xTIMEcomposer User Guide

Document Number: XM009801A

Tools Version: 14.0.x

Publication Date: 2015/10/29
XMOS © 2015, All Rights Reserved.

XMOS

Table of Contents

A Installation 3

1 System requirements for running the xTIMEcomposer 4

2 Installation Instructions 5
2.1 Install the tools . . . 5
2.2 Install the USB drivers . . . 6

B Quick Start 7

3 Get started with xTIMEcomposer 8
3.1 Start xTIMEcomposer Studio . . . 8
3.1.1 Register xTIMEcomposer . . . 8
3.2 Start the command-line tools . . . 10
3.3 Welcome window . . . 11
3.4 Perspective icons . . . 12
3.5 Help Perspective . . . 12

4 Frequently used commands 14
4.1 XCC . . . 14
4.2 XRUN . . . 14
4.3 XGDB . . . 15
4.4 XSIM . . . 15

5 XMOS Programming Guide 16

C Developing in the XDE 17

6 Creating a new project in xTIMEcomposer 18

7 Sharing projects and code in xTIMEcomposer 20
7.1 Import a library . . . 21
7.2 Import an example . . . 21
7.3 Import an xTIMEcomposer project using a ZIP file . . . 21
7.4 Export an xTIMEcomposer project as a ZIP file . . . 21

D Compilation 23

8 Use xTIMEcomposer Studio to build a project 24

9 XCC Pragma Directives 25

10 XCC command-line options 27
10.1 Overall Options . . . 27
10.2 Warning Options . . . 30
10.3 Debugging Options . . . 33

10.4 Optimization Options . . . 34
10.5 Preprocessor Options . . . 34
10.6 Linker And Mapper Options . . . 35
10.7 Directory Options . . . 36
10.8 Environment Variables Affecting XCC . . . 36
10.9 Board Support Provided by <platform.h> . . . 37

11 Using XMOS Makefiles **38**
11.1 Applications and Modules . . . 38
11.1.1 Workspace structure and automatic module detection . . . 39
11.2 The Application Makefile . . . 40
11.2.1 Combined xCORE/ARM applications . . . 41
11.3 The module_build_info file . . . 43
11.3.1 Modules for combined xCORE/ARM applications . . . 43

12 Using XMOS Makefiles to create binary libraries **45**
12.1 The module_build_info file . . . 45
12.2 The module Makefile . . . 46
12.3 Using the module . . . 46

E Timing **47**

13 Use xTIMEcomposer to time a program **48**
13.1 Launch the timing analyzer . . . 48
13.2 Time a section of code . . . 49
13.2.1 Visualize a route . . . 50
13.2.2 The Visualizations view . . . 50
13.3 Specify timing requirements . . . 51
13.4 Add program execution information . . . 51
13.4.1 Refine the worst-case analysis . . . 52
13.5 Validate timing requirements during compilation . . . 52

14 Use the XTA from the command line **54**
14.1 Frequently used commands . . . 54
14.1.1 Loading a binary . . . 54
14.1.2 Routes . . . 54
14.1.3 Endpoints . . . 54
14.1.4 Adding endpoints to source . . . 55
14.1.5 Timing between endpoints . . . 55
14.1.6 Timing functions . . . 55
14.1.7 Timing loops . . . 56
14.1.8 Setting timing requirements . . . 56
14.2 Viewing results . . . 56
14.2.1 Route IDs . . . 56
14.2.2 Node IDs . . . 56
14.2.3 Summary . . . 56
14.2.4 Structure . . . 57
14.2.5 Source code annotation . . . 57
14.2.6 Instruction traces . . . 57
14.2.7 Fetch no-ops . . . 58
14.2.8 Scaling Results . . . 58

14.2.9 Unknowns . . . 58
14.3 Refining timing results . . . 58
14.3.1 Exclusions . . . 59
14.3.2 Loop Iterations . . . 60
14.3.3 Loop path iterations . . . 61
14.3.4 Loop scope . . . 62
14.3.5 Instruction times . . . 64
14.3.6 Function times . . . 64
14.3.7 Path times . . . 65
14.3.8 Active tiles . . . 67
14.3.9 Node frequency . . . 67
14.3.10 Number Of logical cores . . . 67
14.4 Program structure . . . 67
14.4.1 Compiling for the XTA . . . 67
14.4.2 Structural nodes . . . 68
14.4.3 Identifying nodes: code references . . . 69
14.4.4 Reference Classes . . . 70
14.4.5 Back trails . . . 71
14.4.6 Scope of references . . . 72
14.5 Automating the process . . . 73
14.5.1 Writing a script . . . 73
14.5.2 Running a script . . . 73
14.5.3 Embedding commands into source . . . 74
14.6 Scripting XTA via the Jython interface . . . 74

15 XTA command-line manual **75**
15.1 Commands . . . 75
15.1.1 add . . . 75
15.1.2 analyze . . . 77
15.1.3 config . . . 77
15.1.4 clear . . . 79
15.1.5 debug . . . 80
15.1.6 echo . . . 83
15.1.7 exit . . . 83
15.1.8 help . . . 83
15.1.9 history . . . 83
15.1.10 load . . . 84
15.1.11 list . . . 84
15.1.12 print . . . 88
15.1.13 pwd . . . 89
15.1.14 remove . . . 90
15.1.15 scripter . . . 91
15.1.16 set . . . 92
15.1.17 source . . . 94
15.1.18 status . . . 94
15.1.19 version . . . 94
15.2 Pragmas . . . 94
15.3 Timing Modes . . . 95
15.4 Loop Scopes . . . 97
15.5 Reference Classes . . . 97
15.5.1 FUNCTION . . . 97
15.5.2 BRANCH . . . 97

15.5.3 INSTRUCTION . . . 99
15.5.4 ENDPOINT . . . 99
15.5.5 ANY . . . 101
15.5.6 FUNCTION_WITH_EVERYTHING . . . 103
15.5.7 BRANCH_WITH_EVERYTHING . . . 103
15.5.8 INSTRUCTION_WITH_EVERYTHING . . . 105
15.5.9 ENDPOINT_WITH_EVERYTHING . . . 105
15.5.10 ANY_WITH_EVERYTHING . . . 107
15.6 XTA Jython interface . . . 109
15.6.1 Load methods . . . 109
15.6.2 Route creation/deletion methods . . . 109
15.6.3 Add/remove methods . . . 110
15.6.4 Set methods . . . 110
15.6.5 Get methods . . . 111
15.6.6 Config methods . . . 111
15.7 Code reference grammar . . . 112

F Run on Hardware 114

16 Use xTIMEcomposer to run a program 115
16.1 Create a Run Configuration . . . 115
16.2 Re-run a program . . . 116

17 XRUN Command-Line Manual 117
17.1 Overall Options . . . 117
17.2 Target Options . . . 117
17.3 Debugging Options . . . 118
17.4 xSCOPE Options . . . 119

G Application Instrumentation and Tuning 120

18 Use xTIMEcomposer and xSCOPE to trace data in real-time 121
18.1 XN File Configuration . . . 121
18.2 Instrument a program . . . 122
18.3 Configure and run a program with tracing enabled . . . 123
18.4 Analyze data offline . . . 124
18.5 Analyze data in real-time . . . 125
18.5.1 Capture control . . . 125
18.5.2 Signal Control . . . 126
18.5.3 Trigger Control . . . 127
18.5.4 Timebase Control . . . 127
18.5.5 Screen Control . . . 128
18.6 Trace using the UART interface . . . 128

19 xSCOPE performance figures 130
19.1 Transfer rates between the xCORE Tile and XTAG-2 . . . 130
19.2 Transfer rates between the XTAG-2 and Host PC . . . 130

20 xSCOPE Library API 131
20.1 Functions . . . 131

20.2 Enumerations 139

H Simulation **141**

21 Use xTIMEcomposer to simulate a program **142**
21.1 Configure the simulator 142
21.2 Trace a signal 143
21.2.1 Enable signal tracing 143
21.2.2 View a trace file 144
21.2.3 View a signal 144
21.3 Set up a loopback 145
21.4 Configure a simulator plugin 146

