable name	Read	Write	Read Accesses	Write Accesses
0	'inc.c':c	Yes	Yes	8,624	8,625
0x1-0x4	i	Yes	Yes	3,448	3,450
0x5-0x6	i	Yes	Yes	8,620	8,622
0x5	i+0	Yes	Yes	8,620	8,622
0x6	i+1	Yes	Yes	8,620	8,622
0x7	'main.c':c	Yes	Yes	10,350	10,353
0x3ff-0x5ff	Stack memory	Yes	Yes	32,765	32,766
0x50c0	CLK_CKDIVR	No	No	0	0
0x8000-0x807f	_vectab	No	No	0	0

Source View

The **Source View** displays the source file containing a selected function in the text editor. It opens the file at the location where the function appears.

To display the **Source View**, you must highlight the function that you want to look at in the **Functions/Instructions View**, then from the contextual menu, select **Go to Source View**. Alternatively, you can double-click on the function or instruction that you want to examine in the source code.

Figure 208. Data coverage and profiling analysis: source view



The green spot in the margin identifies the lines of code that are executed. The left-hand side of the window displays the **Functions/Instructions View**. You can resize this part of the window or make it disappear completely by clicking on the partition and dragging it to a new position.

Note: The **Source View** tab is not present at the bottom of the **Coverage and Profiling Analysis** window until you have displayed source code for the first time.

9.2.4 Typical examples of use

This section explains how to use the coverage and profiling feature to improve the efficiency and speed of execution of applications that you develop with STVD for ST microcontrollers. Coverage and profiling analysis is an iterative process, where at each new iteration, you can identify and measure the performance gains obtained from your successive code optimizations.

Coverage information has the following practical purposes:

- Detection of holes in the validation test plan by showing that some events are not processed, see [Detecting holes in the validation plan on page 277](#)
- Detection of dead code by showing that some areas of code are never executed and never can be, see [Detecting dead code on page 279](#)
- Detection of global variables that are never used, see [Optimizing variables on page 280](#)
- Detection of global variables that are read but never written to, see [Optimizing variables on page 280](#)
- Checking that non-handled interrupts are never called, see [Processing interrupts more efficiently on page 280](#).

Caution: Code that is not executed because it is not properly covered in the validation test plan can be mistaken for dead code. Therefore, when you are running coverage tests to detect dead code, make sure that the lines that are not executed are not due to an oversight in the test plan before deleting them.

Data occurrence information can be used to:

- Optimize stack usage, see [Monitoring stack space usage on page 279](#)
- Detect that a global variable is located in the wrong memory section, see [Optimizing variables on page 280](#)

Code occurrence information is useful for determining the number of occurrences of an interrupt routine, see [Processing interrupts more efficiently on page 280](#).

Time profiling is the best method for:

- Detecting and investigating bottlenecks, see [Detecting and investigating bottlenecks on page 280](#)
- Assessing the performance of time-critical code, see [Assessing the performance of time-critical code on page 283](#)
- Evaluate the time required to process an interrupt, see [Processing interrupts more efficiently on page 280](#)

Detecting holes in the validation plan

When you run a test suite to validate an application, you must ensure that it covers all areas of code. In the results displayed in the **Coverage and Profiling Analysis** window, the functions that are not called appear in gray, as shown in [Figure 209](#).

Figure 209. Lines not covered by test suite

Function	Location	Time (ns)	% Context	% Total	Avg Time(ns)	Interrupt	% Covered	Calls	Source
computeOutputValue	0x8175	64,690,500	90.72 %	90.72 %	640,500.00	No	100.00 %	101	c:\stm8
initApplication	0x8125	4,000	0.01 %	0.01 %	4,000.00	No	100.00 %	1	c:\stm8
main	0x8156	71,220,020	99.87 %	99.87 %	698,235.44	No	75.00 %	102	c:\stm8
55 void main (void)	0x8156	500	0.00 %	0.00 %	500.00	No		1	
61 initApplication();	0x8157	6,000	0.01 %	0.01 %	6,000.00	No		1	
66 event = waitForEvent();	0x8159	5,813,000	8.16 %	8.15 %	56,990.20	No		102	
68 switch (event)	0x815d	306,020	0.43 %	0.43 %	3,000.20	No		102	
70 case EVENT1: output...	0x8167	64,993,500	91.26 %	91.14 %	643,500.00	No		101	
71 break;	0x816c	101,000	0.14 %	0.14 %	1,000.00	No		101	
73 case EVENT2: output...	0x816e	0	0.00 %	0.00 %	0.00	No		0	
74 break;	0x8171	0	0.00 %	0.00 %	0.00	No		0	
NonHandledInterrupt	0x861d	0	0.00 %	0.00 %	0.00	No	0.00 %	0	c:\stm8
waitForEvent	0x8129	5,558,000	7.79 %	7.79 %	9,081.70	No	100.00 %	612	c:\stm8

In this example, EVENT2 is not processed. The possible causes are:

- EVENT2 is not called by the test suite because it was omitted
- EVENT2 did not occur during the lapse of time during which the test suite was executed.

By looking at the source code for the test suite, shown in [Figure 210](#), it is obvious that EVENT2 is taken into account in the test suite. Therefore, the test suite has not been executed long enough for EVENT2 to occur.

[Figure 211](#) shows the data **Coverage and Profiling Analysis** window after a longer wait. The EVENT2 function appears in green.

Figure 210. Source view of test suite

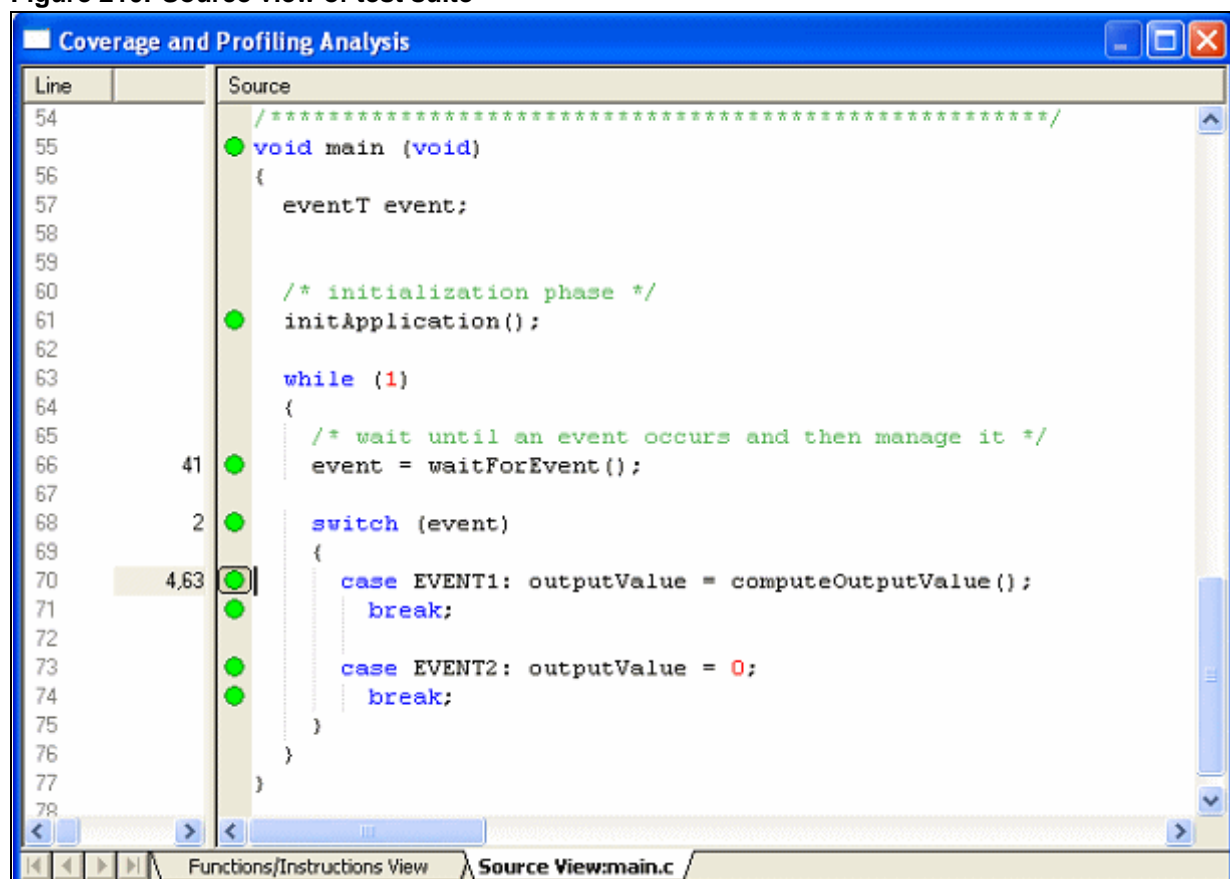


Figure 211. Coverage and Profiling Analysis of test suite after a longer wait

Function	Location	Time (ns)	% Context	% Total	Avg Time(ns)	Interrupt	% Covered
computeOutputValue	0x8175	2,845,591,560	90.83 %	90.83 %	640,466.25	No	100.00 %
initApplication	0x8125	4,000	0.00 %	0.00 %	4,000.00	No	100.00 %
main	0x8156	3,132,614,060	100.00 %	100.00 %	698,309.00	No	100.00 %
55 void main (void)	0x8156	500	0.00 %	0.00 %	500.00	No	
61 initApplication();	0x8157	6,000	0.00 %	0.00 %	6,000.00	No	
66 event = waitForEvent();	0x8159	255,659,000	8.16 %	8.16 %	56,990.42	No	
68 switch (event)	0x815d	13,479,500	0.43 %	0.43 %	3,004.79	No	
70 case EVENT1: outputValue = computeOutputValue();	0x8167	2,858,919,560	91.26 %	91.26 %	643,466.06	No	
71 break;	0x816c	4,442,000	0.14 %	0.14 %	1,000.00	No	
73 case EVENT2: outputValue = 0;	0x816e	64,500	0.00 %	0.00 %	1,500.00	No	
74 break;	0x816e	21,500	33.33 %	0.00 %	500.00	No	
LDW 0x00X	0x816f	43,000	66.67 %	0.00 %	1,000.00	No	
74 break;	0x8171	43,000	0.00 %	0.00 %	1,000.00	No	
NonHandledInterrupt	0x86fd	0	0.00 %	0.00 %	0.00	No	0.00 %
waitForEvent	0x8129	244,444,000	7.80 %	7.80 %	9,081.74	No	100.00 %
34 eventT waitForEvent(void)	0x8129	4,486,000	1.84 %	0.14 %	1,000.00	No	
39 for (i=0; i < 5; i++)	0x812a	195,141,000	79.83 %	6.23 %	7,250.00	No	
41 if (iEventTick > 100)	0x813e	13,436,500	5.50 %	0.43 %	2,995.21	No	
43 iEventTick = 0;	0x8145	64,500	0.03 %	0.00 %	1,500.00	No	
44 return EVENT2;	0x8148	64,500	0.03 %	0.00 %	1,500.00	No	
48 iEventTick++;	0x814c	13,329,000	5.45 %	0.43 %	3,000.00	No	
49 return EVENT1;	0x8153	17,922,500	7.33 %	0.57 %	3,995.21	No	

Detecting dead code

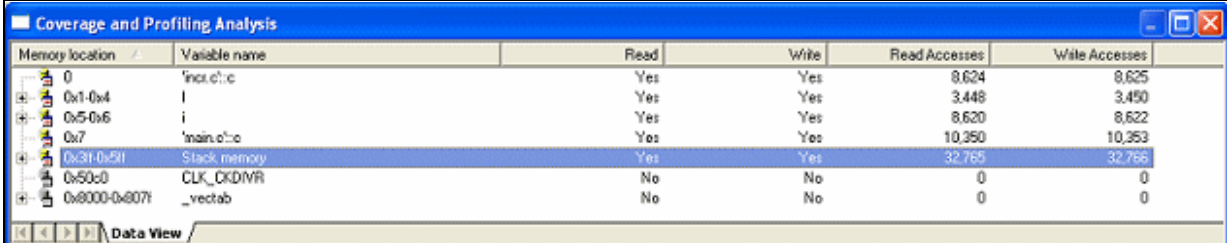
When you run a coverage and profiling session, the areas of code that are not executed are associated with a gray icon on the **Functions/Instructions View** tab in the **Coverage and Profiling Analysis** window. There are two complementary methods for detecting dead code:

- By building the code. During the build, the linker removes any unused functions. This is a static analysis.
- By running a coverage and profiling session. This dynamic coverage analysis shows the functions or logical blocks of code that are dependent upon conditions that never occur.

Monitoring stack space usage

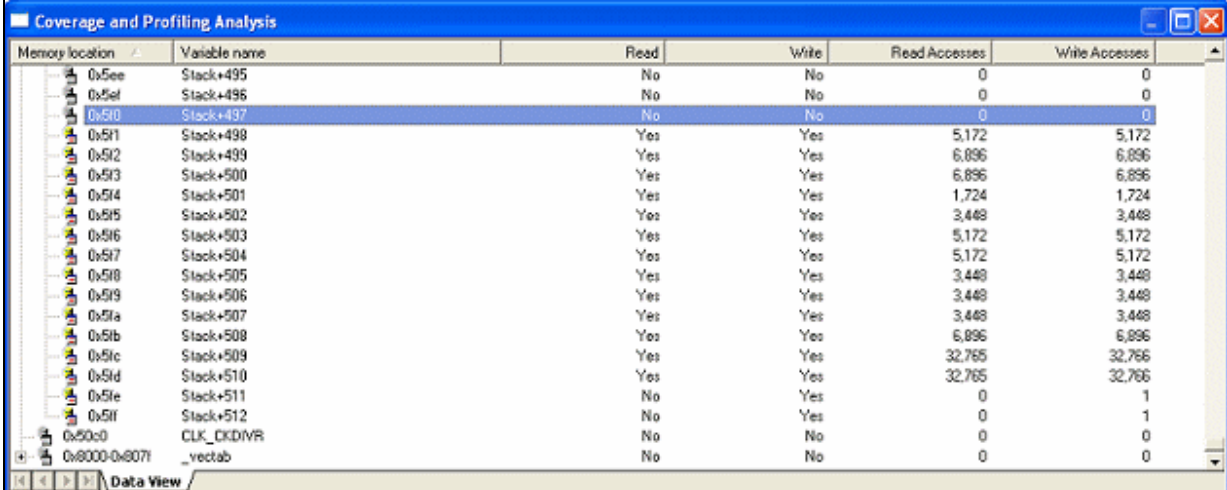
Run a coverage and profiling session with the option **Data coverage and occurrence profiling**. In the **Data View** tab of the **Coverage and Profiling Analysis** window, click on the **Stack memory** item and expand it to see detailed information on stack usage.

Figure 212. Stack memory in Data View tab of the Coverage and Profiling Analysis



Memory location	Variable name	Read	Write	Read Accesses	Write Accesses
0	'incl.c:c	Yes	Yes	8,624	8,625
0x1-0x4	i	Yes	Yes	3,448	3,450
0x5-0x6	i	Yes	Yes	8,620	8,622
0x7	'main.c:c	Yes	Yes	10,350	10,353
0x3f-0x5ff	Stack memory	Yes	Yes	32,765	32,766
0x50c0	CLK_OKDIVR	No	No	0	0
0x8000-0x807f	_vectab	No	No	0	0

Figure 213. Stack space details



Memory location	Variable name	Read	Write	Read Accesses	Write Accesses
0x5ee	Stack+495	No	No	0	0
0x5ef	Stack+496	No	No	0	0
0x5f0	Stack+497	No	No	0	0
0x5f1	Stack+498	Yes	Yes	5,172	5,172
0x5f2	Stack+499	Yes	Yes	6,896	6,896
0x5f3	Stack+500	Yes	Yes	6,896	6,896
0x5f4	Stack+501	Yes	Yes	1,724	1,724
0x5f5	Stack+502	Yes	Yes	3,448	3,448
0x5f6	Stack+503	Yes	Yes	5,172	5,172
0x5f7	Stack+504	Yes	Yes	5,172	5,172
0x5f8	Stack+505	Yes	Yes	3,448	3,448
0x5f9	Stack+506	Yes	Yes	3,448	3,448
0x5fa	Stack+507	Yes	Yes	3,448	3,448
0x5fb	Stack+508	Yes	Yes	6,896	6,896
0x5fc	Stack+509	Yes	Yes	32,765	32,766
0x5fd	Stack+510	Yes	Yes	32,765	32,766
0x5fe	Stack+511	No	Yes	0	1
0x5ff	Stack+512	No	Yes	0	1
0x50c0	CLK_OKDIVR	No	No	0	0
0x8000-0x807f	_vectab	No	No	0	0

By moving to the bottom of the stack, you can see how many bytes of stack have been used. If you are sure you have executed your application in the case of maximum stack use, you can decide to allocate some of your variables to the unused stack zone should the data memory dedicated to variables be full.

This can also help you to know if your function nesting level can be increased or not.

Optimizing variables

Unused variables: The results of a coverage and profiling session can help you examine how you use variables in your code. First, it shows which variables are simply not used at all, and therefore can be removed. Unused variables are shown in gray on the **Data View** tab in the **Coverage and Profiling Analysis** window.

Inadequate variable type: It also shows where you can more effectively replace a variable by another structure. For example, you might be able to simplify a structure type variable (such as the date), by deleting unused fields (day, or month, or year).

In particular, global variables that are only read and never written to can be more efficiently replaced by a constant or a define. The compiler can thus optimize the code and save a RAM variable and one or two bytes of Flash memory.

Location in memory: At last, when a variable is indeed useful, data occurrence counting can help you optimize its placement in memory. In some cases, a variable that appears only once in the code can actually be accessed very often (if it is in a loop for example). By moving this variable to page zero, you will optimize execution speed and save memory.

Processing interrupts more efficiently

Coverage and profiling analysis can help you identify:

- Handled interrupts never called
- Interrupts that occur very often, where optimization of processing time can have a big impact
- The gain obtained from your code optimizations

Detecting and investigating bottlenecks

Detecting bottlenecks in the execution of your code is a prime example of how profiling information helps you make very big gains in efficiency in a small number of key areas of your code. This is a very different approach from the compiler optimizations where the efficiency gain is small throughout the code. These approaches are, therefore, complementary.

This section uses an illustrated example to guide you through the main steps of the process of detecting and investigating bottlenecks

1. Configure the Code coverage and profiling option.

Bottleneck detection is based on time profiling information. Therefore, when you start your profiling session, make sure you have selected the **Code coverage and Profiling** option in the **Coverage and Profiling** settings window (see [Section 9.2.1: Configuring the coverage and profiling settings on page 267](#)).

2. Start a profiling session, and run the application from beginning to end.

For information on running a profiling session, refer to [Section 9.2.2 on page 271](#). When the application has executed to the end, you can stop the profiling session. The **Coverage and Profiling Analysis** window is displayed.

3. Identify the most time consuming code block.

On the **Functions/Instructions View** tab, you can identify the code block that is the most time consuming in the Percentage of Focus column. Expand the block and look at the most time-consuming function or instruction. In [Figure 214](#), the most time-consuming function is `ComputeOutputValue` with 90.83% of the total processing time.

- Expand each code block in turn.

Expand successively the most time-consuming functions and instructions to determine on which lines of code the most time is spent. [Figure 214](#) shows how to expand the main function to locate the ComputeOutputValue call. This call accounts for approximately 91.12% of the processing time of the main function and 99.84% of the processing time of line 70.

Figure 214. Bottleneck detection: top level

Function	Location	Time (ns)	% Context	% Total	Avg Time(ns)	Interrupt	% Covered	Calls	Source file
computeOutputValue	0x8175	4,616,586,660	90.83 %	90.83 %	640,480.94	No	100.00 %	7,208	c:\stm8 appl
initApplication	0x8125	4,000	0.00 %	0.00 %	4,000.00	No	100.00 %	1	c:\stm8 appl
main	0x8156	5,082,305,160	100.00 %	100.00 %	698,214.75	No	100.00 %	7,279	c:\stm8 appl
55 void main [void]	0x8156	500	0.00 %	0.00 %	500.00	No		1	
61 initApplication();	0x8157	6,000	0.00 %	0.00 %	6,000.00	No		1	
66 event = waitForEvent();	0x8159	414,832,000	8.16 %	8.16 %	56,990.25	No		7,279	
68 switch (event)	0x815d	21,872,500	0.43 %	0.43 %	3,004.88	No		7,279	
70 case EVENT1: outputValue = computeOutputValue();	0x8167	4,638,209,660	91.26 %	91.26 %	643,480.81	No		7,208	
CALL computeOutputValue	0x8167	4,631,002,660	99.84 %	91.12 %	642,480.94	No		7,208	
LDW 0x00X	0x816a	7,207,000	0.16 %	0.14 %	1,000.00	No		7,207	
71 break;	0x816c	7,207,000	0.14 %	0.14 %	1,000.00	No		7,207	
73 case EVENT2: outputValue = 0;	0x816e	106,500	0.00 %	0.00 %	1,500.00	No		71	
74 break;	0x8171	71,000	0.00 %	0.00 %	1,000.00	No		71	
NonHandledInterrupt	0x86fd	0	0.00 %	0.00 %	0.00	No	0.00 %	0	c:\stm8 appl
waitForEvent	0x8129	396,634,500	7.80 %	7.80 %	9,081.71	No	100.00 %	43,674	c:\stm8 appl

[Figure 215](#) shows that when you expand the ComputeOutputValue function, the most time-consuming instruction (highlighted in the illustration), is the one that computes the result. By expanding this instruction, you can view the most time-consuming calls, shown in [Figure 216](#).

Figure 215. Bottleneck detection: intermediate level

Function	Location	Time (ns)	% Context	% Total	Avg Time(ns)	Interrupt	% Covered	Calls	Source file
computeOutputValue	0x8175	4,616,586,660	90.83 %	90.83 %	640,480.94	No	100.00 %	7,208	c:\stm8 appl
6 unsigned int computeOutputValue(void)	0x8175	7,208,000	0.16 %	0.14 %	1,000.00	No		7,208	
13 result = (unsigned int) (MAX_PWM_VALUE/2 * (1.0+sin(index*PI/10)))	0x8176	4,555,326,160	98.67 %	89.63 %	631,981.94	No		7,208	
14 if (index >= 20) Index = 0;	0x81a3	25,224,500	0.55 %	0.50 %	3,900.00	No		7,207	
16 return result;	0x81ae	28,828,000	0.62 %	0.57 %	4,000.00	No		7,207	
initApplication	0x8125	4,000	0.00 %	0.00 %	4,000.00	No	100.00 %	1	c:\stm8 appl
main	0x8156	5,082,305,160	100.00 %	100.00 %	698,214.75	No	100.00 %	7,279	c:\stm8 appl
55 void main [void]	0x8156	500	0.00 %	0.00 %	500.00	No		1	
61 initApplication();	0x8157	6,000	0.00 %	0.00 %	6,000.00	No		1	
66 event = waitForEvent();	0x8159	414,832,000	8.16 %	8.16 %	56,990.25	No		7,279	
68 switch (event)	0x815d	21,872,500	0.43 %	0.43 %	3,004.88	No		7,279	
70 case EVENT1: outputValue = computeOutputValue();	0x8167	4,638,209,660	91.26 %	91.26 %	643,480.81	No		7,208	
CALL computeOutputValue	0x8167	4,631,002,660	99.84 %	91.12 %	642,480.94	No		7,208	
LDW 0x00X	0x816a	7,207,000	0.16 %	0.14 %	1,000.00	No		7,207	
71 break;	0x816c	7,207,000	0.14 %	0.14 %	1,000.00	No		7,207	

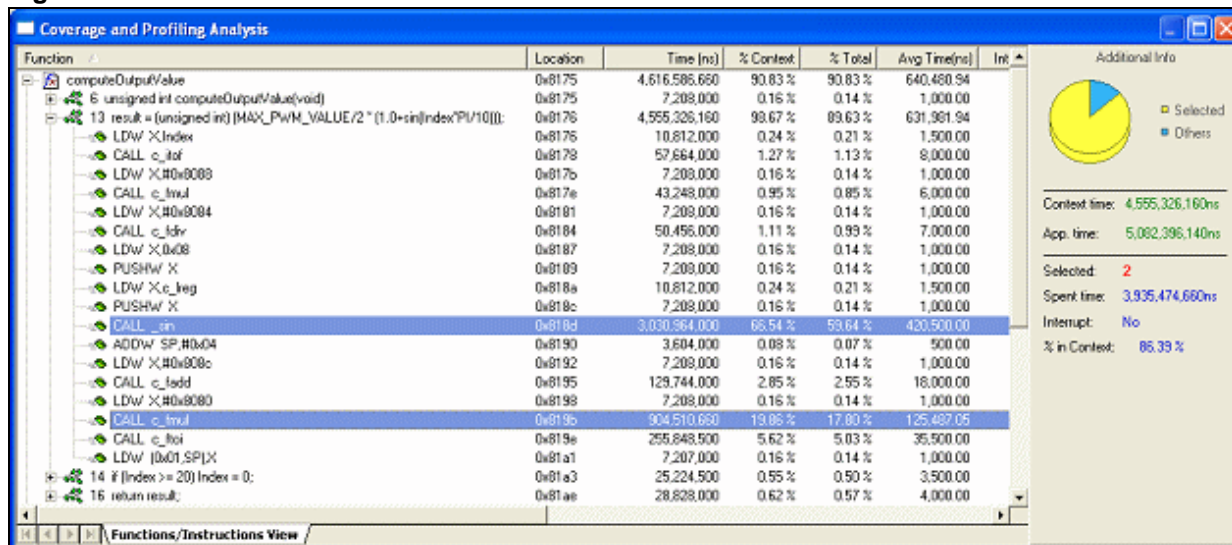
- Determine the cause of the bottleneck.

Examine the most time-consuming lines of code to determine whether you can fulfill the same task with a more efficient structure or method. Thus, by successively zooming in on the code down to the call level, you can identify very precisely the cause of a bottleneck. In [Figure 216](#), the most time-consuming calls are highlighted:

- A call to a sinus function (CALL _sin)
- A call to a float function (CALL c_fm1)

Therefore, an effective way of saving processing time, would be to replace the dynamic call to the sinus function by a static table of computed sinus values, as shown in the code example that follows. You could also decide to compute the result as an integer to avoid calculating a float.

Figure 216. Bottleneck detection: bottom level



```

static unsigned int result=0;
unsigned int Speed = 100;
unsigned int computeOutputValue(void)
{
static const unsigned int SinTab[] = {
(unsigned int) (MIN_SINUS_VALUE+0.500000*SINUS_AMPLITUDE),
(unsigned int) (MIN_SINUS_VALUE+0.654508*SINUS_AMPLITUDE),
(unsigned int) (MIN_SINUS_VALUE+0.793893*SINUS_AMPLITUDE),
(unsigned int) (MIN_SINUS_VALUE+0.904508*SINUS_AMPLITUDE),
(unsigned int) (MIN_SINUS_VALUE+0.975528*SINUS_AMPLITUDE),
(unsigned int) (MIN_SINUS_VALUE+1.000000*SINUS_AMPLITUDE),
(unsigned int) (MIN_SINUS_VALUE+0.975528*SINUS_AMPLITUDE),
(unsigned int) (MIN_SINUS_VALUE+0.904508*SINUS_AMPLITUDE),
(unsigned int) (MIN_SINUS_VALUE+0.793893*SINUS_AMPLITUDE),
(unsigned int) (MIN_SINUS_VALUE+0.654508*SINUS_AMPLITUDE),
(unsigned int) (MIN_SINUS_VALUE+0.500000*SINUS_AMPLITUDE),
(unsigned int) (MIN_SINUS_VALUE+0.345492*SINUS_AMPLITUDE),
(unsigned int) (MIN_SINUS_VALUE+0.206107*SINUS_AMPLITUDE),
(unsigned int) (MIN_SINUS_VALUE+0.095492*SINUS_AMPLITUDE),
(unsigned int) (MIN_SINUS_VALUE+0.024472*SINUS_AMPLITUDE),
(unsigned int) (MIN_SINUS_VALUE+0.000000*SINUS_AMPLITUDE),
(unsigned int) (MIN_SINUS_VALUE+0.024472*SINUS_AMPLITUDE),
(unsigned int) (MIN_SINUS_VALUE+0.095492*SINUS_AMPLITUDE),
(unsigned int) (MIN_SINUS_VALUE+0.206107*SINUS_AMPLITUDE),
(unsigned int) (MIN_SINUS_VALUE+0.345492*SINUS_AMPLITUDE)
};
#define SIN_TAB_SIZE (sizeof(SinTab)/sizeof(SinTab[0]))
static unsigned int Index = 0;
Index++;
if (Index >= SIN_TAB_SIZE) Index = 0;
return SinTab[Index];
}

```

Assessing the performance of time-critical code

Time-critical sections of code are those that are executed most often. Because interrupts occur frequently, they are a good example of time-critical code.

Using the profiling feature, you can identify in your code the areas that are executed most frequently. Gains in these areas will lead to significant performance improvements.


10 EMU3 emulator features

This section explains how to use the features that are specific to the ST7-EMU3 emulator, including:

- [Section 10.1: Trace recording](#)
- [Section 10.2: Using advanced breakpoints](#)
- [Section 10.3: Programming trace recording](#)
- [Section 10.4: Using output triggers](#)
- [Section 10.5: Using analyzer input signals](#)
- [Section 10.6: Performance analysis](#)
- [Section 10.7: Read/write on the fly](#)
- [Section 10.8: Performing automatic firmware updates](#)

10.1 Trace recording

The EMU3 emulator can hold 256000 hardware cycle records in a physical memory module called the trace buffer. STVD's **Trace** window allows you to view recorded hardware cycles that have occurred during the execution of your application.

You can open the **Trace** window either by clicking on  (the **Trace** window icon) in the View toolbar, or from the main menu by selecting **View>Trace**.

Trace recording is activated from the [Trace contextual menu](#). When activated, the **Run**, **Continue** and **Step** commands prompt the trace buffer to save trace information until a breakpoint is reached.

Only information obtained up until the occurrence of the last breakpoint is visible in the **Trace** window.

You can use advanced breakpoints (see [Section 10.2: Using advanced breakpoints on page 291](#)) to define conditions whereby you can filter out those cycles you want recorded in the trace buffer. A number of examples are provided in [Section 10.3: Programming trace recording on page 309](#), showing you how to control the trace recording using advanced breakpoints.

The following sections provide information about:

- [Trace buffer fields](#)
- The commands in the [Trace contextual menu](#)
- [Emulator commands](#)

10.1.1 Trace buffer fields

The **Trace** window presents a table of fourteen fields (plus the Symbol Bar) which together form a single trace record.

At the left is the **Symbol Bar** (which has no column heading) followed in default order by **Record / Address / Memory Location / Data / Event / Hexadecimal / Disassembly / Symbolic Disassembly / Timestamp / (TEv) Timestamp Event / Trace Discontinuity / TIN (Trigger Input) / AI (Analyzer Input) / BEM (Advanced Breakpoints)**. Each is described individually below.

Figure 217. EMU3 trace window

Record	Address	M...	Data	Event	Hexad...	Disassembly	Symbolic ...	Timestamp	T...	Tra...	T...	AI
262134	0x00f060	DEB	0xfe	Operand fetch				25326053600			0	
262135	0x00f060	DEB	0xfe	Discarded				25326054350			0	
→	sample1.c		106			while(1);						
262136	0x00f05f	DEB	0x20	Operation ...	0x0020fe	JRT 0xf05f	JRT 0xf05f	25326055100			0	
262137	0x00f060	DEB	0xfe	Operand fetch				25326055850			0	
262138	0x00f060	DEB	0xfe	Discarded				25326056600			0	
→	sample1.c		106			while(1);						
262139	0x00f05f	DEB	0x20	Operation ...	0x0020fe	JRT 0xf05f	JRT 0xf05f	25326057350			0	
262140	0x00f060	DEB	0xfe	Operand fetch				25326058100			0	
262141	0x00f060	DEB	0xfe	Discarded				25326058850			0	
262142	0x00f05f	DEB	0x20	Discarded				25326059600			0	

Figure 217 shows most of the trace columns. You can preselect which of these columns are displayed (see [Column display](#)).

- **Symbol Bar:** This column contains icons such as source line markers (→) and/or bookmarks.

Note: When a source line marker occurs, the fields described hereafter are used to display source line information such as the name of the source file, the line of source code, and the instruction call.

- **Record:** Trace record numbering starts at 1, which corresponds to the earliest cycle to be recorded, and ends at the latest cycle recorded (to a maximum of 256000 cycles). This means that if you have an application that runs for 264000 cycles before there is a break, and you record the trace continuously, cycles 8000 to 264000 stay recorded in the trace buffer, but they are numbered 1 to 256000 in the **Record** column. Record numbers are not recalculated when you remove events from the display by filtering (see [Filter lines](#)).
- **Address:** The memory location accessed.
- **Data:** The hexadecimal value on the data bus.
- **Memory Location:** The physical memory in which this address is located. There are four kinds of memory:
 - **EMU** = Dedicated Emulation Board memory (emulator)
 - **TARGET** = Target Emulation Board memory (emulator)
 - **In MCU** = Internal (microcontroller) memory
 - **USER** = User (on-application board) memory
- **Event:** The type of microprocessor event—for example, stack read, stack write, direct memory read, direct memory write. In addition, some hardware cycles that are automatically recorded to trace have a special identifier in this field that allows you to filter them out if you want—such as **Extra timestamp record** or **Discarded**. These hardware cycles exist but do not correspond to either a specific instruction or operand.

The event column serves chiefly to allow you to filter out the hardware cycles of interest using the **Filter Lines** option in the Trace contextual menu, described in [Filter lines](#).

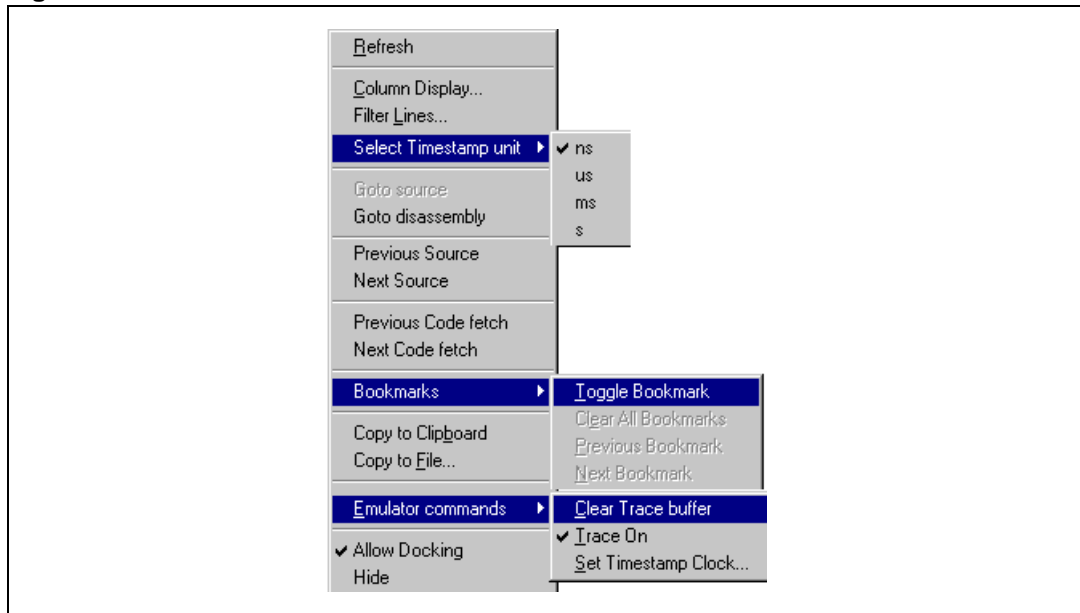
- **Hexadecimal:** The instruction in hexadecimal format, if this is a Fetch instruction cycle.
- **Disassembly:** The instruction in assembly language mnemonics, if this is a Fetch instruction cycle.
- **Symbolic Disassembly:** The instruction in assembly language mnemonics with symbolic operands, if this is a Fetch instruction cycle.
- **Time Stamp:** The value in nanoseconds of the time marker at this event. The time stamp is not recalculated when you remove events from the display by filtering (see [Filter lines](#)).
- **(TEv) Timestamp Event:** Signals the occurrence of events that affect the validity of the value in the Timestamp column—in particular:
 - **Restart:** An automatic restart of the 30-bit Timestamp counter. This means that the 30-bit Timestamp counter register has been filled and has started counting again, and the fact has been registered by the trace. **Note that a trace recording is forced each time a Timestamp counter restart occurs.** This allows you to be able to calculate the absolute amount of time passed between the beginning of the run and any given trace record. However, after a Restart message is shown, remember that the value in the **Time Stamp** column is only relative to the last Restart message, not absolute. To obtain the true value of time passed, refer to [Example 5—Measuring long time periods between events](#) for an example of how to measure the time passed between events, even when the Time Stamp counter has been restarted.
 - **Discontinuity:** Interruption of the timestamp counter's ability to write to the trace, due to the emulator's processor being in energy-saving mode, such as after a **halt** or a **wait for** instruction. While the timestamp counter continues, there is no way to force the trace to record any restart messages that occur when the counter restarts. Because of this, there is no way to determine the time elapsed between two events when a Discontinuity message occurs between them. However, the relative time elapsed between two events occurring before the discontinuity, or two events occurring after the discontinuity can still be calculated and is valid.
- **Trace discontinuity:** Indicates that the trace record in the buffer is not continuous. This is the case when the trace is turned on or off from the **BEM** or from the **stand-alone viewer**, to record only certain events. It is not the case when the Trace display has been filtered using the **Filter Lines** option, as the record in the buffer is unaffected.
- **TIN (Trigger IN):** External Trigger input.
- **AI (Analyzer Input):** Input from the Analyzer input connector on the EMU3 probe.
- **BEM (Advanced Breakpoints):** Shows the level and the logic states of the **Bus Event Machine (BEM)** as they become **TRUE**. These are reported in the form **BEM I[1-4] [e1 / e2 / if / e3 / e4 / else]**. For example: *BEM I2 e1 if* indicates that on *Level 2* of the sequencer, *Event 1* is **TRUE** and at the same time the condition *IF* is **TRUE** (only one event is necessary to fulfill the *IF* condition in this case).

10.1.2 Trace contextual menu

The **Trace** window contextual menu contains commands for trace operations plus several trace window configuration options.

Right-click anywhere within the **Trace** window to open the **Trace** contextual menu and access the following options and commands (see [Figure 218](#)).

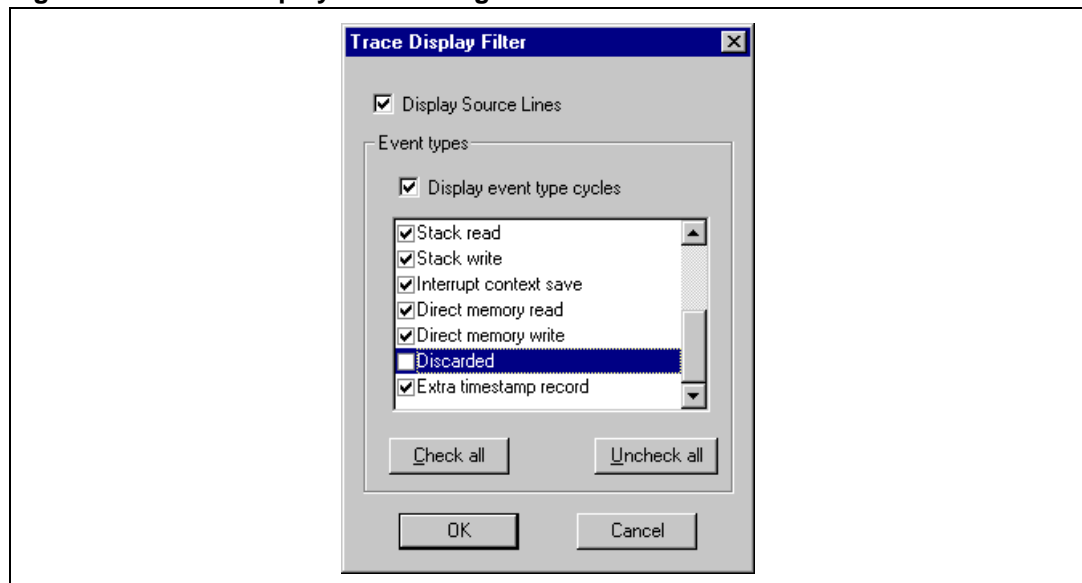
Figure 218. Trace contextual menu



Filter lines

Trace records can be filtered using the **Trace Display Filter** dialog box ([Figure 219](#)). To open this window, select **Filter Lines** from the Trace window contextual menu. The line filter is used to restrict the trace display to the operation that you are interested in. Only the selected trace entries appear in the **Trace** window (all others are hidden). The record number and timestamp are not recalculated when lines are removed from the display.

Figure 219. Trace Display Filter dialog box



- **Display Source Lines:** This option includes the source line in the Trace information. The **Address** column contains the name of the source file and the source line content is displayed in the **Disassembly** window.
- **Display event type cycles:** Includes the recorded cycles in the trace information. When enabled, the microprocessor actions selected in the **Trace Display Filter** dialog box are included. When disabled, only the source lines are shown in the **Trace** window.

In the above example, the **Discarded** event type is unchecked, so that no trace record with a “Discarded” event type is displayed. Discarded events are additional cycles that result from the normal functioning of the microcontroller, or from the emulator’s execution of certain debugging actions (break of execution, stepping). By filtering these out, you can see the true trace of memory operations. Keep in mind, however, that the timestamps (execution times) are not recalculated when events are filtered out of the display.