22 xSIM command-line manual **147**
22.1 Overall Options 147
22.2 Warning Options 147
22.3 Tracing Options 148
22.4 Loopback Plugin Options 150
22.5 xSCOPE Options 151

23 XSIM Testbench and Plugin Interfaces **152**
23.1 Implementing a Plugin 153
23.2 Plugin Notifications 154
23.3 Implementing a testbench 155
23.4 Plugin API 155
23.4.1 Interfacing with the Simulator 157
23.5 Testbench API 161
23.5.1 Interfacing with a Simulator 163

I Debugging **167**

24 Use xTIMEcomposer to debug a program **168**
24.1 Launch the debugger 169
24.2 Control program execution 169
24.3 Examine a suspended program 170
24.4 Set a breakpoint 172
24.5 View disassembled code 173

25 Debug with printf in real-time **174**
25.1 Redirect `stdout` and `stderr` to the xTAG 175
25.2 Run a program with xTAG output enabled 176
25.3 Output using the UART interface 176

J Flash Programming **178**

26 Design and manufacture systems with flash memory **179**
26.1 Boot a program from flash memory 179
26.2 Generate a flash image for manufacture 180
26.3 Perform an in-field upgrade 180

26.3.1 Write a program that upgrades itself . . . 180
26.3.2 Build and deploy the upgrader . . . 182
26.4 Customize the flash loader . . . 182
26.4.1 Build the loader . . . 183
26.4.2 Add additional images . . . 183

27 libflash API **184**
27.1 General Operations . . . 184
27.2 Boot Partition Functions . . . 186
27.3 Data Partition Functions . . . 188
27.3.1 Page-Level Functions . . . 189
27.3.2 Sector-Level Functions . . . 190

28 libquadflash API **192**
28.1 General Operations . . . 192
28.2 Boot Partition Functions . . . 194
28.3 Data Partition Functions . . . 196
28.3.1 Page-Level Functions . . . 197
28.3.2 Sector-Level Functions . . . 198

29 List of devices natively supported by libflash **200**

30 List of devices natively supported by libquadflash **201**

31 Add support for a new flash device **202**
31.1 Libflash Device ID . . . 203
31.2 Page Size and Number of Pages . . . 203
31.3 Address Size . . . 204
31.4 Clock Rate . . . 204
31.5 Read Device ID . . . 205
31.6 Sector Erase . . . 206
31.7 Write Enable/Disable . . . 206
31.8 Memory Protection . . . 207
31.9 Programming Command . . . 208
31.10 Read Data . . . 209
31.11 Sector Information . . . 209
31.12 Status Register Bits . . . 210
31.13 Add Support to xTimeComposer . . . 211
31.14 Select a Flash Device . . . 212

32 XFLASH Command-Line Manual **213**
32.1 Overall Options . . . 213
32.2 Target Options . . . 214
32.3 Security Options . . . 215
32.4 Programming Options . . . 215

K Security and OTP Programming **217**

33 Safeguard IP and device authenticity **218**
33.1 The xCORE AES module . . . 219
33.2 Develop with the AES module enabled . . . 220

33.3 Production flash programming flow . . . 221
33.4 Production OTP programming flow . . . 222

34 XBURN Command-Line Manual 223
34.1 Overall Options . . . 223
34.2 Target Options . . . 223
34.3 Security Options . . . 224
34.4 Security Register Options . . . 224
34.5 Programming Options . . . 225

L Programming xCORE-XA devices 227

35 xTIMEcomposer support for xCORE-XA devices and boards 228
35.1 3rd-party ARM tool chain . . . 228
35.2 3rd-party ARM libraries . . . 229
35.3 Additional software requirements . . . 229
35.4 Library support for communication and device boot . . . 229
35.4.1 xCORE-ARM bridge library . . . 229
35.4.2 xCORE-ARM boot library . . . 229
35.5 Multi architecture project support . . . 230
35.5.1 Project structure . . . 230
35.5.2 Makefile extensions . . . 230
35.5.3 Building projects . . . 230
35.6 xTIMEcomposer support . . . 230
35.7 Setting up the ARM GDB Server . . . 230
35.8 Running applications on hardware . . . 232
35.8.1 Command line . . . 232
35.8.2 xTIMEcomposer Studio . . . 232
35.9 Debugging applications on hardware . . . 233
35.9.1 Command line . . . 233
35.9.2 xTIMEcomposer Studio . . . 233
35.10 Using xTIMEcomposer launch groups . . . 233
35.11 Deploying applications into xCORE-XA flash memory . . . 234

36 xCORE ARM bridge library 235
36.1 xCORE ARM bridge . . . 235
36.1.1 Features . . . 235
36.2 Functional overview . . . 236
36.3 ARM API . . . 238
36.3.1 Interrupt usage . . . 240
36.3.2 DMA usage . . . 241
36.4 xCORE API . . . 243
36.4.1 The interface APIs . . . 245

37 xCORE ARM boot library 247
37.1 Functional overview . . . 247
37.2 Booting an xCORE from the ARM core . . . 248
37.3 ARM API . . . 248
37.3.1 ARM API . . . 248

M Programming in C/XC 251

38 Calling between C/C++ and XC 252
38.1 Passing arguments from XC to C/C++ . . . 252
38.2 Passing arguments from C/C++ to XC . . . 252

39 XC Implementation-Defined Behavior 253

40 C Implementation-Defined Behavior 255
40.1 Environment . . . 255
40.2 Identifiers . . . 256
40.3 Characters . . . 256
40.4 Floating point . . . 257
40.5 Hints . . . 257
40.6 Preprocessing directives . . . 257
40.7 Library functions . . . 257
40.8 Locale-Specific Behavior . . . 261

41 C and C++ Language Reference 264
41.1 Standards . . . 264
41.2 Books . . . 264
41.3 Online . . . 264

42 XC Library 265
42.1 print.h functions . . . 265
42.2 safestring.h functions . . . 270
42.3 xccompat.h typedefs . . . 276

N Programming in Assembly 278

43 Inline Assembly 279

44 Make assembly programs compatible with the XMOS XS1 ABI 281
44.1 Symbols . . . 281
44.2 Alignment . . . 281
44.3 Sections . . . 282
44.3.1 Data . . . 282
44.3.2 Arrays . . . 283
44.4 Functions . . . 283
44.4.1 Parameters and return values . . . 283
44.4.2 Caller and callee save registers . . . 284
44.4.3 Resource usage . . . 284
44.4.4 Side effects . . . 285
44.5 Elimination blocks . . . 286
44.6 Typestrings . . . 286
44.7 Example . . . 287

45 Using the XTA With Assembly 289
45.1 Assembly Directives . . . 289
45.2 Branch Table Example . . . 289
45.3 Core Start/Stop Example . . . 290

46 Assembly Programming Manual **291**
46.1 Lexical Conventions . . . 291
46.1.1 Comments . . . 291
46.1.2 Symbol Names . . . 291
46.1.3 Directives . . . 291
46.1.4 Constants . . . 292
46.2 Sections and Relocations . . . 292
46.3 Symbols . . . 292
46.3.1 Attributes . . . 292
46.4 Labels . . . 293
46.5 Expressions . . . 293
46.6 Directives . . . 294
46.6.1 add_to_set . . . 294
46.6.2 max_reduce, sum_reduce . . . 295
46.6.3 align . . . 295
46.6.4 ascii, asciiz . . . 295
46.6.5 byte, short, int, long, word . . . 296
46.6.6 file . . . 296
46.6.7 loc . . . 296
46.6.8 weak . . . 297
46.6.9 globl, global, extern, locl, local . . . 297
46.6.10 globalresource . . . 298
46.6.11 typestring . . . 298
46.6.12 ident, core, corerev . . . 298
46.6.13 section, pushsection, popsection . . . 299
46.6.14 text . . . 300
46.6.15 set, linkset . . . 300
46.6.16 cc_top, cc_bottom . . . 301
46.6.17 scheduling . . . 301
46.6.18 issue_mode . . . 302
46.6.19 syntax . . . 302
46.6.20 assert . . . 302
46.6.21 Overlay Directives . . . 302
46.6.22 Language Directives . . . 303
46.6.23 XMOS Timing Analyzer Directives . . . 304
46.6.24 uleb128, sleb128 . . . 305
46.6.25 space, skip . . . 305
46.6.26 type . . . 305
46.6.27 size . . . 305
46.6.28 jmptable, jmptable32 . . . 306
46.7 Instructions . . . 307
46.7.1 Data Access . . . 309
46.7.2 Branching, Jumping and Calling . . . 310
46.7.3 Data Manipulation . . . 310
46.7.4 Concurrency and Thread Synchronization . . . 311
46.7.5 Communication . . . 312
46.7.6 Resource Operations . . . 312
46.7.7 Event Handling . . . 313
46.7.8 Interrupts, Exceptions and Kernel Calls . . . 313
46.7.9 Debugging . . . 313
46.7.10 Pseudo Instructions . . . 314
46.8 Assembly Program . . . 315