Selecting the Timestamp unit

By clicking on the **Select Timestamp unit** option, you can choose to have the Timestamp value shown in nanoseconds (ns), microseconds (µs), milliseconds (ms) or seconds (s).

Goto Source, Goto Disassembly

You may use these commands in the contextual menu to jump to either an editor window or the **Disassembly** window under the following conditions:

- If you highlight an entry in the trace where the address is a line of source code, you can use the **Goto Source** command to jump to that line of code in the Editor window.
- If you highlight a trace record where the event is an opcode fetch, you can use the **Goto Disassembly** command to jump to the appropriate address in the **Disassembly** window.

Previous Source, Next Source commands

If you highlight an entry in the trace where the address is a line of source code, you can use the **Previous Source** and **Next Source** commands in the contextual menu to jump to, respectively, the previous or next source code entries in the trace recording.

Previous Code Fetch, Next Code Fetch commands

If you highlight a trace record of an opcode fetch, you can use the **Previous Code Fetch** and **Next Code Fetch** commands in the contextual menu to jump to, respectively, the previous or next opcode fetch records in the trace recording.

Bookmarks

Clicking on the **Bookmarks** commands lets you access the following commands:

- **Toggle Bookmark:** Allows you to place or remove a bookmark at any trace recording entry.
- **Clear All Bookmarks:** Clears all bookmarks in the **Trace** window.
- **Previous Bookmark:** Allows you to jump to the previous bookmark in the **Trace** window.
- **Next Bookmark:** Allows you to jump to the next bookmark in the **Trace** window.

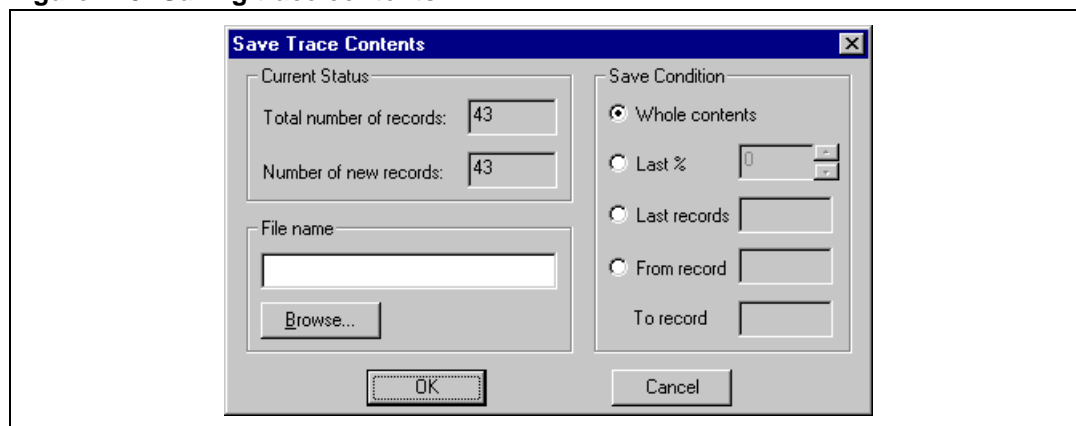
Copying and saving the trace contents

The trace record may be copied to the clipboard or copied to file via options in the Trace contextual menu.

The **Copy to Clipboard** command copies the highlighted trace record(s) to the clipboard.

The command **Copy to File** opens the **Save Trace Contents** dialog box, with details of saved trace files and user options for save parameters.

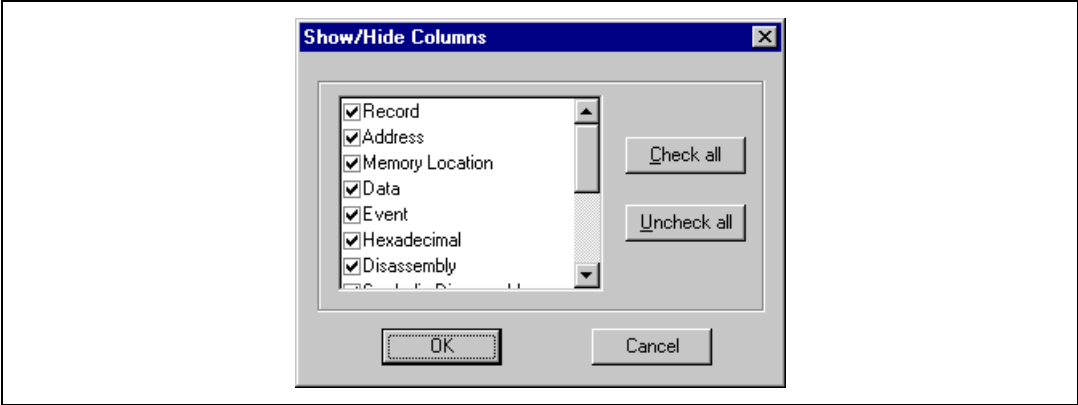
Figure 220. Saving trace contents



Column display

Columns may be disabled if they not required and/or do not display any trace information for the particular trace situation. To activate/disable a column, select **Column Display** in the Trace contextual menu. This opens a list of all the columns available in the trace record.

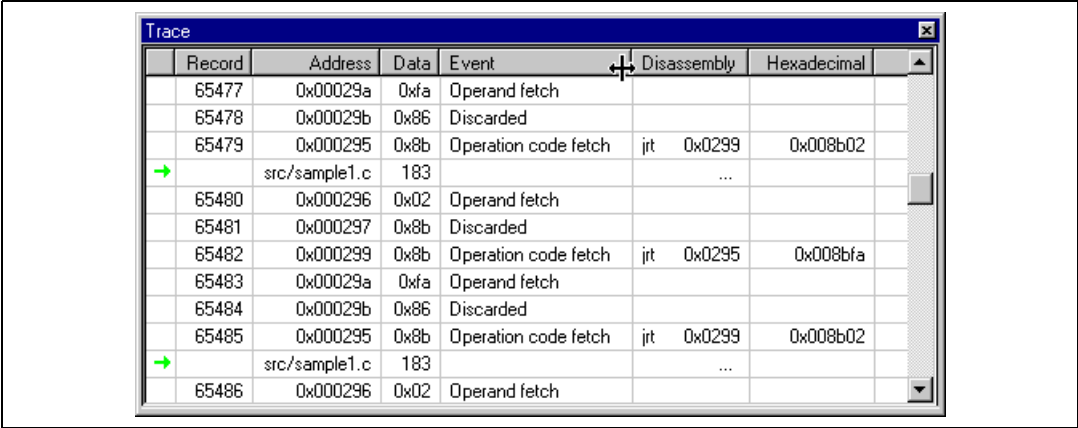
Figure 221. Show/hide columns



Columns may also be shifted right or left for convenience of use. Pick up the column header with the left mouse button and drag to the location required.

The **Trace** window in [Figure 222](#) has been set up to show only the **Record**, **Address**, **Data**, **Event**, **Disassembly** and **Hexadecimal** columns.

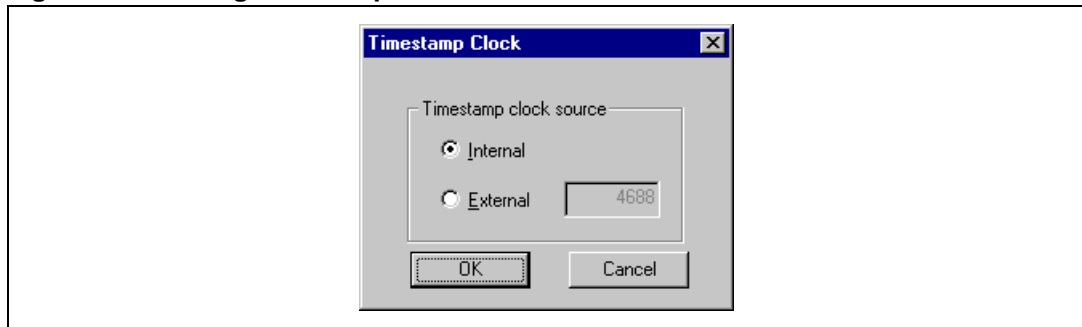
Figure 222. Customized trace window



10.1.3 Emulator commands

This option in the Trace contextual menu gives access to the following commands:

- **Clear Trace buffer:** This option erases all information currently contained in the Trace buffer.
- **Trace On:** When the trace function is on, records are numbered sequentially and stored in memory. The **Trace** window is automatically updated when the program is stopped. This option turns the Trace function on—re-clicking the option toggles the trace off. Note that you may also activate the Trace using the **Advanced Breakpoints** window (via the BEM)—but doing so disables the Trace On/Off command in the Trace Contextual menu.
- **Set Timestamp Clock...:** This option opens the **Timestamp Clock** dialog box shown in [Figure 223](#).

Figure 223. Setting timestamp clock

The timestamp clock frequency determines the granularity of the timer—the inverse of the clock frequency ($1/f$) equals the period of each timer tick. In this dialog box, you can choose the timestamp clock you want to use.

- **Internal:** Selects the emulator's internal 20 MHz timestamp clock (therefore the period of each timer tick equals 50 ns).
- **External:** Selects a timestamp clock external to the MCU. An external clock frequency must be input via the MCU's OSCIN pin, and provided by either the probe or a TTL clock on the application. (Note that the XTAL clock source on the probe may not be used as the external timestamp clock.)

If the external option is selected, the clock frequency in MHz must be given.

Note: You must ensure the accuracy of this frequency, as it is not verified by the debugger.

10.2 Using advanced breakpoints

The advanced breakpoint functionality provides great flexibility of use. You can set simple or multi-level breakpoint conditions, control the recording of the trace and send signals from the emulator output triggers. A four-level logical sequencer allows you to perform specific actions upon the occurrence of a specific event or series of events.

Events can be defined using any of the following parameters:

- a specific address or range of addresses,
- a specific data value with bit mask,
- a read, write or read/write access,
- an opcode fetch,
- external event(s) monitored using one or all of the nine input triggers,
- trace full information,
- a DMA memory access,
- an interruption,
- a stack operation access,
- any combination of the above.

Actions that may be performed upon the occurrence of the defined event or sequence of events can be defined as:

- a break in the execution of the program (a breakpoint),
- the outputting of a waveform to one or both of the two output triggers,
- the enabling, or disabling of trace recording,
- the recording of a snapshot in the trace,
- continuing to another level of conditions, defined by another event or series of events,
- a set of the above actions.

When the conditions you have programmed are met, the sequencer carries out the preset actions. Up to four levels of conditional terms can be linked to construct logical functions that define precisely the event to be tracked and the action to be carried out, within even the most convoluted of program sequences. It is this adaptability which makes the sequencer such a valuable real-time tool (see [Section 10.2.1: Defining advanced breakpoints](#)).

The heart of the Advanced Breakpoints sequencer is the **bus event machine (BEM)**. This hardware system is user-transparent and equipped with its own memory and bus connections. The BEM functions in parallel with the emulation hardware system controlling the program under test. In this way, the EMU3 emulator is able to carry out **real-time** debugging actions.

Because the Advanced Breakpoints features uses separate BEM hardware to “spy” on the running of the program on the emulator, the inclusion of advanced breakpoints does not interfere with the real-time running of the program. However, there is a short delay between the time the “spy” identifies a breakpoint condition has occurred and the moment the program execution actually stops. Typically, a few additional instruction lines are executed during this delay.

This behavior contrasts with that of Instruction Breakpoints where, because breakpoint conditions (set using the Counter and Condition options) are evaluated internally as if the evaluation were part of the program execution, the use of these breakpoints means that the program cannot run in real-time. However, Instruction breakpoints cause the program to stop exactly at the instruction line that satisfies the break condition.

This section provides information to help you use your EMU3 emulators's advanced breakpoint features, including:

- [Defining advanced breakpoints](#)
- [Memory access events](#)
- [Other types of events](#)
- [Enabling advanced breakpoints](#)
- [Starting with trace ON](#)
- [The configuration summary](#)
- [Synoptic representation of advanced breakpoints](#)
- [Saving and loading advanced breakpoints](#)
- [Advanced breakpoint examples](#)

10.2.1 Defining advanced breakpoints

Advanced breakpoints use logic levels to structure a sequence of conditions that will trigger a response from STVD, such as stopping your application or starting a trace recording.

Each level can incorporate up to four events. The level defines the relationship between these events (specific user-defined conditions) and the actions to be carried out by the debugger. A counter may also be set.


Up to four levels can be combined to structure advanced breakpoint definitions.

Each of the four levels has the structure (IF..THEN) (ELSE IF..THEN). Accordingly, there are four vertically-aligned buttons in the **Programming Level** area that, when clicked with the mouse, give access to each statement in the logical sequence.

Combination operators: AND, OR, NAND, NOR, XAND and XOR can also be used.

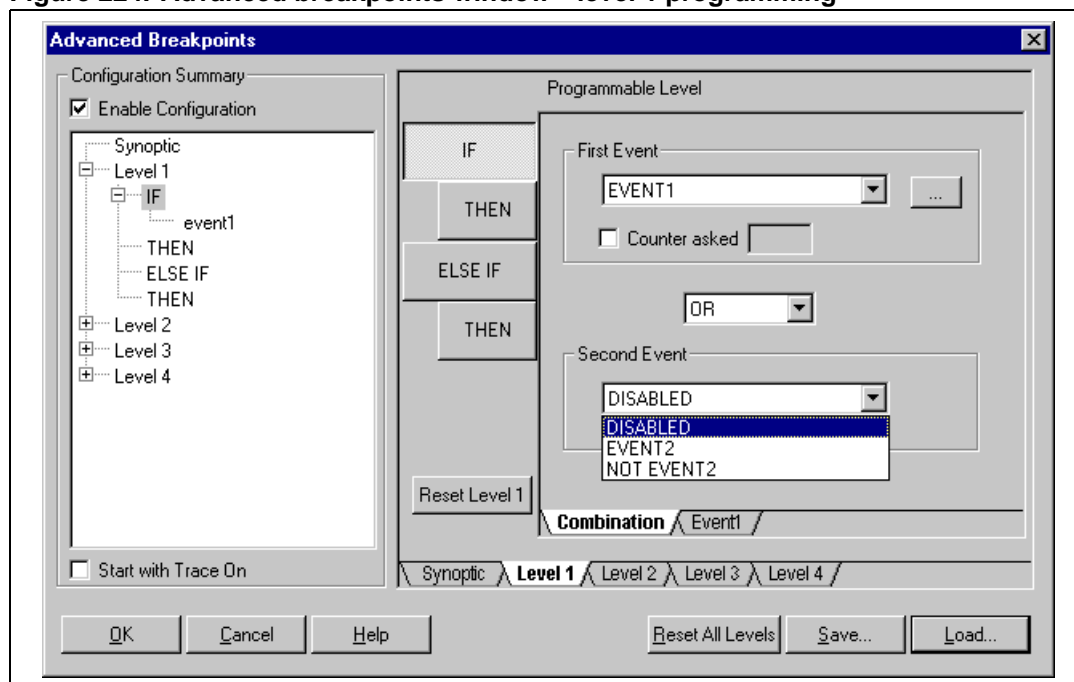
Therefore, the definition of each level has the following structure:

```
IF [NOT] <[counter] event1> [<combination> [NOT] <event2>]
  THEN actions
ELSE IF [NOT] <[counter] event3> [<combination> [NOT] <event4>]
  THEN actions
```

Open the **Advanced Breakpoints** window by selecting **Debug Instrument > Advanced Breakpoints** from the main menu, or by clicking on  (the Advanced Breakpoints icon) in the Emulator toolbar.

In the **Advanced Breakpoints** window, click on the **Level 1** tab located towards the bottom of the window. (You can also access Levels, Events and Combination windows using the **Configuration Summary**).

Figure 224. Advanced breakpoints window—level 1 programming



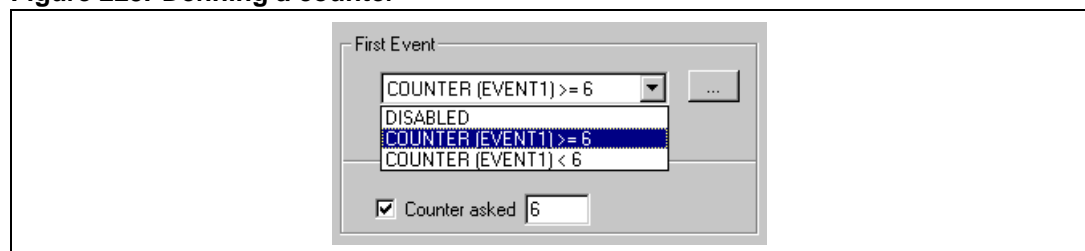
The sequencer level being programmed is indicated at the top of the **Programming Level** area, and a reset button for the level is located at the bottom. This is useful if you want to completely erase the programming level.

Four buttons, **IF**, **THEN**, **ELSE IF** and the final **THEN** allow access to these four portions of the level definition.

When you click on the **IF** button in the **Programmable Level** portion of the window, as shown in [Figure 224](#), you are presented with the **Combination** window. It contains fields that allow you to define the conditions upon which you want certain action(s) to be taken. Proceed as follows:

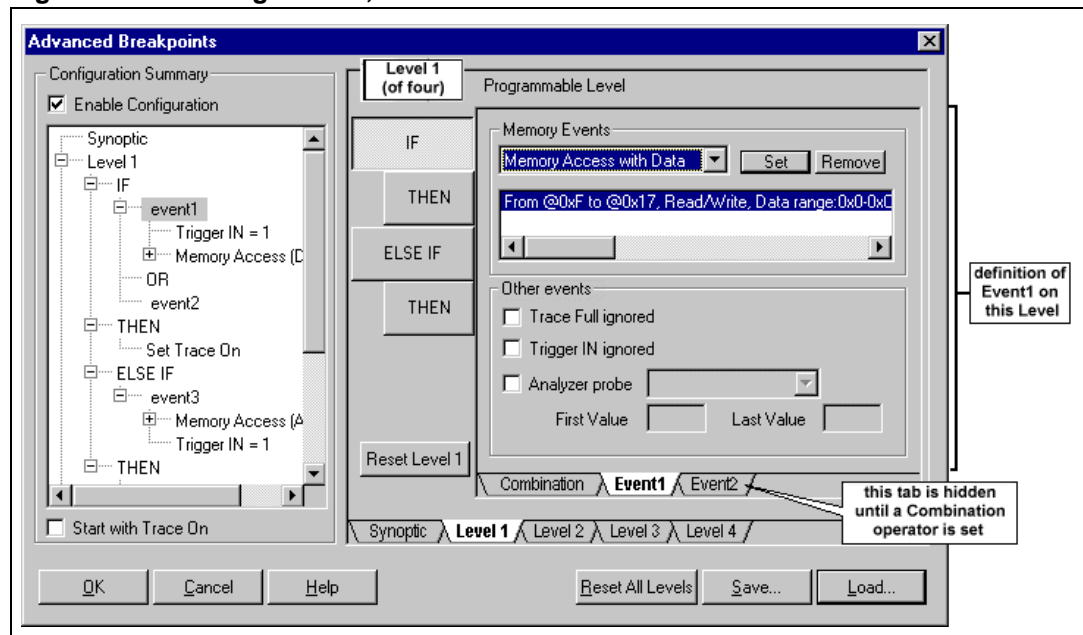
1. First ask yourself: "What type of condition results in an action?". Is it a single occurrence of an event, a number of occurrences of the same event, the non-occurrence of an event or a combination of dissimilar events that trigger the action?
2. In the field, **First Event**, choose either **EVENT 1** or **NOT EVENT 1**. If you choose **EVENT 1**, the condition is satisfied when this event is true. When you choose **NOT EVENT 1**, the condition is satisfied when this event is false. (If you leave the **First Event** as **DISABLED**, no event is defined and the Level remains void. Choosing anything other than **DISABLED** causes an **Event1** tab to appear in the window.)
3. If you want the same event to occur more than once before an action is taken, choose **EVENT 1**, and check the **Counter asked** option, then set the number of times you want the event to occur before an action is taken. Once these fields are completed, you can choose between the following options (as shown in [Figure 225](#)):
 - If **counter [event] >= n** is selected, then an action is taken when the event has occurred **n** or more times (six or more times in the example).
 - If **counter [event] < n** is selected, then an action is taken (the condition is TRUE) while the event has occurred less than **n** times and then the action ceases (the condition becomes FALSE). Use of the counter in this way is useful when you want to control trace recording, or the sending of signals to an output trigger, but has no meaning when applied to breakpoints.

Figure 225. Defining a counter



4. If you want to combine the occurrence (or non-occurrence) of event1 with another event (event2) using a logical operator (OR, XOR, NOR, AND, NAND) choose the operator from the field in the middle of the window. Choosing anything other than **NONE** causes an **Event2** tab to appear in the window.
5. If you chose a logical operator in the previous step, you can choose between **EVENT 2** and **NOT EVENT 2** in the **Second Event** field. Note that no counter is available in this field. Only two counters per level are available, associated with event1 and event3 of each level.
6. The next step is to define event1 and event2 (if present). Click on the **Event1** tab.
7. The **Event** tabs allow you to define each event, as shown in [Figure 226](#).

Figure 226. Defining Level 1, Event 1



An event may incorporate any of the following:

- A **Memory Event**. You may set an event on any of the following memory events: **Any Memory Access, Memory Access with Data, Opcode Fetch, Opcode Fetch with Data, Stack Memory Access, DMA Memory access, IT Memory Access**. A Memory Access event would be, for example, a read access of memory address 0xfd00.
- One or more **Other events**. Events can be set on the detection of specific values of signals or other parameter. You can choose one or any combination of the following: trace buffer (full or not full), input trigger signal value (0 or 1), and Analyzer probe (enter a specific value or range of values).
- A combination of a memory event and signal events.

Note:

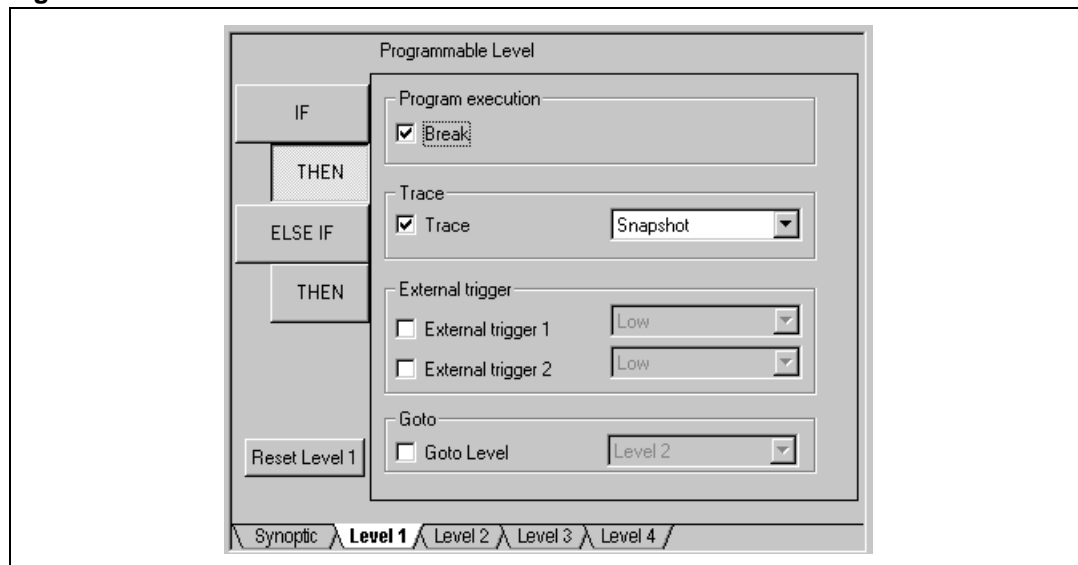
All values can be entered either in unsigned integer format, or in hexadecimal format. All entries in unsigned integer format will be automatically converted into hexadecimal format.

8. Once you have defined event1 and event2 (if present), the next step is to click **THEN** to define the actions to take if the defined conditions occur during the running of your program.

The **THEN** window allows you to define an **action** as any of the following:

- Program Execution **Break**: Stops the execution of the program.
- **Trace**: Start or stop trace recording, or take a snapshot of the trace (a trace snapshot is a trace recording of one full program loop). You may opt to start running your program with the trace ON using the **Start with Trace ON** option. Examples of how to control the trace recording using Advanced Breakpoints are given in [Section 10.3 on page 309](#).
- **External Trigger 1** and **External Trigger 2**: Send a signal to one or both of the two external triggers (OUT1 and OUT2). Examples of how to control the trigger output using Advanced Breakpoints are given in [Trigger programming examples](#).
- **Goto Level**: Jump to another **Level** of event/action definitions.

Figure 227. Then window



9. Continue defining the level using the remaining **Else If** and **Then** commands in the same manner as described above. Remember that:
 - **Else If** conditions are only considered when the **If** conditions are not true at the end of one memory cycle.
 - If any of your **Then** actions include a **Goto Level** command, you must also define this level.
10. Once you have defined all levels required, **remember to enable the advanced breakpoints** by checking the **Enable Configuration** box in the top left corner of the window.
11. Click **OK** to validate your Advanced Breakpoints settings, and exit the dialog box.
 The current Advanced Breakpoints settings are saved as part of the workspace settings. If you have made any errors in programming, a warning message is shown. For example, a warning message would arise if you have asked for two incompatible actions simultaneously, such as a **Break** and a **Goto Level**. The warning allows you to go back and edit your sequencer programming.

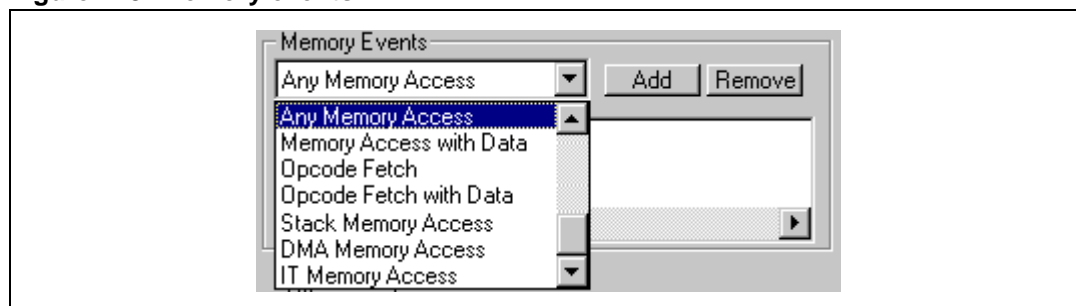
Note: By default, your Advanced Breakpoints configuration is enabled every time you create, load or modify a configuration. This means that the next time you run your application, the Advanced Breakpoints program you have just validated (by clicking OK) will be active.

You can enable and disable your Advanced Breakpoints configuration by checking or unchecking the **Enable Configuration** box at the top left-hand side of the Configuration Summary. Additionally, the status of the configuration (enabled, modified or disabled) is displayed in a message in the **Synoptic view**.

10.2.2 Memory access events

Memory access options are selected from the pull-down menu in the **Memory Access** dialog box as shown in [Figure 228](#).

Figure 228. Memory events

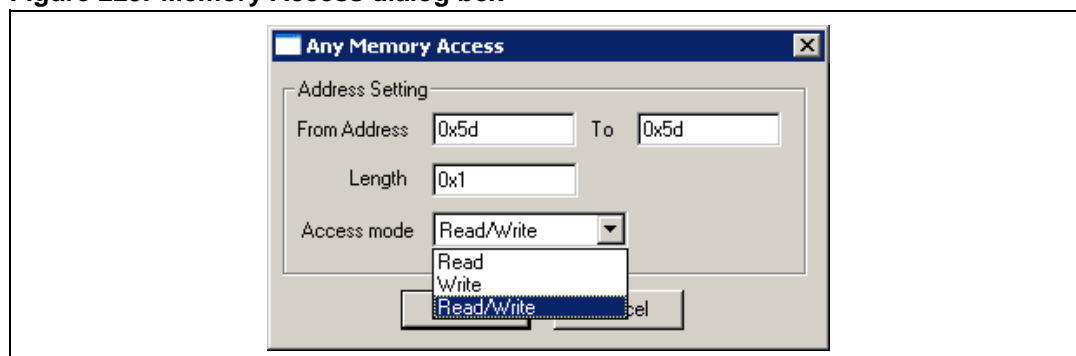


An event may be defined as one of the following types of memory access:

- **Any Memory Access:** A specified memory address or range of addresses is accessed by a read or a write command.

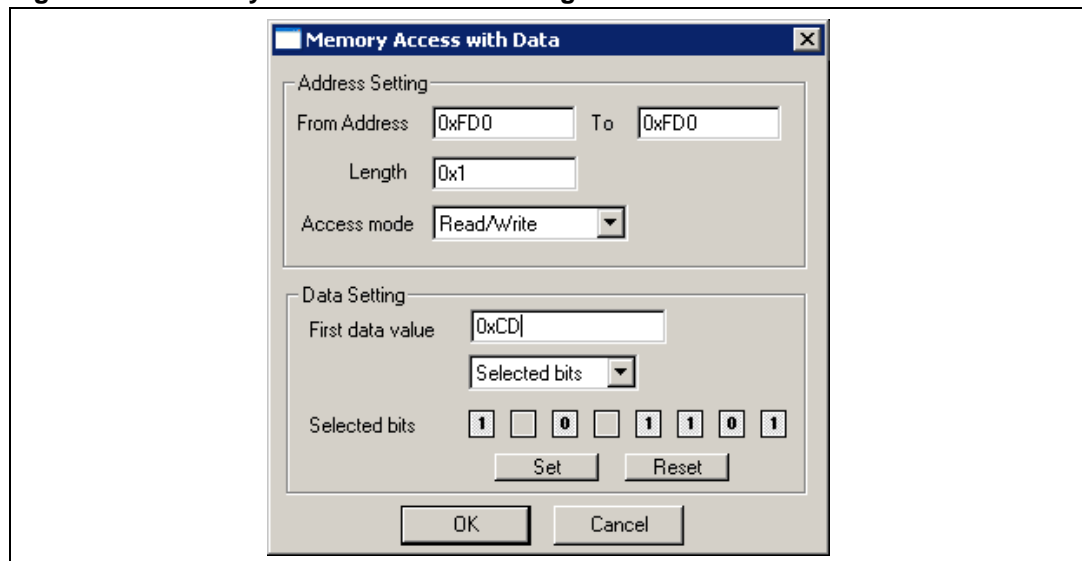
In the dialog box shown in [Figure 229](#), you must specify two of the following fields: a start address (for the start of a range of addresses), an end address (the end of a range of addresses) or the length of the address range. If you want to monitor a specific address, enter the address in the **From Address** field, and enter a **Length** of zero ("0x1" in hexadecimal format). You may also specify the access mode: a read access, a write access, or either a read or write access.

Figure 229. Memory Access dialog box



- **Memory Access with Data:** A specified data value or range (or specified bit mask) is accessed (read/written) at a specified memory address or specified address range.

Figure 230. Memory Access with Data dialog box



The dialog box at right is the same as for the **Memory Access** dialog box with the addition of a field to specify a data value or range.

You must enter a **First data value** and then, in the drop-down box, specify whether you want to give a **Data range**, or specify **Selected bits**.

If you select **Data range**, you will have to enter a **Last data value** (to complete the range).

If you select **Selected bits**, the 8-bit data register appears, and you can choose to ignore the value of certain bits in the particular data register you specified.

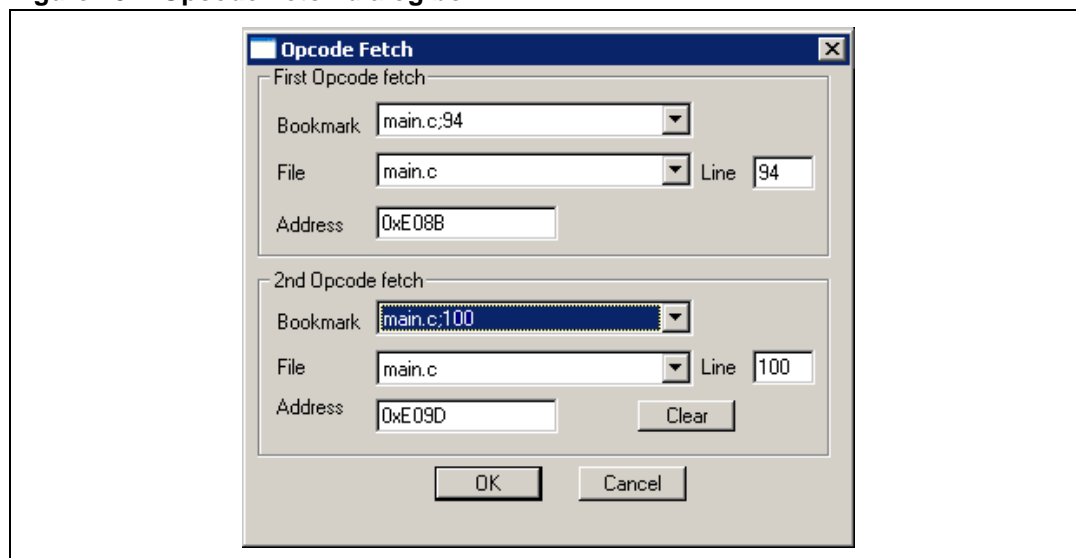
In the example shown in [Figure 230](#), the first data value given is CD (or 205 in decimal format). By masking the second and fourth bits as shown, we are actually specifying *any* of the following values: 8D (or 141), 9D (or 157), DD (or 221), and of course, the original value, CD (or 205).

- **Opcode Fetch:** An opcode fetch is carried out from a specified memory address or range of addresses. In the dialog box shown in [Figure 231](#), you must define a first address under **First Opcode fetch**.

The address can be entered directly, or if you have placed a bookmark at the address of interest, you can simply choose the bookmark (and the file in which it occurs) and the address field is automatically calculated.

If you only specify one address, the Opcode Fetch must occur at this address. If you want to specify a range of addresses which are monitored for opcode fetches, fill in the **2nd Opcode fetch** section of the dialog box. Note that you can also enter the name of the symbol in the **Address** field, and the address will be identified by STVD.

Figure 231. Opcode Fetch dialog box



- Opcode Fetch with Data:** An opcode fetch of a specified value or range (or specified bit mask) is carried out at a specified address, or within a specified range of addresses. The dialog box shown in [Figure 232](#) is the same as for the **Opcode Fetch** dialog box with the addition of a field to specify a data value or range. You must enter a **First data value** and then, in the drop-down box, specify whether you want to give a **Data range**, or specify **Selected bits**.
 If you select **Data range**, you will have to enter a **Last data value** (to complete the range). If you select **Selected bits**, the 8-bit data register appears, and you can choose to ignore the value of certain bits in the particular data register you have specified.

Figure 232. Opcode Fetch with Data dialog box

Note: When **Any Memory Access** or **Opcode Fetch** is selected as the type of access, then any number of access events may be included in the list for that event. If **Opcode Fetch with Data** or **Memory Access with Data** are selected as the type of Memory Access, then only a single entry is allowed in the list for that event. (This limitation applies only to the current event, and not between different events in the same or different Levels.)

10.2.3 Other types of events

Other events can be selected in the lower half of the event definition area in each **Event** window. Each event type has a tristate checkbox, which allows you to choose between particular states or signal values, and ignoring the signals.

Figure 233. Other events

- **Trace Full:** Values **True**, **False** or **Ignored** may be selected.
- **Trigger In:** Values **1**, **0**, or **Ignored** may be selected.
- **Analyzer Input:** Signals from the Analyzer Input connector (located on the EMU3 Probe) can be used as conditions (for more information, see [Section 10.5 on page 317](#)). A single value, a range of values, or selected bits may be defined. An example

that shows how to specify a value with selected bits masked, is given in [Section 10.5.1 on page 318](#).

Note: Ignored means that the signal is not monitored as part of the event.

10.2.4 Enabling advanced breakpoints

There is an **Enable Configuration** checkbox at the top left of the **Advanced Breakpoints** window, above the user configuration summary. This must be checked to activate the breakpoint program.

When breakpoint information is restored at the beginning of a new emulation session, this checkbox is disabled (unchecked), it must be set to active again for the current breakpoint program to run.

10.2.5 Starting with trace ON

The checkbox **Start with Trace ON**, is located at the bottom left of the **Advanced Breakpoints** window below the user program summary. This checkbox corresponds to the **Trace On** option which is available in the Trace contextual menu under **Emulator Commands** (refer to [Section 10.1.2 on page 286](#)).

If **Advanced Breakpoints** is enabled, then control of the trace is via the **Advanced Breakpoints** sequencer and the **Trace On/Off** command in the Trace contextual menu is disabled.

Figure 234. Emulator commands in trace contextual menu



When this option is active, the trace is turned on when the application program runs. When inactive, the trace is off until turned on explicitly by an **Advanced Breakpoints** programmed event (set in a **THEN** section of one of the Program Levels).

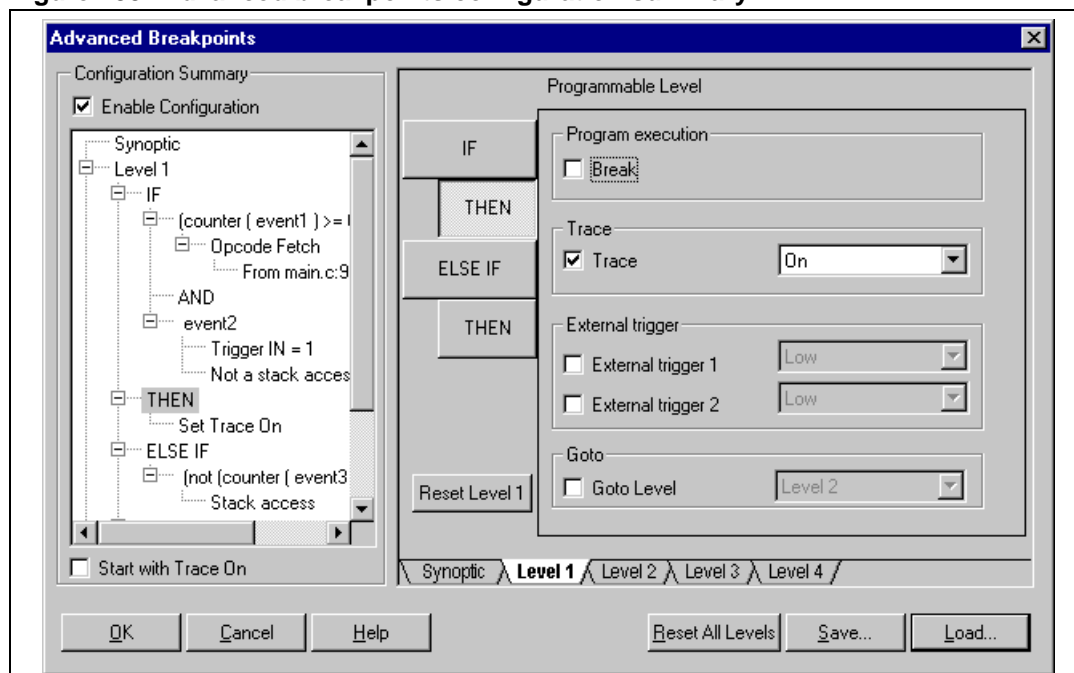
10.2.6 The configuration summary

On the left of the **Advanced Breakpoints** window, the **Configuration Summary** shows the logic structure, event definitions and actions to take for all the levels. As you complete the definition of each level, the information that you entered appears in the **Configuration Summary**.

You can use the **Configuration Summary** to jump to your definitions—any element may be selected by a single mouse click on the corresponding identifier. The text is highlighted, and information on this single level and the particular element within it, appears in the **Programming Level** area to the right of the Window.

For example, in [Figure 235](#) you can see a structured view of Level 1 in the **Configuration Summary** and Event 2 of Level 1 has been highlighted by clicking on it, thus opening the Level 1, Event 2 programming area on the right-hand side of the window.

Figure 235. Advanced breakpoints configuration summary

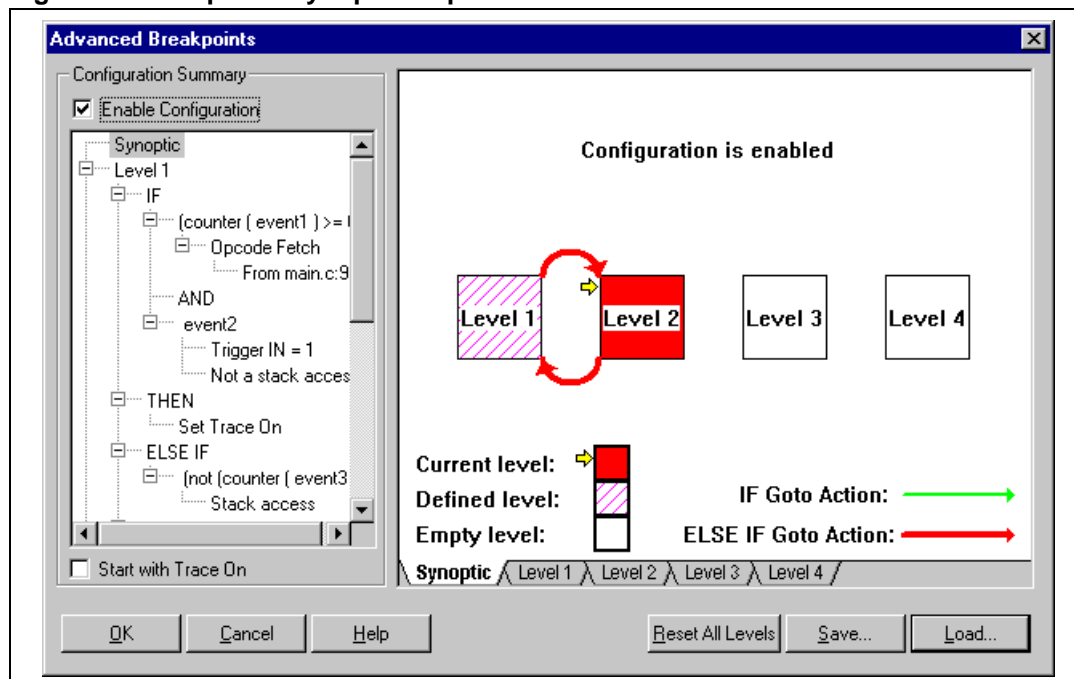


10.2.7 Synoptic representation of advanced breakpoints

The **Synoptic** window may be viewed by clicking on the **Synoptic** tab at the bottom of the window, or by clicking on “Synoptic” in the **Configuration Summary**. It illustrates the logical links programmed into the levels.