O Programming for XS1 Devices 316

47 XCC Target-Dependent Behavior for XS1 Devices 317
47.1 Support for Clock Blocks . . . 317
47.2 Support for Ports . . . 318
47.2.1 Serialization . . . 318
47.2.2 Timestamping . . . 318
47.2.3 Changing Direction of Buffered Ports . . . 319
47.3 Channel Communication . . . 319

48 XS1 Data Types 320

49 XS1 port-to-pin mapping 321

50 XS1 Library 323
50.1 Data types . . . 323
50.2 Port Configuration Functions . . . 324
50.3 Clock Configuration Functions . . . 343
50.4 Port Manipulation Functions . . . 350
50.5 Clock Manipulation Functions . . . 356
50.6 Logical Core/Tile Control Functions . . . 357
50.7 Channel Functions . . . 372
50.8 Predicate Functions . . . 381
50.9 XS1-S Functions . . . 384
50.10 Miscellaneous Functions . . . 386

51 xCORE 32-Bit Application Binary Interface 393

P Platform Configuration 394

52 Describe a target platform 395
52.1 Supported network topologies . . . 395
52.2 A board with two packages . . . 395

53 xConnect link naming convention manual 400
53.1 xCORE “L” series example . . . 400
53.2 xCORE-200 series example . . . 401

54 XN Specification 402
54.1 Network Elements . . . 402
54.2 Declaration . . . 402
54.3 Package . . . 403
54.4 Node . . . 404
54.4.1 Tile . . . 405
54.4.2 Port . . . 406
54.4.3 Boot . . . 406
54.4.4 Source . . . 406
54.4.5 Bootee . . . 407
54.4.6 Bit . . . 407
54.4.7 Link . . . 407
54.4.8 Service . . . 408

54.4.9 Chanend 408
54.5 Link 409
54.5.1 LinkEndpoint 409
54.6 Device 410
54.6.1 Attribute 410
54.7 JTAGDevice 412

Part A

Installation

CONTENTS

- System requirements for running the xTIMEcomposer
- Installation Instructions

1 System requirements for running the xTIMEcomposer

The xTIMEcomposer tools are officially supported on the following platforms:

Windows XP SP3

- *32-bit with 32-bit JRE*

Windows 7 SP 1

- *32-bit with 32-bit JRE*
- *64-bit with 32-bit JRE*

Windows 8

- *32-bit with 32-bit JRE*
- *64-bit with 32-bit JRE*

Windows 10

- *32-bit with 32-bit JRE*
- *64-bit with 32-bit JRE*

Mac OS X 10.6 +

- *Intel Processors*

Linux CentOS 5.8 and Ubuntu 12.04 LTS

- *32-bit with 32-bit JRE*
- *64-bit with 64-bit JRE*

The tools also work on many other versions of Linux, including RedHat and Ubuntu. For up-to-date information on known compatibility issues, see:

- http://www.xmos.com/tools

You must also have a Java Runtime Environment (JRE) version 1.5 or later installed, which can be downloaded from:

- http://java.sun.com/javase/downloads

2 Installation Instructions

IN THIS CHAPTER

- Install the tools
- Install the USB drivers

xTIMEcomposer and related drivers are provided in a single platform-specific downloadable file.

2.1 Install the tools

To install the tools on your PC, follow these steps:

On Windows:

1. Download the Windows installer from:
   - http://www.xmos.com/tools
2. Double-click the installer to run it. Follow the on-screen prompts to install the tools on your PC.

On Mac:

1. Download the Macintosh installer from:
   - http://www.xmos.com/tools
2. Double-click the downloaded installer to open it, and then drag the xTIMEcomposer icon into your Applications folder.

   The installer copies the files to your hard disk.
3. Unmount the installer.

On Linux:

1. Download the Linux archive from:
   - http://www.xmos.com/tools
2. Uncompress the archive to an installation directory, for example by entering the following command:
   - `tar -xzf archive.tgz -C /home/`*user*

2.2 Install the USB drivers

xTIMEcomposer interfaces to development boards over USB. Some boards provide a completely integrated debugger with the xCORE device, while others require an external xTAG adapter that connects to the board via an XSYS connector.

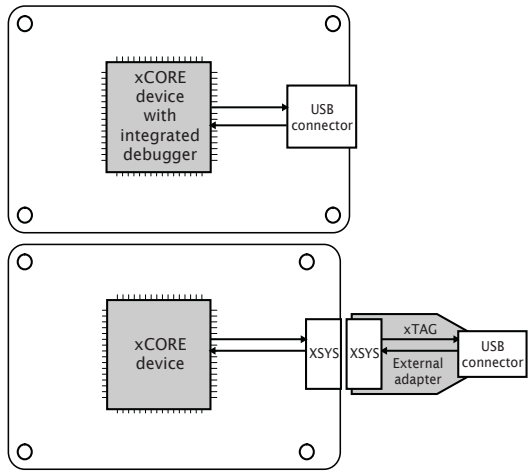


Figure 1: Adapter configurations used with xCORE development boards

Consult your board manual to determine which driver to use.

On Windows:

The JTAG drivers are installed by the tools installer. Plug your xCORE development board in after an installation to load the drivers.

On Mac:

USB driver support is provided natively on OS X.

On Linux:

USB driver support is provided natively on some versions of Linux. In some cases the driver must be enabled, see *Enable USB drivers on Linux*[1]

[1] http://www.xmos.com/published/enable-usb-drivers-linux

Part B

Quick Start

CONTENTS

- Get started with xTIMEcomposer
- Frequently used commands
- XMOS Programming Guide

3 Get started with xTIMEcomposer

IN THIS CHAPTER

- Start xTIMEcomposer Studio
- Start the command-line tools
- Welcome window
- Perspective icons
- Help Perspective

3.1 Start xTIMEcomposer Studio

To start xTIMEcomposer Studio:

In Windows:

Choose **Start** ▶ **Programs** ▶ **XMOS** ▶ **xTIMEcomposer_14** ▶ **xtimecomposer**.

In **OS X**:

Open a new Finder window, navigate to the **Applications** folder, open the folder **XMOS_xTIMEcomposer_14** and double-click on the **xtimecomposer.app** icon.

In Linux:

Open a terminal window, change to the installation directory and enter the following commands:

- `source SetEnv`
- `xtimecomposer`

3.1.1 Register xTIMEcomposer

The first time you start xTIMEcomposer Studio, you are required to register the tools with your XMOS account so you must be connected to the internet.

Once you have registered xTIMEcomposer you can use the tools offline although we recommend that you remain connected as often as possible in order to download the latest versions of software and documentation.

Registration provides benefits such as automatic notifications of document and software updates directly within the Studio, and the option to manage account settings from within the tools.

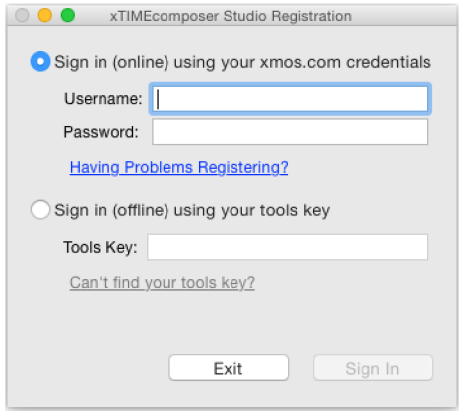


Figure 2: xTIMEcomposer Studio registration window

If you have problems registering, click the **Having problems registering** link in the xTIMEcomposer Studio Registration window. This window has four options to help you get started:

- **Forgotten your password** - Opens a page on the xmos.com website that you can use to request a new password.
- **Configure the proxy server settings** - Opens the system *Network Connections* window so you can configure your proxy settings.
- **Disable security certificate checking** - Opens the *xTIMEcomposer Preferences* window where you can disable security certificate checking.

You can also sign into the tools using the **Sign in using your tools key** option if you have no internet connection. Your tools key is emailed to you when you register on xmos.com. It can also be found on the Account tab of your MyXMOS account.

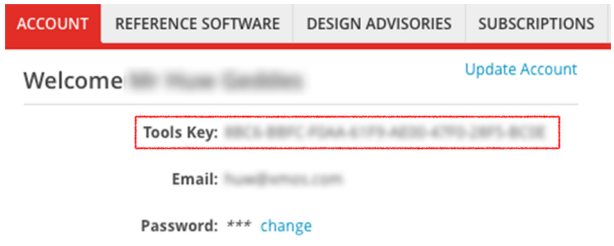


Figure 3: MyXMOS Account: Tools Key

3.2 Start the command-line tools

The xTIMEcomposer command-line tools use a set of environment variables when searching for header files, libraries and target devices (see §10.8). To add the xTIMEcomposer tools to the path and configure the default set of environment variables:

In **Windows**:

Choose **Start** ▶ **Programs** ▶ **XMOS** ▶ **xTIMEcomposer_14** ▶ **Command Prompt**.

In **OS X**:

Open a Terminal window, change to the installation directory and enter the following command:

- `SetEnv.command`

In **Linux**:

Open a Terminal window, change to the installation directory and enter the following command:

- `source SetEnv`

You can now run any of the tools by entering its name and command-line options. Some of the most common commands are summarized in the following section.

3.3 Welcome window

The **Welcome** window in xTIMEcomposer Studio provides a convenient starting point for all users, including developers who are new to XMOS and experienced users.

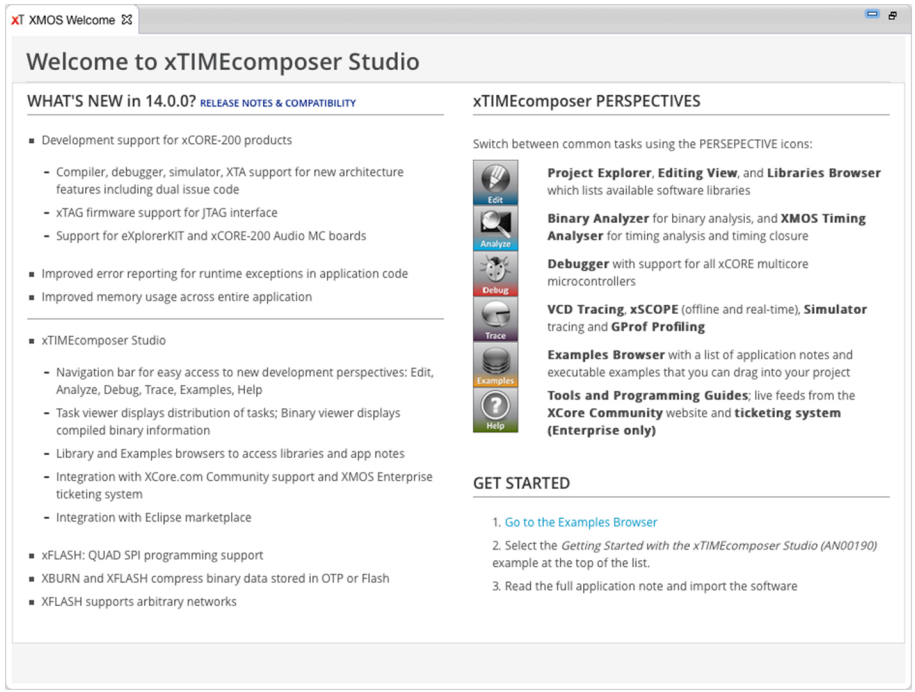


Figure 4: xTIMEcomposer Studio Welcome window

Developers with an xCORE development board, can use the page to check their board is working correctly, download firmware and find kit-specific documentation or tutorials. Those who do not have a board you can follow a tools tutorial using the simulator or download xTIMEcomposer related documentation and programming guides.

In xTIMEcomposer Studio, choose **Help ▶ Welcome** to view the Welcome window at any time.

3.4 Perspective icons

The Eclipse interface has been updated to make it easier to switch between common tasks using the *perspective* icons displayed on the left of the xTIMEcomposer window.

EDIT: *Project Explorer*, *Editing View*, and *Libraries Browser* which lists available software libraries.

ANALYZE: *Binary Analyzer* for binary analysis, and *XMOS Timing Analyzer* for timing analysis and timing closure.

DEBUG: *Debugger* with support for all xCORE multicore microcontrollers.

TRACE: *VCD Tracing*, *xSCOPE* (offline and real-time), *Simulator* tracing and *GProf Profiling*.

EXAMPLES: *Examples Browser* with a list of application notes and executable examples that you can drag into your project.

HELP: *Tools and Programming Guides*; live feeds from the XCore Community website and ticketing system (Enterprise only).

3.5 Help Perspective

The xTIMEcomposer *Help Perspective* contains a web bowser that displays all the information developers need when writing real-time multicore applications, including:

- a live view of the XMOS website (xmos.com), allowing users to download the latest documentation from within the development tools, as well as managing their XMOS account;
- a live link to the XCore Community Q&A website (Community tools) or XMOS Ticketing system (Enterprise Tools).

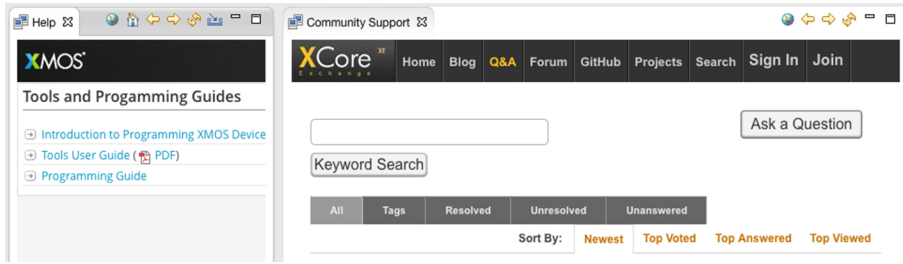


Figure 5: xTIMEcomposer Studio Help Perspective

Developers navigate the content using the toolbar, which provides the following options:

Home loads the default Developer Column home page that displays content related to the tools, development boards and silicon devices.

Local Home loads the home page for xTIMEcomposer and programming content that has been downloaded and stored offline using the *Update offline content* option.

Back and Forward navigate between the next and previous web pages in the Developer Column.

Refresh reloads the current web page.

Update offline content downloads to your local drive the documentation for xTIMEcomposer and programming C applications so they can be used in the Developer Column when xTIMEcomposer is offline.

The *Developer Column* can be displayed at any time using **Window ▶ Show View ▶ Developer Column**.

4 Frequently used commands

IN THIS CHAPTER

- XCC
- XRUN
- XGDB
- XSIM

This document summarizes a number of frequently-used commands that can be run using the command line.

4.1 XCC

To compile a program for your development board, enter the following commands:

1. `xcc -print-targets`

   XCC displays a list of supported development boards.

2. `xcc <file> -target=<board> -o <binary>`

   XCC compiles the file, generating an executable binary for your target board.