The Synoptic graphic for a typical sequence is shown in [Figure 236](#).

Figure 236. Graphical synoptic sequence

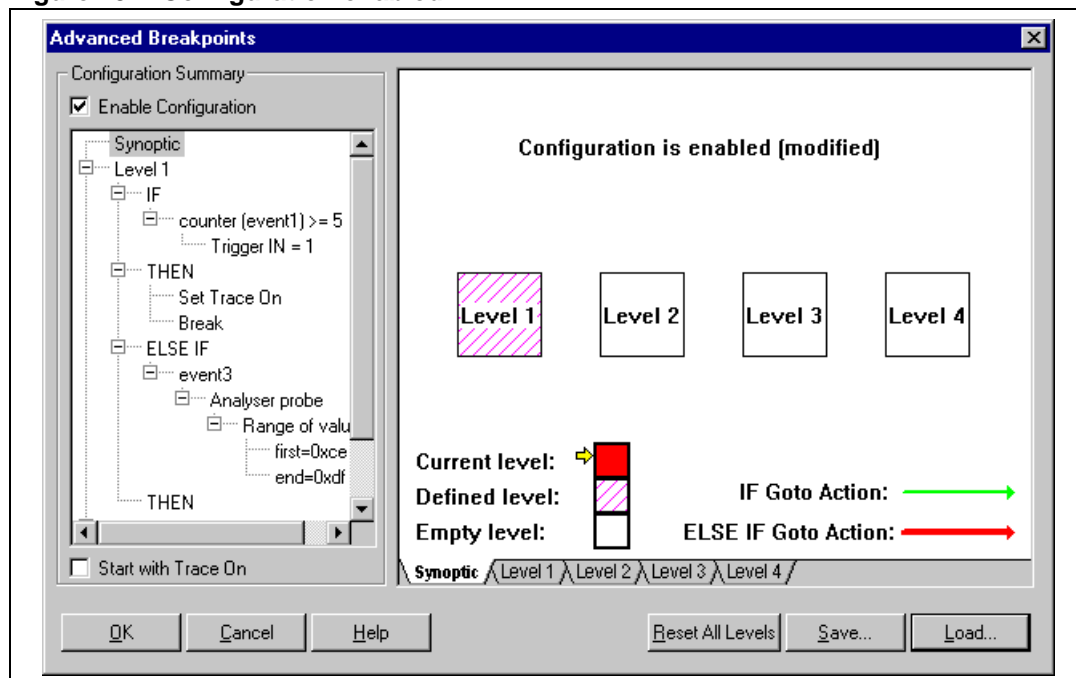


Programmed and unused levels are differentiated by color coding. The currently active Level is color-coded and indicated by an arrow. Links between levels are color-coded to differentiate between IF and ELSE IF logic.

The Synoptic view is a block diagram of the four levels with the programmed logical links displayed graphically. When the configuration is enabled (using the **Enable Configuration** checkbox at the top left of the **Advanced Breakpoints** window), programmed and unused levels are differentiated by color coding as shown in [Figure 236](#). The currently active level is color-coded and indicated by an arrow. Links between levels are color-coded to differentiate between IF and ELSE IF logic.

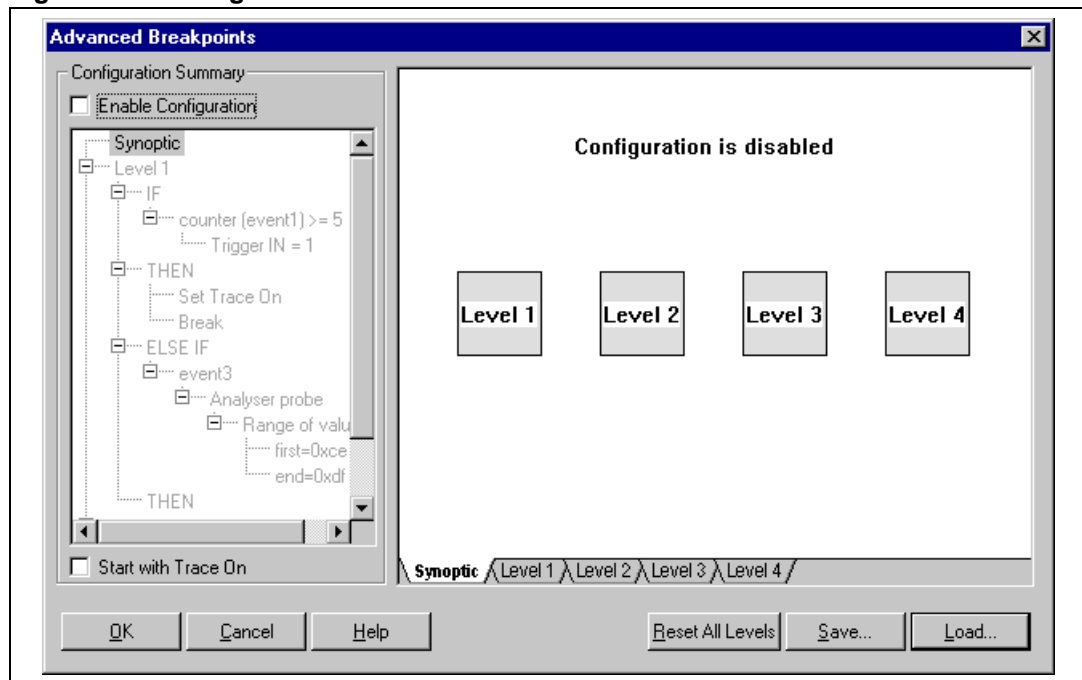
In addition to summarizing the logic of your Advanced Breakpoints configuration, the Synoptic view also tells you the current status of your configuration. If you have programmed your Advanced Breakpoints configuration, but have not yet validated it (by clicking on the **OK** button), the Synoptic view appears with the message “**Configuration is enabled (modified)**” as shown in [Figure 237](#).

Figure 237. Configuration enabled



If you disable your configuration, the Synoptic view shows the message “**Configuration is disabled**”, and the levels are shown in gray to indicate that while the configuration has been saved, it is currently disabled ([Figure 238](#)).

Figure 238. Configuration disabled



10.2.8 Saving and loading advanced breakpoints

You can store the definition for your advanced breakpoint to a file (*.bem), which can be reloaded by any STVD project, providing that you are using the EMU3 emulator to debug the application. This will allow you to save time if you have to switch between advanced breakpoints while debugging your application.

To save your advanced breakpoint, click on the **Save** button at the bottom of the Advanced Breakpoint window. In the next window, select the folder where you will save your advanced breakpoint, type a name for the .bem file and then click on **Save**.

To load an advanced breakpoint that you have already saved, click on the Load button in the Advanced Breakpoint window. In the next window, browse to locate the .bem file that you want, select it and then click on **Open**.

10.2.9 Advanced breakpoint examples

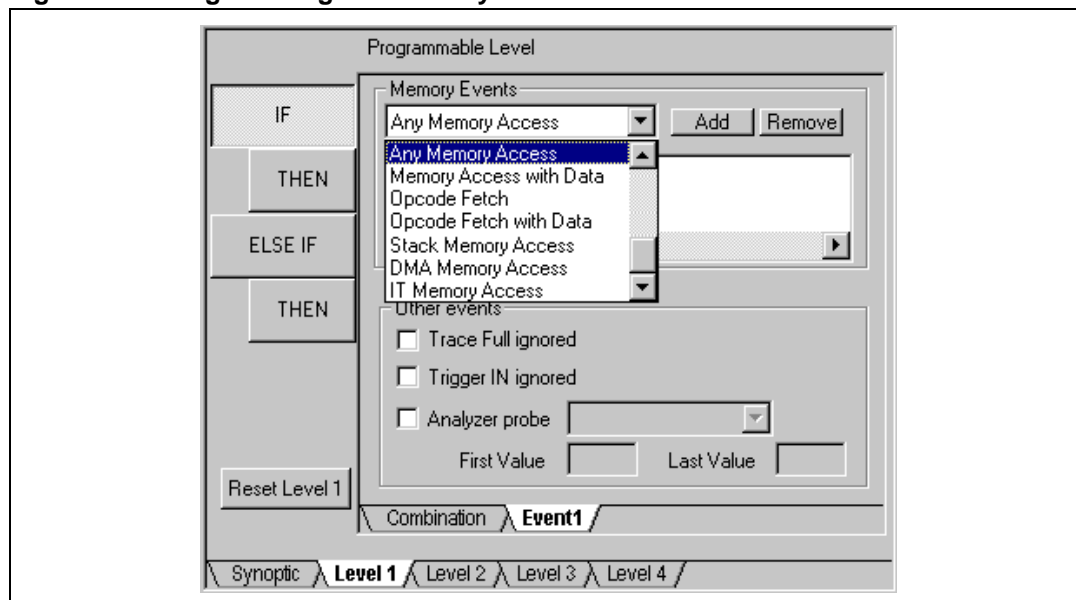
Example 1—Break on memory access without data masking

This example sets a breakpoint on memory access (either a read or a write) within a specific range of addresses. This sort of breakpoint is very useful if you want to ensure that certain memory zones are never read or written to.

Imagine that we want to make sure that there are no memory accesses in the memory address zone from 0x40 to 0x47. We'll program the **Advanced Breakpoints** window so that if a memory access in this address range occurs, the program is halted at the instruction that provoked the access. Proceed as follows:

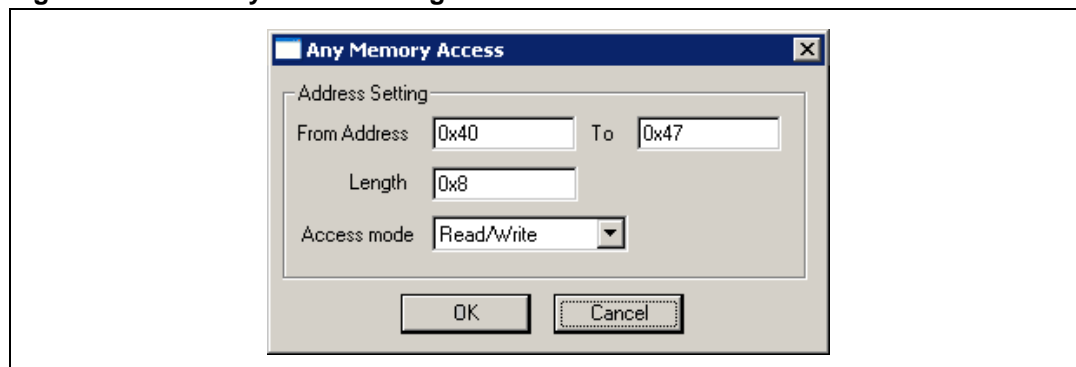
1. From the main menu, select **Debug Instrument > Advanced Breakpoints**.
2. In the **Advanced Breakpoints** window, click on the **Level1** tab, then on the **Event1** tab.
3. In the **Memory Events** area, select **Any Memory Access** from the drop down menu as shown in [Figure 239](#).

Figure 239. Programming the memory access



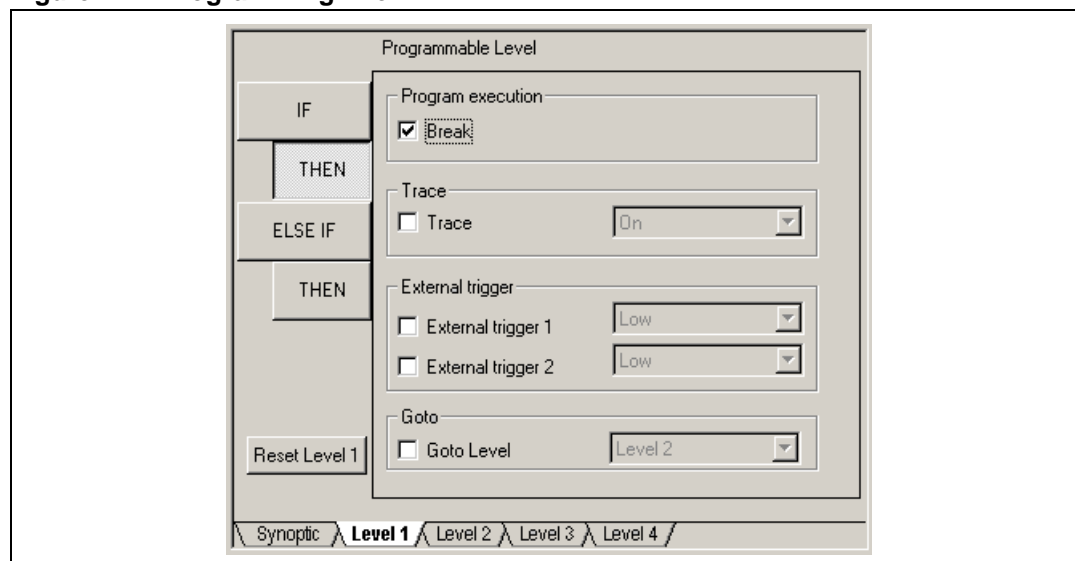
The **Memory Access** dialog box opens. To define the memory zone, enter the starting address, 0x40, in the **From Address** field, and the end address, 0x47, in the **To** field. The length of the zone is automatically calculated and entered in the **Length** field. In the **Access mode** drop-down list, select **Read/Write**, so that either type of access triggers a breakpoint.

Figure 240. Memory access dialog box



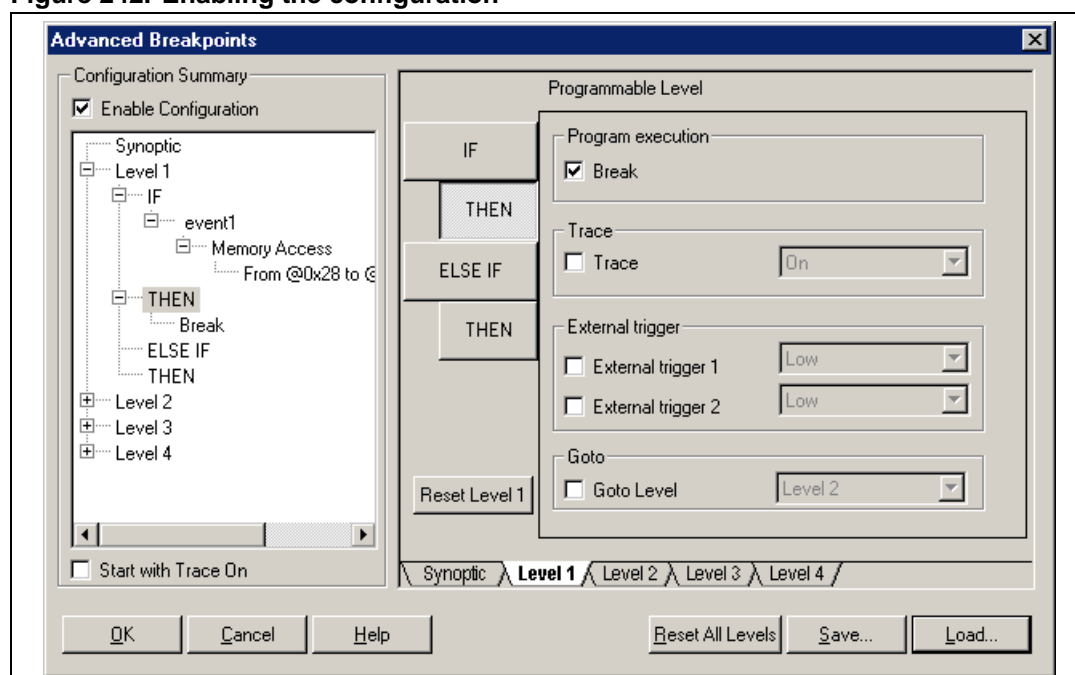
4. Click **OK**.
5. Click on the first **THEN** button to open the tab to define the action to take if the memory event defined is TRUE. Check the **Break** option as shown in [Figure 241](#).

Figure 241. Programming Then



6. Check the **Enable Configuration** box at the top left corner of the **Advanced Breakpoints** window (as shown in [Figure 242](#)) to enable the program. Click **OK**.

Figure 242. Enabling the configuration



Advanced breakpoint programming is now enabled. [Table 73](#) provides a summary of steps.

Table 73. Advanced breakpoint programming summary

Step	Description
START with TRACE	ON or OFF
LEVEL 1	IF (access at memory location [0x40-0x47]) THEN break;
LEVEL 2-4	(empty)

Example 2—Break on memory access with data masking

This example is an elaboration of the previous example. In this case, we set a breakpoint on the reading or writing of a particular data value to a particular memory address or range of memory addresses.

Imagine that you want to be alerted if a null value is written to the memory address 0x40.

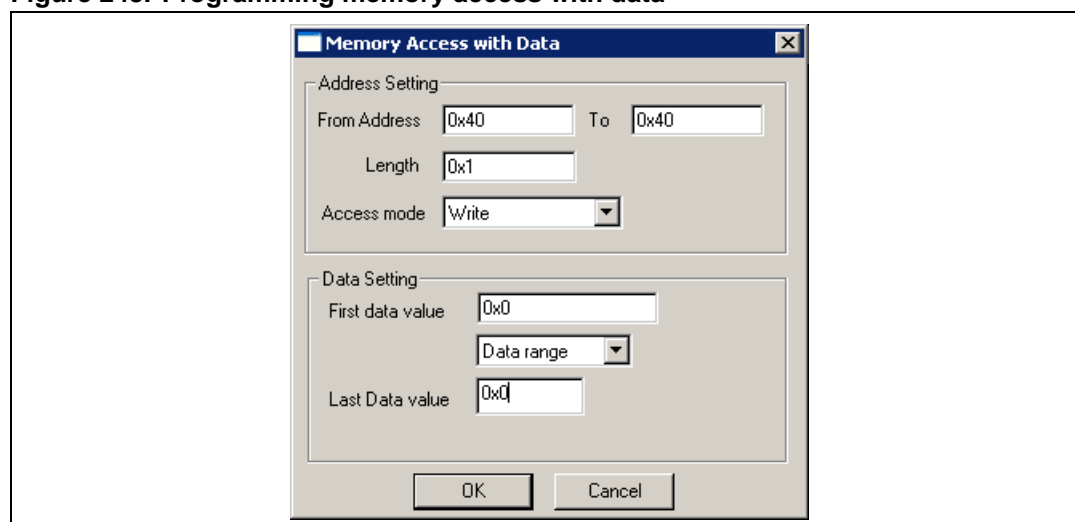
The programming of the **Advanced Breakpoints** window is summarized in [Table 74](#).

Table 74. Advanced breakpoints summary

Step	Description
START with TRACE	ON or OFF
LEVEL 1:	IF (access at memory location [0x40] with data=0x00) THEN break;
LEVEL 2-4:	(empty)

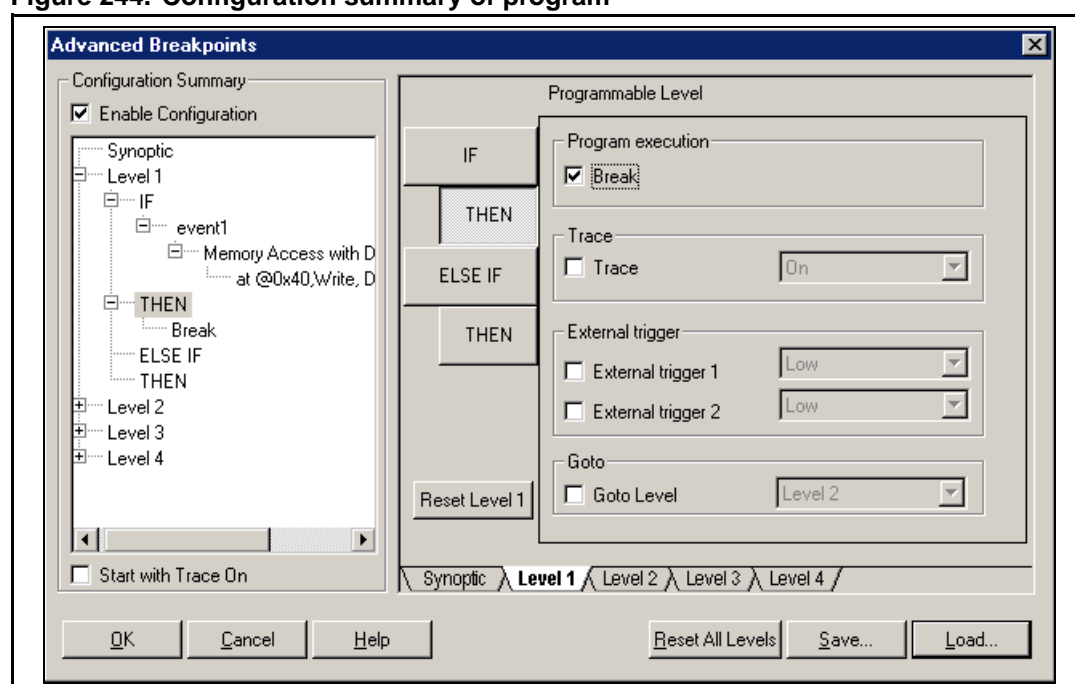
1. In the **Advanced Breakpoints** window, click on the **Level1** tab, then on the **Event1** tab.
2. In the **Memory Event** field, choose **Memory Access with Data**.
3. Set the memory event as shown in [Figure 243](#).

Figure 243. Programming memory access with data



4. In the **Level 1** tab, click **Then** and set the defined action as a breakpoint, as shown in [Figure 244](#). Note the program is summarized in the **Configuration Summary** window on the left.

Figure 244. Configuration summary of program



Example 3—Break on *Condition* after function call

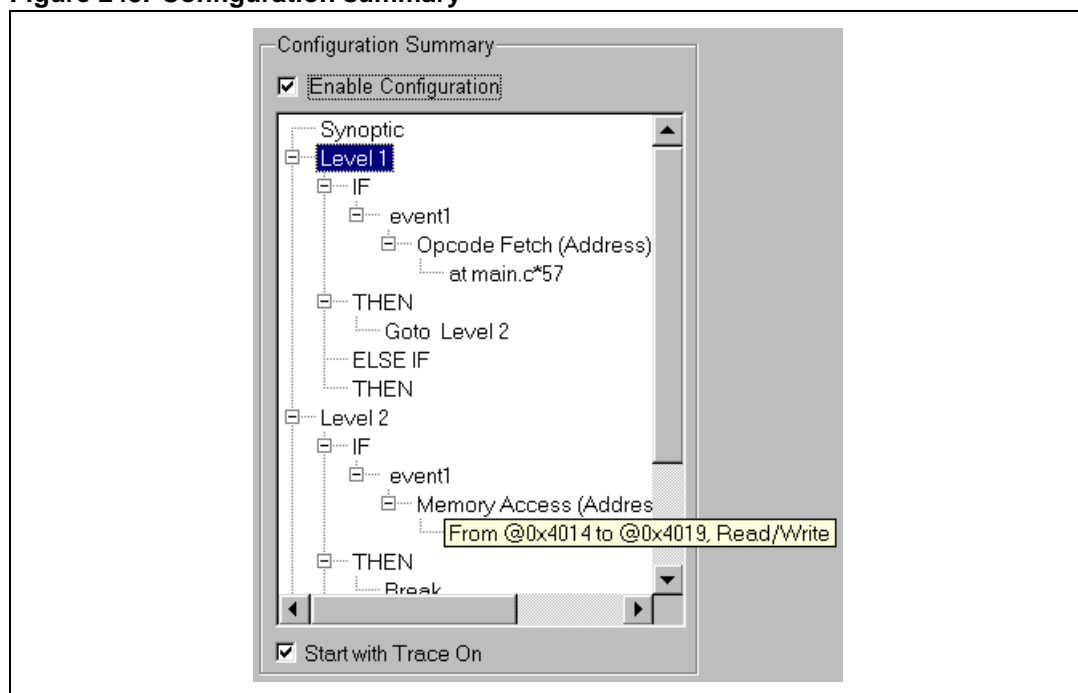
In the example below, a breakpoint must be set on a Read within a given block of memory but the break should not take place until after a specified function is called. Use is made of the starting address of the function. An opcode *fetch* at that address indicates the opening of the function.

Program the **Advanced Breakpoints** window as summarized in [Table 75](#).

Table 75. Advanced breakpoints summary

Step	Description
START with TRACE	ON or OFF
LEVEL 1:	IF (opcode fetch at main.c*57) THEN Goto Level 2;
LEVEL 2:	IF (read at memory address 0x004014-0x004019) THEN break;
LEVEL 3-4:	(empty)

The **Configuration Summary** of the **Advanced Breakpoints** window should appear as shown in [Figure 245](#).

Figure 245. Configuration summary

Note: There are more examples on how to use advanced breakpoints to control the output triggers and the trace recording in [Trigger programming examples](#) and [Programming trace recording](#) respectively.

10.3 Programming trace recording

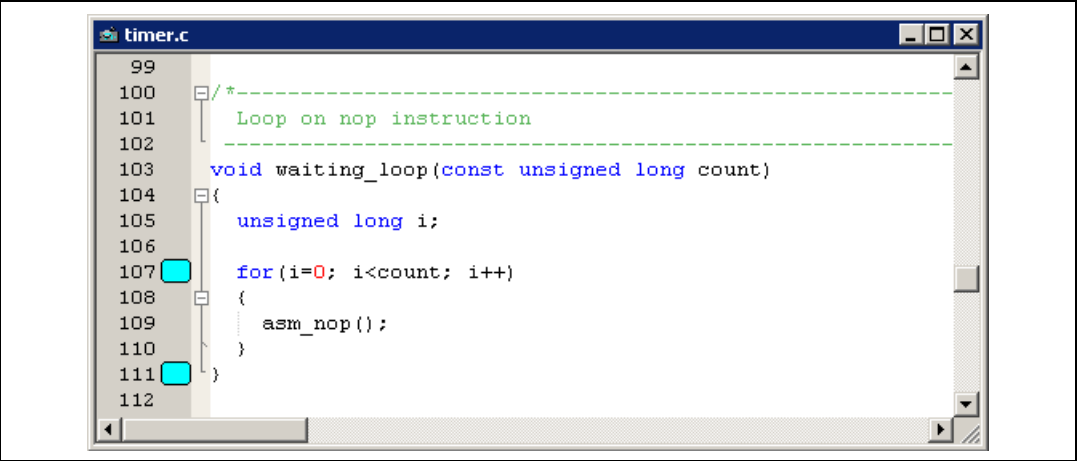
While you can simply turn the trace recording on and off from the [Trace contextual menu](#), you can also program the trace recording so that you record those hardware cycles which interest you most. This is done using the **Advanced Breakpoints** window (for general information on the Advanced Breakpoints feature, refer to [Section 10.2 on page 291](#)). The following examples show you how to program the recording of the trace using advanced breakpoints.

Example 1—Trace ON inside a function

This example shows you how to program the **Advanced Breakpoints** window so that the trace is recorded only during the running of a particular program function.

Imagine that you have a program called `sample1.c`, which calls the function `my_lib` at line 170 (as shown in [Figure 246](#)) and that you want to have the trace recorded while the program is executing that function.

Figure 246. Calling function *my_lib*



- Proceed as follows:
1. Open the **Advanced Breakpoints** window.
 2. You must start application execution with the trace OFF, to do this remove the checkmark next to the **Start with Trace On** option.
 3. Program your Advanced Breakpoint as described in the table below.

Table 76. Advanced breakpoints summary

Step	Description
START with TRACE	OFF
LEVEL 1:	IF (opcode fetch at sample1.c:107) THEN trace ON ELSE IF (opcode fetch at sample3.c:111) THEN trace OFF;
LEVEL 2-4:	(empty)

4. Set **Event 1** to first instruction of *function ()*, and set **Event 2** to last instruction of *function ()*. Trace recording takes place only within the function.

In the Trace window, the discontinuity event in the *Trace Event* column allows you to distinguish each pass inside the function.

Example 2—Programmable trigger position in trace

In this example, we will program the recording of the trace to begin when a specific condition is met, then record the trace for 128 cycles, then stop the program.

Proceed as follows:

1. Open the **Advanced Breakpoints** window.
2. Configure your **Advanced Breakpoints** window as summarized in the table below.

Table 77. Advanced breakpoints summary

Step	Description
START with TRACE	ON
LEVEL 1:	IF (opcode fetch at sample.c:164) THEN (Goto Level 2);
LEVEL 2:	IF (Counter (Trigger IN == 0) >= 128) THEN Break;
LEVEL 3-4:	(empty)

Note: The input trigger is used here to provide an event which is always true. For an unconnected trigger the statement trigger input = 0 is always TRUE.

Example 3—Measure frequency of interruptions

This Advanced Breakpoint program records a snapshot each time the event occurs (such as an interrupt). Then, by calculation from the timing mark of the trace, the frequency of the event can be determined. It may be useful to export the trace record as a text file. The file can then be manipulated using any convenient spreadsheet software, to be presented in graph form.

Program the **Advanced Breakpoints** window as summarized in the table below.

Table 78. Advanced breakpoints summary

Step	Description
START with TRACE	OFF
LEVEL 1:	IF (opcode fetch at main.c:57) THEN (Trace snapshot);
LEVEL 2-4:	(empty)

You must start with the Trace OFF so that snapshots are the only records in the trace buffer.

Example 4—Run until trace full

This simple program allows you to start recording the trace at the start of the program, running until the trace buffer is full, and then stopping the program.

Table 79. Advanced breakpoints summary

Step	Description
START with TRACE	ON
LEVEL 1:	IF (Trace buffer is full) THEN Break;
LEVEL 2-4:	(empty)

Example 5—Measuring long time periods between events

The trace recording includes a **Timestamp** field, but the Timestamp's counter is only 30-bits, while the maximum trace recording length is 256 Kbytes. This means that if you fill the trace buffer, the Timestamp counter is filled and automatically reset many, many times, and the **Restart** message appears in the Timestamp Event field each time the counter is reset.

50 seconds is the approximate time, using the internal 20 MHz clock, that the 30-bit Timestamp counter can count before the counter is filled and it is reset. Therefore, if you want to measure the time passed between two events and this time is longer than 50 seconds, the best thing to do is to set up an Advanced Breakpoint so that the trace is only recorded between the two events of interest. ()

*Note: This is almost true - in addition to the trace records that result from your Advanced Breakpoints program, you may also have some forced trace recordings that occur every time a Restart message occurs. This is because the timestamp counter is started at the same time the program run is started. If the time passed between the start of the run and the occurrence of the first event of interest is longer than about 50 s, there will be a trace record that is blank except for the Restart message in the **Timestamp Event** column.*

For this example, let's assume that we are interested in the time passed between the reception of a certain input trigger signal and the outputting of a certain output signal. We expect these two events to take place within the space of 256 Kbyte cycles.

We will program our **Advanced Breakpoints** window as described in the table below.

Table 80. Advanced breakpoints summary

Step	Description
START with TRACE	OFF
LEVEL 1:	IF (input_probe value == 0xF) THEN (Trace ON), GoTo Level 2
LEVEL 2:	IF (output_probe value == 0xF) THEN (Trace OFF), Break ELSE IF (trace_full = 1) THEN Break;
LEVEL 3-4:	(empty)

*Note: It is necessary to start with **Trace OFF** so that the trace starts to be recorded after the required event occurs and continues either until the second event occurs or the trace is full.*

Once the two events of interest have occurred and you have a trace record spanning the entire time between the events, proceed as follows:

1. Open the **Trace** window.
2. Right-click the mouse in the **Trace** window and select **Show/Hide Columns** from the contextual menu. Ensure that you are able to view the **Record**, **Event**, **Timestamp**, **Timestamp Event** and **TIN (Trigger Input)** fields.
3. Go to the record in the **Trace** window that corresponds to the first event. This record should have **Event 1** in the **Event** field and a value of **0xF** in the **TIN** field. Note the value (**a**) in the **Timestamp** column for this record on a piece of paper.
4. Go to the last record in the **Trace** window (that corresponds to the second event). This record should have **Event 2** in the **Event** field. Note the value (**b**) in the **Timestamp** column for this record.
5. Next, you can search your trace record and count the number of times (**n**) that the Restart message was found.

Each time the Restart message occurs, it means that the 30-bit timestamp counter was filled and automatically reset. A 30-bit counter can measure up to $(2^{30} - 1)$ time periods.

Each time period is equal to the inverse of your Time Stamp Clock frequency. For example, using an internal 20 MHz Time Stamp clock results in a period of 50 ns. So the 30-bit Time Stamp counter is capable of measuring $(2^{30} - 1)$ time periods \times 50 ns = **approximately 53.687 seconds**. If you are using an external Timestamp clock, you must recalculate this according to the clock frequency.

So, to have an absolute measurement of the time passed between the two events of interest, perform this calculation:

- a) **Multiply *n*** (the number of times the Restart message was found) **by 53.687 seconds** (or whatever you have calculated as the maximum time measurable by the 30-bit counter if you are using an external timestamp clock with a frequency other than 20 MHz).
- b) Then **add *b*** (the value found in the Timestamp column of the trace record corresponding to event 2).
- c) Finally, **subtract *c*** (the value found in the Timestamp column of the trace record corresponding to event 1).

Then you will have the true time between the two events of interest.

10.4 Using output triggers

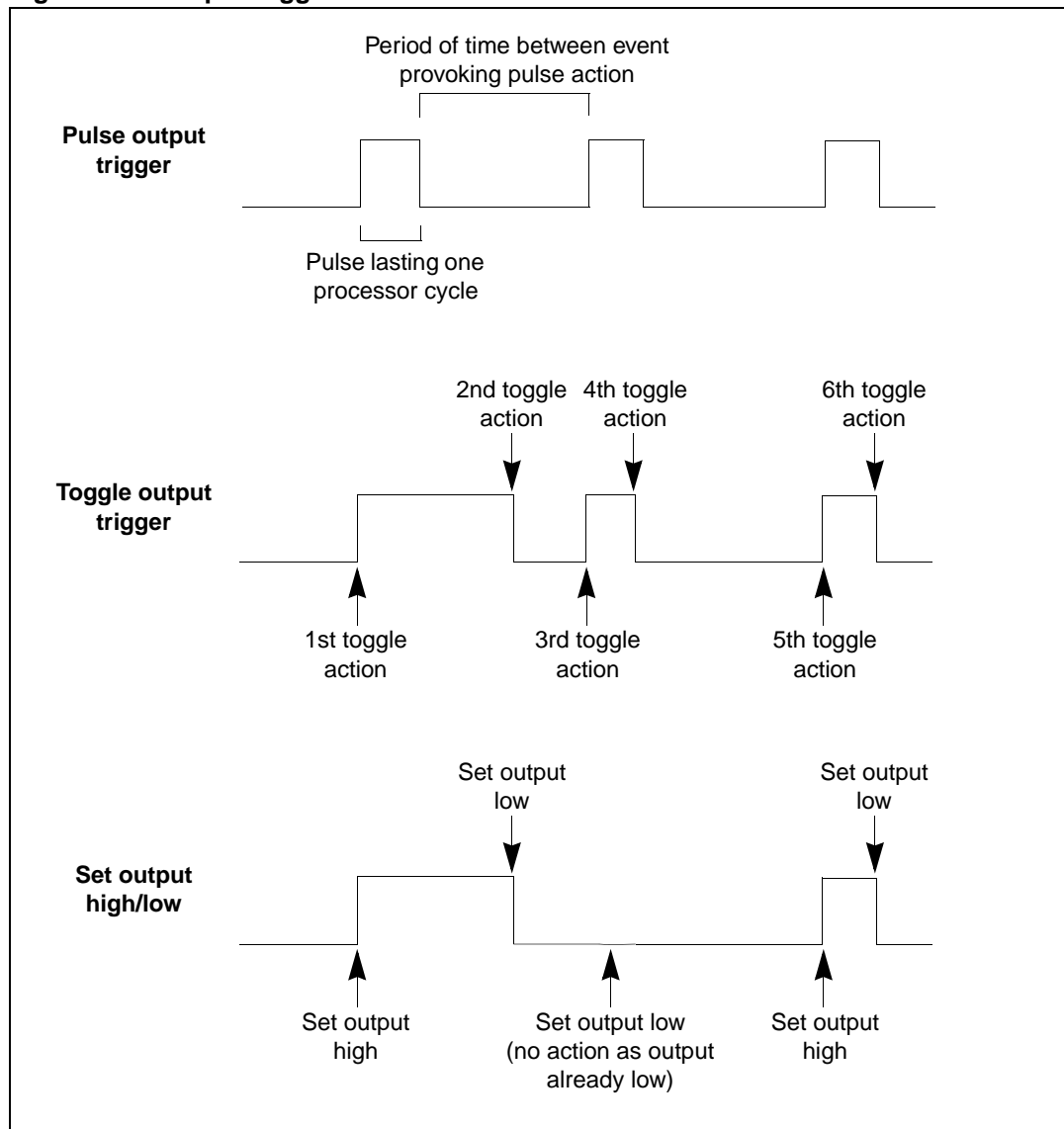
Your ST7 EMU3 emulator has two output triggers, OUT1 and OUT2. You can program the output signals to these triggers using **Advanced Breakpoints** in STVD.

From within Advanced Breakpoints, you can set a trigger output action with the following options (schematically represented in [Figure 247](#)):

- **output a pulse**—where a signal of value 1 is sent during exactly one memory cycle.
- **toggle**—where the signal value is changed from 0 to 1 or from 1 to 0.
- **set the output high**—where the output signal is set equal to 1.
- **set the output low**—where the output signal is set equal to 0.

The following sections provide [Trigger programming examples](#) to help you learn to use Advanced Breakpoints to control the sending of signals to the output triggers.

Figure 247. Output trigger action modes



10.4.1 Trigger programming examples

The following examples show you how to control the trigger output signals by programming the **Advanced Breakpoints** window.

Example 1: Sending a pulsed signal to synchronize an oscilloscope

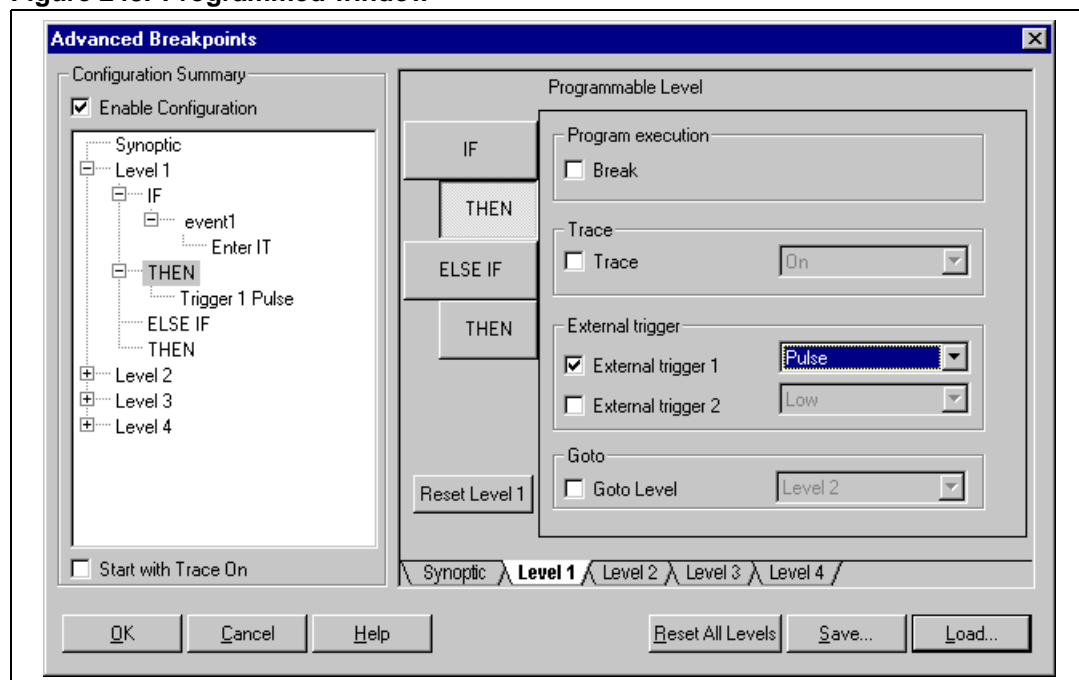
Sometimes it is useful to send a pulsed signal from one of the output triggers to synchronize an external device upon the occurrence of a particular event. For example, say that you want to synchronize an oscilloscope upon the occurrence of an entry into an interrupt loop.

Program your **Advanced Breakpoints** window as summarized in the table below.

Table 81. Advanced breakpoints summary

Step	Description
START with TRACE	ON or OFF
LEVEL 1:	IF (Enter IT) THEN (Trigger 1 Pulse);
LEVEL 2-4:	(empty)

The **Advanced Breakpoints** window should appear as shown in [Figure 248](#).

Figure 248. Programmed window**Example 2: Toggling a trigger output signal**

Suppose that you want the output trigger signal to be toggled (go from 0 to 1 or from 1 to 0) upon a certain conditional event.

In this example, the event is the detection of an opcode fetch at a specific address corresponding to a function call at line 170 in `sample.c`.