4.2 XRUN

To load a compiled program onto your development board, enter the following commands:

1. `xrun -l`

   XRUN prints an enumerated list of all JTAG adapters connected to your PC and the devices on each JTAG chain, in the form:

   ```
   ID        Name        Adapter ID        Devices
   --        ----        ----------        -------
   ```

2. `xrun --id <n> --io <binary>`

   XRUN loads your binary onto the hardware connected to the adapter with the specified ID.

   The `--io` option causes XRUN to remain connected to the adapter, providing the standard output stream from your hardware to the terminal.

4.3 XGDB

To compile and debug your program, enter the following commands:

1. `xcc <file> -target=<board> -o <binary> -g`

   XCC compiles your file with debugging information enabled.

2. `xgdb bin.xe`

   GDB loads with a prompt.

3. `list-devices`

   GDB prints an enumerated list of all JTAG adapters connected to your PC and the devices on each JTAG chain, in the form:

   ```
   ID        Name         Adapter ID        Devices
   --        ----         ----------        -------
   ```

4. `connect --id <id>`

   GDB connects to your target hardware.

5. `load`

   GDB loads your binary.

6. `break main`

   GDB adds a breakpoint to the function main.

7. `continue`

   GDB runs the program until it reaches main.

4.4 XSIM

To run your program on the simulator, enter the following command:

- `xsim <binary>`

To launch the simulator from within the debugger, at the GDB prompt enter the command:

- `connect -s`

You can then load your program onto the simulator in the same way as if using a development board.

5 XMOS Programming Guide

To help you access the real-time hardware features of XMOS multicore microcontrollers, XMOS has published a separate *XMOS Programming Guide*. This document shows how to use the additional multicore language extensions for C (which collectively form the xC language) to handle task based parallelism and communication, accurate timing and I/O, and safe memory management.

The document contains many examples, including a simple Hello World application, handling I/O and button presses, a Tic-Tac-Toe game, an example of how to use safe pointers for string processing, and a double buffering example.

To view and download the *XMOS Programming Guide* go to:

http://www.xmos.com/published/xmos-programming-guide

Part C

Developing in the XDE

CONTENTS

- Creating a new project in xTIMEcomposer
- Sharing projects and code in xTIMEcomposer

6 Creating a new project in xTIMEcomposer

When you first open xTIMEcomposer, the Project Explorer and Exiting view are empty. To get started you need to create a project to store your files in, which xTIMEcomposer Studio can build and run on the target hardware or simulator.

1. Select **File ▶ New ▶ xTIMEcomposer Project** ( ).
2. Enter a **Name** for your project, and select a **Workspace**.

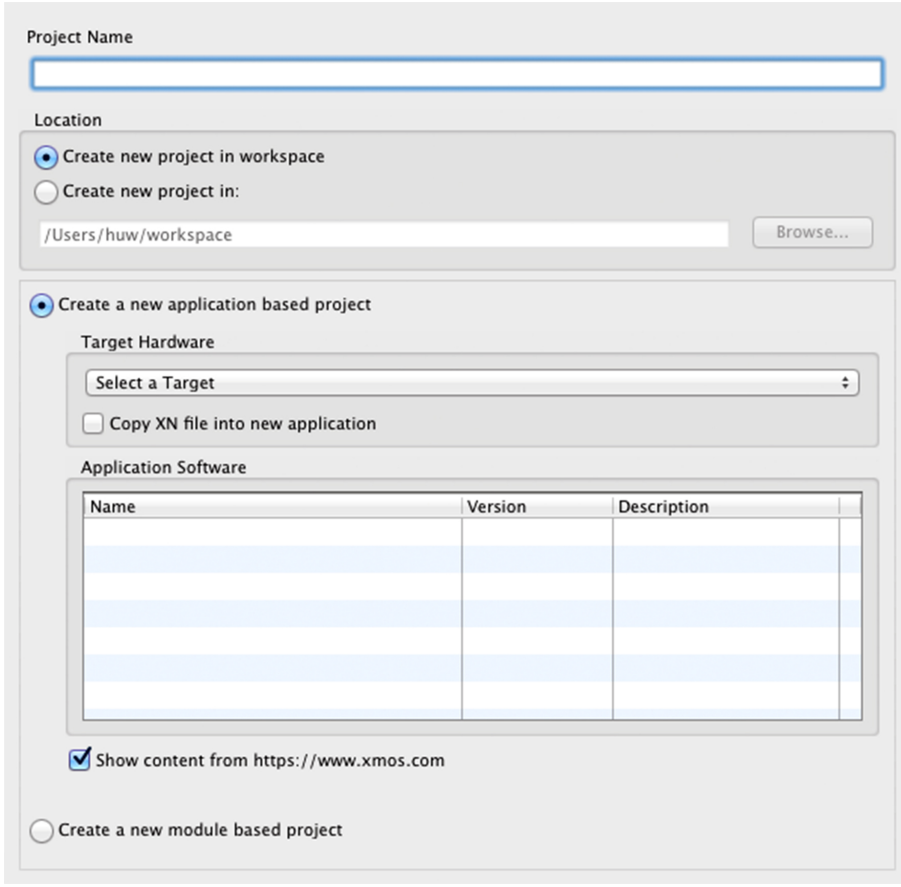


Figure 6: Create xTIMEcomposer Project window

3. Select your **Target Hardware**. This might be an xCORE device or a development board. If you don't have the hardware available you can develop your application and run it on the simulator.

4. xTIMEcomposer Studio includes an XN file for all the hardware targets, which defines the system configuration and boot process. If you want to edit the XN file select **Copy XN file into new application**. You can import the XN file after you create the project if necessary.

5. Select **Application Software ▶ Empty XC File** to create a project with an empty source file.

6. Click **Finish** to create your project and source file.

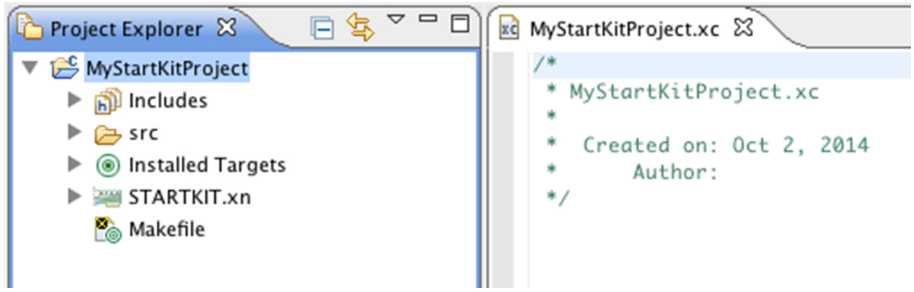


Figure 7: xTIMEcomposer Project Explorer and empty XC file

The Makefile at the bottom of the project tree is used to configure the compiler options for your project. Double-click the **Makefile** to open it in the *Editor view*.

You can now start editing the empty XC source file, or import a libraries/example/application note.

7 Sharing projects and code in xTIMEcomposer

IN THIS CHAPTER

- Import a library
- Import an example
- Import an xTIMEcomposer project using a ZIP file
- Export an xTIMEcomposer project as a ZIP file

xTIMEcomposer Studio has a set of views that you can use to import software into your project from different sources including:

- **Libraries**: browse the complete set of XMOS libraries and then drag them into your project. The Libraries view appears on the *Edit perspective*.

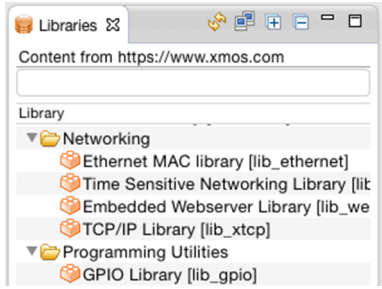


Figure 8: Libraries browser

- **Examples**: browse a library of application notes and examples that show how to do the most common tasks in multicore applications. Many of the examples have executable code that you can drag into your project.