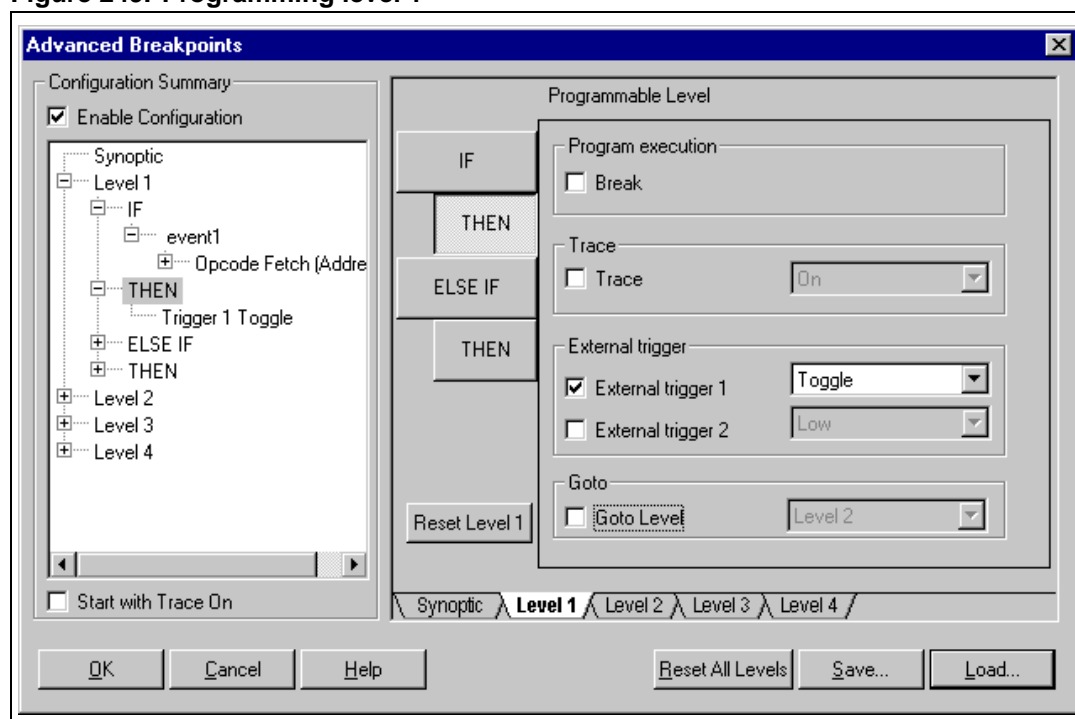
We need to program the **Advanced Breakpoints** window as summarized in [Table 82](#).

Table 82. Advanced breakpoints summary

Step	Description
START with TRACE	ON or OFF
LEVEL 1:	IF (opcode fetch at <code>sample.c:170</code>) THEN (Trigger 1 Toggle);
LEVEL 2-4:	(empty)

The programmed **Advanced Breakpoints** window should appear as shown in [Figure 249](#).

Figure 249. Programming level 1

**Example 3: Setting the trigger output high or low**

In the previous example, upon each occurrence of a specific event (an opcode fetch at `sample.c`, line 170) the trigger is toggled—meaning that the new trigger value is the opposite of the old trigger value. Toggling changes the relative value of the trigger.

In this example, we show that we can also effectively toggle the trigger high or low using the set high and set low options. The advantage of toggling in this way is that you can set a specific trigger value upon the occurrence of a particular event.

This is advantageous if you are not sure of the order in which events occur during the running of your application.

Imagine that we have an external device attached to our input probe and we want to make certain that an output signal of value “1” is sent from Output Trigger 1 every time we receive a positive signal value from the input trigger.

However, at the same time, if our application enters an interrupt loop, we want this event to override all else and send a signal of value “0” from Output Trigger 1.

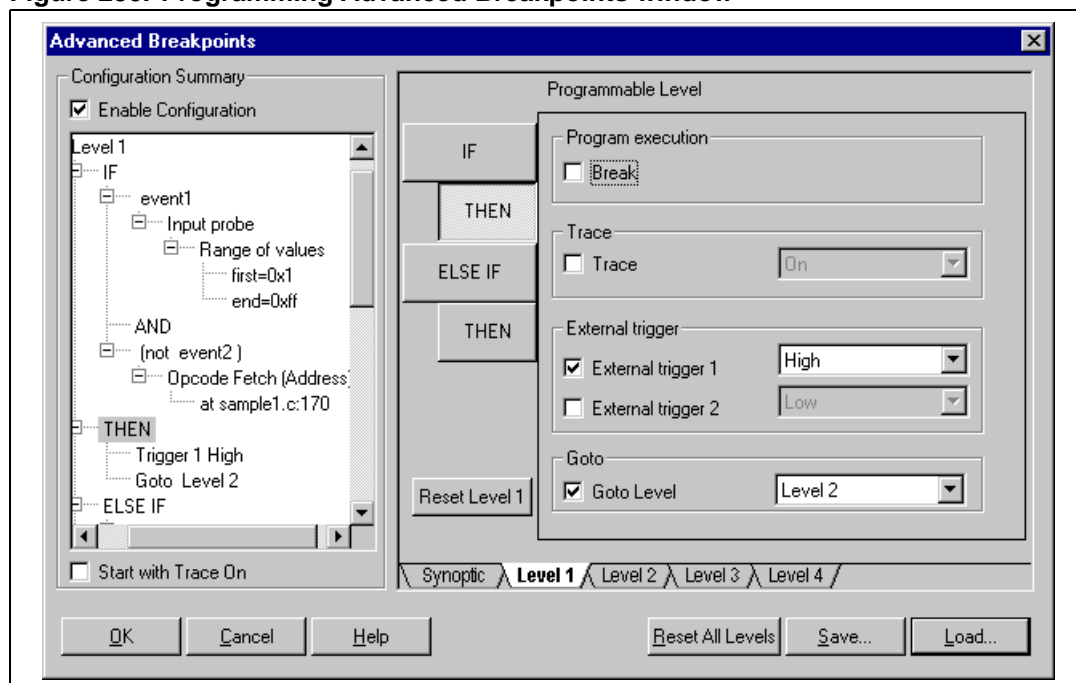
While neither of these events has occurred, we would also like to send a signal of value “0” from Output Trigger 1.

Program your **Advanced Breakpoints** window as summarized in [Table 83](#).

Table 83. Advanced breakpoints summary

Step	Description
START with TRACE	ON or OFF
LEVEL 1:	IF (event 1: Probe input = Range of values, first=0x1, last=0xff) AND (not event 2: Opcode Fetch (Address) at sample.c:170) THEN (Trigger 1 High) AND (GoTo Level 2); ELSE IF (event 3: Input probe = 0x0) OR (event 4: Opcode Fetch (Address) at sample.c:170) THEN (Trigger 1 Low)
LEVEL 2:	IF (event 1: Opcode Fetch (Address) at sample.c:170) THEN (Trigger 1 Low) AND (GoTo Level 1)
LEVEL 3-4:	(empty)

Your **Advanced Breakpoints** window should appear as shown in [Figure 250](#).

Figure 250. Programming Advanced Breakpoints window

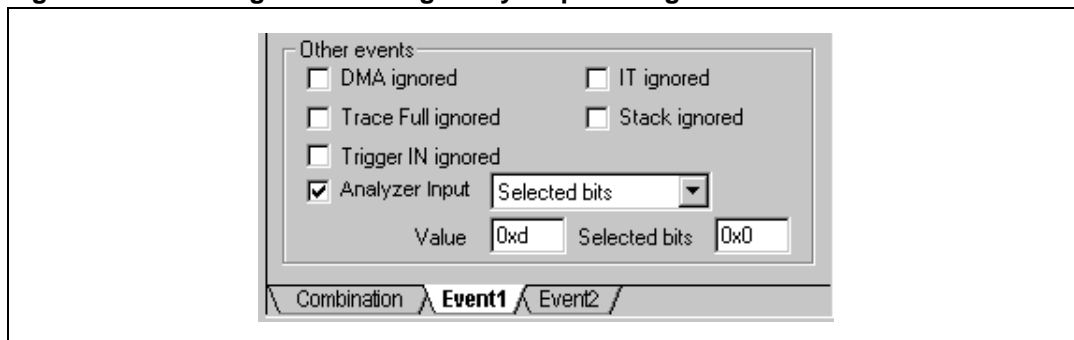
10.5 Using analyzer input signals

The ST7 EMU3 emulator allows you to use eight external input signals (TTL level). These input signals can be used as other events (see [Section 10.2.3 on page 300](#)) in the programming of Advanced Breakpoints.

You can view the value of the Analyzer probe input at any time, using STVD. From the main menu, select **View>Trace**. The input signal value is listed in the **Analyzer Probe** column.

STVD's **Advanced Breakpoints** allows you to use these input signals to define an event. From the main menu, select **Debug Instrument > Advanced Breakpoints** to open the dialog box.

Figure 251. Defining events using Analyzer probe signals



As shown in [Figure 251](#), you can provide a specific value (with or without bit masking) or a range of values (see [Analyzer input examples](#)).

10.5.1 Analyzer input examples

Example 1- Specifying an Analyzer probe value with selected bits masked

The Analyzer probe has an 8-bit input. Imagine that you want to specify an input value of 91 (5B in hexadecimal format). You can simply specify the value “91”. A value of “91” received on an 8-bit input looks like:

0	1	0	1	1	0	1	1
---	---	---	---	---	---	---	---

However, suppose that you want to specify the value “91” but mask certain bits. These bits are shaded in the figure below:

0	1	0	1	1	0	1	1
---	---	---	---	---	---	---	---

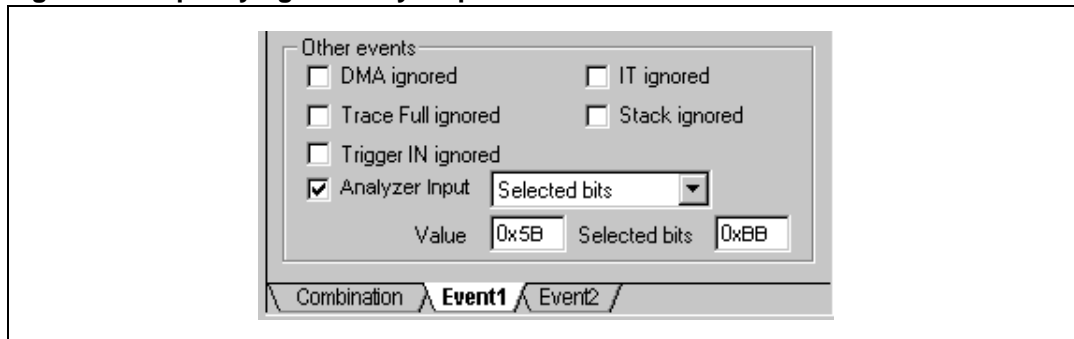
The selected bit mask is therefore represented by:

1	0	1	1	1	0	1	1
---	---	---	---	---	---	---	---

which corresponds to the value “BB” in hexadecimal format, or “187” in decimal format.

So, to specify a value of “91” with the second and sixth bits masked, you would complete the **Advanced Breakpoints** window as shown in [Figure 252](#).

Figure 252. Specifying an Analyzer probe value with bit mask



Example 2—Setting an advanced breakpoint on an input probe value


This program allows you to make the trace recording conditional on a specific input probe signal, and then record the trace until the trace buffer is full. Program the **Advanced Breakpoints** window as summarized below:

Table 84. Advanced breakpoints summary

Step	Description
START with TRACE	OFF
LEVEL 1:	IF (input_probe value == 0xF) THEN (Trace ON)
LEVEL 2:	IF (trace_full = 1) THEN Break;
LEVEL 3-4:	(empty)

Note: You must start with Trace OFF so that the trace starts to be filled after the required event occurs and until the trace is full.

10.6 Performance analysis

You can open the **Performance Analysis** window by clicking on  (the Performance Analysis icon) from the Menu Bar, or by selecting **Debug Instrument > Performance Analysis**.

The **Performance Analysis** window allows you to measure the time spent in a given portion of your code in order to evaluate the efficiency of your application.

In the **Performance Analysis** window, you select a section of code (for example, a function) by choosing a start and end bookmark, or by defining the start and end addresses.

Enter the number of passes to be analyzed. The program is run and timed. The **Total time** for the number of passes is recorded. The percentage time inside the defined start and end points (**% time in code**) is calculated, as is the **Maximum time**, **Minimum time**, and **Average time** for a single pass.

The results are displayed in the **Statistics** tab of the **Performance Analysis** window. They may also be viewed in graphical form on the **Graphic** tab, and a number of graphical manipulations carried out to facilitate analysis.

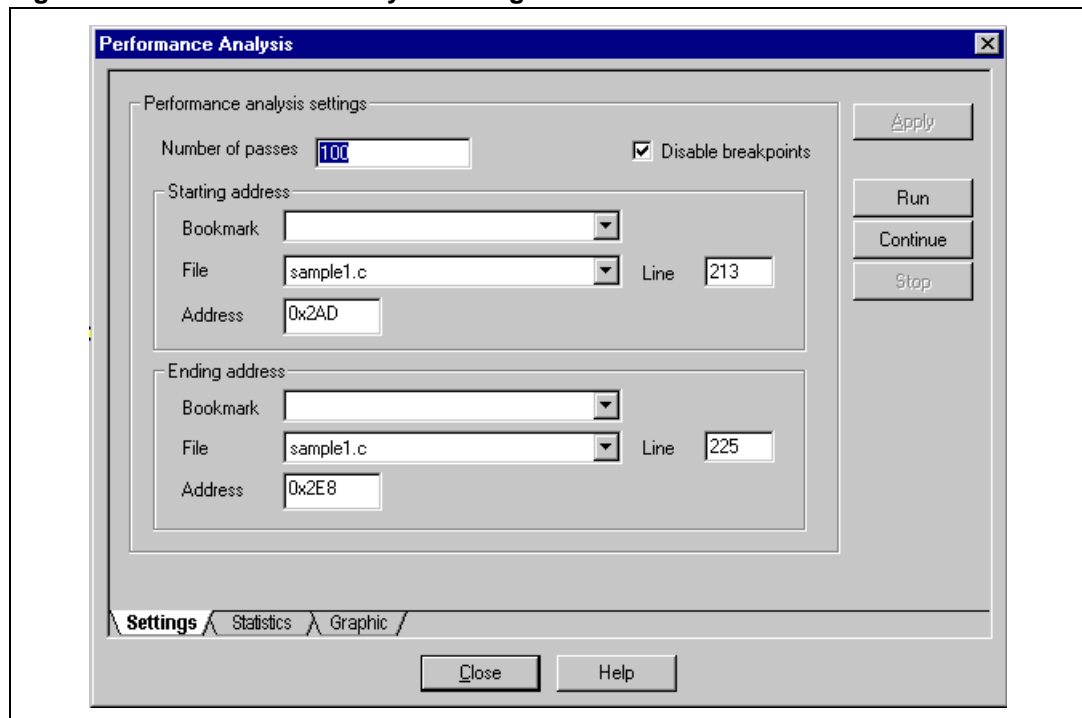
This section provides information about:

- [Running a performance analysis](#)
- [Using the Run/Stop/Continue commands](#)
- [Viewing results](#)

10.6.1 Running a performance analysis

The **Performance Analysis** dialog box appears as shown in [Figure 253](#).

Figure 253. Performance Analysis dialog box



In this dialog box, you must specify the following:

- **Disable breakpoints:** this checkbox disables all current breakpoints during the test.
- **Number of passes:** the number of times that the specified sequence of code will be run for the test.
- **Starting address:** specifies the start address of the sample and the name of the module in which it is located. The start address may be specified as a bookmark location in a specific file, or an address. If any **Bookmark**, **Line**, or **Address** field is completed, the remaining fields are specified automatically.
 - **Bookmark:** current bookmarks are listed in the **Bookmark** pull-down menu and may be selected to define the start point for the program sample (click on a bookmark definition to set it as the start address).
 - **File:** project executable files are listed in this pull-down menu, or may be entered by hand.
 - **Line:** the line number may be entered here.
 - **Address:** the address may be entered in decimal or hexadecimal format.
- **Ending address:** as for the **Starting address** (above), but specifying the final address of the program section to be tested.

10.6.2 Using the Run/Stop/Continue commands

When the definition is complete, use the **Run**, **Stop** and **Continue** buttons at the top right of the window to execute/stop the program:

- **Run:** runs the executable. This command carries out a chip reset and runs the program from the reset vector.
- **Continue:** restarts the program from the current Program Counter location. When this command is used, the performance analysis is also restarted. The elapsed time and

number of passes executed are reset to zero and a new record is made, starting from the Program Counter location.

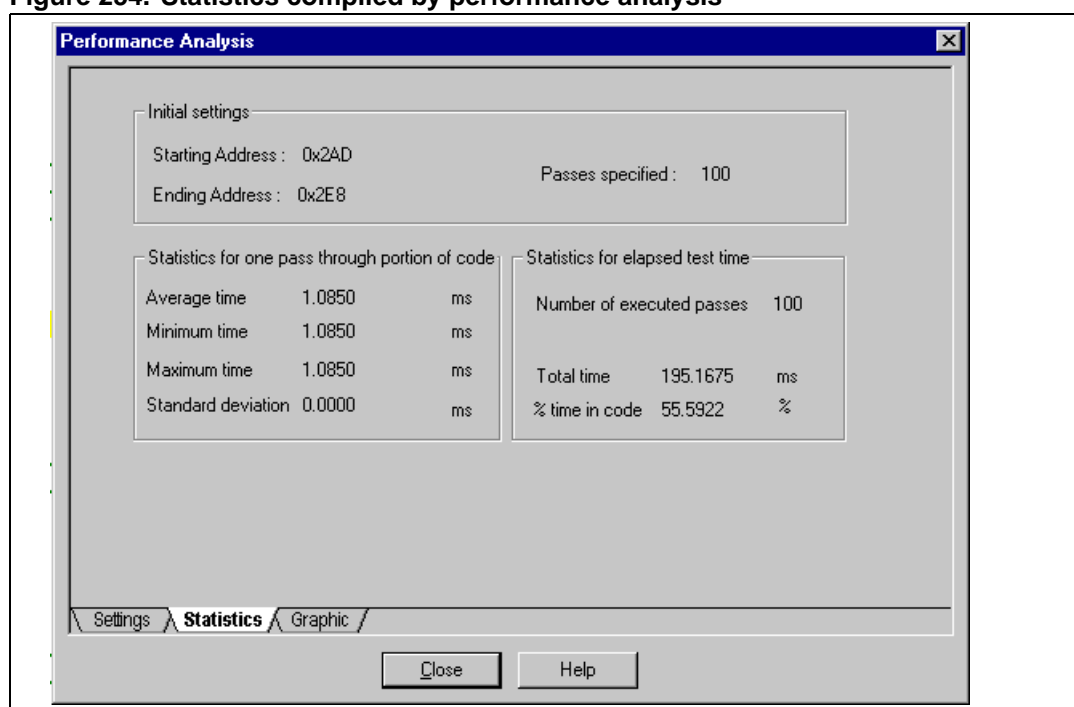
- **Stop:** stops program execution and shows performance analysis results to this point.

When the program execution terminates, the **Statistics** tab opens automatically to display the results of the analysis.

10.6.3 Viewing results

The **Statistics** tab (shown in [Figure 254](#)) sets out the numerical data resulting from the performance analysis. The results on this page are expressed in unit that varies depending on the duration of the events analyzed. The results may be expressed in seconds (s), milliseconds (ms), microseconds (μ s), or nanoseconds (ns).

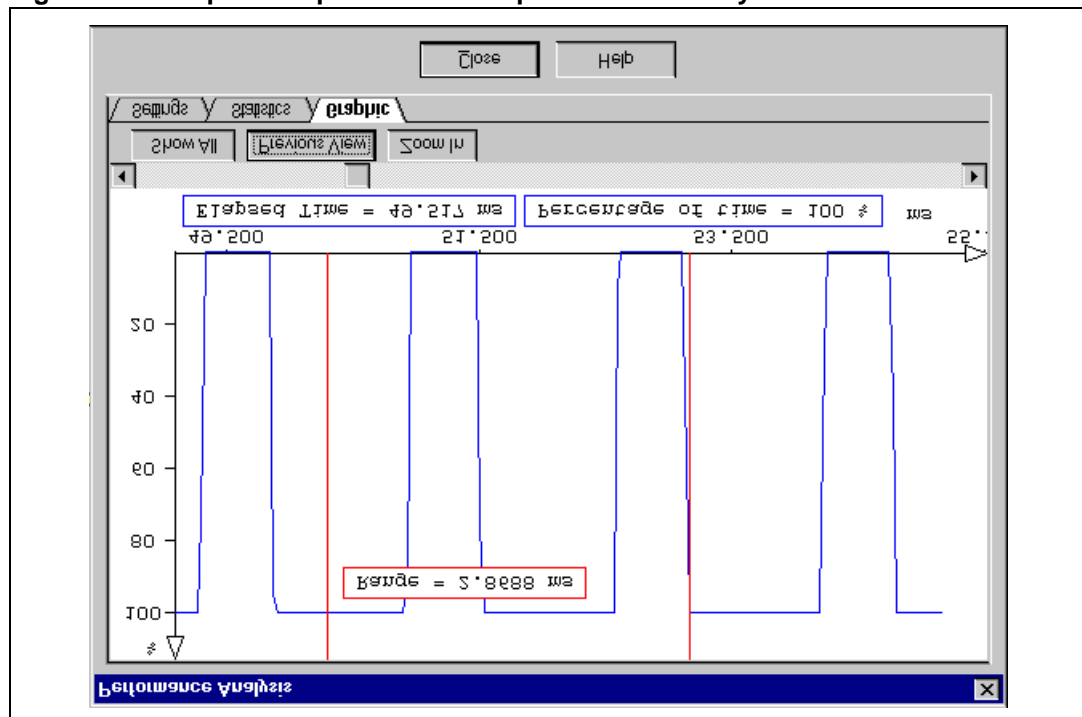
Figure 254. Statistics compiled by performance analysis



- **Initial settings:** The initial settings that you entered are summarized here, defining the section of code under examination.

- **Statistics for one pass through portion of code:** These values concern timing data for the selected portion of code (between the chosen start-point and end-point).
 - **Average time:** Average time for a single passage between start- and end-points.
 - **Minimum time:** The most rapid timing recorded between the two points.
 - **Maximum time:** The slowest pass between the two points.
 - **Standard deviation:** Standard deviation for the range of timings obtained over the number of passes carried out, expressed as a deviation from the average time for a single passage.
- **Statistics for elapsed test time:** These results express overall timing values:
 - **Number of executed passes:** The total number of passes through the selected code carried out during the analysis.
 - **Total time:** The runtime, from start (Run/Continue) to the occurrence of the last significant event.
 - **% time in code:** The percentage of the total time which is passed executing the selected code (between the user-defined start-point and end-point).

Figure 255. Graphical representation of performance analysis



The **Graphics** tab sets out the graphical data resulting from the performance analysis, as shown in [Figure 255](#). The graphical display has the following features:

- **Horizontal axis** shows the elapsed time measured from the start of runtime.
- **Vertical axis** shows the percentage of time (within the defined section of code).
- **Units of time** on the graph are selected automatically depending on the duration of the events analyzed; values may be expressed in seconds (s), milliseconds (ms), microseconds (μ s), or nanoseconds (ns).
- **The cross-hair pointer:** The current location of the pointer on the graph is interpreted at the bottom of the graph in the boxes **Elapsed Time** and **Percentage of time**.

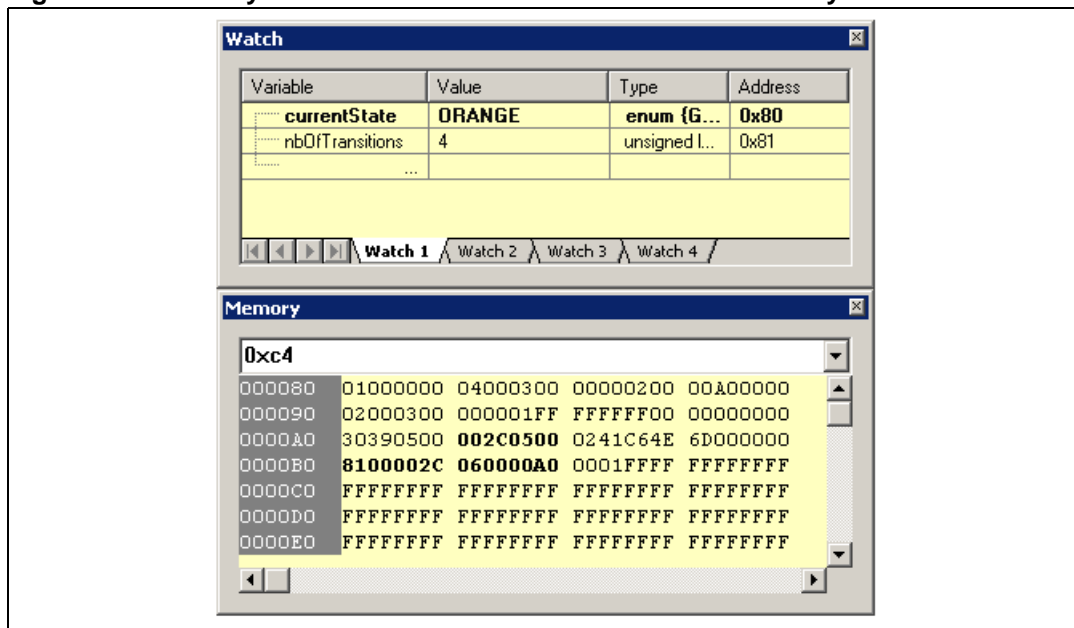
- **Show All** modifies the scale of the graph to include the whole record in the visible area.
- **Previous View** restores the previous graph view (for example to zoom out after a **Zoom In**).
- **Zoom In**: left-click on the mouse to position a vertical marker on the graph. Two of these markers may be positioned (left-click in each case) and the view zoomed to the region between them using the **Zoom In** button.

10.7 Read/write on the fly

The **Read/Write on the Fly** option permits you to non-intrusively read or write a value in the RAM memory of the emulation chip while the application is executing.

When this option is selected in either the **Memory** window or the **Watch** window (indicated by a checkmark beside the option in the contextual menu), all of the memory entries shown in the window are continuously refreshed at one-second intervals while the program is running. In these windows, the background turns yellow if the **Read/Write on the Fly** option is active.

Figure 256. Memory and Watch windows with read/write on the fly



Note that when you scroll in the **Memory** Window, the newly visible memory entries turn bold, to show that they are being updated, then change back to normal, indicating that they are current and valid.

When the **Read/Write on the Fly** option is active, the only time that the values shown in the **Memory** window may not be valid is during the direct editing of a memory value during the running of a program. In this event, the edited value appears in red, indicating that it has not yet been taken into account by the program, it then turns to bold black, to indicate that the program is adjusting accordingly, and then finally, all entries are in normal weight black, indicating that the window is valid and updated.

10.8 Performing automatic firmware updates

Your emulation hardware contains programmable logic devices for the emulation of specific MCUs or families of MCUs. This firmware may need to be updated if you have changed the Target Emulation Board in your EMU3 probe, or if you have updated your version of STVD. STVD checks your emulator's firmware each time you start a debug session, and updates them if older versions are detected.

After launching STVD, the software automatically checks the firmware versions when you start a debugging session by selecting **Debug > Start Debugging**. If STVD detects an old firmware version, you will receive the following prompt:

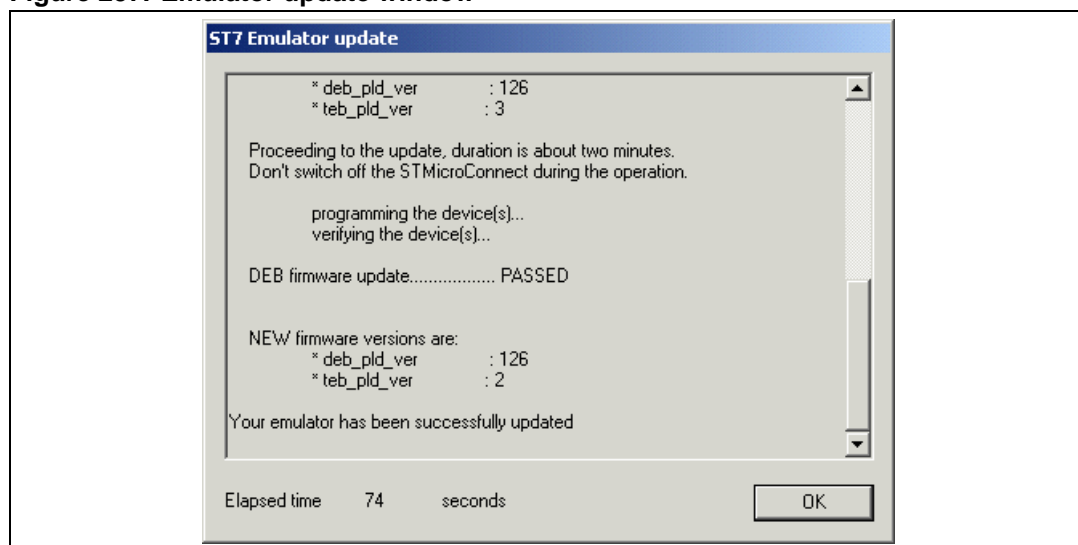
```
Your emulator needs to be updated, would you like to proceed
now?
```

To proceed with the update, click **Yes**. The **Emulator update** window will open on your screen (see [Figure 257](#)). This window provides you with the status of the firmware update. It also tells you what firmware is being updated and the new versions, and indicates how much time the update took. The update will take up to two minutes, depending on your host PC and connection type.

If you choose not to update your firmware and click **No** when prompted to update, you will receive an error message indicating that your emulator's firmware needs to be updated. If you do not want to update its firmware, you must use an older version of STVD.

During the update, you must not disconnect your emulator. Disturbing the connection between the emulator and the host PC will result in an incomplete update of the firmware, which will affect the integrity of your emulator.

Figure 257. Emulator update window



While STVD is capable of detecting and updating out-of-date firmware, it is also possible to have a previous version of STVD that does not support more recent firmware versions. If this is the case, you will receive a message indicating that your emulator is too recent to be managed by the STVD version you are using.

To avoid this problem, be sure to install the most recent version of STVD when you begin using a new emulator or TEB. The most recent version of STVD is provided with your emulator and can also be downloaded from the ST web site www.st.com.

11 Program

Once you have finished debugging your application and have built the final version, you can program it to your target microcontroller using STVD's Programmer.

This feature is based on ST Visual Programmer (STVP). It offers basic programming features and allows you to program your target microcontroller using a programming board or in-circuit programming (ICP) hardware.

To access the Programmer, select **Tools > Programmer** from the main menu bar. The **Programmer** window opens.

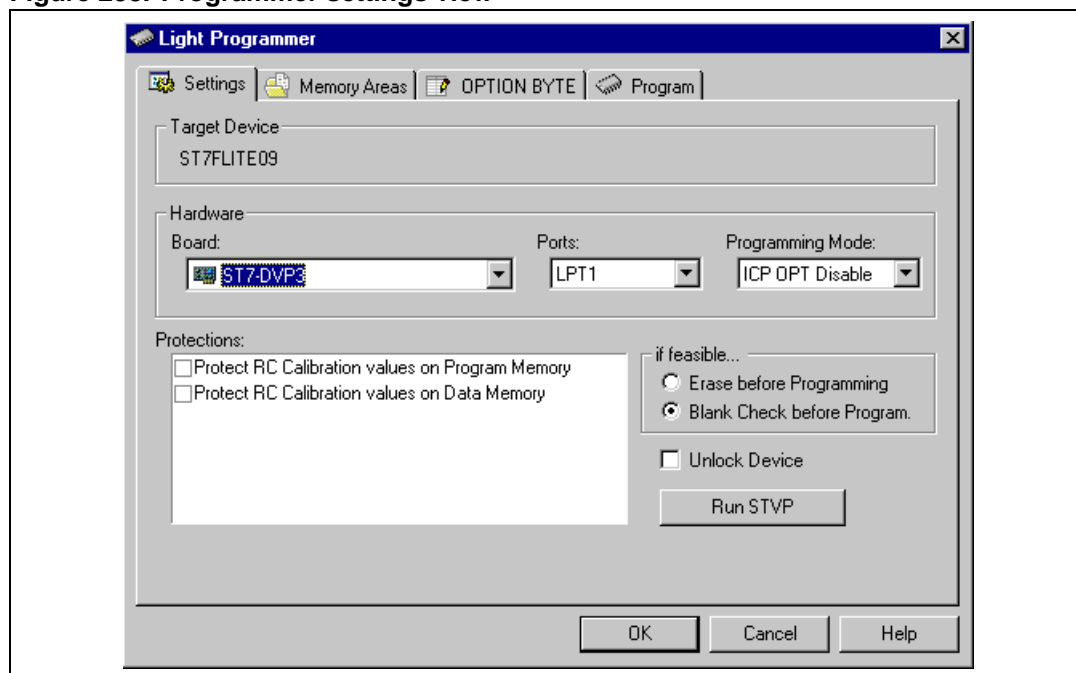
The following sections provide information about this programming interface, including:

- [Section 11.1: Configuring general programming settings](#)
- [Section 11.2: Assigning files to memory areas](#)
- [Section 11.3: Configuring option byte settings](#)
- [Section 11.4: Starting the programming sequence](#)

11.1 Configuring general programming settings

When you access the Programmer (**Tools>Programmer**), the window opens to the **Settings** tab ([Figure 258](#)). In this tab you can configure the general settings for programming your microcontroller.

Figure 258. Programmer settings view



The options in the list boxes will vary according to the target MCU that you identified for your project when you built your application (for more information, refer to the section about the [Selecting your MCU](#) of the **Project Settings** window). When programming your MCU, the target microcontroller is displayed in the **Target Device** field of the **Settings** tab.

Depending on your target microcontroller you will be able to configure the following:

- [The programming hardware and connection port](#)
- [The programming mode](#)
- [Protections](#)
- [Erase before programming option](#)
- [Unlock device](#)

In addition to these settings, the **Run STVP** button allows you to launch STVP, instead of using STVD's Programmer. To exit the Programmer, click on **Run STVP**. STVD prompts you to confirm that you want to close the Programmer window and start an STVP session. Click on **OK** to confirm.

The programming hardware and connection port

The fields in the hardware section of the **Settings** tab allow you to specify the programming hardware and the connection port to use. To configure these settings:

1. Identify your programming hardware from the **Board** list box.
Only the programming hardware that supports your target MCU is listed.
2. Identify the port that you will use to connect to your programming hardware in the **Ports** list box.
Only the ports for the connection types supported by your programming hardware are listed.

Note: *While some emulators may support Parallel, USB and Ethernet connections for emulation, not all connection types are supported for programming. When programming, you will have to use a supported port listed in the **Ports** list box.*

The programming mode

The Programmer supports multiple programming modes depending on your target MCU and your programming hardware. Select the programming mode from the list of options in the **Programming Mode** list box. Only the modes supported by your MCU and programming hardware are listed. The available modes can include:

- **Socket mode:** This programming mode is used when you are programming a device in an EPB or DVP socket. If there are several sockets, take care to insert only one device at a time.
- **ISP mode:** In-situ programming mode can be used when your device is soldered on your application board. The application board must be designed with the necessary connection hardware.
For more information about connection requirements, refer to the user manual for your programming hardware.
- **ICP modes:** In-circuit programming modes are used to program XFlash and HDFlash devices. For ICP modes, you have the choice of starting with option bytes disabled (ICP OPT disable) or enabled (ICP OPT enable).

Caution: Before selecting one of the in-circuit programming modes, you should determine whether or not the device has come directly from the factory. Devices delivered directly from the factory are programmed with factory default options. If you are not certain of the option byte settings, or you know the settings are not consistent with your application board configuration, choose **ICP OPT Disable** to avoid MCU start-up failures. For option byte descriptions and default settings, refer to your microcontroller's datasheet.

ICP mode with Option Bytes Disabled is a safe programming mode that allows you to start your MCU while ignoring the current option byte values. This is to avoid start-up failures that result from option byte settings that are not compatible with the application configuration. For example, you can use it when the clock on your application board doesn't match the oscillator type specified by the MCU's option byte.

To use this mode you **must** connect your MCU to either the ICCOSC clock signal from the ICC connector, or a clock signal from your application board that has a square wave form between 0V and VDD. For more information about these connections, refer to the user manual for your programming hardware.

When using **ICP mode** or **ICP mode with Option Bytes Enabled**, the MCU's option byte settings are taken into account upon startup. Use this programming mode if your MCU has the factory default settings, and if you are certain that the clock source on your application board matches the oscillator type specified by the MCU's option bytes.

In this mode, the ICCOSC clock from the ICC connector is not required.

Protections

Some devices allow you to protect specific values that may already be loaded in a register in your MCU's memory when programming. These types of values are specific to one of your MCU's features, such as RC calibration values. If your MCU supports a feature with that can be protected, a protection option will appear the **Protections** field. You can enable the option by clicking to place a checkmark in the box next to it.

RC Calibration values

Some microcontrollers store values for calibrating the internal RC oscillator in the RC Control Register. To avoid erasing these values, the option "Protect RC calibration values in Program Memory/Data Memory" appears in the **Protections** field.

Caution: Any program or data in the memory are lost when Read Out Protection is disabled.

Restoring the default RC calibration values in the microcontroller's Data and Program memory can be done by enabling and disabling Read Out Protection. If Read Out Protection is already disabled, set it to ON in the option byte settings and program the option byte. Then set Read Out Protection back to OFF and reprogram the option byte. The default RC calibration values are written to the appropriate addresses in Program and Data memory. Refer to your microcontroller's datasheet for the default values and addresses.

Erase before programming option

This option is only available when programming Flash devices. If you check this box, the programmer will erase the memory areas that you have identified for programming, before starting the programming sequence. When you erase the Flash device, the memory area being erased is set to FFh.

Note: For information about selecting memory areas for programming, refer to [Section 11.2: Assigning files to memory areas](#).

If you enable this option, Erase replaces the Blank Check step in the programming sequence (see [Section 11.4: Starting the programming sequence](#)). In this case, the Blank Check is not performed.

Unlock device

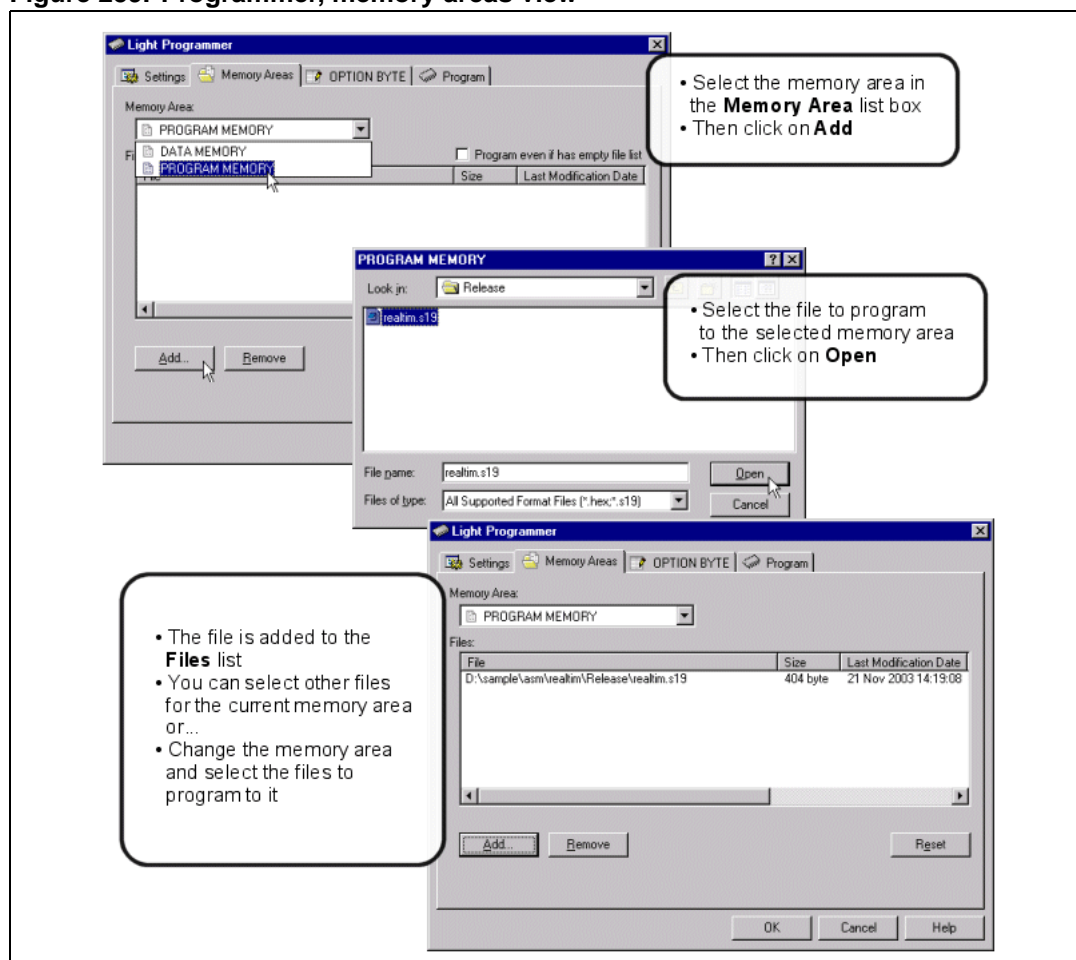
The **Unlock Device** option is only available for specific Flash devices. If you check this option, the programmer will unlock the microcontroller before starting any other programming activities (such as Erase, Blank check, Program, Verify).