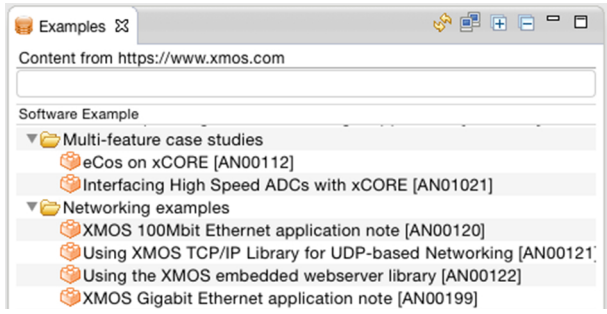


Figure 9: Examples browser

7.1 Import a library

To import a library follow these steps:

1. Select the **Libraries** view in the bottom left corner of the xTIMEcomposer window (**Window ▶ Show View ▶ Libraries**).
2. Double-click the library you want to import.
3. Select the version you require. The latest version is the default option.
4. Click **Finish**.
   Documentation for the library is displayed in the *Libraries Info* window.

7.2 Import an example

To import a example follow these steps:

1. Select the **Examples** perspective.
2. Double-click the example you want to import.
3. Select the version you require. The latest version is the default option.
4. Click **Finish**.
   Documentation for the example is displayed in the *Examples info* window.

7.3 Import an xTIMEcomposer project using a ZIP file

To import a project follow these steps:

1. Choose **File ▶ Import**.
2. Double-click on the **General** option, select **Existing Projects into Workspace** and click **Next**.
3. In the **Import** dialog box, click **Browse** (next to the **Select archive file** text box).
4. Select the archive to import and click **Open**.
5. Click **Finish**.

7.4 Export an xTIMEcomposer project as a ZIP file

To export a project follow these steps:

1. Choose **File ▶ Export**.
2. Double-click on the **General** option, select **Archive File** and click **Next**.

3. Select the projects you wish to export in the top-left panel. You can exclude files by deselecting them in the top-right panel.
4. Enter a name for the archive in the **To archive file** text box.
5. Click **Finish**.

Part D

Compilation

CONTENTS

- Use xTIMEcomposer Studio to build a project
- XCC Pragma Directives
- XCC command-line options
- Using XMOS Makefiles
- Using XMOS Makefiles to create binary libraries

8 Use xTIMEcomposer Studio to build a project

To build your project, select your project in the **Project Explorer**, click the arrow next to the **Build** buttonand select either **Debug** or **Release**.

xTIMEcomposer uses the Makefile in your project to determine the configuration settings used with the compiler.

Double-click the project **Makefile** in the *Project Explorer* to open it in the Makefile Editor, where you can set the compiler options. The *XCC Command-Line Manual* (see §10) lists all supported compiler options.

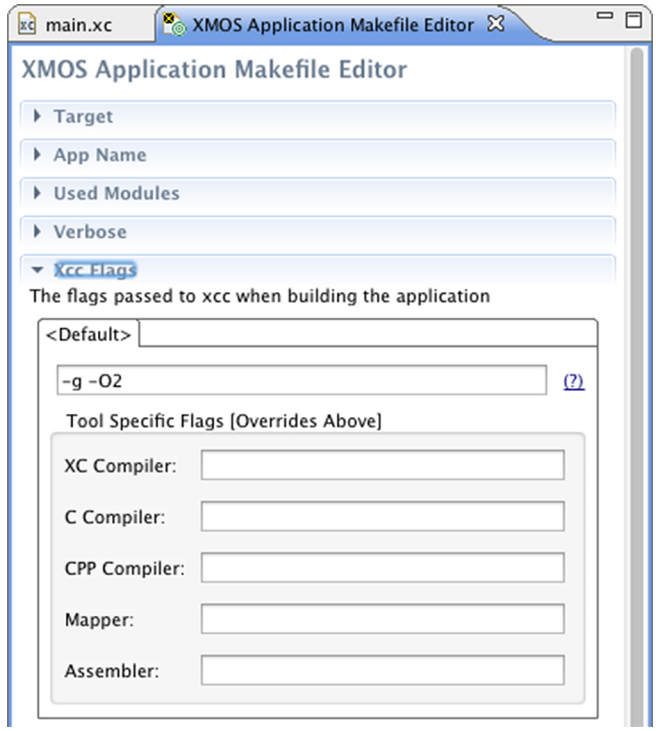


Figure 10: Makefile Editor

If there are no errors in your program, xTIMEcomposer adds the compiled binary file to the **Binaries** folder in your project.

Errors are reported in the **Console**. Double-click a message highlighted red to locate it in the editor.

9 XCC Pragma Directives

xTimeComposer supports the following pragmas.

`#pragma unsafe arrays`

(XC Only) This pragma disables generation of run-time safety checks that prevent dereferencing out-of-bounds pointers and prevent indexing invalid array elements. If the pragma appears inside a function it applies to the body of the next `do`, `while` or `for` statement in that function. If the pragma appears outside a function the scope it applies to the body of the next function definition.

`#pragma loop unroll (n)`

(XC only) This pragma controls the number of times the next `do`, `while` or `for` loop in the current function is unrolled. `n` specifies the number of iterations to unroll, and unrolling is performed only at optimization level 01 and higher. Omitting the `n` parameter causes the compiler to try and fully unroll the loop. Outside of a function the pragma is ignored. The compiler produces a warning if unable to perform the unrolling.

`#pragma stackfunction n`

This pragma allocates `n` words (`int`s) of stack space for the next function declaration in the current translation unit.

`#pragma stackcalls n`

(XC only) This pragma allocates `n` words (`int`s) of stack space for any function called in the next statement. If the next statement does not contain a function call then the pragma is ignored; the next statement may appear in another function.

`#pragma ordered`

(XC only) This pragma controls the compilation of the next `select` statement. This select statement is compiled in a way such that if multiple events are ready when the select starts, cases earlier in the select statement are selected in preference to ones later on.

`#pragma select handler`

(XC only) This pragma indicates that the next function declaration is a select handler. A select handler can be used in a select case, as shown in the example below.

```
#pragma select handler
void f(chanend c, int &token, int &data);

...
select {
  case f(c, token, data):
    ...
    break;
}
...
```

The effect is to enable an event on the resource that is the first argument to the function. If the event is taken, the body of the select handler is executed before the body of the case.

The first argument of the select handler must have transmissive type and the return type must be `void`.

If the resource has associated state, such as a condition, then the select will not alter any of that state before waiting for events.

`#pragma fallthrough`
(XC only) This pragma indicates that the following switch case is expected to fallthrough to the next switch case without a break or return statement. This will suppress any warnings/errors from the compiler due to the fallthrough.

`#pragma xta label "name"`
This pragma provides a label that can be used to specify timing constraints.

`#pragma xta endpoint "name"`
(XC only) This pragma specifies an endpoint. It may appear before an input or output statement.

`#pragma xta call "name"`
(XC only) This pragma defines a label for a (function) call point. Use to specify a particular called instance of a function. For example, if a function contains a loop, the iterations for this loop can be set to a different value depending on which call point the function was called from.

`#pragma xta command "command"`
(XC only) This pragma allows XTA commands to be embedded into source code. All commands are run every time the binary is loaded into the XTA. Commands are executed in the order they occur in the file, but the order between commands in different source files is not defined.

`#pragma xta loop (integer)`
(XC only) This pragma applies the given loop XTA iterations to the loop containing the pragma.

10XCC command-line options

IN THIS CHAPTER

- Overall Options
- Warning Options
- Debugging Options
- Optimization Options
- Preprocessor Options
- Linker And Mapper Options
- Directory Options
- Environment Variables Affecting XCC
- Board Support Provided by `<platform.h>`

XCC is the front-end to the xCORE C, C++ and XC compilers. Typical usage results in preprocessing, compilation, assembly, linking, and mapping code and data onto tiles. Some options allow this process to be stopped at intermediate stages and other options are passed to one stage of processing. Most options have negative forms (for example, `-fno-`*option*). A space between an option and its argument is permitted.

Build settings for an application are defined in the application Makefile. Double click the Makefile in the *Project Explorer* to open it in the *Makefile Editor*.

10.1 Overall Options

The four possible stages of compilation are preprocessing, compilation proper, assembly and linking/mapping. The first three stages are applied to an individual source file, producing an object file. Linking and mapping combine the object files and an XN file into a single executable XE file, which contains the code and data segments for each tile.

`source-file` The suffix of a source file determines how it is handled by default.

Extension	Type of File	Preprocessed by XCC
`.xc`	XC source code	Y
`.c`	C source code	Y
`.cpp`	CPP source code (for compatability, the extensions `cc`, `cp`, `c++`, `C` and `cxx` are also recognized)	Y
`.S`	Assembly code	Y
`.xta`	xCORE Timing Analyzer script	N
`.xscope`	xSCOPE configuration file	N
`.xn`	xCORE Network Description	N
`.xi`	XC source code	N
`.i`	C source code	N
`.ii`	C++ source code	N
`.s`	Assembly code	N
other	Object file .o be given to the linker	N

Figure 11: File extensions recognized by XCC and their meaning

`-x`*language* Specifies the language for the following input files. This option applies to all following input files until the next `-x` option. Supported values for *language* are:

`xc`
`c`
`c++`
`assembler`
`assembler-with-cpp`
`xn`
`xta`
`xscope`
`none` (turn off language specification)

`-std=`*standard*
Specifies the language variant for the following input C or C++ file. Supported values for *standard* are:

`c89`
: ISO C89

`gnu89`
: ISO C89 with GNU extensions

`c99`
: ISO C99

`gnu99`
: ISO C99 with GNU extensions (default for C programs)

`c++98`
: ISO C++ (1998)

`gnu++98`
: ISO C++ (1998) with GNU extensions (default for C++ programs)

`-fsubword-select`
: In XC, allows selecting on channel inputs where the size of the desstination variable is less than 32 bits.

  This is default for targets based on XS1-L devices. It is not default for targets based on XS1-G devices. For further details, see §47.3.

`-target=platform`
: Specifies the target platform. The platform configuration must be specified in the file `platform.xn`, which is searched for in the paths specified by the `XCC_DEVICE_PATH` environment variable (see §10.8).

`-foverlay`
: Enable support for memory overlays. Functions marked as overlay roots are placed in external memory and are loaded on demand at runtime. The option should be passed when compiling and linking. An overlay runtime should be supplied in the application.

`-foverlay=flash`
: Enable support for memory overlays linking in the flash overlay runtime. Overlays are only enabled on tiles which boot from flash.

`-foverlay=syscall`
: Enable support for memory overlays linking in the syscall overlay runtime. Overlay are enabled on all tiles. Overlays are loaded from a host machine using a system call.

`-fxscope[=link|uart]`
: Enable support for tracing using xSCOPE (defaults to link). The XN file of the target must contain an xSCOPE link. The option should be passed when compiling and linking.

`-funroll-loops`
: Unroll loops with small iteration counts. This is enabled at -O2 and above.

`-finline-functions`
: Integrate simple functions into their callers. This is enabled at -O2 and above and also at -Os.

`-pass-exit-codes`
: Returns the numerically highest error code produced by any phase of compilation. (By default XCC returns 1 if any phase of the compiler returns non-success, otherwise it returns 0.)

`-c`
: Compiles or assembles the source files, producing an object file for each source file, but does not link/map. By default the object filename is formed by replacing the source file suffix with `.o` (for example, `a.c` produces `a.o`).

`-S` Stops after compilation proper, producing an assembly code file for each nonassembly input file specified. By default the assembly filename is formed by replacing the source file suffix with .s.

Input files not requiring compilation are ignored.

`-E` Preprocesses the source files only, outputting the preprocessed source to stdout.

Input files not requiring preprocessing are ignored.

`-ofile` Places output in *file*.

If `-o` is not specified, the executable file is placed in `a.xe`, the object file for `source.suffix` in `source.o`, its assembly code file in `source.s`, and all preprocessed C/C++/XC source on standard output.

`-v` Prints (on standard error) the commands executed at each stage of compilation. Also prints the version number of XCC, the preprocessor and the compiler proper.

`-###` The same as `-v` except that the commands are not executed and all command arguments are quoted.

`--help` Prints a description of the supported command line options. If the `-v` option is also specified, `--help` is also passed to the subprocesses invoked by XCC.

`--version` Displays the version number and copyrights.

10.2 Warning Options

Many specific warnings can be controlled with options beginning `-W`. Each of the following options has a negative form beginning `-Wno-` to turn off warnings.

`-fsyntax-only`
Checks the code for syntax errors only, then exits.

`-w` Turns off all warning messages.

`-Wbidirectional-buffered-port`
Warns about the use of buffered ports not qualified with either `in` or `out`. This warning is enabled by default.

`-Wchar-subscripts`
Warns if an array subscript has type char.

`-Wcomment` Warns if a comment-start sequence `/*` appears in a `/*` comment, or if a backslash-newline appears in a `//` comment. This is default.

`-Wimplicit-int`
Warns if a declaration does not specify a type. In C also warns about function declarations with no return type.

`-Wmain` Warns if the type of `main` is not a function with external linkage returning `int`. In XC also warns if main does not take zero arguments. In C also warns if `main` does not take either zero or two arguments of appropriate type.

`-Wmissing-braces`
Warns if an aggregate or union initializer is not fully bracketed.

`-Wparentheses`
Warns if parentheses are omitted when there is an assignment in a context where a truth value is expected or if operators are nested whose precedence people often find confusing.

`-Wreturn-type`
Warns if a function is defined with a return type that defaults to `int` or if a return statement returns no value in a function whose return type is not `void`.

`-Wswitch-default`
Warns if a `switch` statement does not have a `default` case.

`-Wswitch-fallthrough`
(XC only) Warns if a case in a `switch` statement with at least one statement can have control fall through to the following case.

`-Wtiming` Warns if timing constraints are not satisfied. This is default.

`-Wtiming-syntax`
Warns about invalid syntax in timing scripts. This is default.

`-Wunused-function`
Warns if a static function is declared but not defined or a non-inline static function is unused.

`-Wunused-parameter`
Warns if a function parameter is unused except for its declaration.

`-Wunused-variable`
Warns if a local variable or non-constant static variable is unused except for its declaration.

`-Wunused` Same as `-Wunused-function`, `-Wunused-variable` and `-Wno-unused-parameter`.

`-Wall` Turns on all of the above `-W` options.

The following `-W...` options are not implied by `-Wall`.
`-Wextra`
`-W` Prints extra warning messages for the following:

- A function can return either with or without a value (C, C++ only).

- An expression statement or left-hand side of a comma expression contains no side effects. This warning can be suppressed by casting the unused expression to `void` (C, C++ only).
- An unsigned value is compared against zero with < or <=.
- Storage-class specifiers like `static` are not the first things in a declaration (C, C++ only).
- A comparison such as `x<=y<=z` appears (XC only).
- The return type of a function has a redundant qualifier such as `const`.
- Warns about unused arguments if `-Wall` or `-Wunused` is also specified.
- A comparison between signed and unsigned values could produce an incorrect result when the signed value is converted to unsigned. (Not warned if `-Wno-sign-compare` is also specified.)
- An aggregate has an initializer that does not initialize all members.
- An initialized field without side effects is overridden when using designated initializers (C, C++ only).
- A function parameter is declared without a type specifier in K&R-style functions (C, C++ only).
- An empty body occurs in an `if` or `else` statement (C, C++ only).
- A pointer is compared against integer zero with <, <=, >, or >=. (C, C++ only).
- An enumerator and a non-enumerator both appear in a conditional expression. (C++ only).
- A non-static reference or non-static const enumerator and a non-enumerator both appear in a conditional expression (C++ only).
- Ambiguous virtual bases (C++ only).
- Subscripting an array which has been declared `register` (C++ only).
- Taking the address of a variable which has been declared `register` (C++ only).
- A base class is not initialized in a derived class' copy constructor (C++ only).

`-Wconversion`

Warns if a negative integer constant expression is implicitly converted to an unsigned type.

`-Wdiv-by-zero`

Warns about compile-time integer division by zero. This is default.

`-Wfloat-equal`
: Warns if floating point values are used in equality comparisons.