11.2 Assigning files to memory areas

When you program your microcontroller, only the memory areas that you select are programmed. All areas, except those dedicated to option bytes, are selected for programming by associating them with one, or more files. Files contain the values to be programmed to a memory area.

With the **Programmer** window open (**Tools>Programmer**), select the **Memory Areas** tab. In this tab you can associate the files with your microcontroller's memory areas for programming ([Figure 259](#)).

Figure 259. Programmer, memory areas view



The memory areas that are available may vary according to the target microcontroller that you specified when you built your application.

To assign a file for programming to a memory area:

1. Select a memory area from the **Memory Area** list box.
2. Click on the **Add** button to open a browse window.
3. Select the file to be programmed to this memory area, then click on **Open**.

The file is added to the **Files** field.

“Program even if has empty file list” option

If you don't have a file to program to a memory area, you can still select it for programming by checking the **Program even if has empty file list** checkbox. This checkbox is only visible when the **Files** field is empty.

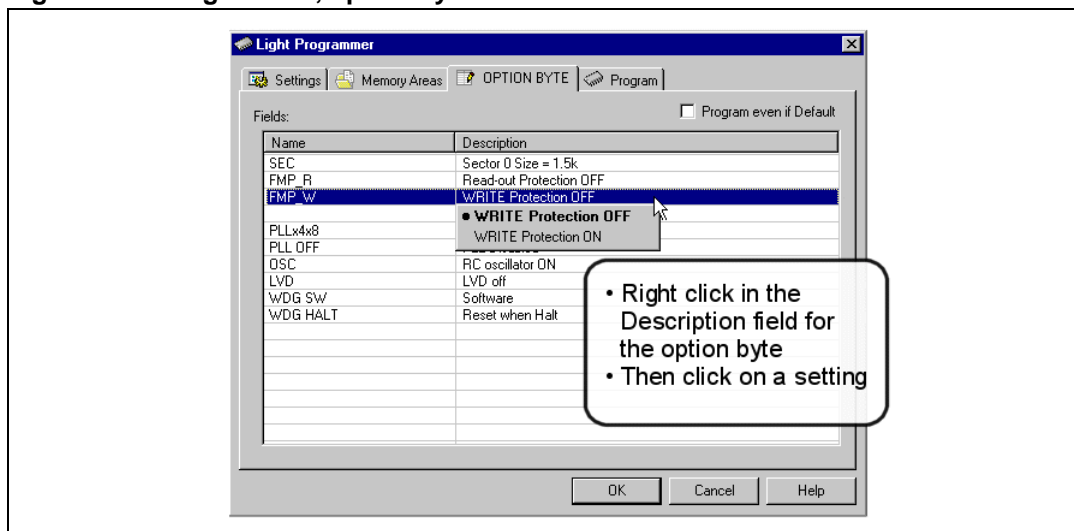
This option can be useful when programming a device that has already been programmed (when the memory contains values other than default values). In the Memory Area tab, you can select the file to use when programming the Program Area. Then, to ensure that no data remains in the Data Area, you can check the **Program even if has empty file list** option for that area. As a result your application will be programmed to the Program Area and the Data Area will be programmed with the default value, FFh.

11.3 Configuring option byte settings

Option bytes control a variety of configurable features for ST7 Flash devices such as memory write protection, clock source, watchdog and low voltage detection. The option byte settings that you want to program to your target MCU are configured in the **OPTION BYTE** tab.

With the **Programmer** window open (**Tools>Programmer**), select the **OPTION BYTE** tab (Figure 260). The tab is divided into a **Name** field, which contains the abbreviated name for each option byte, and **Description** field, which contains the settings that will be programmed to your target MCU.

Figure 260. Programmer, option byte view



Note: The values shown are the factory default settings for your microcontroller. These settings may not be the same as the settings in your actual target microcontroller.

To change the setting for an option byte:

1. Right-click in the option byte's **Description** field to see the pop-up list of possible settings.
2. Click on the new setting in the pop-up list.

The new setting for the option byte is displayed in the **Description** field.

“Program even if default” option

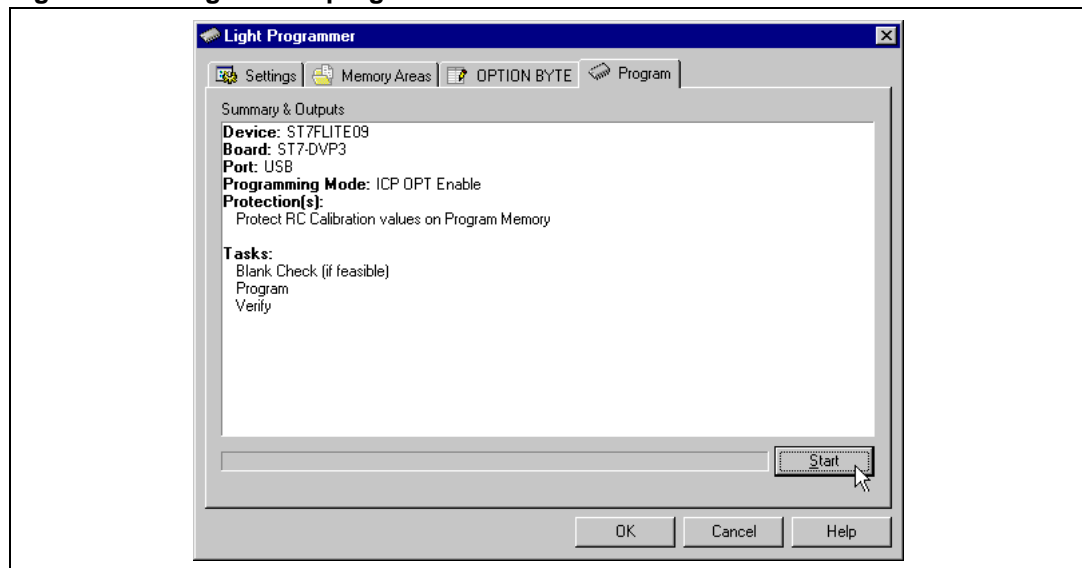
The Programmer only programs the Option Byte Area of memory if the settings indicated in the **Option Bytes** tab are different from the factory default settings. However, you can check the **Program even if default** option and the Programmer will program the default option byte settings shown in the **OPTION BYTE** tab.

This option is useful when you are programming to a microcontroller that has already been programmed. By using this option, you can ensure that the option bytes are set to the factory defaults.

11.4 Starting the programming sequence

With the **Programmer** window open (**Tools>Programmer**), select the **Program** tab. In this tab, you can confirm the programming settings, start the programming sequence and monitor the programmers progress ([Figure 261](#)).

Figure 261. Programmer program view



Before you begin the programming sequence, the **Summary and Outputs** field contains information about the settings you have applied. This information includes:

- **Device** (the target MCU)
- **Board** (your programming hardware)
- **Port**
- **Programming Mode**
- **Tasks** (the sequence of tasks that the programmer will execute)

To start the programming sequence, click on the **Start** button. The Programmer connects to your target MCU and begins the programming sequence.

During programming, the **Summary and Outputs** field displays any status messages and errors that result from the execution of the programming tasks.

12 STM8 C tutorial

The STM8 tutorial illustrates some of the main features of STVD by guiding you through the process of building and debugging a sample application in C language.

Each procedure presented in this tutorial focuses on a functional aspect of the build and debug processes. Steps should be performed in the sequence in which they are presented; failure to do so can result in errors in the steps that follow.

After the successful completion of a procedure, you should save your workspace by selecting **File>Save Workspace As...** from the main menu. By doing so, you will be able to return to a previous, successfully completed stage if necessary.

The tutorial example is based on an application for a traffic signal. It is designed to be built using the Raisonance or Cosmic C toolset for STM8. You can download limited versions of these toolsets for free from **st.mcu.com**.

This application uses features that are common to all STM8 microcontroller families (STM8L, STM8A, STM8S) so that emulator users can choose a microcontroller that is supported by their emulator to follow most of the tutorial. The following files are provided to build the application:

1. Setup: only minimum source files are provided.
 - *timer.c*: application source file written in C language. This is an application for a traffic signal. A bug is generated during the execution of the application.
 - *mcuregs.h*: generic include file for microcontroller-specific constants.
 - *st.toolset/include/<mcuname>.h*: microcontroller-specific constants.
 - *Raisonance/R_stm8_interrupt_vector.c*: file containing the interrupt vector table.
 - *Cosmic/stm8_interrupt_vector.c*: file containing the interrupt vector table.
2. Build.
 - *tutorial.stw*, *tutorial.wed*: workspace information files.
 - *cosmic/cosmic.stp*, *cosmic/cosmic.dep*: cosmic project information files.
 - *raisonance/raisonance.stp*, *raisonance/raisonance.dep*: raisonance project information files.

These files are created during step 1, but are provided in the directory *step2_build*, so that you may start the tutorial from step 2.

Note: For standard installations of STVD, the files listed above are saved under:
C:\Program Files\STMicroelectronics\st_toolset\stvd\Examples\tutorial_STM8

Some lessons of the debug tutorial relate to features that are specific to a debug instrument, while others relate to all debug instruments.

Table 85. Tutorial sections and applicable debug instrument

Section	Debug instrument
<i>Setup</i>	All
<i>Build</i>	All
<i>Start debugging</i>	All
<i>Instruction breakpoints</i>	All

Table 85. Tutorial sections and applicable debug instrument

Section	Debug instrument
View execution	All
Perform memory mapping	All
View program execution history	EMU3, STice
Use read/write on-the-fly	EMU3, STice, SWIM
Set an advanced breakpoint	EMU3 (and STice in the near future)
Run a coverage and profiling session	STice

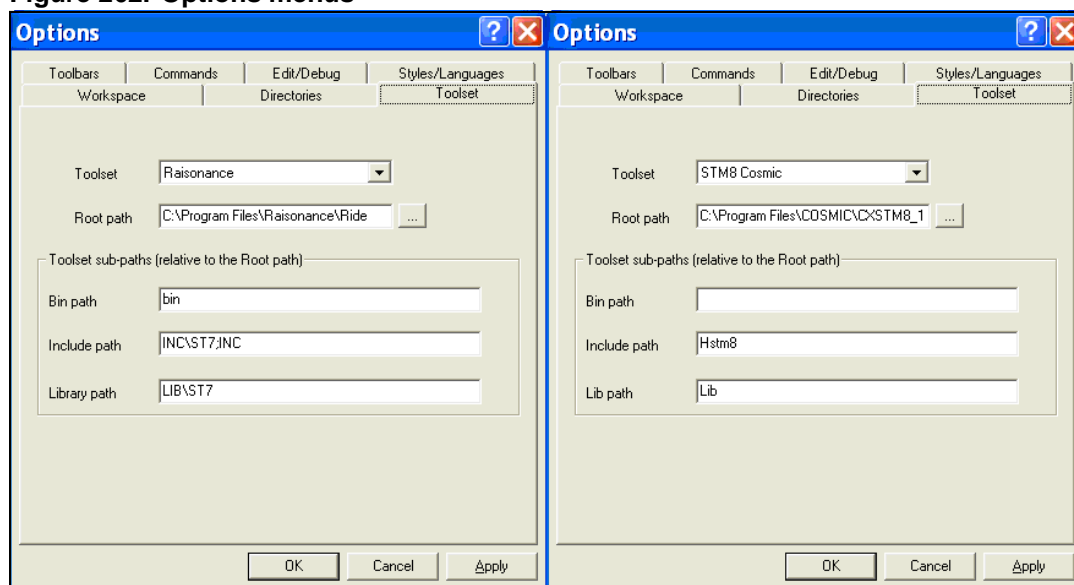
12.1 Setup

In this lesson you will learn how to setup the workspace environment so that you can build, debug and program the application to a microcontroller. Here, you will:

- [Create a workspace](#)
- [Create a project](#)
- [Add the source files](#)
- [Create a folder to organize files in the project](#)

12.1.1 Install the toolset and configure the STVD

It is recommended to install your toolset (Cosmic or Raisonance) before you install STVD, so that STVD is automatically configured with your environment. If you install STVD first, you must configure it manually through the menu **Tools>Options>Toolset** ([Figure 262](#)).

Figure 262. Options menus

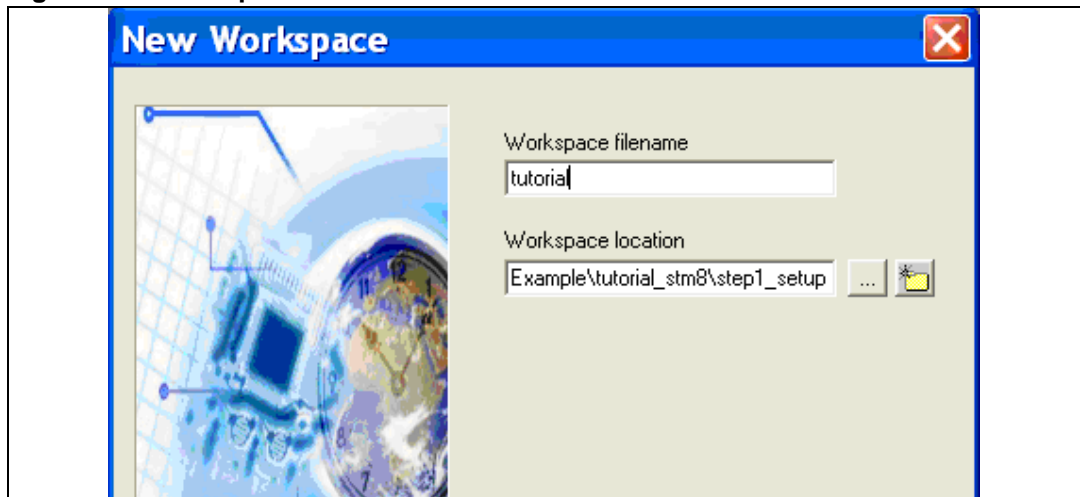
Information is common to all workspace (especially paths, which have priority over workspace-defined paths, we will see an example later using the Raisonance toolset).

12.1.2 Create a workspace

Before you can build or debug an application with STVD, you must create a workspace to contain your project and its source files. A workspace can contain several projects.

1. In the main menu bar select **File>New Workspace**.
2. In the **New Workspace** window, click on the **Create Empty Workspace** icon and then **OK**.

Figure 263. Workspace information



3. Enter a name in the **Workspace filename** field.
4. Enter the pathname where you want to store the workspace in the **Workspace location** field, then click on **OK**.
You may choose `<st_toolset>\stvd\Example\Tutorial_stm8\step1_setup` for this example, but for future work you should use a destination reserved for user data (not Program Files).
5. Save your workspace, select **File>Save Workspace**.

RESULT:

STVD's **Workspace** window contains a workspace icon.

The working directory that you have identified now contains the files, `<workspace name>.stw` and `<workspace name>.wed`.

However, you do not have access to the commands in the **Build** menu, because your workspace does not contain a project and source files, yet.

12.1.3 Create a project

The source files for your application are contained in a project. The **Project Settings** interface gives you access to the options for building your application with a specific toolset.

If you expect to use both Cosmic and Raisonance toolsets in an evaluation phase you must create one project for each, but you may have a single workspace.

1. In the main menu bar, select **Project>Add New Project to Workspace**.
2. In the **New Project** window, click on the **New Project** icon and then **OK**.
3. Enter the name `cosmic` or `raisonance` in the **Project Filename** field, depending on your toolset. In future, if you have only one toolset you can choose a name more pertinent to your project application.
4. Enter the pathname where you want to store the project and the resulting files in the **Project Location** field. The path for the workspace is used by default. Choose the **Raisonance** or **Cosmic** subdirectory (eventually, after having created it) for better disk organization.
5. Next, select **STM8 Cosmic** or **Raisonance** from the **Toolchain** list box.
6. Enter the path for the toolset in the **Toolchain Root** field. The toolchain root is by default the path defined in the STVD option-toolset ([Section 12.1.1](#)). You can define a project-specific toolset path here. It can also be changed later, through the **Project Settings** window. Then click on **OK**.
7. In the **MCU Selection** window, select the MCU that you will build your application for and click on **Select**, then on **OK**. We have selected the STM8L101G2P. Later we will see how you can change the MCU selection in the **Project Settings** window. You can use the **Filter** field to reduce the MCU selection list size.

Figure 264. Project parameters

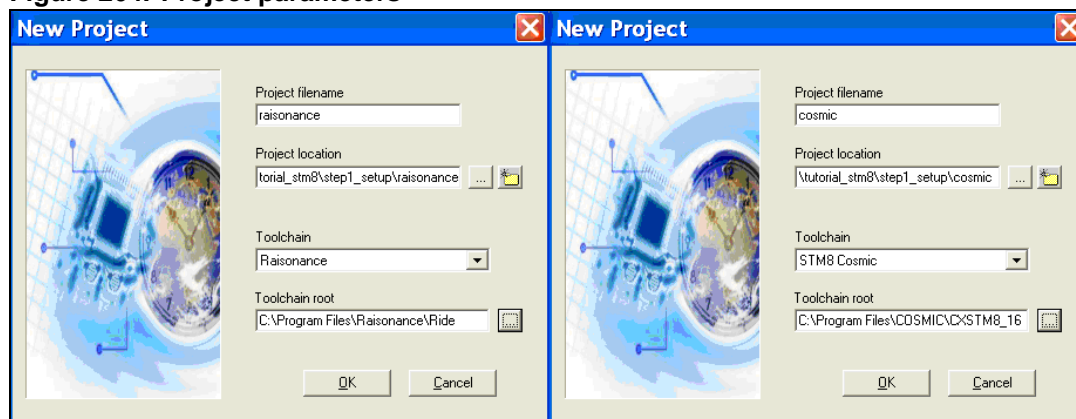
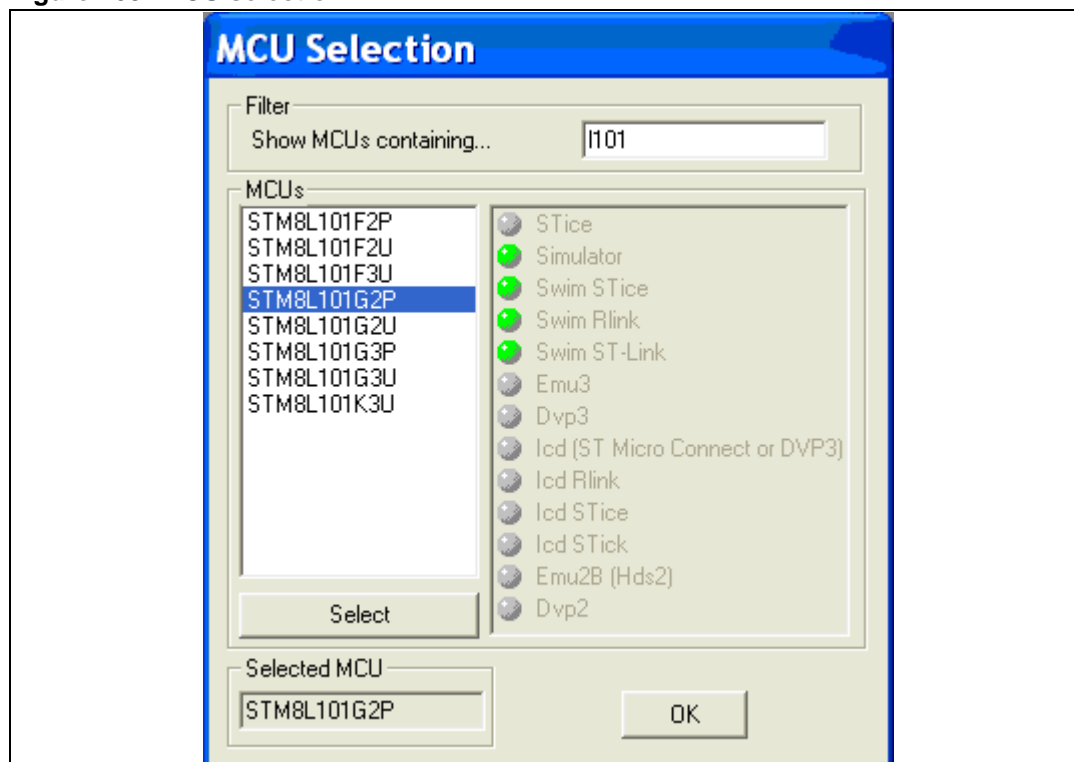


Figure 265. MCU selection

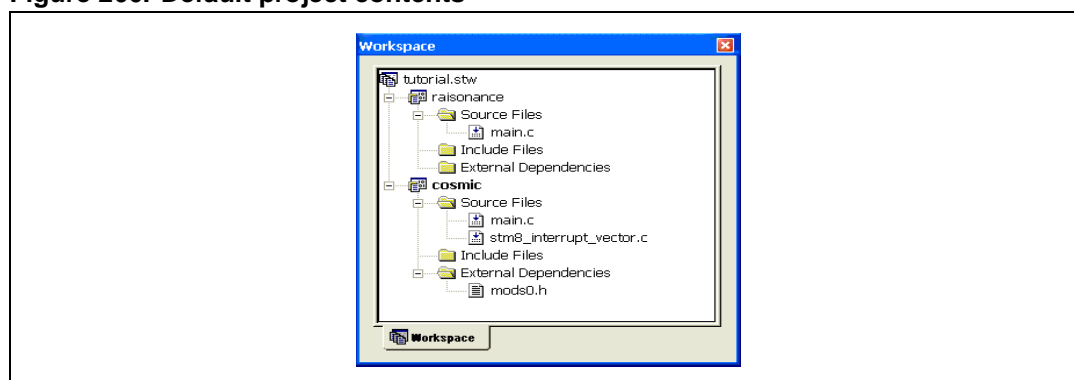


8. Save your workspace; select **File>Save Workspace**.

RESULT:

<toolset>.stp has been added to the workspace in the STVD **Workspace** window. By default the project contains folders called **Source Files**, **Include Files** and **External Dependencies**. These folders are used to organize your project within STVD. They do not exist in the working directory. The file *main.c* is added to the **Source Files** folder by default. With Cosmic the file *mods0.h* is added to the **External Dependencies** folder and */stm8_interrupt_vector* is added to the **Source Files** folder by default

Figure 266. Default project contents



The working directory contains the following:

- three files: *main.c*, and
 - for Cosmic: *cosmic.stp*, *stm8_interrupt_vector.c*
 - for Raisonance: *raisonance.stp*, *R_stm8_interrupt_vector.c*
- and two folders: **Debug** and **Release**. These folders are the default locations for storing your application and any intermediate files depending on whether you build using the Debug or Release configuration.

Note: The workspace view tree does not necessarily reflect the directory tree on the disk.

Now that the workspace contains a project, you will notice that the build commands in the **Build** menu are available (they are no longer gray). STVD automatically adds a minimal *main.c* to allow a project to be built immediately after creation.

12.1.4 Add the source files

To build the application, you must identify the source files for STVD. The source files can be written in either C or Assembler language. For standard installations of STVD these files are located in *C:\ProgramFiles\STMicroelectronics\st_toolset\stvd\Example\tutorial_stm8*.

For this tutorial, the source files required are:

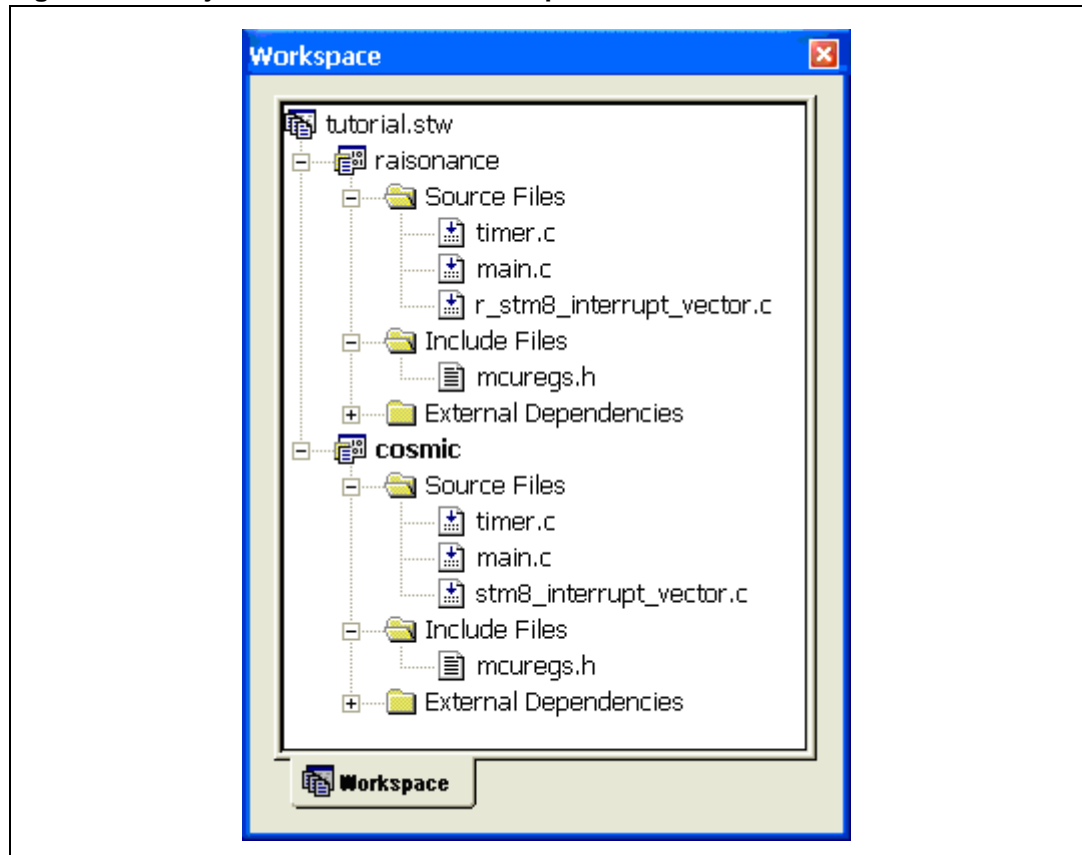
- *src/timer.c*, *src/mcuregs.h*
- *cosmic/stm8_interrupt_vector.c*, *raisonance/R_stm8_interrupt_vector.c*.

You can add files into your STVD project using one of the following methods:

- Select **Project>Insert Files Into Project** from the main menu bar, then browse the file to add. The file will be added at project's level; you may then move it into **Source Files** or **Include Files** folder, or
- Go into the **STVD workspace** window, right click on a folder (**Source Files** or **Include Files**), then select **Add file to folder** and browse the file to add. It will directly be added into the folder, or
- Drag and drop files from an explorer window into the **STVD workspace** window.

You should add all required files into the workspace, in order to reach the state as depicted below:

Figure 267. Project with source file and dependencies

**RESULT:**

Your project workspace now contains the application source code file. Any dependencies specified in the source code are displayed in the **External Dependencies** folder. The list of external dependencies is built by a dedicated call of the compiling toolset, after each modification of the project files list.

The file *main.c*, which was automatically added by STVD, will be removed later.

12.1.5 Create a folder to organize files in the project

The default *main.c* created by STVD does not fit our needs. The `main` function is defined in *timer.c*.

1. Right-click on the project and select **New Folder** from the contextual menu.
2. Enter the name "Obsolete" in the **Name of the New Folder** field.
3. Click on **OK** to create the folder and close the window.
4. Drag and drop *main.c* from **Source Files** folder to **Obsolete** folder.
5. Both *main.c* and *timer.c* are still part of the project, as a result any build attempt at this stage fails because of multiple `main` function definitions.

12.2 Build

In this lesson you will learn how to configure project settings and build the application for debugging. Here, you will:

- [Specify the target STM8](#)
- [Customize the C compiler options](#)
- [Change build settings for a specific file](#)
- [Build the application](#)

12.2.1 Specify the target STM8

The choice of the target microcontroller affects the building, debugging and programming phases of development, including:

- The assignment of application code to specific addresses in the MCU's memory
- The choice of debug instrument
- The choice of programming hardware and protocol

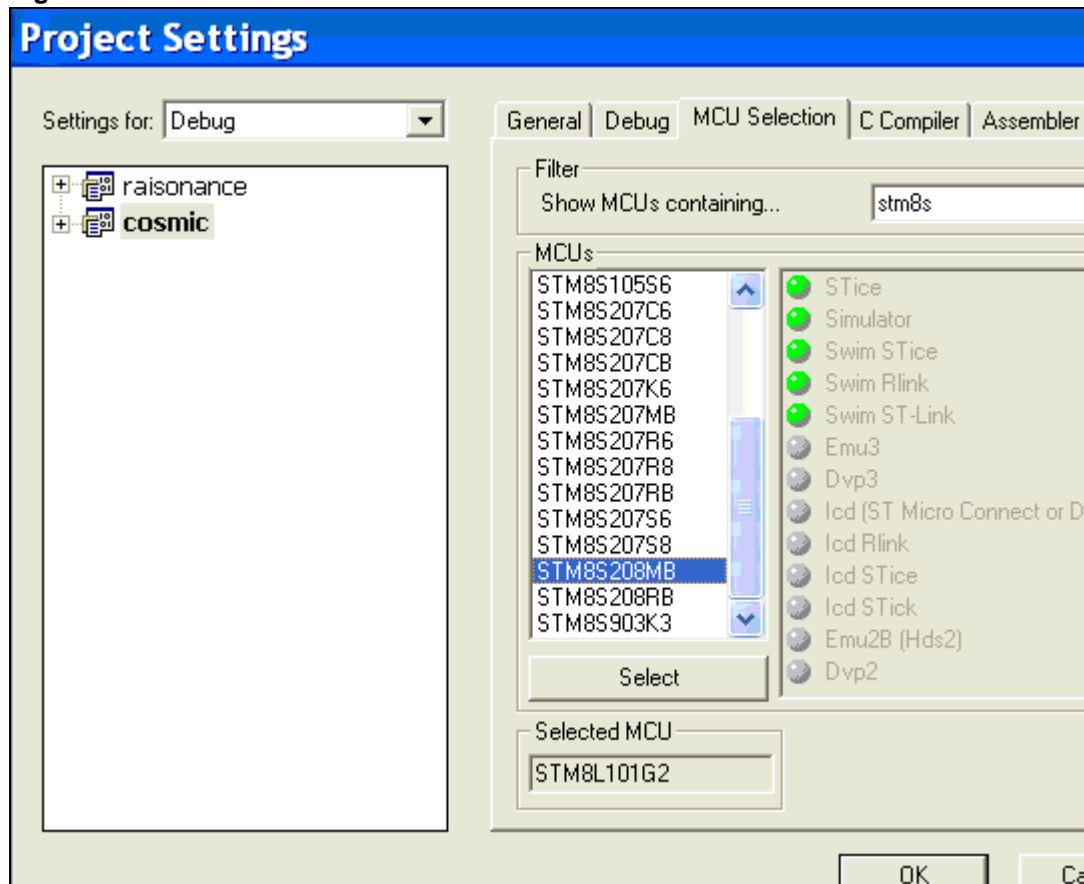
We have already associated the project with a target MCU. In this step change the target MCU selection in the **Project Settings** window, to an MCU that is supported by your debug instrument. Keep in mind that, whenever you change the target device in the Project Settings window, you must build/rebuild your application for the memory mapping and peripheral specification of the new target device.

For this tutorial we will specify the **STM8S208MB** and give an overview of the impact that a change in MCU family would have on the user application.

To specify the MCU:

1. In the main menu bar, select **Project>Settings**.
2. In the **Project Settings** window, click on the **MCU Selection** tab
3. Place the cursor in the **Filter** field and type S20. The list of target microcontrollers is reduced to those that match what you have typed.

Figure 268. Select the MCU



4. Click on the name of the MCU (STM8S208MB), then click on the **Select** button. The name of the MCU now appears in the **Selected MCU** field.
5. Edit the file *mcuregs.h*. As we changed to the STM8S MCU family, activate the definition of the `STM8S_FAMILY` constant and change the definition of the `STM8L_FAMILY` constant into a comment.
6. Changing the MCU part number may introduce differences in some peripheral register addresses. To limit your effort, ST provides `Include` files for all part numbers, defining at least register addresses. These files are available in `<st_toolset>\include`. To adapt the tutorial to your microcontroller you must just modify the file to include in *mcuregs.h*.

RESULT:

Your project is now associated with a specific microcontroller. STVD only allows you to connect to hardware that supports the target MCU. If you change the MCU, you must rebuild the application for the change to be valid.

12.2.2 Customize the C compiler options

In this step we will change the options used by the C compiler, to add the path to STMicroelectronics' include files.

1. In the **Project Settings** window, **C Compiler** tab, select **Preprocessor** in the **Category** list box.

Note that STVD automatically fills in the **Additional include directories** when you add files to your project.

2. Use the button on the right to browse `<st_toolset>/include`.
Note that the change only applies to the current build configuration (**Debug** in this case). When you make these changes do not forget to repeat them for all build configurations. Select **Release** build configuration and repeat the operation.

Caution: **TRAP:** The user defined paths come **after** the toolset paths in the general command line. Before adding any include files, ensure they do not already exist at toolset level. If they do, you may either change the filename, modify the toolset paths as explained in [Section 12.1.1](#) (impacting all projects), or modify the toolset path only for the project (**Project specific toolset path** in **General** tab of **Project Settings**).

Figure 269. C compiler options for Raisonance

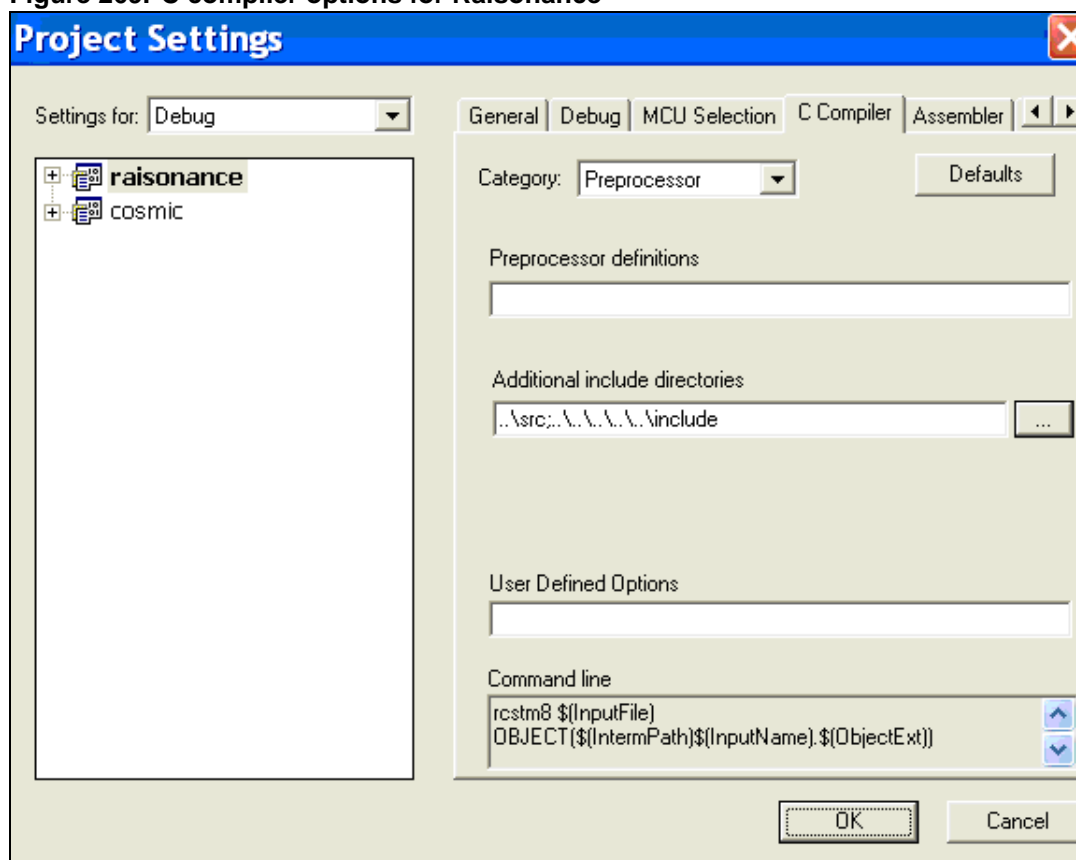
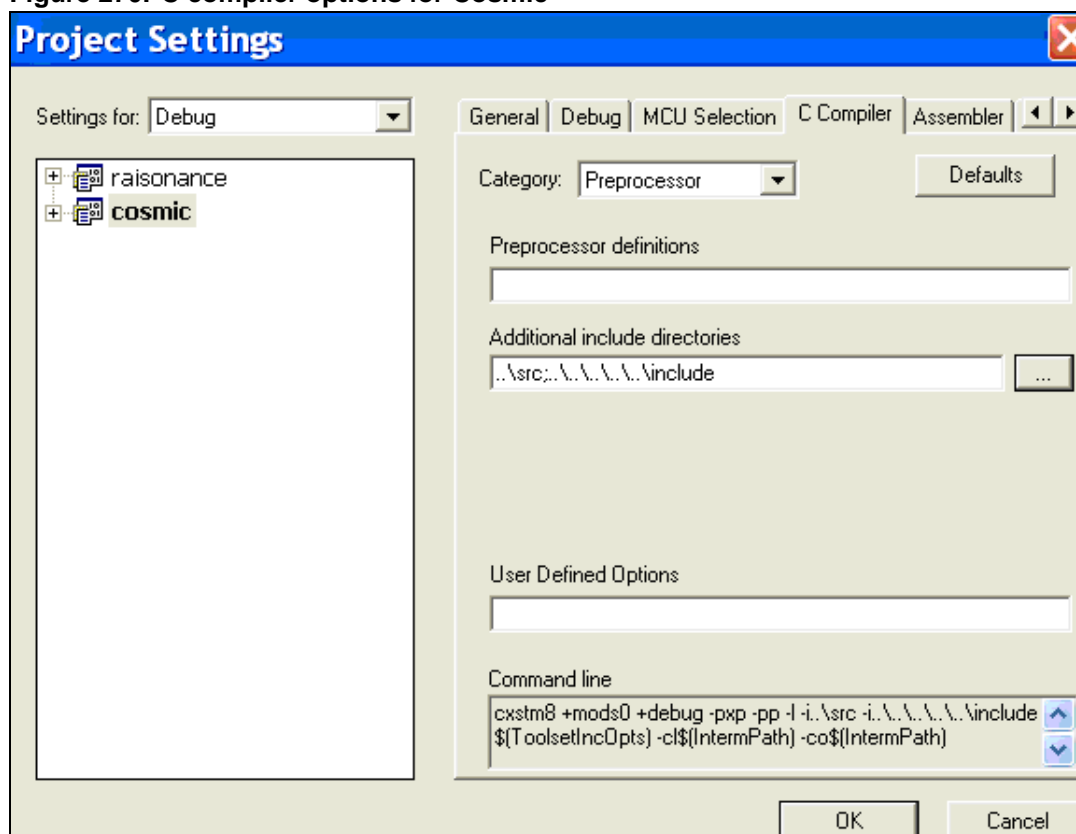


Figure 270. C compiler options for Cosmic

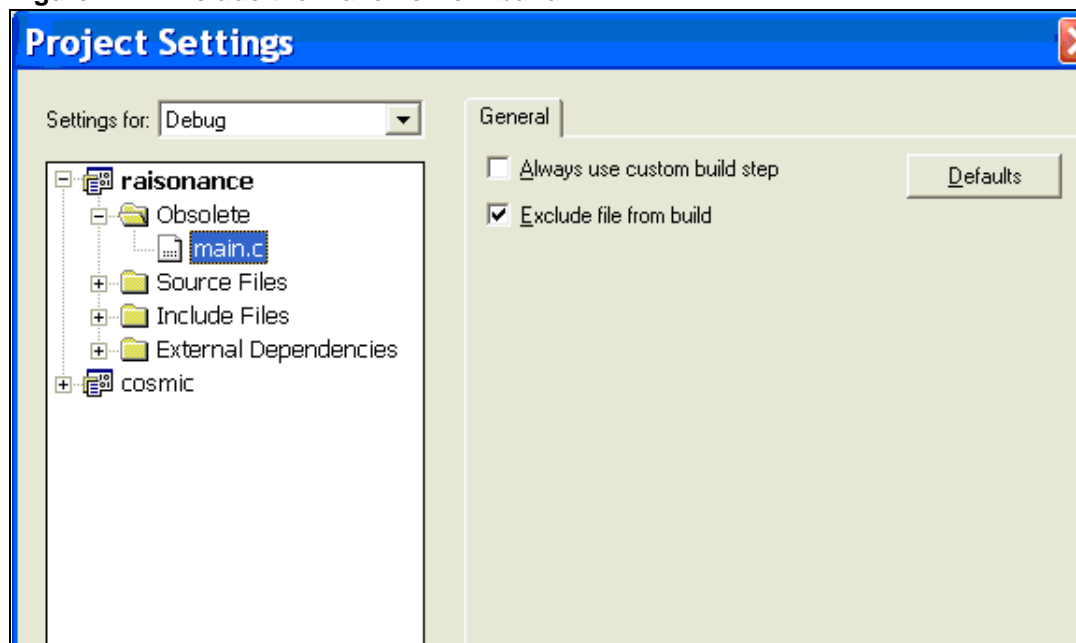


12.2.3 Change build settings for a specific file

In addition to applying settings at the project level, you can also specify settings for a file. As described in an earlier step, we have included the file `main.c` as a reference for understanding STVD's **Project Settings** interface. However, we do not want to include this file in the build. In this step we will exclude this file from the build, by applying a file level setting.

1. Right-click on `main.c`, then select **Settings** from the contextual menu.
2. In the resulting **Project Settings** window, place a checkmark in the box for **Exclude file from build**.

Figure 271. Exclude the makefile from build



Note: At the file level the **Project Settings** window only displays the tabs that pertain to the selected file. For example, the tabs for a selected C source file are **General** and **C compiler**.

3. Click on **OK**.

RESULT:

Project information file (*.stp) is updated.

12.2.4 Build the application

Now, you can see the final result of the settings that we applied in the **Project Settings** window. To build the application:

1. If the **Output** window is not already open, select **View>Output Window** from the main menu bar.
2. In the **Output** window click on the **Build** tab
3. To generate your application, select **Build>Build** from the main menu bar

RESULT:

The commands invoked, error messages and warnings that occurred during the build are displayed in the **Build** tab of the **Output** window.

The **C compiler** is invoked for each of the source files in the project. It is not invoked for the file that we excluded from the build.

The **Linker** is invoked once, to link the object files.

All generated files are located in the projects **Debug** subdirectory.

12.3 Start debugging

In this lesson you will learn how to start a debugging session and run the application. Here, you will:

- [Start the debugging session](#)
- [Run and stop the application](#)
- [Step through the application](#)

Note: If you are already familiar with these basic features, feel free to go to the next lesson, [Instruction breakpoints](#).

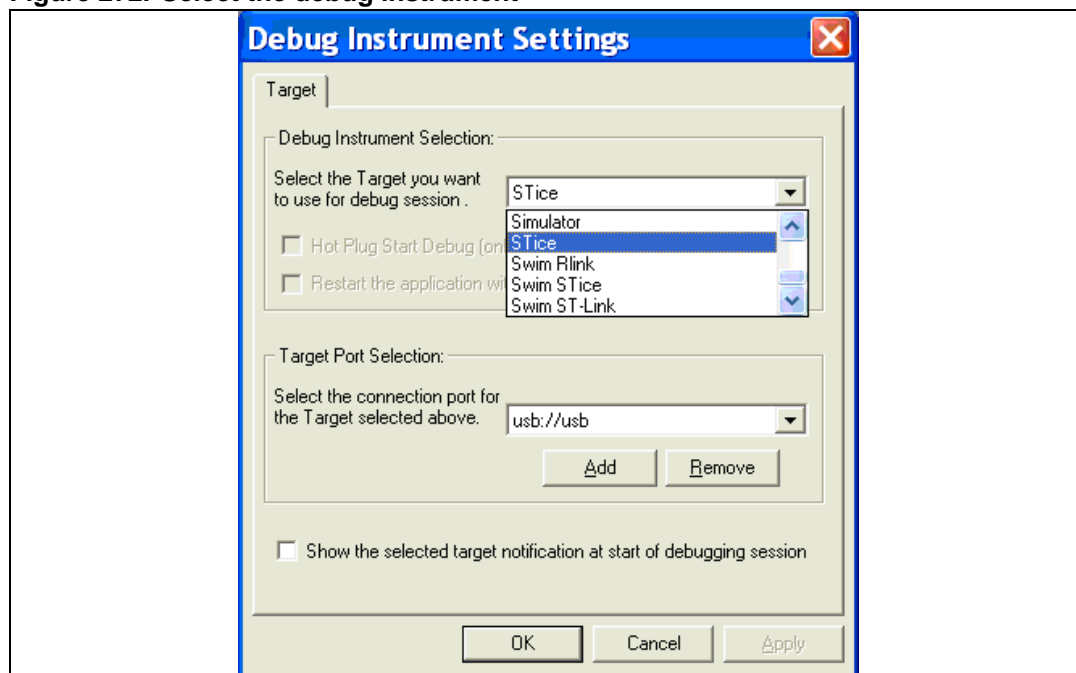
12.3.1 Start the debugging session

Select the debug Instrument

You must select a debug instrument before you can start debugging. For this part of the tutorial we use the STice (you should select your own tool).

1. From the main menu bar, select **Debug Instrument>Target Settings**.
2. Select **STice** from the **Debug Instrument Selection** list box.
3. Click on **OK**.

Figure 272. Select the debug instrument



Start the debugging session

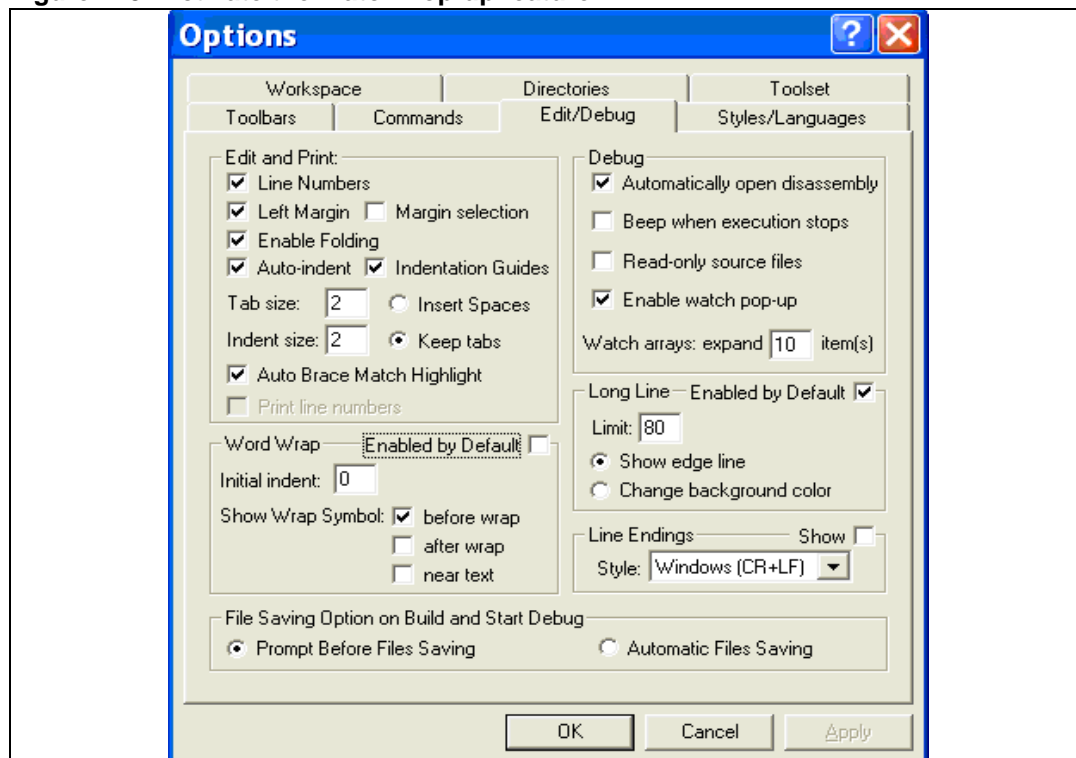
1. From the main menu bar, select **Debug>Start Debugging**.
2. Confirm the debug instrument by clicking **OK** in the **Target Selection** pop-up window.
3. STVD restores the debug context. The MCU configuration window may open at this point, simply click **OK** to close it. We will learn more about this feature later.

Activate a feature

You can enable and disable some debug options. For this session we want to use the **Watch Pop-up** feature that allows us to see the current value of a variable in our source file.

1. From the main menu bar, select **Tools>Options**.
2. Click on the **Edit/Debug** tab in the **Options** window.
3. Place a checkmark in the box next to **Enable watch pop-up**, then click on **OK**.

Figure 273. Activate the Watch Pop-up feature



RESULT:

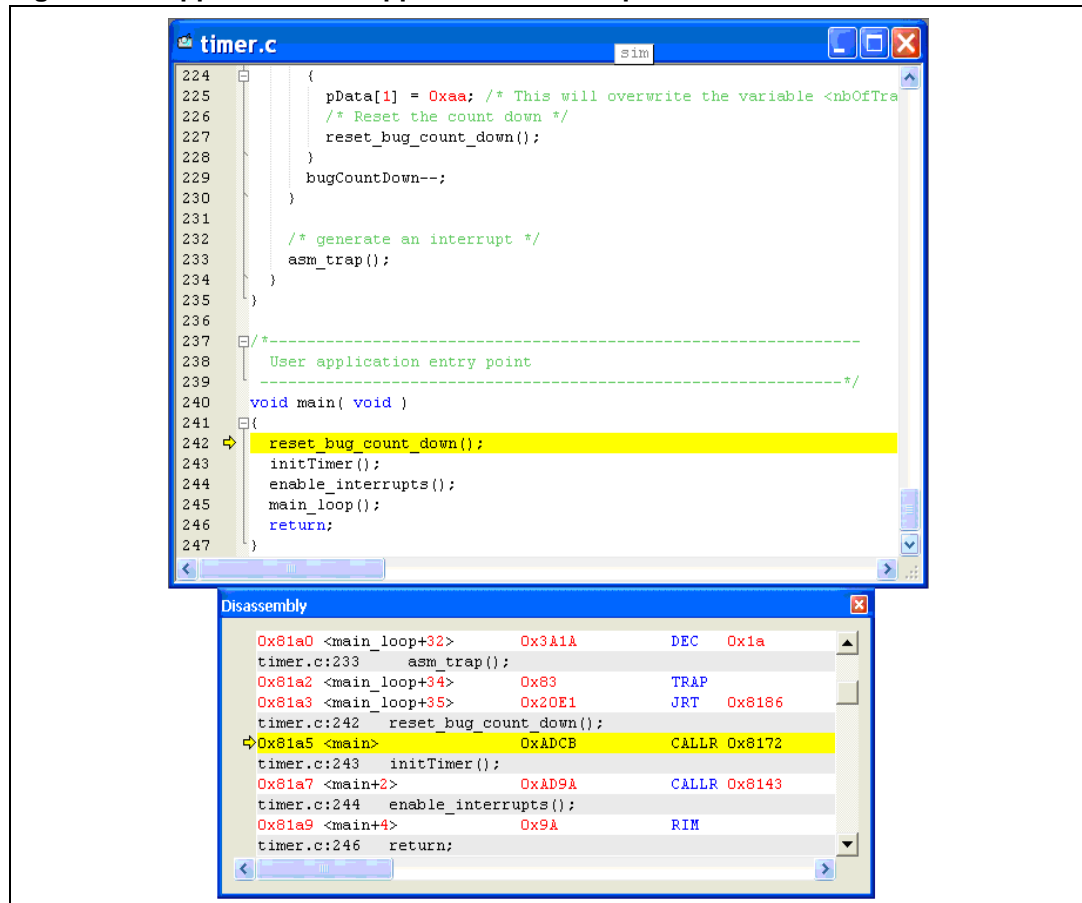
The debug session has started. Double-click on `timer.c` to open the source file in an **Editor** window. Scroll to **line 75**, then hover your cursor over the variable `currentState`. A pop-up will appear with the variable's current value.

12.3.2 Run and stop the application

You can run and stop the application using the commands in the **Debug** menu. You can use **Debug>Run** and **Debug>Stop Program** to start and stop the execution of your application. However, to stop the application there are many more accurate ways of controlling where and when the application stops. For Example:

1. If it is not already open, double-click on `timer.c` to open the source file in an **Editor** window.
2. Scroll to **line 242** of the source code (`timer.c`) and click to place your cursor at the beginning of this line. You may use the **Go to** function of the contextual menu in the editor (right click) in order to navigate easily.
3. Select **Debug>Run to Cursor**. The application executes up to line **242** of the code. The **Program Counter** (yellow arrow and highlighting) shows the next instruction to be executed in the source code and the corresponding position in the disassembled code.

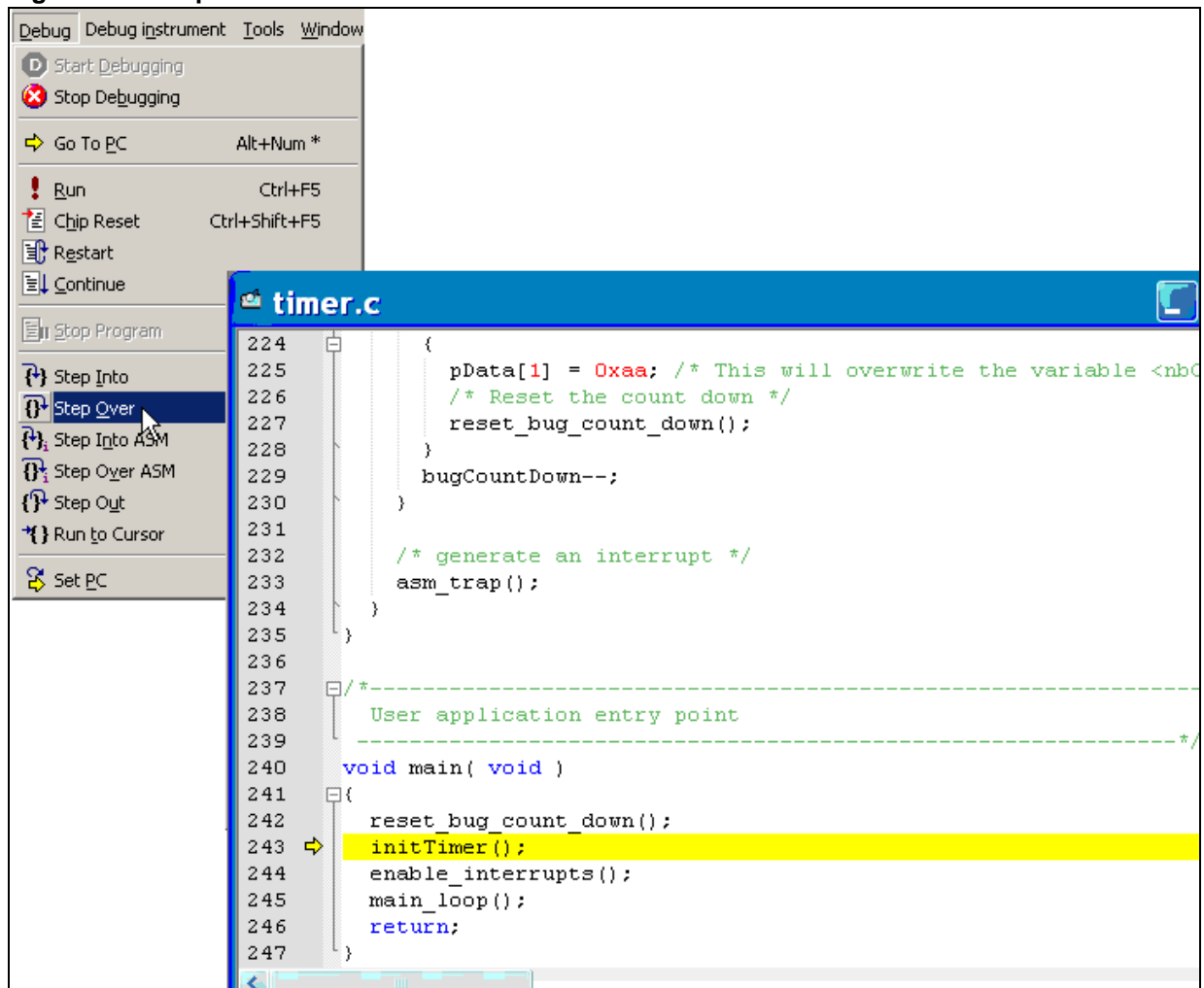
Figure 274. Application is stopped at the cursor position



12.3.3 Step through the application

Stepping instructions allow you to execute the application one instruction at a time and to step into or over functions as required. For example, we are currently on **line 242** of the source code.

Figure 275. Step over line 242 to line 243



1. Select **Debug>Step Over** to step over the `reset_bug_count_down` function. To view the current line of assembly code to be executed, see the Disassembly window
2. Select **Debug>Step Into** to step into `main_loop`.
3. Select **Debug>Step Out** to step out of `init_timer` (back to line 244).

12.4 Instruction breakpoints

An instruction breakpoint stops program execution when a specific instruction is reached. If a condition and/or a counter is set on an instruction breakpoint, program execution is stopped to evaluate the condition/counter.

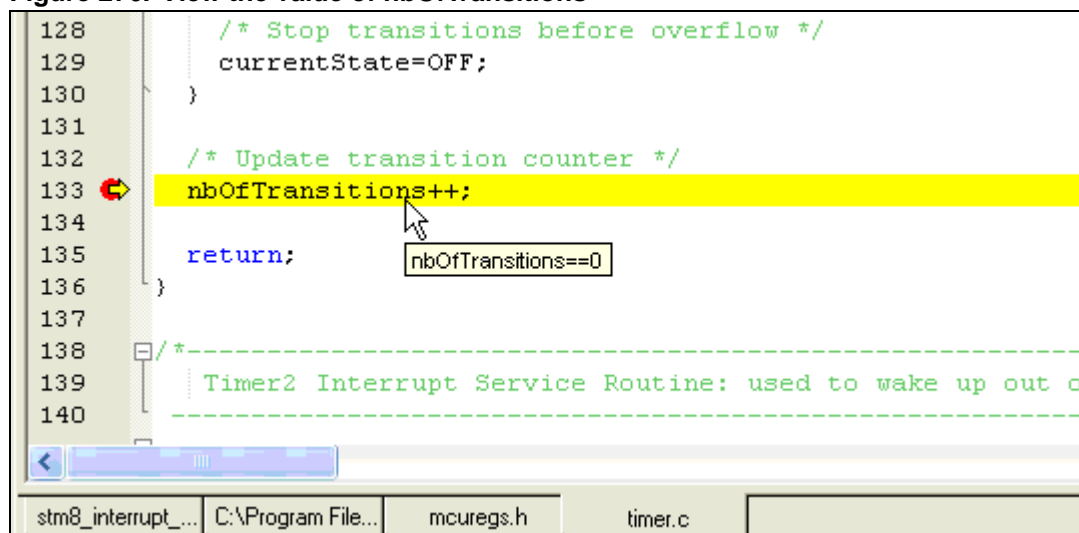
In this lesson you will learn how to set instruction breakpoints with counters and conditions. Here, you will:

- [Set an instruction breakpoint](#)
- [Set a counter on an instruction breakpoint](#)
- [Set a condition on an instruction breakpoint](#)

12.4.1 Set an instruction breakpoint

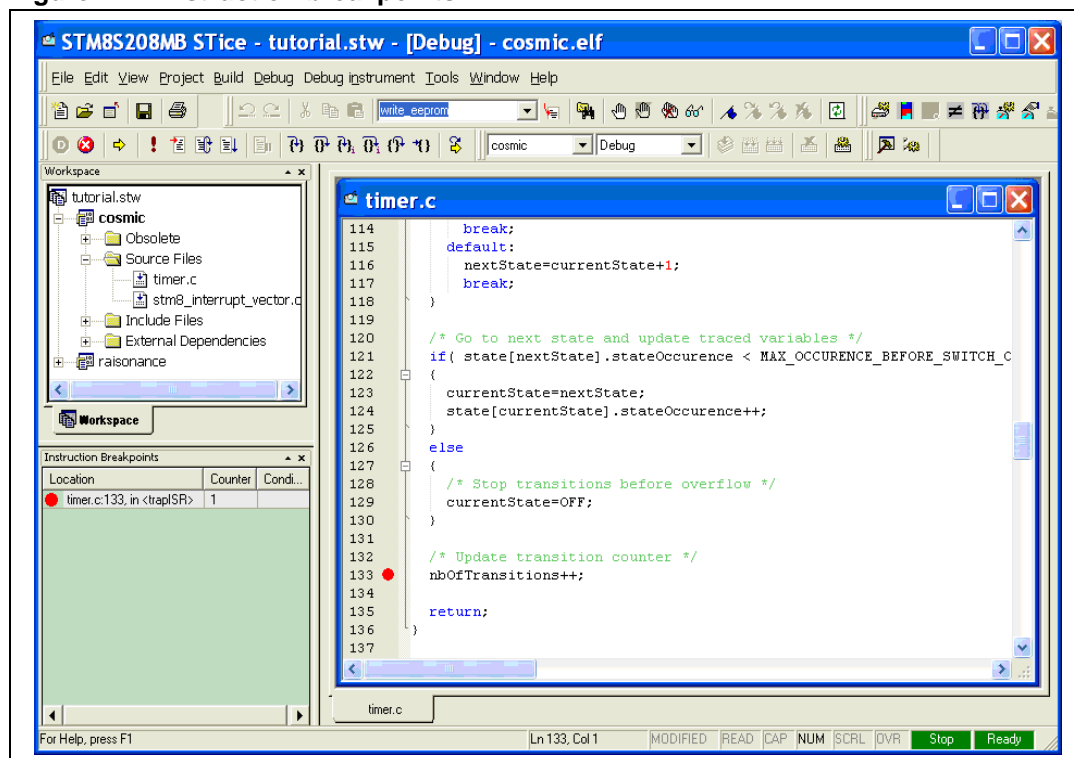
1. If it is not already open, double-click on `timer.c` to open the source file in an Editor window.
2. Select **View>Instruction Breakpoints** to open the *Instruction breakpoints* window.
3. In the Editor window, place the cursor on the `nbOfTransitions` variable (line 133) to view its current value. A pop-up box is displayed with the value.

Figure 276. View the value of `nbOfTransitions`



4. Click in the margin of the editor window next to **line 133** of the source code. The instruction breakpoint is indicated by a red dot in the margin. It is also displayed in the **Instruction Breakpoints** window.

Figure 277. Instruction breakpoints

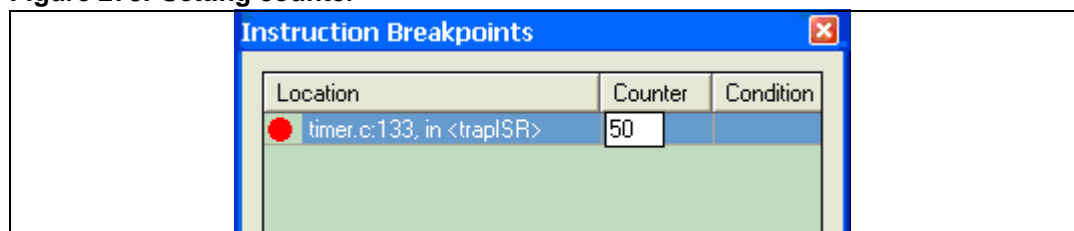


5. Select **Debug>Continue** to run the application from line 107 to the breakpoint.
6. Check the `nbOfTransitions` value in the pop-up. The variable `nbOfTransitions` = 0.

12.4.2 Set a counter on an instruction breakpoint

1. In the **Instruction Breakpoints** window double-click on the **Counter** field and enter 50.

Figure 278. Setting counter

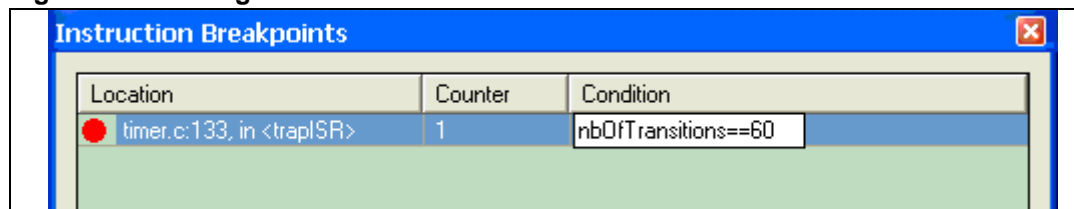


2. Click on **Debug>Continue**.
3. When the application stops, check the new value of the `nbOfTransitions` variable. The program has stopped at the breakpoint after 50 passes (1...50).

12.4.3 Set a condition on an instruction breakpoint

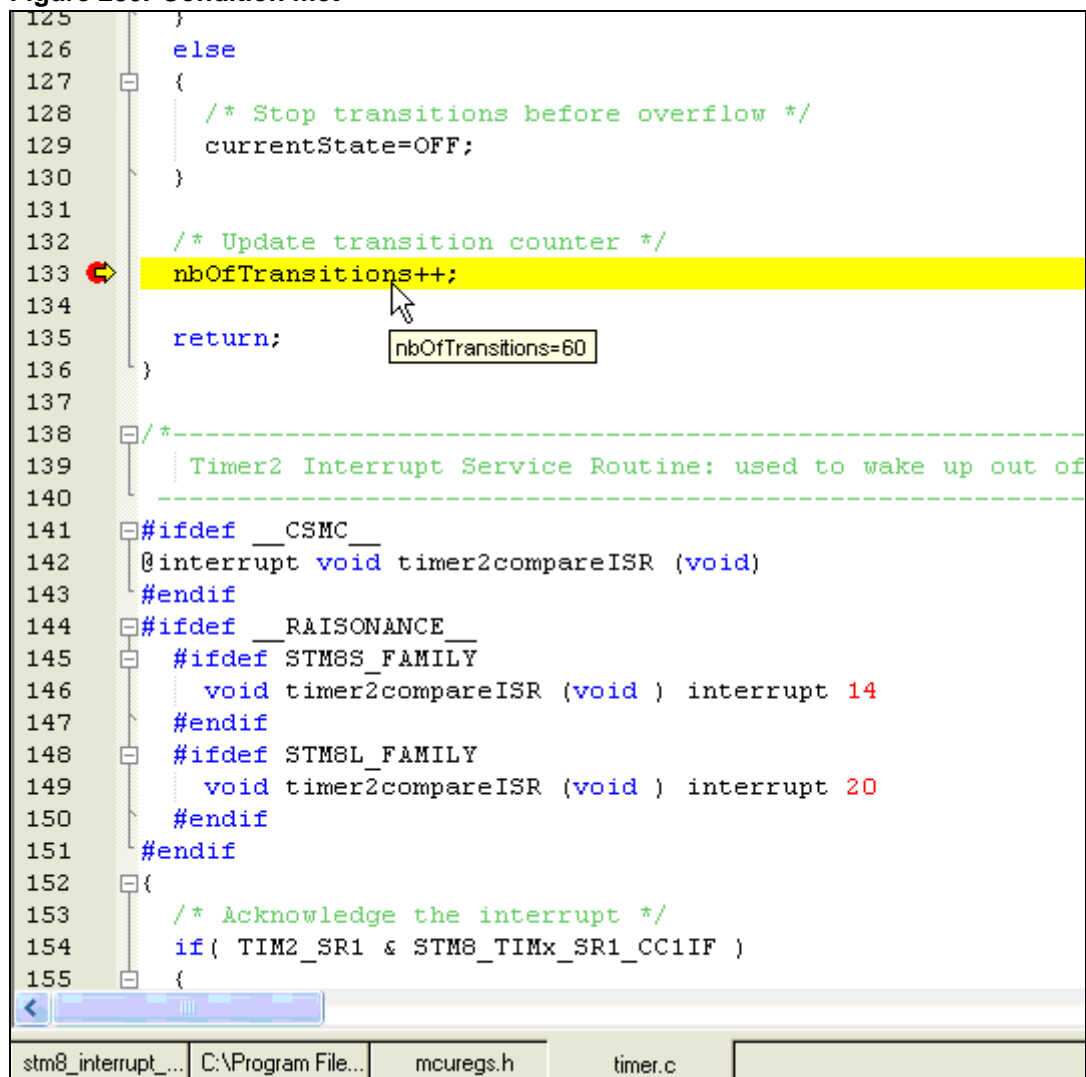
1. In the **Instruction Breakpoints** window double-click on the **Condition** field and enter `nbOfTransitions == 60` at the flashing cursor.

Figure 279. Setting condition



2. Click on **Debug>Continue** to continue execution of the application.
3. In the Editor window place the cursor over the `nbOfTransitions` variable. Its value is now 60. The program stopped after the condition was found to be true, that is `nbOfTransitions == 60`.

Figure 280. Condition met



12.5 View execution

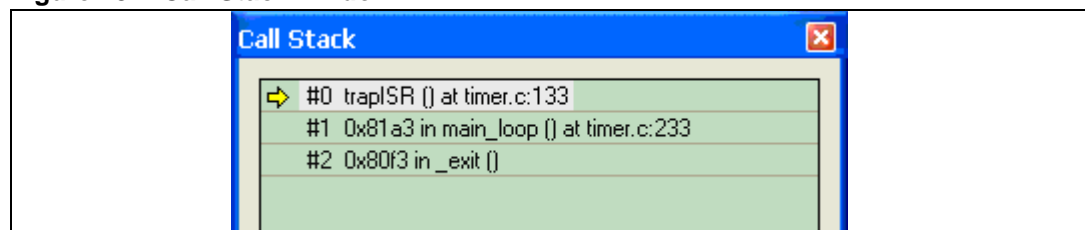
In this lesson you will learn the STVD features that allow you to view the application instructions that have been executed by the debugger. Here, you will:

- [View calls to the stack](#)
- [View and change local variables](#)
- [View variables using the Watch, QuickWatch and Memory windows](#)

12.5.1 View calls to the stack

1. Select **View>Call Stack** to display the [Call stack window](#). This window shows the function calls stored in the microcontroller call stack from the most recent function calls, traced back to the earliest call.

Figure 281. Call Stack window



2. Double-click on entry **#1**. A call stack frame indicator ► appears in the **Disassembly** window to indicate where the call was made in the disassembled source code. You can return to any previous stack call in this manner.
3. Select **View>Local Variables** to view the [Local variables window](#), this shows the values of the variables as they were at the moment the selected function call was performed.
4. In the **Call Stack** window, double-click on entry **#0** to return to the current Program Counter position.
5. Before going on, remove the breakpoint at **line 133** by selecting **Edit>Remove All Breakpoints**.

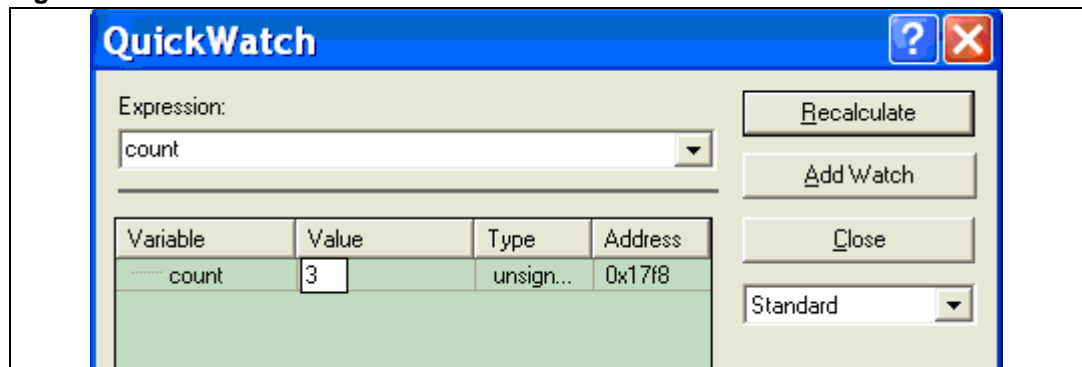
12.5.2 View and change local variables

1. Select **Debug>Step Out** to step out of trapISR and into main_loop.
2. Set an instruction breakpoint on the call to waiting_loop function (line 218).
3. Select **Debug>Continue**.
4. Select **Debug>Step Into** to step into the waiting_loop function.
5. Place the cursor on the count variable (line 186), select **Edit>QuickWatch** to open the [QuickWatch window](#).

The **QuickWatch** window is opened in front of the active Editor window. The main display area displays the count variable and its current values.

6. Double-click in the **Value** field of the count variable, change its value to 3. and click on **Close**.

Figure 282. QuickWatch window



7. In the Editor window place the cursor over the `count` variable, a pop-up should appear indicating that its value is 3.

12.5.3 View variables using the Watch, QuickWatch and Memory windows

In this procedure you will learn how to use the [Watch window](#), [QuickWatch window](#) and [Memory window](#) to inspect variables.

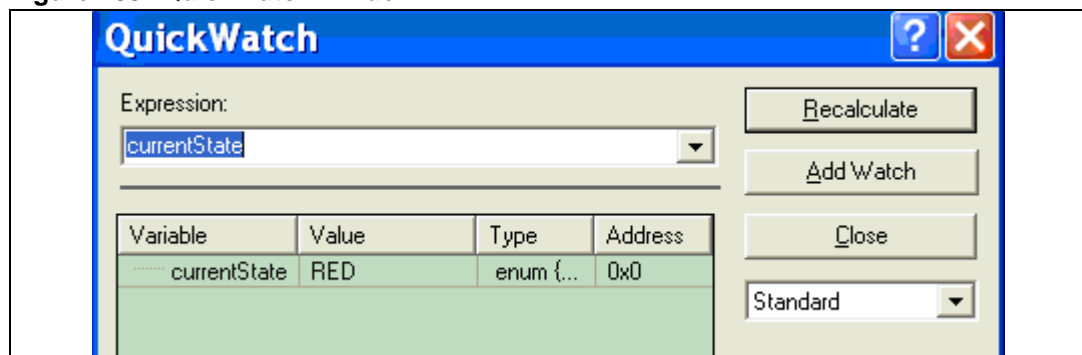
The [Watch window](#) displays the current value of selected program variables when the executable is stopped.

The [QuickWatch window](#) provides rapid contextual access to the most commonly used Watch functions. Values are easily transferred from the QuickWatch window to the Watch window.

The [Memory window](#) provides views of selected areas of memory contents.

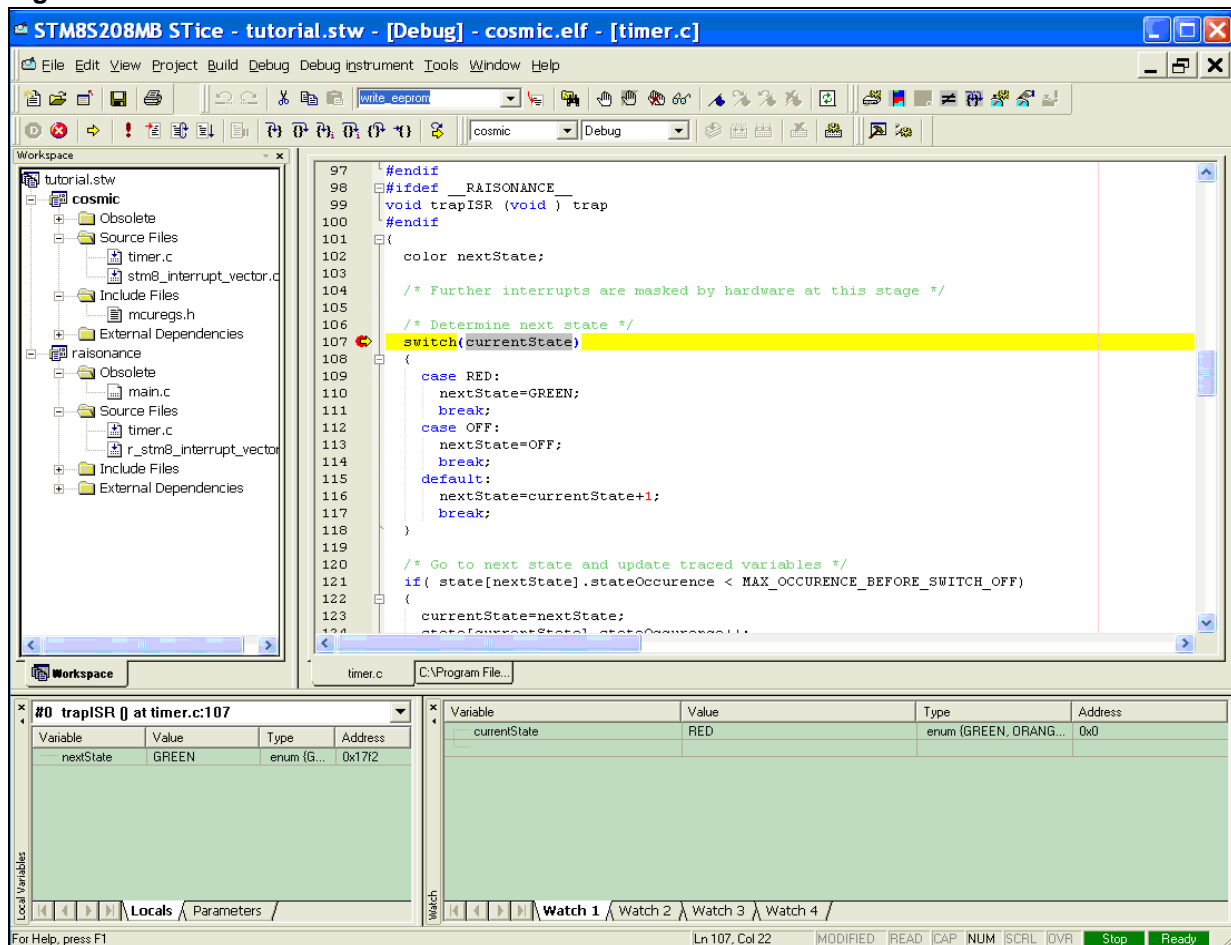
1. Set a breakpoint at the beginning of the `trapISR` function (line 107).
2. Click on **Debug>Continue** to continue execution of the application up to this breakpoint.
3. Click to place the cursor on the `currentState` variable, select **Edit>QuickWatch** to open the QuickWatch window.

Figure 283. QuickWatch window



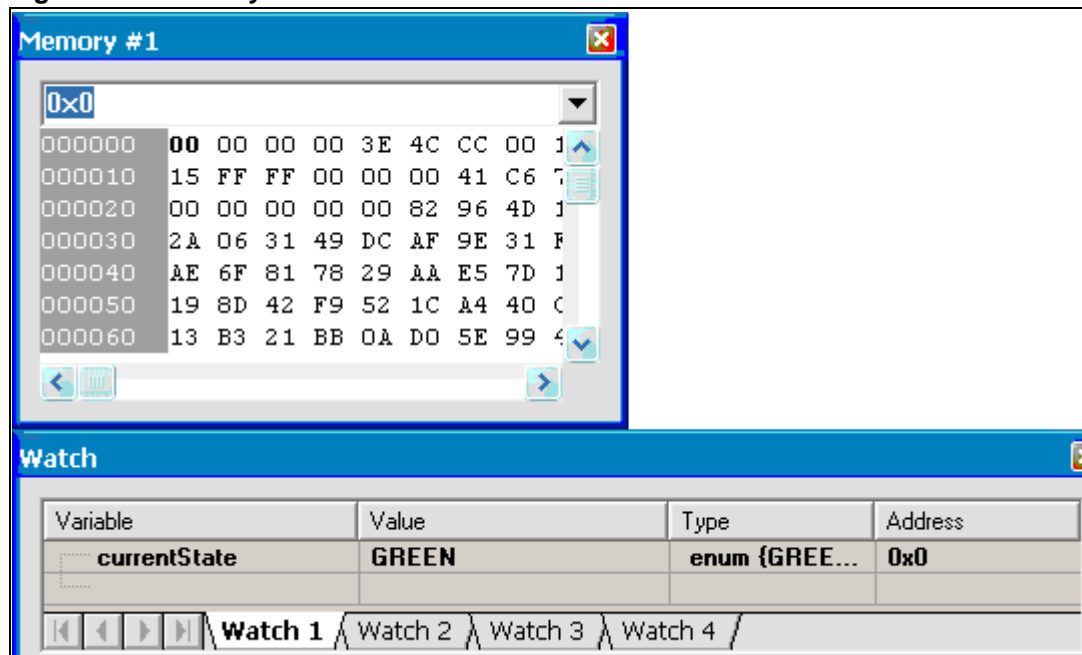
4. In the **QuickWatch** window select **Standard** from the drop down menu and click on the **Add Watch** button.
5. The **QuickWatch** window closes and the **Watch** window appears with `currentState` displayed.

Figure 284. Watch window



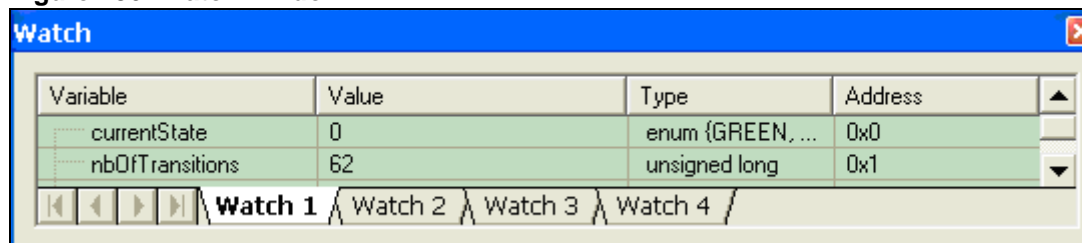
6. In the Editor window, click to place the cursor on line 123.
7. Click on **Debug>Run To Cursor**.
8. Select **View>Memory** to display the Memory window.
9. In the Memory window, set the start address to the address of the currentState variable (this address is displayed in the Watch window as shown above). Note that the display format is set to double-word — this is controlled by right-clicking the mouse in the Memory window and selecting **Format>DWord** from the contextual menu that appears.
10. Select **Debug>Step Over** to step over the current line of code.
11. In the **Watch** window and **Memory** window you can see the values that have just changed, they are displayed in **bold** type.

Figure 285. Memory and Watch windows



12. In the **Watch** window right-click to display the **Watch** window contextual menu, select **Display All>Decimal** to see the decimal value of `currentState`, this should correspond to the value displayed in the Memory window.
13. Select **Debug>Step Over** to step to line 133.
14. In the Editor window highlight `nbOfTransitions` with the cursor and use the left mouse button to drag and drop it to the **Watch** window. It now appears in the **Watch** window as shown below:

Figure 286. Watch window



15. Remove all breakpoints by right clicking in the *Instruction breakpoints* window and selecting **Remove All Breakpoints**.
16. Click on **Debug>Continue** to continue execution of the application. The Status bar should indicate the debugger status as **Running**.
17. After a period of 5 to 10 seconds stop the program by selecting **Debug>Stop Program**. The status bar should indicate the debugger status as **Stopped**.
18. Use the **Watch** window to view the values of the `nbOfTransitions` and `currentState` variables. The value of `nbOfTransitions` should have increased and the value of `currentState` may have changed.

12.6 Perform memory mapping

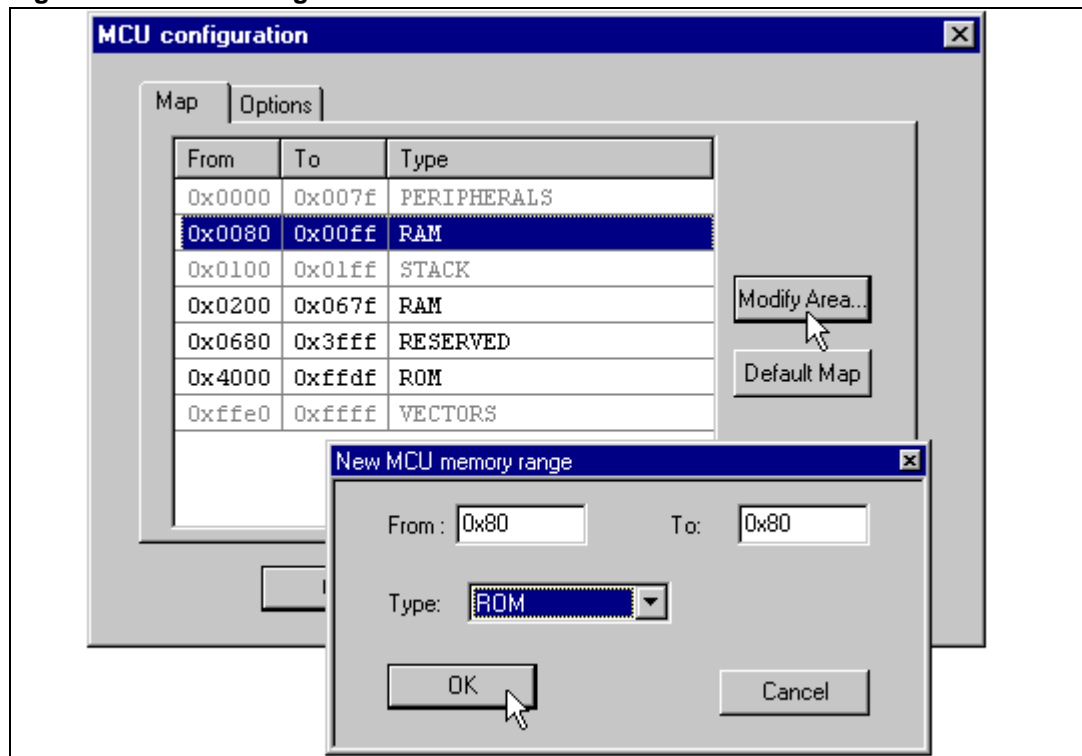
Memory mapping is the process of defining which areas of memory can be accessed by the application and the type of access permitted, (for example, read only or read/write). This allows verification of the application's use of memory, ensuring that only memory locations which will exist in the target controller are addressed.

In this lesson you will learn how modify the default memory mapping. Specifically, we are going to create a new range of ROM within an existing RAM memory range.

To do this:

1. Select **Debug Instrument>MCU Configuration**.
2. Select the range of RAM that contains the address 0x00.
3. Click on the **Modify Area** button.
4. In the **New MCU Memory Range** dialog box type the address 0x00 in both the **From** and **To** fields, and change the memory type to **ROM** for the single address range.
5. Click on **OK** in the **New MCU Memory Range** dialog box.
6. Click on **OK** in the **MCU Configuration** window.
7. From the main menu select **Debug>Run**.
8. A memory access error is displayed, because the address 0x00 corresponds to the address of the `currentState` variable in our tutorial program. When we ran our program, the error occurred when the program tried to write the new value of `currentState` to its address 0x0000, which we mapped as a read only section of memory.

Figure 287. MCU configuration



9. Click on **OK** in the error message box. In the **MCU Configuration** window reset the memory area selected above to type: **RAM**.

12.7 Advanced emulator features for EMU3 (ST7) and STice (STM8)

In this lesson you will learn about using some of the STice emulators advanced features while debugging the sample application that you build in previous lessons. The advanced features covered here include how to:

- [View program execution history](#)
- [Use read/write on-the-fly](#)
- [Set an advanced breakpoint](#)
- [Run a performance analysis](#)

For this lesson use the `cosmic2.stw` that is provided upon installation of STVD. For standard installations of STVD, this file is saved at the directory location:

```
C:\Program Files\STMicroelectronics\st_toolset\stvd\examples\tutorial_cosmic\
```

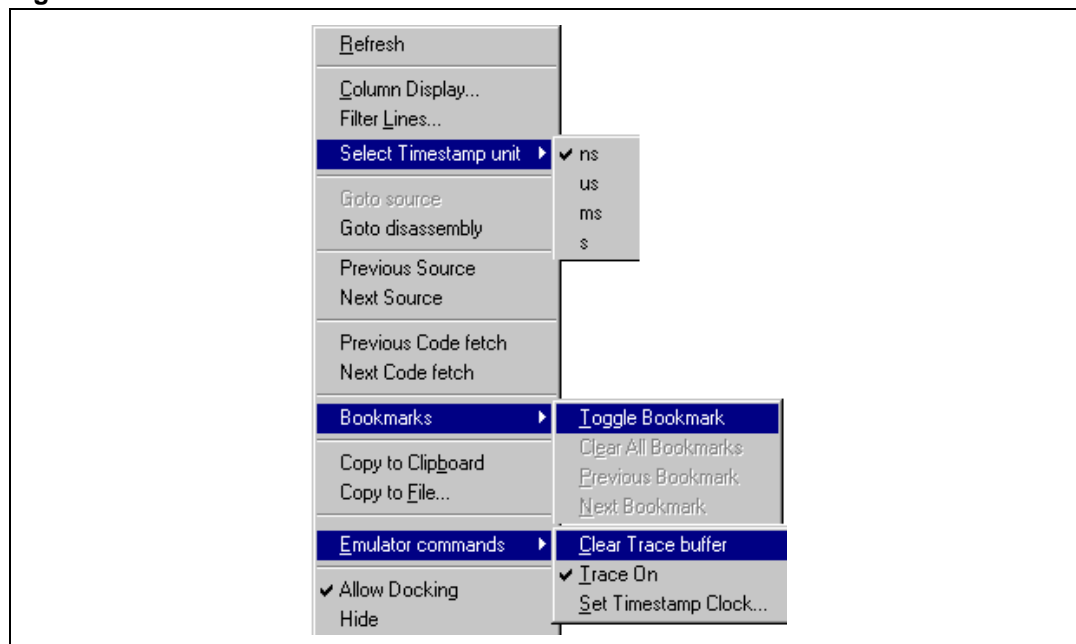
If you are not already in the debug context, select **Debug>Start Debugging** from the main menu. The application file is loaded into the emulator and the previously saved settings are restored.

12.7.1 View program execution history

The Trace window is used to view the contents of the trace buffer. The Trace Buffer is a physical memory module in the emulator that records the hardware cycles that occur when a program is executed.

1. Select **View>Trace** to display the **Trace** window.
2. Right-click in the **Trace** window to open the contextual menu.

Figure 288. Trace contextual menu

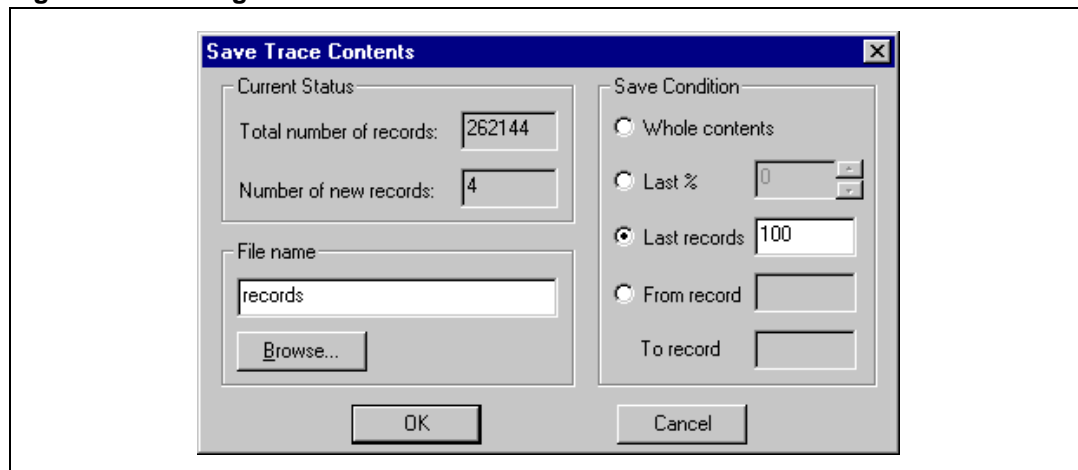


3. Select **Emulator commands>Clear Trace buffer**.
4. Click on **Debug>Step Over** a few times in order to generate trace record.

Note: If you have executed a Chip Reset (debug>Chip Reset) prior to Step Over, STVD may not be able to terminate the execution of the step command. To regain control, select Debug>Stop Program from the main menu bar.

5. From the Trace Contextual menu select **Column Display....**
6. Click on **Uncheck all** and then check the following: **Record, Address, Data, Symbolic Disassembly**. Click on **OK** to view the results.
7. From the Trace Contextual menu select **Copy to File....**
8. In the Save Trace Contents window click on the **Last records** button and enter **100** as the number of records to save. Enter a filename in the **File name** field. Click on **OK** to save the records.

Figure 289. Saving trace contents



The last 100 records are saved to file where they can be read using a text editor.

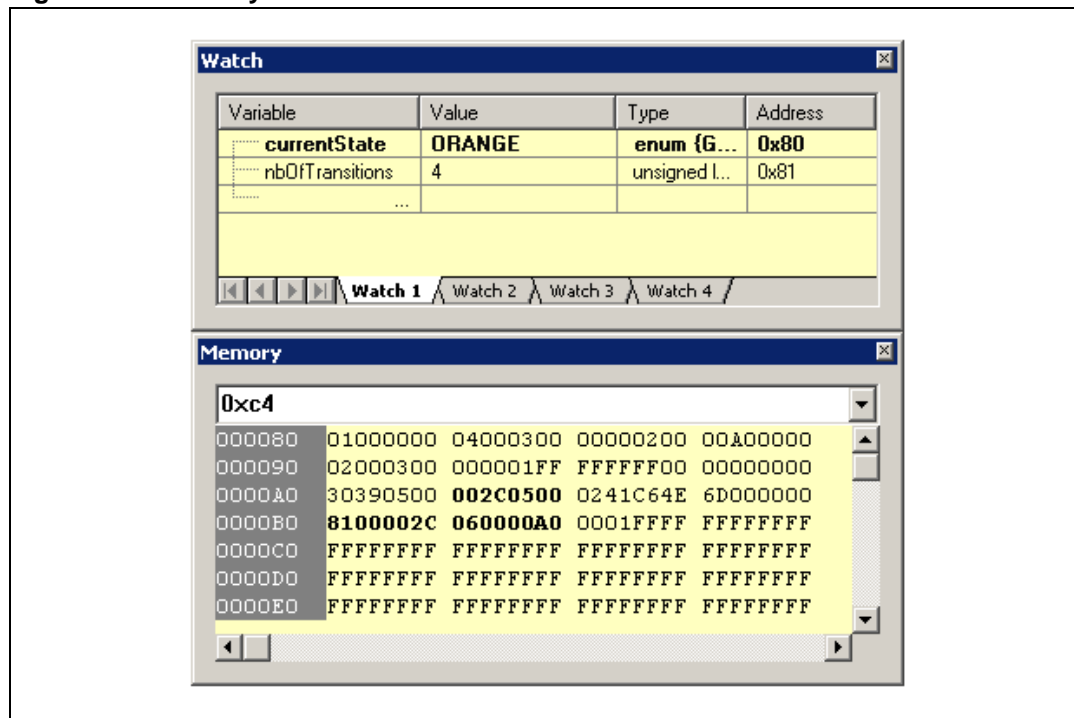
12.7.2 Use read/write on-the-fly

When the *Read/write on the fly* option is selected, the *Watch window* and *Memory window* are continuously refreshed at one-second intervals and any changing values are displayed in **bold** type.

This procedure describes how to enable the read/write on the fly option and how it may be used to watch variable values being read or written to while your program is running on the emulator. Here, we also show how our program has a bug which we will use later to demonstrate advanced breakpoints.

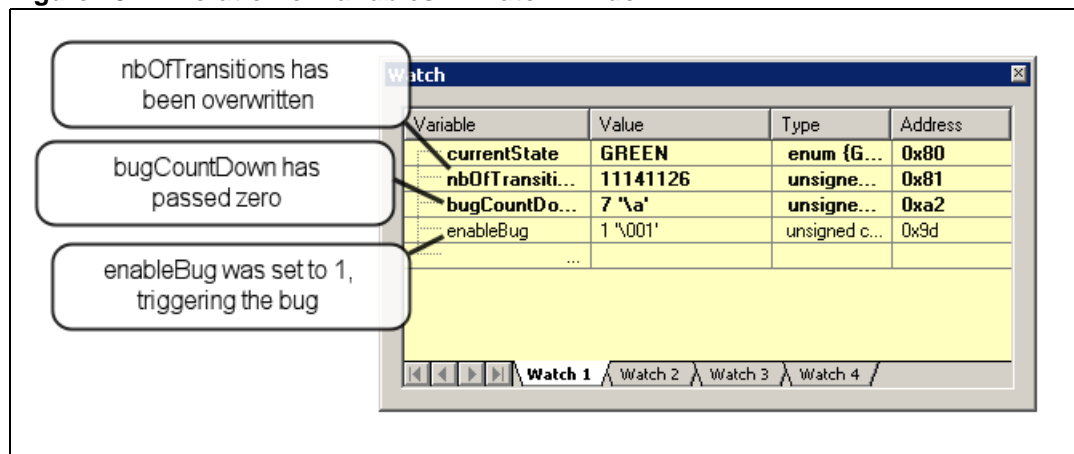
1. Use the **View** menu to open the *Watch window* and *Memory window* if they are not already displayed.
2. In both the Watch and Memory windows, right-click to open the contextual menu and enable the **Read/Write on the Fly** option.
3. When the **Read/Write on the Fly** option is enabled, the window's background changes to yellow.
4. Select **Debug>Run** from the main menu.
5. You can now see memory entries in the Watch window and variable values in the Memory windows being updated periodically (values are shown in bold when updated).

Figure 290. Memory and Watch windows



6. In the **Editor** window, locate the `bugCountDown` variable, select it and drag and drop it to the **Watch** window.
7. In the **Editor** window, locate the `enableBug` variable, select it and drag and drop it to the **Watch** window.
8. In the **Watch** window, double-click in the **Value** field of the `enableBug` variable and enter 1 at the flashing cursor. The bug is now enabled and the bug countdown begins.
9. In the **Watch** window watch the value of `bugCountDown` decrease, after it has reached 0 the variable `nbOfTransitions` is overwritten—we have a bug! This is described in more detail in the next procedure.

Figure 291. Evolution of variables in Watch window



10. Stop the program by selecting **Debug>Stop Program**.

12.7.3 Set an advanced breakpoint

The **Advanced Breakpoints** window gives access to a powerful function based on a programmable multi-level logic Sequencer. Advanced breakpoints allow you to set simple or multi-level breakpoint conditions, in addition to controlling trace recording and trigger output.

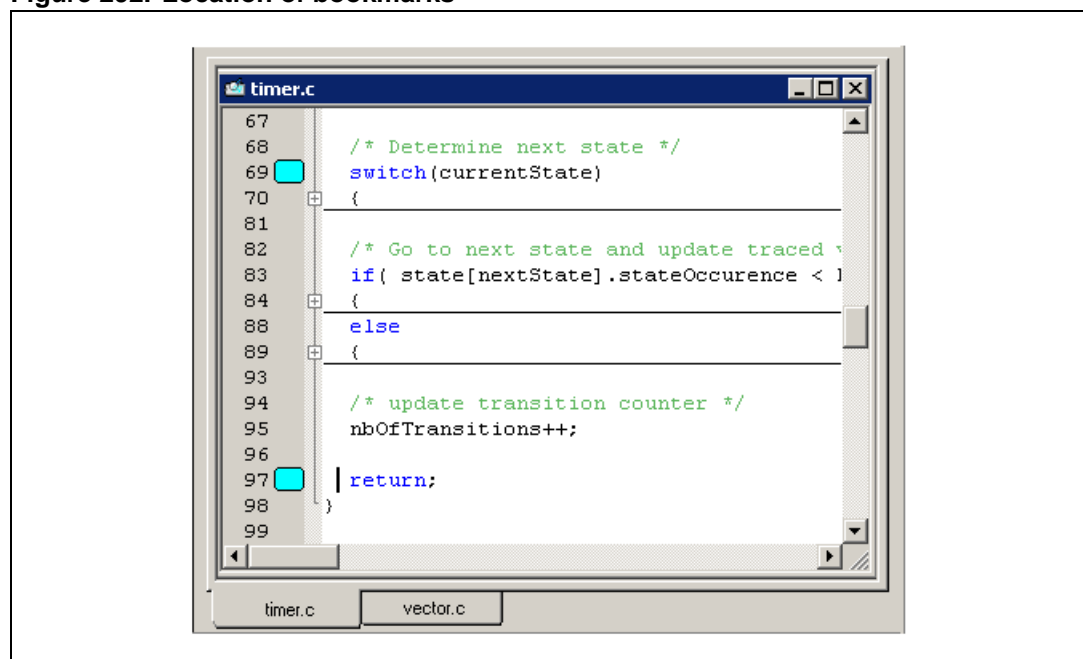
In this lesson, you will learn how to set a two level advanced breakpoint. If this program did not have the bug described in the previous procedure then write access to the `nbOfTransitions` variable would only be possible from inside the `trapISR` function. However, because of the bug, write access to the `nbOfTransitions` variable is possible from outside of the `trapISR` function. In this procedure we use the advanced breakpoint feature to detect write access to the `nbOfTransitions` variable by an instruction located outside of the `trapISR` function.

The advanced breakpoint you will set has two levels. When the application executes code outside of `trapISR` and write access to `nbOfTransitions`, “Level 1” of the advanced breakpoint stops the application. When executing code inside `trapISR`, the write accesses are not tracked. This is defined by “Level 2” of the advanced breakpoint.

The advanced breakpoint program must be activated when the program counter is outside of the `trapISR` function. We do this by setting a breakpoint outside of the `trapISR` function and running the program up to this breakpoint.

1. Set an instruction breakpoint in the `waiting_loop` function (line 107).
2. Select **Debug>Continue** from the main menu.
3. When the program stops, remove the instruction breakpoint.
4. Set bookmarks on the first and last instruction of `trapISR` (line 69 and line 97 respectively) by placing the cursor on the instruction line and selecting **Edit>Toggle Bookmark** from the main menu.

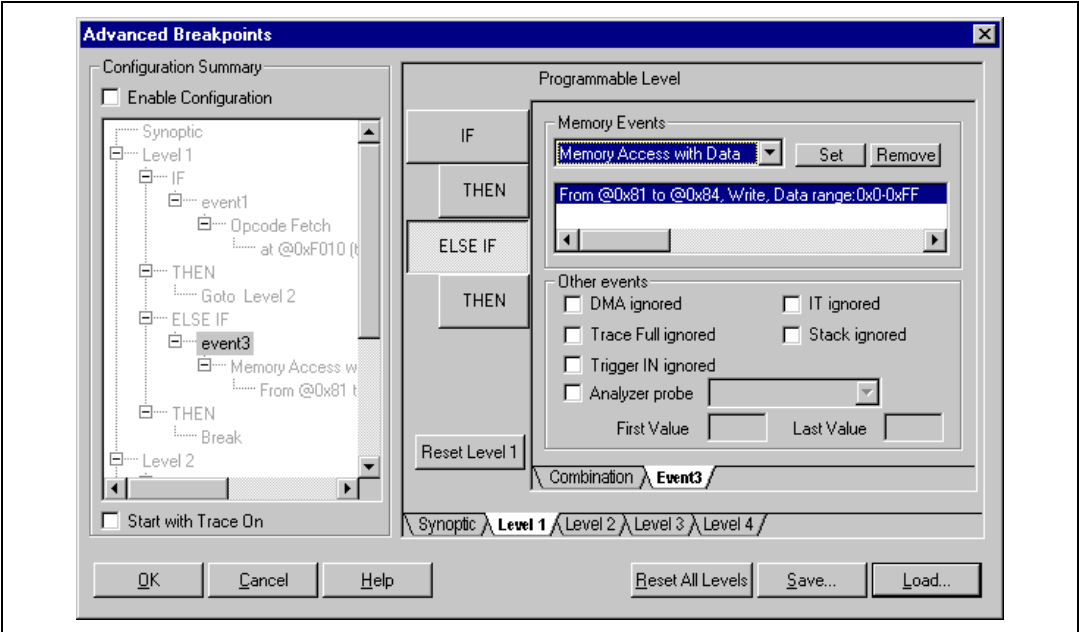
Figure 292. Location of bookmarks



5. Select **Debug Instrument>Advanced Breakpoints...** to open the **Advanced Breakpoints** window.

- The **Configuration Summary** field shows the logic structure, event definitions and actions to take for all the levels, this can be navigated using the navigation buttons (+). Use the tabs at the bottom of the right-hand window to switch between the Synoptic and any of the four programmable levels: Level 1 to Level 4. The following diagram shows the settings for level 1.

Figure 293. Setting advanced breakpoints



- The **Advanced Breakpoints** window is configured as summarized below.

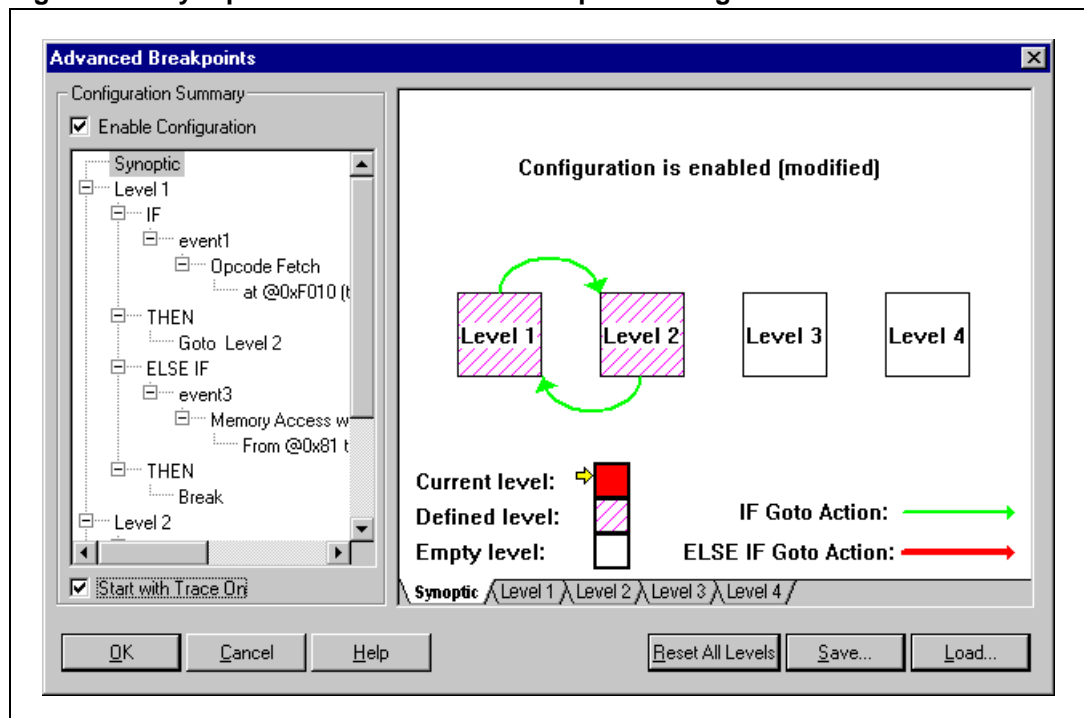
Table 86. Advanced breakpoint configuration

Level	Configuration summary	How to configure it...
LEVEL 1	IF (opcode fetch at timer.c:69	<ol style="list-style-type: none"> Click on Level 1 tab and IF. In First Event, select Event1 then click on the button [...] In Memory Events, select Opcode Fetch then click on Add In Opcode Fetch, select Bookmark, timer.c;69 then click OK.
	THEN (Goto Level 2);	<ol style="list-style-type: none"> Click on THEN and check Goto Level 2.
	ELSE IF (data memory access from nbOfTransitions, length = 4 bytes, access = write, data value from 0 to FF (for each byte))	<ol style="list-style-type: none"> Click on ELSE IF. In First Event, select Event3 then click on the button [...] In Memory Event, select Memory Access with Data then Set Enter address range (0x81-0x84), length in bytes, select access mode (Read/Write), enter data value 0 to FF with Data Range option selected, then click OK.
	THEN Break	<ol style="list-style-type: none"> Click on THEN and check Break.

Table 86. Advanced breakpoint configuration (continued)

Level	Configuration summary	How to configure it...
LEVEL 2	IF (opcode fetch at timer.c:97)	1. Select Level 2 tab. 2. In First Event , select Event1 then click on the button [...] 3. In Memory Event , select Opcode Fetch then click on Add 4. In Opcode Fetch , select timer.c:97 then click OK .
	THEN (Goto Level 1);	1. Click on THEN and check Goto Level 1 .
LEVEL 3-4	(empty)	

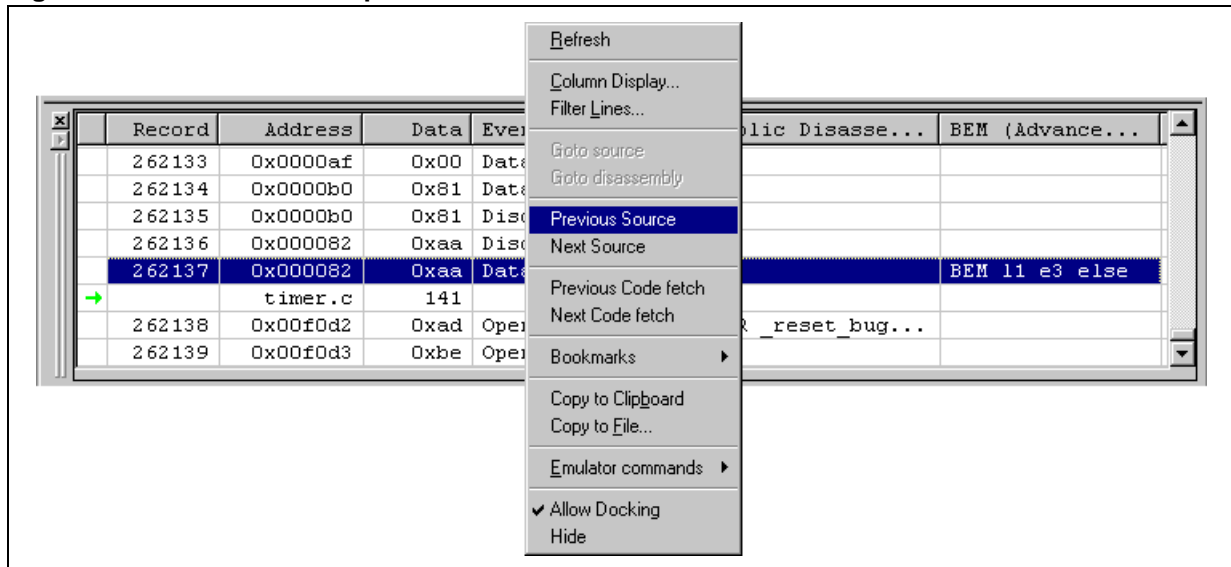
8. In the **Advanced Breakpoints** window click on the **Synoptic** tab. This tab displays a block diagram of the four levels with the programmed logical links.
9. Check the **Enable Configuration** checkbox, to activate the breakpoint.
10. Check the **Start with Trace On** checkbox, this ensures that the trace is turned on when the application starts.

Figure 294. Synoptic view of advanced breakpoint configuration

11. Click on **OK** and select **Debug>Continue** from the main menu.
12. Select **View>Watch** to display the **Watch** window if it is not already displayed.
13. In the **Watch** window watch the value of `bugCountDown` decrease, after it reaches 0 the variable `nbOfTransitions` is overwritten. The status bar indicates that a break has occurred—the advanced breakpoint has worked.
14. Select **View>Trace** to display the **Trace** window if it is not already displayed.

15. From the Trace contextual menu select **Column Display...** Click on **Uncheck all**, then check the following: **Record**, **Address**, **Data**, **Event**, **Symbolic Disassembly**, **BEM**. Click on **OK**.
16. Starting from the last record, find the record for which the BEM (Advanced breakpoints) description is "BEM I1 e3 else". Select this record and select **Previous Source** from the Trace Contextual menu.

Figure 295. Advanced breakpoint info in trace



17. From the Trace Contextual menu select **Goto source** to jump to the line of source code that sets off the bug in the Editor window.
18. Select **Debug Instrument>Advanced Breakpoints...** and uncheck the **Enable Configuration** checkbox. Click on **OK**.

12.7.4 Run a coverage and profiling session

12.7.5 Run a performance analysis

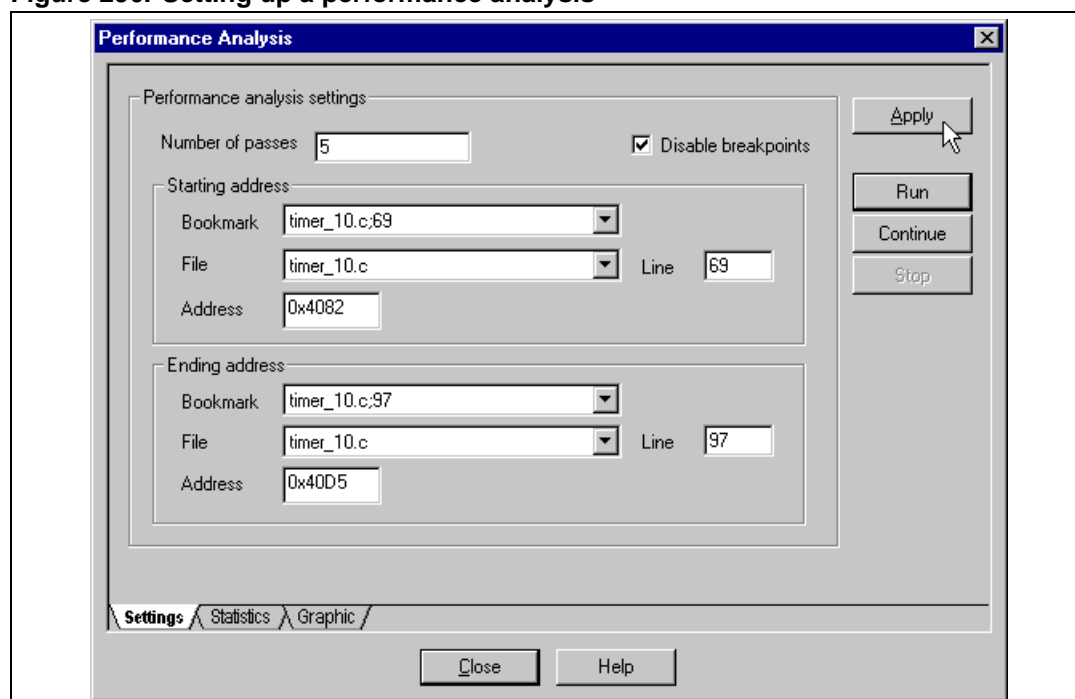
The **Performance analysis** feature displays statistics on certain performance criteria regarding the running of the application program. It measures the time spent in a section of code, defined by its start and end addresses. Data such as the minimum execution time, maximum execution time and the execution time as a percentage of the total overall execution of all code is provided.

In this procedure you will learn how to use the Performance Analysis feature to measure the time spent in a function (the trapISR function).

To run the performance analysis:

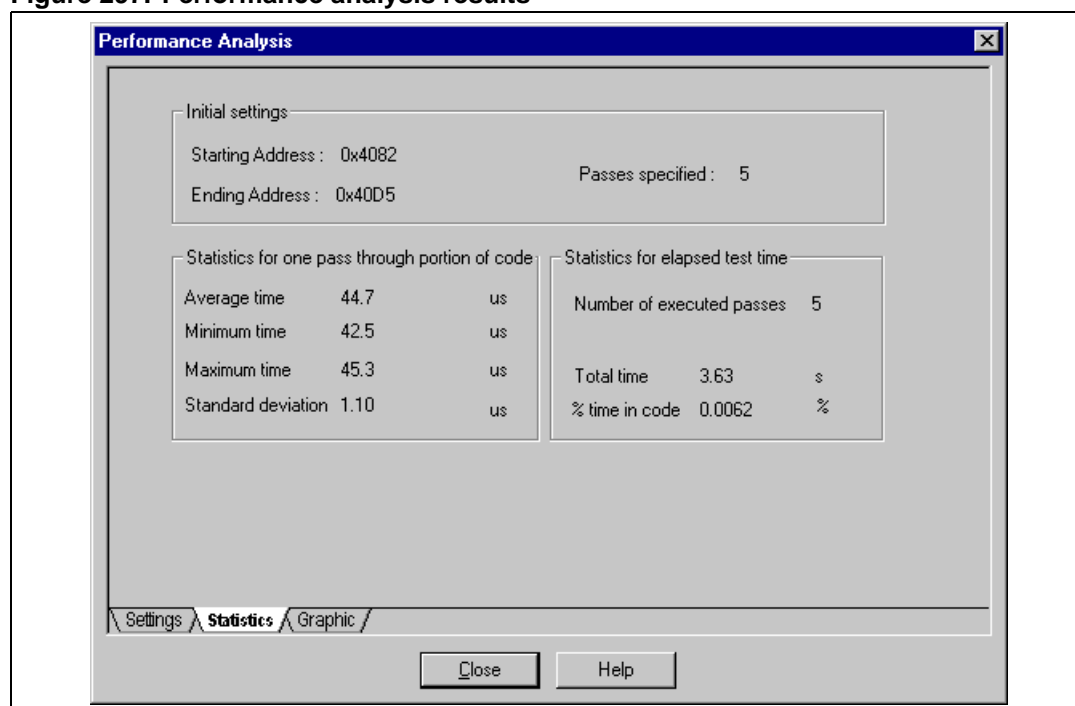
1. Select **Debug Instrument>Performance Analysis**.
2. In the **Performance Analysis** window enter 5 in the **Number of passes** field, check the **Disable breakpoints** checkbox.
3. Select **timer.c;69** and **timer.c;97** from the **Bookmarks** fields for **Starting Address** and **Ending Address**, respectively. These mark the first and last instruction of trapISR function.

Figure 296. Setting up a performance analysis



4. Click on **Apply** and then **Run**.
5. When the program execution terminates, the **Statistics** tab opens automatically to display the results of the analysis. In addition, the **Graphic** tab provides a display of the results.
6. After viewing the performance analysis results click on **Close**.

Figure 297. Performance analysis results

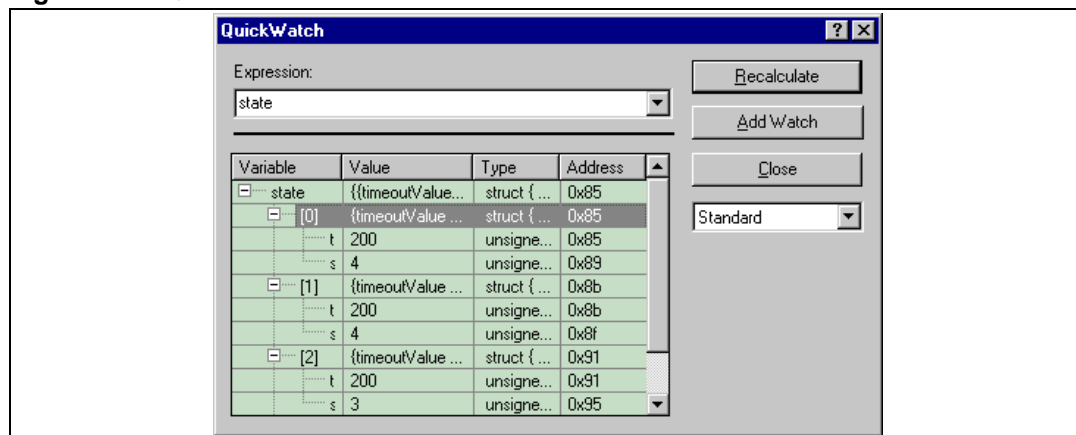


To make the trap interrupt occur more frequently:

1. Restart the application.
2. Locate the state variable in the Editor window (line 132).
3. Place the cursor on the `state` variable.
4. Select **Edit>QuickWatch** to view it in the QuickWatch window.

In the **QuickWatch** window change the `timeoutValue` value for elements 0, 1 and 2 of this array to a lower value (for example 200).

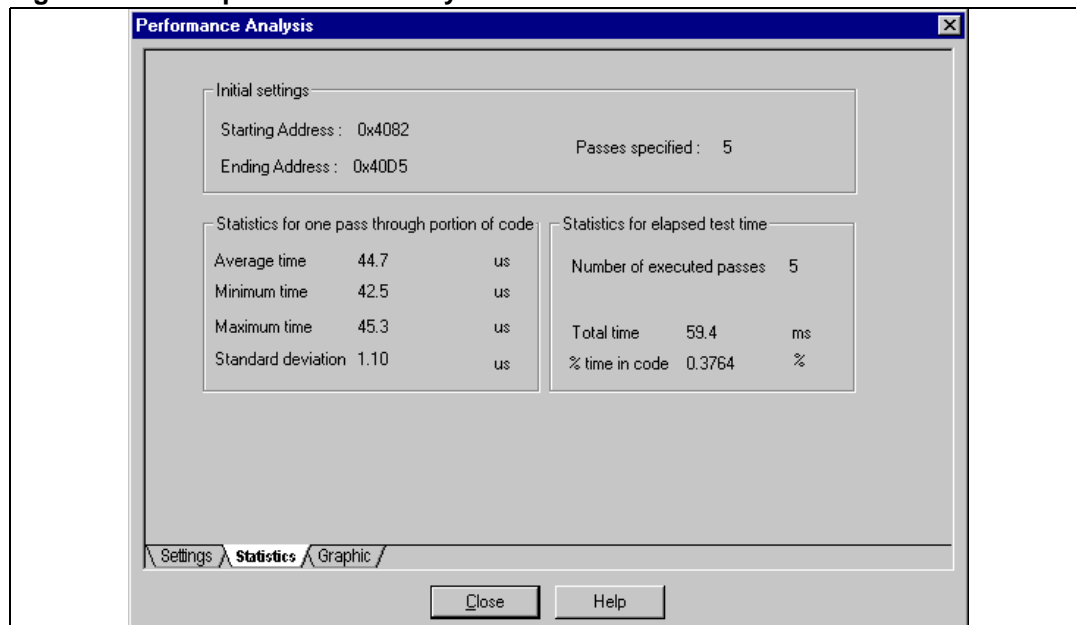
Figure 298. QuickWatch



Then select **Debug Instrument>Performance Analysis** and click on **Continue** to execute the program (do not click on **Run** because you would reset the chip and lose the values previously set). The results of the performance analysis appear much more quickly.

1. Select **Debug>Stop Debugging** from the main menu.
2. Select **File>Close Workspace** to close the workspace. The current workspace is closed and automatically saved.

Figure 299. New performance analysis results



13 ST Assembler/Linker build tutorial

This build tutorial takes you through the steps of setting up workspaces, creating projects and building assembler source code into an application that can be debugged with STVD.

Your objective in this tutorial is to build the provided source code in assembler language into an **executable in Motorola S format** for the **ST72311N4**.

To do this, we have provided you with the following files:

- **tim_rtc.asm** — the application source file in assembler language. This is a Real Time Clock application that uses the Output Compare Interrupt software driver of the ST72311xx Timer. With this application, a timer interrupt is generated every 100mS when fCPU = 4MHz.
- **tim_rtc.bat** — this is the .bat file that is typically used when building with the ST Assembler Linker.
- **st72311n4.asm** — a source file in assembler language that specifies the memory mapping of the ST72311n4 peripheral registers, which is required for the application.
- **st72311n4.inc** — an include file that defines the addresses of the ST72311n4 peripheral registers, which is required for the application.

For standard installations of STVD, the files listed above are saved at the directory location:

C:\Program Files\STMicroelectronics\st_toolset\stvd\Examples\tutorial_asm

*Note: The **st72311n4.asm** and **st72311n4.inc** have been copied to the examples directory, however these included files are also provided in the directory ...**st_toolset\asm\include** for supported ST7 devices. For typical use you should use the provided ST7 include files from their original location unless you want to modify them. If you want to modify an include file for a specific application, first copy it to your working directory so as to preserve an unmodified version of the original include file.*

13.1 Create a new workspace with the New Workspace wizard

Before you can build or debug an application with STVD, you must create a workspace to contain any projects and their source files.

1. In the main menu bar select **File>New Workspace**.
2. In the **New Workspace** window, click on the **Create an Empty Workspace** icon.
3. Click on **OK**, the view in the New Workspace window changes.

In the resulting view of the **New Workspace** window, you must enter a name for your workspace and identify the directory pathname.

1. Enter the name `my_workspace` in the **Workspace Filename** field.
2. Enter the pathname to identify the directory where you want STVD to store your workspace.
3. You can use the browse button to find the location
4. You can create a folder for your new workspace by clicking on the **Create folder** icon. Enter the folder name `my_workspace` in the **Create Folder** window and click on **OK**.
5. Click on **OK** to create the workspace (`.stw`).
6. Before going any further, save your workspace. In the main menu bar, **File>Save Workspace**.

RESULT:

The **Workspace** window now contains the workspace `my_workspace.stw`. Note, however, that in the **Build** menu, you do not have access to the build commands, because your workspace does not contain a project, yet.

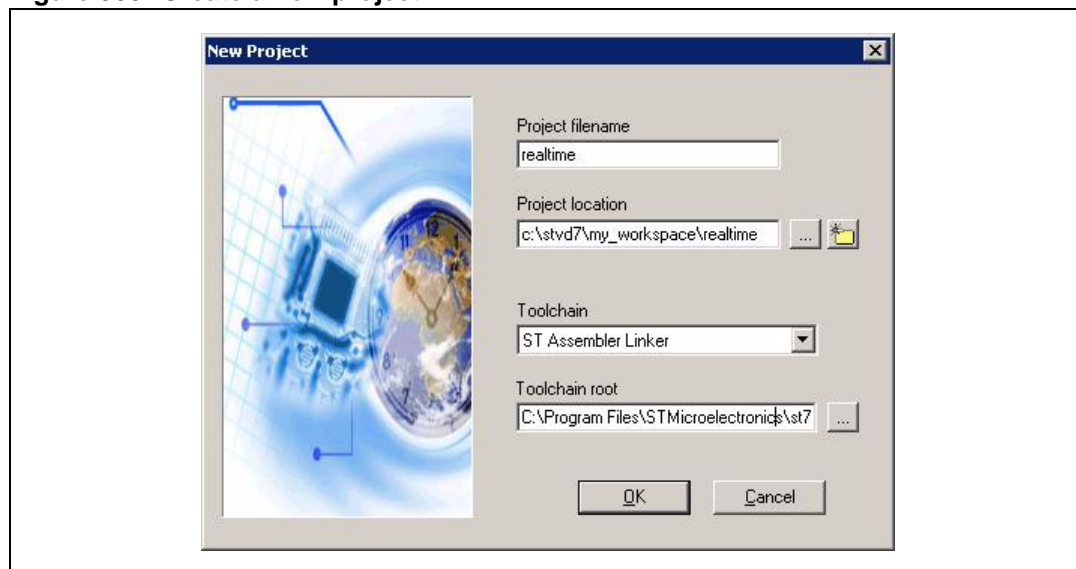
The working directory that you have identified now contains the file `my_workspace.stw`.

13.2 Create a project using Add New Project to Workspace command

Before you can access build and debug commands, you must create a project or insert an existing project in your workspace. As we have only provided you with the source files for the application (not a project), you must create a project from scratch.

1. In the main menu bar, select **Project>Add New Project to Workspace**.
2. In the **New Project** window, click on the **New Project** icon.
3. Click on **OK**, the view in the **New Project** window changes.

Figure 300. Create a new project



In the resulting **New Project** window, you must enter a name for your project and associate the project with the toolchain that you will use to build your application. For this tutorial, you will use the ST Assembler Linker, which was installed along with STVD.

To do this:

1. First, enter the name `realtime` in the **Project Filename** field
2. Enter the pathname to identify the directory where you want STVD to store your project in the **Project Location** field. By default, this will be the same as the working directory that you identified when you created your workspace.

You can use the browse button to find a location if necessary.

You can create a folder for your new project by clicking on the Create folder icon. Enter the folder name `realtime` in the **Create Folder** window and click on **OK**.

3. Next, select **ST Assembler-Linker** from the **Toolchain** list box.
4. Enter the path for the toolchain in the **Toolchain Root** field. You can use the browse button to find it if necessary.

For standard installations of STVD the toolchain root is:

```
C:\Program Files\STMicroelectronics\st_toolset\asm
```

5. Click on **OK** to apply these settings.
6. The **MCU Selection** window opens. Select **ST72311N4** from the list, click on **Select**, then click **OK**.
7. Save the workspace again. In the main menu bar, **File>Save Workspace**.

RESULT:

The workspace contains the project, **realtime.stp**. By default this project contains folders called **Source Files**, **Include Files** and **External Dependencies**. The folders that you see in the **Workspace** window are used to organize your project within STVD. They do not exist in the working directory.

However, the working directory that you have specified does contain the file **realtime.stp**. When you build the application this directory will also contain one or two folders (**Debug** and/or **Release**) depending on the build configuration that you use. These folders are the default locations for storing your application and any intermediate object files and listings.

You can now access the project settings. The project settings allow you to control the building of your application using the options that are specific to your toolchain. From the main menu bar select **Project>Settings** to open the **Project Settings** window. In the **General tab**, you will see that the project `realtime.stp` is associated with the ST Assembler Linker toolchain. The path for locating the toolchain is displayed in the **Root Path** field. Exit this window by clicking on **OK**.

In addition the project is associated with an MCU: the **ST72311N4**. Your project will be built for this MCU, taking into account its peripherals and memory mapping. When selecting your debug instrument and programming hardware you will be limited to those tools that support this MCU. You can always change the MCU selection later in the project settings interface, but you must rebuild the application.

13.3 Inserting files in your project

To build your application, you must identify the source files for STVD. You will do this by inserting the source files using the **Insert Files into Project** command. In addition, when building with the ST Assembler Linker, your files may need to be built in a specific order. Once placed in the correct order you can protect the order using the **Add Sorted Elements** option.

Adding source files to the project

1. Selecting **Project>Insert Files Into Project** from the main menu bar, this opens a browse window.
2. Select the following files:
 - st72311n4.asm
 - st72311n4.inc
 - tim_rtc.asm

For standard installations of STVD these files are located at:

C:\Program Files\STMicroelectronics\st_toolset\stvd\examples\tutorial_asm\

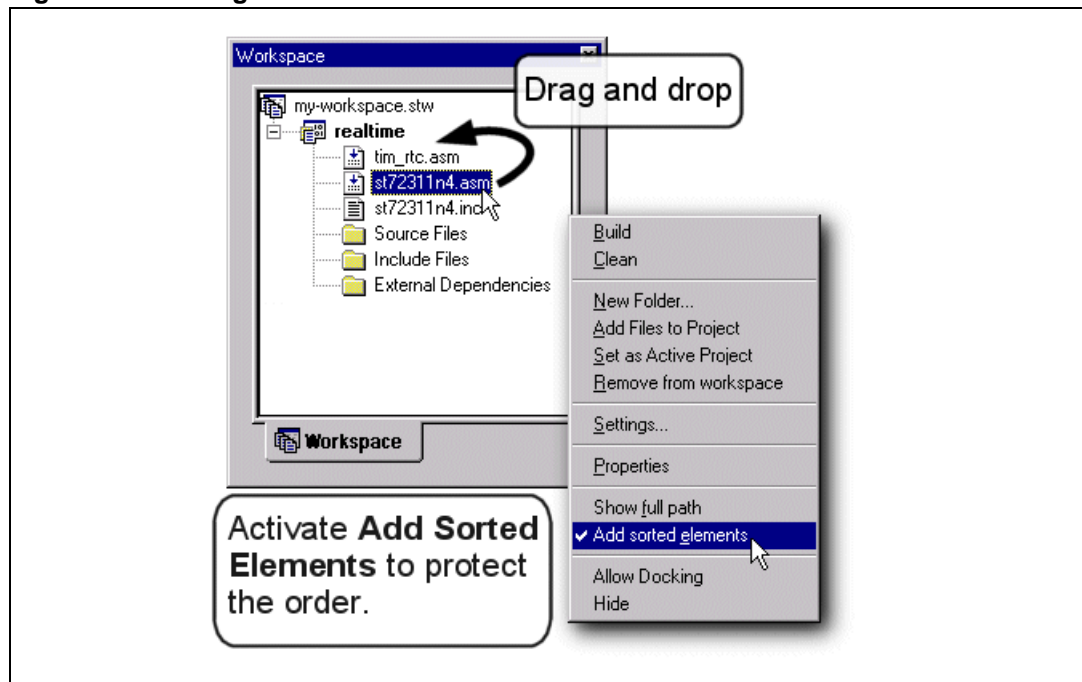
3. Click on **Open**.

The files are added to the project in the **Workspace** window, however they have not been placed in specific folders and they are not necessarily in the correct order. We will build the files in the alphabetical order: st72311n4.asm, st72311n4.inc, tim_rtc.asm.

To correct the ordering of the files:

1. Right-click on the project to open the contextual menu and click to remove the checkmark next to the **Add Sorted Elements** option.
2. Drag the files into the required order, first to last from top to bottom (that is, the first file, st72311n4.asm, on top, and the last file, tim_rtc.asm, on the bottom).

Figure 301. Arrange the files



3. Right-click on the project to open the contextual menu and click to place the checkmark next to the **Add Sorted Elements** option. Their order is now protected.
4. Now, drag the source files st72311n4.asm and tim_rtc.asm to place them in the **Source Files** folder. Note that STVD maintains the ordering of the files if you move them one at a time.

5. Drag the file `st72311n4.inc` to the **Include Files** folder.
6. You can save your workspace by selecting, **File>Save Workspace**.

RESULT:

Open the Project Settings window by selecting **Project>Settings** in the main menu bar. In the Project Settings window, select the **Debug** tab. The **Source Directories** field shows the pathnames that the ST Assembler Linker will use to locate your source files, `tim_rtc.asm` and `st72311n4.asm`.

Now select the **ST7 ASM** tab. The **Include Paths** field shows the pathname that the ST Assembler Linker will use to locate the include file `st72311n4.inc` that you inserted in the project.

13.4 Creating a folder to organize files in the project

You can also create folders within your project to help you organize files. To do so:

1. Right-click on the project to open the contextual menu and select **New Folder**.
2. In the **New Folder** window, enter the name "Obsolete" in the **Name of the New Folder** field.
3. Click on **OK** to create the folder and close the window.

You can add files directly to the new folder using the **Add Files to Folder** command. To do so:

1. Right-click on the **Obsolete** folder and select **Add Files to Folder** from the contextual menu.
2. In the browse window, select the file, `tim_rtc.bat` to add to the folder and click on **Open**.
For standard installations of STVD this file is located at:

```
C:\Program Files\STMicroelectronics\st_toolset\stvd\examples\tutorial_asm\
```

RESULT:

The file `tim_rtc.bat` has been added to your project. This is the type of `.bat` file that was used when building with previous versions, however this file is no longer necessary for building with STVD7 3.0 and later versions. We have included this file, which contains the command lines and build options used when building with a previous version so that you can easily view it and compare the command lines and options that result from using STVD's **Project Settings** window. This file is only included as a reference and we will specify not to use it in the build later in this tutorial.

13.5 Project settings

STVD's **Project Settings** window allows you to view the command lines and options that will be used when building your application. From this window you can also change options and apply user-defined options and commands that are supported by your toolset.

Changing build settings

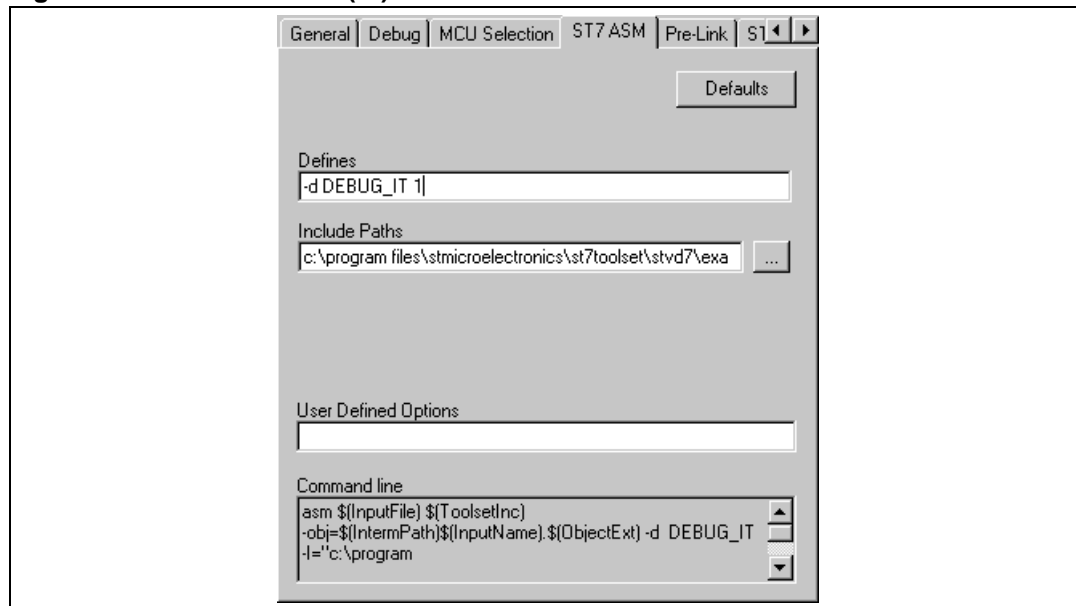
Settings can be applied to the entire project or to specific files. In the following procedure we will use the **define (-d)** option in order to activate a special instruction in the code for the debug version of our application.

The application source contains a routine under the `#ifdef DEBUG_IT` (lines 153-155 of `tim_rtc.asm`). When activated, this routine generates an output signal on the pin PFDR whenever a timer interrupt occurs. To activate this routine we will use the define option to assign a value other than zero to `DEBUG_IT` (When `DEBUG_IT` is zero the routine is ignored).

To do this:

1. Select the **ST7 ASM** tab.
2. Type `DEBUG_IT 1` in the **Defines** field.

Figure 302. Define a value (-d)



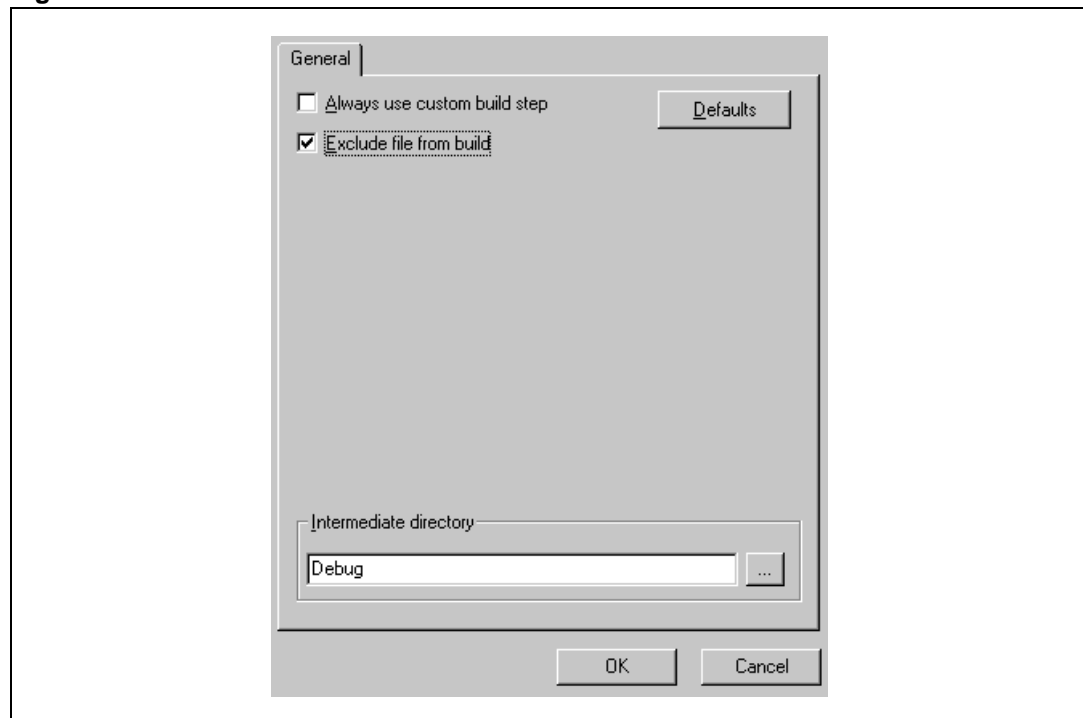
RESULT:

When we build the application, the instruction `INC PFDR`, which is assembled if `DEBUG_IT` is a value other than zero, will be included in the final application. When we run the application during a debug session the application will generate a signal on the PFDR pin each time a timer interrupt occurs (every 100ms).

Changing settings for a specific file

In addition to applying settings at the project level, you can also specify settings for a file. As describe in an earlier step, we included the `tim_rtc.bat` as a reference for understanding STVD's **Project Settings** interface. However, we do not want to include this file in the build of our application. In this step we will exclude this file from the build, by applying a file level setting via the **Project Settings** window.

Figure 303. Exclude file from build



To do this:

1. In the field on the left side of the **Project Settings** window, click on the project to display all the folders it contains.
2. Click on the **Obsolete** folder that we created earlier in order to view the folder's contents
3. Select the **tim_rtc.bat** file, the **Project Settings** window changes to show only those tabs with settings that apply specifically to this file.
4. In the **General** tab, click to place a checkmark in the **Exclude File from Build** checkbox.
5. Click on **OK** to close the **Project Settings** window.

RESULT:

During the build the assembler will not be invoked for the `tim_rtc.bat` file. Because we have done this, it will have no impact on the building of the application.

13.6 Build a version for debugging

You will see the final result of the settings that we have applied in the **Project Settings** window, now, when we build the application. To build the application:

1. If the **Output** window is not already open, select **View>Output Window** from the main menu bar.
2. In the **Output** window, click on the **Build** tab
3. To generate your application, select **Build>Build** from the main menu bar.

RESULT:

The commands invoked, error messages and warnings that occur during the build are displayed in the **Build** tab of the **Output** window.

The **Assembler** is invoked for each of the `.asm` source files in the project. It is not invoked for the file that we excluded from the build (`tim_rtc.bat`). The Assembler command line is:

```
asm -sym -li=$(IntermPath)$(InputName).lsr $(InputFile) $(ToolsetInc) -  
obj=$(IntermPath)$(InputName).$(ObjectExt) -d DEBUG_IT 1 -I="C:\Program  
Files\STMicroelectronics\st_toolset\stvd\Example\tutorial_asm"
```

The **Linker** is invoked once, to link the object files. The linker command line is:

```
lyn Debug\st72311n4.obj+Debug\tim_rtc.obj, Debug\realtime.cod, " "
```

OBSEND is invoked once to generate the final output `realtime.s19` in Motorola S-Record format. The OBSEND command line is:

```
obsend Debug\realtime.cod,f,Debug\realtime.s19,s
```

The **Post-Link Step** is run in order to create the listing of absolute addresses required by the debugger.

The following files have been generated in the **Debug** folder in the working directory:

- `realtime.s19`
- `realtime.map`
- `realtime.sym`
- `realtime.cod`
- `st72311n4.lsr, tim_rtc.lsr`
- `st72311n4.lst, tim_rtc.lst`
- `st72311n4.obj, tim_rtc.obj`

The application `realtime.s19` includes the routine activated by the `-d` option, which will generate a signal on the PFDR pin each time a timer interrupt occurs. You can confirm this by running the application on the Simulator and plotting the output of the PFDR pin.

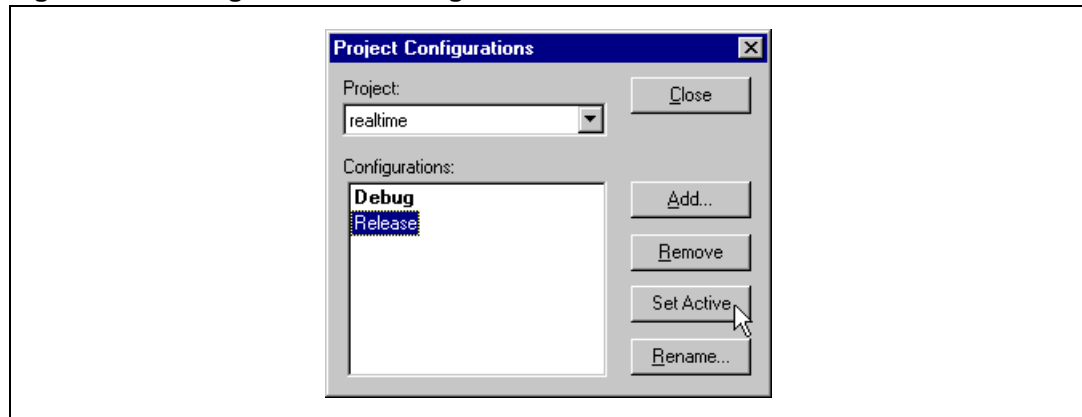
13.7 Building a version for programming

STVD is designed to allow you to customize and change project settings quickly and easily. Build configurations allow you to save and change all the settings associated with a build. For example the default configurations allow you to switch between settings for generating a version for debugging or a final, optimized version of the application. In addition, the tabs in the **Project Settings** window also allow you to enter user defined commands that are supported by your toolchain.

Changing the build configuration

Project settings can be saved as configurations allowing you to quickly change and recall a full range of settings. STVD has two preset configurations: **Debug** and **Release**. By default, when you create a project in STVD, the configuration is set to **Debug**. Now we are going to generate a final version using the **Release** configuration.

Figure 304. Change the build configuration



To change the build configuration:

1. From the main menu bar, select **Build>Configurations**.
2. In the **Project Configurations** window, the active project name is in the **Project** field. The current configuration is in bold type (Debug should be in bold type).
3. Select **Release** in the **Configuration** field and click on **Set Active**.
4. Click on **Close**.

RESULT:

In the **Project Settings** window (**Project>Settings...**), the **Settings For** field is now set to Release. In the **General** tab, the **Output File** specifies that output from the build be saved to the Release folder in the working directory.

Note: Other toolchains allow you to apply a range of optimizations to improve the performance or the size of your final application. Some of these optimizations have been applied by default in the Release configuration, not the Debug configuration.

Applying project settings

The project settings that we applied in the debug configuration were not carried over into the Release configuration. We must now go back and apply any necessary settings that were not carried over.

1. Select **Project>Settings** from the main menu.
2. Click on the project **realtime** in the left hand field of the **Project Settings** window, in order to view all of its folders.
3. Click on the **Obsolete** folder to view its contents.
4. Click on the `tim_rtc.bat` file, to see the tabs that are specific to this file.
5. Click to place a checkmark in the **Exclude File from Build** checkbox in the **General** tab.

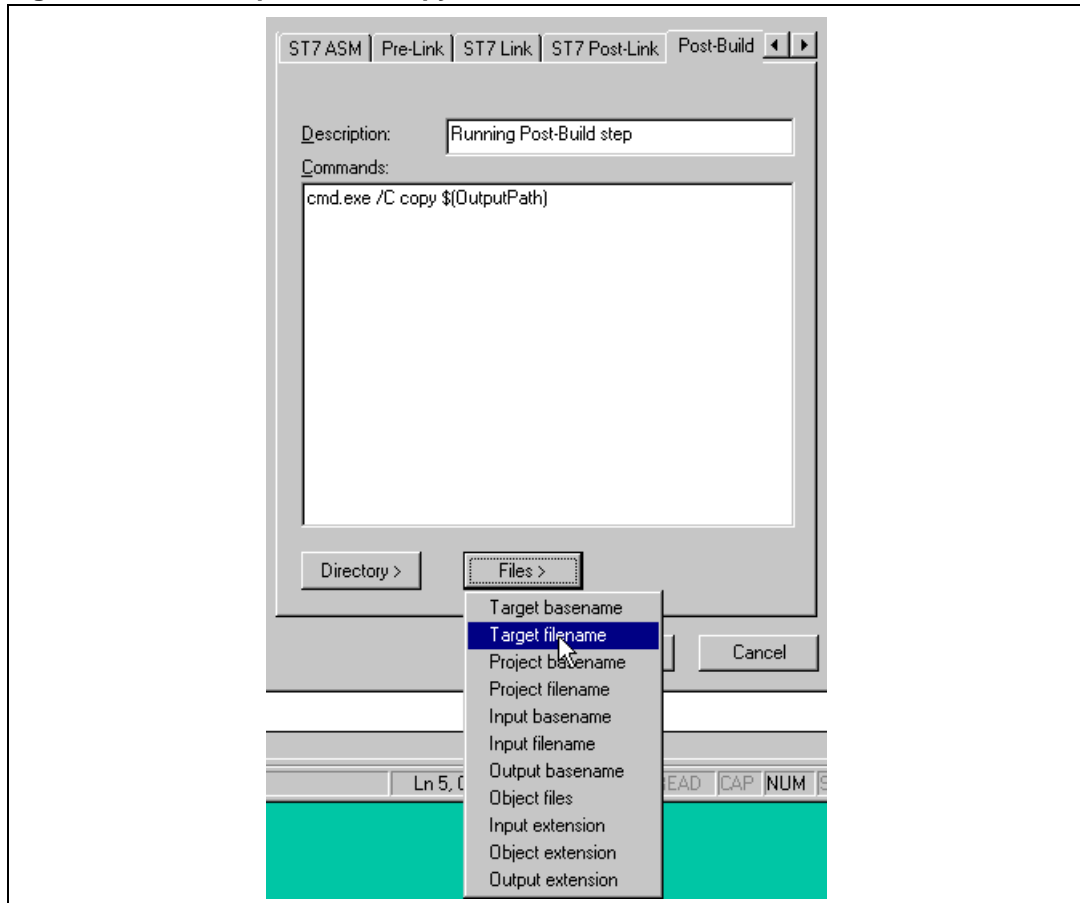
You have now re-applied the project settings as they were for the Debug version of the application, except for `-d DEBUG_IT 1`, which we used to illustrate the use of the `-d` option.

Before continuing, click on the **realtime** in the left hand field of the **Project Settings** window, so that we can see all the settings tabs for the project. You can now go to the next step, [Entering a user-defined command](#).

Entering a user-defined command

User-defined commands such as those applied in the **Post-Build** tab, can allow you to specify tasks to accomplish in addition to the typical build process. Such a task could be to copy a version of the final application to another directory.

Figure 305. Enter a post build copy command



To do this:

1. Create a new folder on your hard drive. For the purposes of this example, I created a file with the pathname C:\final.
2. In the **Project Settings** window (**Project>Settings...**), click on the **Post-Build** tab
3. Click to place the cursor in the **Commands** field.
4. Type `cmd.exe /C copy` followed by a space.
5. Click on the **Directory** button and select **Output Folder**. This macro (`$OutputPath`) tells STVD to look in the project's output folder (Release) for the file to copy.
6. Click on the **File** button and select **Target File Name**. This macro `$(TargetFName)` tells STVD to look for the target application file that is specified during linking.

7. Type a space then the pathname for the folder where you want to save the copy of the application file.

The resulting command should be:

```
cmd.exe /C copy $(OutputPath)$(TargetFName) c:\final
```

In this command **c:\final** is the pathname for the folder to which the file will be copied.

8. Click on **OK** to apply the new settings and close the **Project Settings** window.
9. Now to build the release version of your application, select **Build>Build** from the main menu bar.

The **Assembler** is invoked for each of the .asm source files in the project. However this time the executable for the application is generated in the **Release** folder of the working directory. This folder is created automatically by STVD, as with the **Debug** folder.

After the **Linker** and **OBSSEND** are invoked, STVD runs the command you entered in the post-build step.

```
cmd.exe /C copy $(OutputPath)$(TargetFName) c:\final
```

As a result, the file `realtime.s19` has been copied to the folder `c:\final`. The following files are regenerated in the `Release` folder:

- `realtime.s19`
- `realtime.map`
- `realtime.sym`
- `realtime.cod`
- `st7321ln4.obj, tim_rtc.obj`
- `st7321ln4.lst, tim_rtc.lst`

Appendix A Product support

If you experience any problems with this product, or if you need spare parts or repairs, contact the distributor or the STMicroelectronics sales office where you purchased the product. Phone numbers for major sales regions are provided on the www.st.com web site.

From the www.st.com site, select **Products > Microcontrollers** to obtain a complete online selection guide, as well as documentation, software downloads and user discussion groups to help you answer questions and stay up to date with our latest product developments.

A.1 Software updates

All our latest software and related documentation are available for download from the STMicroelectronics microcontrollers support site at www.st.com.

If you are using software from a third-party tool provider, please refer to the third-party for software product support and downloads.

A.2 Hardware spare parts

Your development tool comes with the hardware you need to set it up, connect it to your PC and connect to your application. However, some components can be bought separately if you need additional ones. You can order extra components, such as sockets and adapters, from STMicroelectronics, from the component manufacturer or from a distributor.

To help you find what you need, a listing of accessories for STMicroelectronics development tools is available on the STMicroelectronics Internet site, www.st.com.

A.2.1 Sockets

Complete documentation and ordering information for P/LQFP sockets from Yamaichi, Ironwood, CAB and Enplas are provided at their respective Internet sites.

A.2.2 Connectors

Complete documentation and ordering information for SAMTEC connectors is provided at their Internet site.

A.3 Getting prepared before you call

Collect the following information about the product before contacting STMicroelectronics or your distributor:

1. Name of the company where you purchased the product.
2. Date of purchase.
3. Order Code: Refer to the side your emulators box. The order code will depend on the region in which it was ordered (for example, the UK, Continental Europe or the USA).
4. Serial Number: The serial number is found located on the rear panel of the ST Micro Connect box and is also listed on the Global Reference card provided with the emulator.
5. TEB (Target Emulation Board) hardware and firmware versions: The hardware and firmware versions can be found by opening an STVD7 session, entering the debug context and selecting **Help > About** from the main menu. The TEB version numbers are given in the *Target box* – scroll downwards until you find the TEB version (hardware) and TEB PLD version (firmware).
6. Target Device: The sales type of the ST microcontroller you are using in your application.

Revision history

Table 87. Document revision history

Date	Revision	Changes
	1	Initial release.
1- Feb-2001	2	Added support of EMU3.
1-Aug-2001	2.3	Updated for improved version of Simulator.
1-Oct-2001	2.4	Updated for release of new emulators.
1-Sep-2002	2.5	Updated for simplified USB connection. Updated for In-Circuit Debugging with ST7-ICD (STMC).
1-Dec-2003	2.5.4	Updated with In-Circuit Debugging with EMU3. Added device Discrepancies window.
1-Apr-2004	3.0	One user manual for all debug instruments. Added "What's New" section. Added migration procedure for users of STVD7 2.x. Added "Build" section. Added Build tutorials.
1-Nov-2004	3.1	Added Table 65 – Document revision history table. Added Section 2.9 – Help and support features. Added Section 6 – In-Circuit Debugging with DVP3.
15-Jun-2005	4.0	Updated Host PC system requirements on page 8 – host PC requirements. Updated Section 2.5 – with new editor features. Updated Find features on page 61 – with description of Regular Expressions. Updated Section 2.7 – with descriptions of new Edit/Debug and Languages tabs in Tools Options window. Updated Section 3.5.1 – descriptions of Output and Intermediate directories. Added Section 4.14 – description of MSCI Tools window. Updated Section 10.2.4 – tutorial procedure for Cosmic linker options.
24-Jan-2006	5.0	What's new with STVD7 3.3 Patch1 with new feature information. <i>Updated Section 6.3: Using the simulation plotter</i> – descriptions of new Plotter interface and features. Updated Section 3.6.2: ST7 Link tab , Section 3.7.3: Cosmic C linker tab , Section 4.9.3: Metrowerks linker tab , with descriptions of segment and section customization. Updated MCU selection tab on page 225 – changes to LED colors, update screenshots. Updated Introduction on page 1 and Selecting the debug instrument on page 161 – for support of RLink in-circuit debugger/programmer. Updated Section 6.4: Trace recording – description of discarded events, description of filtering function.

Table 87. Document revision history (continued)

Date	Revision	Changes
22-Nov-2007	6	Added section What's new with STVD 4.0.1 on page 24 . Deleted Section 6.4: Trace recording , now obsolete. Included information on Raisonance C compiler now supported by STVD7. Updated document format.
06-Mar-2008	7	Updated introduction to include information on STice and STM8 support. Updated Section 4: Project creation and build to include information on STM8 support. Added Section 9: STice features on page 258 .
13-May-2008	8	Updated Section 7: In-circuit debugging with information on the SWIM protocol.
22-Sep-2008	9	Added Section 9.2: Coverage and profiling .
10-Aug-2009	10	Removed section What's new with STVD 4.0.1 on page 24 . Updated Section 4: Project creation and build . Updated Section 9.2: Coverage and profiling . Updated Section 12: STM8 C tutorial .
01-Apr-2010	11	Updated Section 4.7.3: Cosmic C linker tab in subsection Customizing linker input settings . Updated Section 4.6: Customizing build settings for ST Assembler/Linker toolset . Updated Section 13.6: Build a version for debugging .

Index

A

advanced breakpoints	19
EMU3	
analyzer input signals	317
configuration summary	301
creating	292
defining levels	292
enabling	301
examples	304
memory access event	296
other events	300
programming triggers	314
synoptic representation	302
ICD	230
allocating resources for	229
settings	232

B

bookmarks	47
icon	47
insert, remove, navigate	36
breakpoints	19
advanced breakpoints (EMU2)	248
advanced breakpoints (ICD)	230
break on TRIGIN (ICD)	233
data breakpoints	186
setting	186
window	187
with conditions	187
with counters	187
icon	173
instruction breakpoints	183
setting	183
with conditions	184
with counters	184
software vs. hardware (ICD)	228
building	75
commands	158
configurations	83
debug settings	86
general settings	85
microcontroller selection	86
output window	57
project settings	84
settings for files	156
settings for folders	156

C

call stack frame indicator	173
configurations	83
build project settings	83
contextual menu	40
allow docking option	34
editor debug actions	170
editor window	49
EMU3 trace	286
file	42
folder	42
hardware event window	238
peripheral registers	197
project	42
workspace window	42
Cosmic C	93
assembler	107
compiler	95
linker	110
project settings	93
Script LKF	112

D

data breakpoints	19
window	186
debug instrument	161
add connection ports	162
port selection	162
selection	30
debugging	160
advanced features	19
call stack window	188
common features	160
console	58
data breakpoints	186
editor debug actions	169
hardware configurations	18
instruction breakpoints	183
output triggers	237
DVP	255
output window	58
peripheral registers window	196
simulator	202
symbols	172
symbols browser window	195
watch window	191
dependencies	158

- between files157
- between projects158
- directory85
 - intermediate85, 157
 - output85, 157
- documentation
 - conventions22
 - for hardware configurations21
- DVP emulator237
 - debugging features237
 - hardware events237
 - hardware test246
 - in-circuit debugging223
 - stack control254

E

- editor43
 - auto completion50
 - auto-completion37
 - auto-indent51
 - block indent51
 - bookmarks36, 47
 - brace matching52, 62
 - complete word37, 50
 - contextual menu49
 - cut, copy, paste36
 - debug icons172
 - debug option63
 - debugging actions169
 - drag and drop52
 - enable folding62
 - end of line characters63
 - file folding46
 - find in files36, 54
 - find text string52
 - find, replace36
 - go to36, 55
 - indentation51
 - inserting breakpoints36
 - instruction syntax37, 50
 - line length63
 - line numbering46
 - line selection46
 - margin46
 - margin selection62
 - match brace36
 - parameter info37
 - parameter information50
 - print options62
 - quick watch36
 - refresh36

- regular expressions55
- replace53
- status bar information48
- syntax highlighting50, 64
- tabs, indents62
- text selection52
- undo/redo36
- watch pop-up63
- window position48
- wrapping options62
- editor window43
 - customizing61
- EMU2 (HDS2) emulator237
 - debugging features237
 - hardware events237
 - hardware test246
 - Logical Analyser248
- EMU3 emulator284
 - analyzer input317
 - debugging features284
 - in-circuit debugging223
 - performance analysis319
 - read/write on the fly323
- end of line characters63
- error logs70
- evaluating your application (EMU3)319

F

- files41, 44
 - adding to project83
 - dependencies157
 - settings for156
 - sorted elements88
 - toolset specific75
 - types supported by editor44
- find52
 - regular expressions55
- find in files58
- folders41
 - adding to project83
 - settings for156

G

- GDB58, 199
 - commands199
 - help200
 - load filename199
 - pin -output_file199
 - pin -stimuli199
 - save_memory200
 - symbol-file200

console58
 go to55
 program counter170

H

hardware events237
 hardware test246

I

ICD19
 features223
 functional limitations/discrepancies234
 in-circuit debugging223

L

Logical Analyser for EMU2248

M

makefile77
 create project from82
 create workspace from79
 migration71
 MCU86
 debug instrument support87
 memory map164
 on-chip peripherals166
 selection86
 menus36
 adding custom commands67
 build menu38
 contextual menus40
 debug instrument38
 debug menu38
 edit menu36
 file menu36
 help menu39
 project menu37
 tools menu38
 view menu37
 window menu39
 Metrowerks C131
 assembler143
 compiler133
 linker148
 linker PRM file153
 project settings131
 microcontroller86
 configure peripherals166
 debug instrument support87

ICD limitations235
 memory map164
 memory types165
 on-chip peripherals166
 selection86
 migration71
 full vs. partial71
 procedure72
 MSCI tools (EMU, ICD)194

O

online assembler function178
 output triggers237, 255
 output window56
 console tab58
 debug tab58
 find in files58

P

path information76
 performance analysis (EMU3)319
 peripherals
 configuring target166
 value of peripheral registers196
 program counter173
 go to170
 programming325
 hardware configurations19
 project41
 common settings84
 create80
 debug settings86
 dependencies158
 general settings85
 microcontroller selection86

Q

quick watch174

R

read/write on the fly19
 simulator220
 read/write on the fly (EMU3)323
 running applications167
 run commands167
 run to cursor171
 status bar indications169
 step commands167
 stepping modes168

S

simulator	202
read/write on the fly	220
software updates	377
ST Assembler Linker	88
assembler	89
linker	90
project settings	88
stack	188
call stack window	188
stack control (DVP)	254
status bar	34, 169
step commands	167
modes	168
STice	
emulator	258
STice emulator	
debugging features	258
STVD	
supported hardware	18
supporting documentation	21
user options and preferences	39
support	21
information file	70

T

text editor	43
clipboard operations	52
toolbars	65
creating custom	66
customizing	65
moving	66
popup tips	67
rearranging	67
toolset	75-76
Cosmic C	93
file types	75
Metrowerks C	131
path information	23
project specific paths	85
ST Assembler Linker	88
tooltips	69
trace	19
hardware events	237
recording EMU3 trace	284
view in trace window	198
trace recording	19
DVP	242
EMU2 (HDS2)	242
EMU3	284
start with trace on	301

triggers

hardware events	237
output triggers (EMU2/DVP)	237, 255
output triggers(EMU3)	313
TRIGIN (ICD)	233

V

variables

intrusive read/write	220, 323
viewing local variables	190
viewing variables	191
symbols browser	195

W

windows	40
call stack window	188
disassembly window	175
docking	34, 41
instruction breakpoints	183
local variables window	190
memory window	180
output window	56
peripheral registers window	196
ST registers	192
symbols browser	195
text editor	43
trace window	198
watch window	191
workspace window	41
workspace	41
.wsp	77
create	78
migration	71
new workspace wizard	23
workspace environment	34
customizing	60, 75
status bar	34

Please Read Carefully:

Information in this document is provided solely in connection with ST products. STMicroelectronics NV and its subsidiaries ("ST") reserve the right to make changes, corrections, modifications or improvements, to this document, and the products and services described herein at any time, without notice.

All ST products are sold pursuant to ST's terms and conditions of sale.

Purchasers are solely responsible for the choice, selection and use of the ST products and services described herein, and ST assumes no liability whatsoever relating to the choice, selection or use of the ST products and services described herein.

No license, express or implied, by estoppel or otherwise, to any intellectual property rights is granted under this document. If any part of this document refers to any third party products or services it shall not be deemed a license grant by ST for the use of such third party products or services, or any intellectual property contained therein or considered as a warranty covering the use in any manner whatsoever of such third party products or services or any intellectual property contained therein.

UNLESS OTHERWISE SET FORTH IN ST'S TERMS AND CONDITIONS OF SALE ST DISCLAIMS ANY EXPRESS OR IMPLIED WARRANTY WITH RESPECT TO THE USE AND/OR SALE OF ST PRODUCTS INCLUDING WITHOUT LIMITATION IMPLIED WARRANTIES OF MERCHANTABILITY, FITNESS FOR A PARTICULAR PURPOSE (AND THEIR EQUIVALENTS UNDER THE LAWS OF ANY JURISDICTION), OR INFRINGEMENT OF ANY PATENT, COPYRIGHT OR OTHER INTELLECTUAL PROPERTY RIGHT.

UNLESS EXPRESSLY APPROVED IN WRITING BY AN AUTHORIZED ST REPRESENTATIVE, ST PRODUCTS ARE NOT RECOMMENDED, AUTHORIZED OR WARRANTED FOR USE IN MILITARY, AIR CRAFT, SPACE, LIFE SAVING, OR LIFE SUSTAINING APPLICATIONS, NOR IN PRODUCTS OR SYSTEMS WHERE FAILURE OR MALFUNCTION MAY RESULT IN PERSONAL INJURY, DEATH, OR SEVERE PROPERTY OR ENVIRONMENTAL DAMAGE. ST PRODUCTS WHICH ARE NOT SPECIFIED AS "AUTOMOTIVE GRADE" MAY ONLY BE USED IN AUTOMOTIVE APPLICATIONS AT USER'S OWN RISK.

Resale of ST products with provisions different from the statements and/or technical features set forth in this document shall immediately void any warranty granted by ST for the ST product or service described herein and shall not create or extend in any manner whatsoever, any liability of ST.

ST and the ST logo are trademarks or registered trademarks of ST in various countries.

Information in this document supersedes and replaces all information previously supplied.

The ST logo is a registered trademark of STMicroelectronics. All other names are the property of their respective owners.

© 2010 STMicroelectronics - All rights reserved

STMicroelectronics group of companies

Australia - Belgium - Brazil - Canada - China - Czech Republic - Finland - France - Germany - Hong Kong - India - Israel - Italy - Japan - Malaysia - Malta - Morocco - Philippines - Singapore - Spain - Sweden - Switzerland - United Kingdom - United States of America

www.st.com