`-Wlarger-than-len`
: Warns if an object of larger than *len* bytes is defined.

`-Wpadded`
: Warns if a structure contains padding. (It may be possible to rearrange the fields of the structure to reduce padding and thus make the structure smaller.)

`-Wreinterpret-alignment`
: Warns when a reinterpret cast moves to a larger alignment.

`-Wshadow`
: Warns if a local variable shadows another local variable, parameter or global variable or if a built-in function is shadowed.

`-Wsign-compare`
: Warns if a comparison between signed and unsigned values could produce an incorrect result when the signed value is converted to unsigned.

`-Wsystem-headers`
: Prints warning messages for constructs found in system header files. This is not default. See §10.7.

`-Wundef`
: Warns if an undefined macro is used in a `#if` directive.

`-Werror`
: Treat all warnings as errors.

`-Werror=option`
: Turns a warning message into an error. The option should be one of the warning options to the compiler that can be prefixed with `-W`.

  By default, the flag `-Werror=timing-syntax` is set. Turning this warning into an error implies that timing warnings (`-Wtiming`) are also errors and vice versa.

10.3 Debugging Options

`-g`
: Produces debugging information.

`-fxta-info`
: Produces timing information for use with XTA. This is default.

`-fresource-checks`
: Produces code in the executable that traps if a resource allocation fails. This causes resource errors to be detected as early as possible.

`-save-temps`
: Saves the intermediate files. These files are placed in the current directory and named based on the source file.

`-fverbose-asm`
: Produces extra compilation information as comments in intermediate assembly files.

`-dumpmachine`
: Prints the target machine and exit.

`-dumpversion`
: Prints the compiler version and exit.

`-print-multi-lib`
: Prints the mapping from multilib directory names to compiler switches that enable them. The directory name is seperated from the switches by ';', and each switch starts with a '@' instead of the '-', without spaces between multiple switches.

`-print-targets`
: Prints the target platforms supported by the compiler. The target names correspond to strings accepted by the `-target` option.

10.4 Optimization Options

Turning on optimization makes the compiler attempt to improve performance and/or code size at the expense of compilation time and the ability to debug the program.

`-O0`
: Do not optimize. This is the default.

`-O`
`-O1`
: Optimize. Attempts to reduce execution time and code size without performing any optimizations that take a large amount of compilation time.

`-O2`
: Optimize more. None of these optimizations involve a space-speed tradeoff.

`-O3`
: Optimize even more. These optimizations may involve a space-speed tradeoff; high performance is preferred to small code size.

`-Os`
: Optimize for the smallest code size possible.

`-fschedule`
: Attempt to reorder instructions to increase performance. This is not default at any optimization level.

10.5 Preprocessor Options

The following options control the preprocessor.

`-E`
: Preprocesses only, then exit.

`-Dname`
: Predefines *name* as a macro with definition 1.

`-Dname=definition`
: Tokenizes and preprocesses the contents of *definition* as if it appeared in a `#define` directive.

`-Uname`
: Removes any previous definition of *name*.

`-D` and `-U` options are processed in the order given on the command line.

`-MD`
: Outputs to a file a rule suitable for `make` describing the dependencies of the source file. The default name of the dependency file is determined based on whether the `-o` option is specified. If `-o` is specified, the filename is the basename of the argument to `-o` with the suffix `.d`. If `-o` is not specified, the filename is the basename of the input file with the suffix `.d`. The name of the file may be overriden with `-MF`.

`-MMD`
: The same as `-MD` expect that dependencies on system headers are ignored.

`-MF file`
: Specifies the file to write dependency information to.

`-MP`
: Emits phony targets for each dependency of the source file. Each phony target depends on nothing. These dummy rules work around errors `make` gives if header files are removed without updating the `Makefile` to match.

`-MT file`
: Specifies the target of the rule emitted by dependency generation.

10.6 Linker And Mapper Options

The following options control the linker/mapper.

`-llibrary`
: Searches the library library when linking. The linker searches and processes libraries and object files in the order specified. The actual library name searched for is `lib`*library*`.a`.

  The directories searched include any specified with `-L`.

  Libraries are archive files whose members are object files. The linker scans the archive for its members which define symbols that have so far been referenced but not defined.

`-nostartfiles`
: Do not link with the system startup files.

`-nodefaultlibs`
: Do not link with the system libraries.

`-nostdlib`
: Do not link with the system startup files or system libraries.

`-s`
: Removes all symbol table and relocation information from the executable.

`-default-clkblk clk`
: Use *clk* as the default clock block. The clock block may be specified by its name in `<xs1.h>` or by its resource number.

  The startup code turns on the default clock block, configures it to be clocked off the reference clock with no divide and puts it into a running state. Ports declared in XC are initially attached to the default clock block. If this option is unspecified, the default clock block is set to `XS1_CLKBLK_REF`.

`-Wm,option`
: Passes *option* as an option to the linker/mapper. If *option* contains commas, it is split into multiple options at the commas.

  To view the full set of advanced mapper options, type `xmap --help`.

`-Xmapper option`
: Passes option as an option to the linker/mapper. To pass an option that takes an argument use `-Xmapper` twice.

`-report`
: Prints a summary of resource usage.

10.7 Directory Options

The following options specify directories to search for header files and libraries.

`-Idir`
: Adds *dir* to the list of directories to be searched for header files.

`-isystemdir`
: Searches *dir* for header files after all directories specified by `-I`. Marks it as a system directory.

  The compiler suppresses warnings for header files in system directories.

`-iquotedir`
: Searches *dir* only for header files requested with `#include "file"` (not with `#include <file>`) before all directories specified by `-I` and before the system directories.

`-Ldir`
: Adds *dir* to the list of directories to be searched for by `-l`.

10.8 Environment Variables Affecting XCC

The following environment variables affect the operation of XCC. Multiple paths are separated by an OS-specific path separator (';' for Windows, ':' for Mac and Linux).

`XCC_INCLUDE_PATH`
: A list of directories to be searched as if specified with `-I`, but after any paths given with `-I` options on the command line.

`XCC_XC_INCLUDE_PATH`
`XCC_C_INCLUDE_PATH`
`XCC_CPLUS_INCLUDE_PATH`

`XCC_ASSEMBLER_INCLUDE_PATH`
: Each of these environment variables applies only when preprocessing files of the named language. The variables specify lists of directories to be searched as if specified with `-isystem`, but after any paths given with `-isystem` options on the command line.

`XCC_LIBRARY_PATH`
: A list of directories to be searched as if specified with `-L`, but after any paths given with `-L` on the command line.

`XCC_DEVICE_PATH`
: A list of directories to be searched for device configuration files.

`XCC_EXEC_PREFIX`
: If set, subprograms executed by the compiler are prefixed with the value of this environment variable. No directory seperated is added when the prefix is combined with the name of a subprogram. The prefix is not applied when executing the assembler or the mapper.

`XCC_DEFAULT_TARGET`
: The default target platform, to be located as if specified with `-target=`. The default target platform is used if no target is specified with `-target=` and no XN file is passed.

10.9 Board Support Provided by `<platform.h>`

During compilation of a program, the compiler generates a temporary header file named `platform.h` that contains variable and macro definitions, as defined by the target XN file, which includes:

- Declarations of variables of type `tileref` (see §54.2).
- Macro definitions of port names (see §54.4.2).

11 Using XMOS Makefiles

IN THIS CHAPTER

- Applications and Modules
- The Application Makefile
- The module_build_info file

Projects created by xTIMEcomposer Studio have their build controlled by Makefiles. These Makefiles execute the build using the program `xmake` which is a port of Gnu Make[2]. The build is executable either from within xTIMEcomposer or from the command line by calling `xmake` directly.

You do not need to understand the Gnu Makefile language to develop applications using xTIMEcomposer. The **common XMOS Makefile** provides support for building applications and source code modules. You need only specify the required properties of the build in **Application Makefiles** and **Module build info files**.

11.1 Applications and Modules

An application is made up of source code unique to the application and, optionally, source code from modules of common code or binary libraries. When developing an application, the working area is described in terms of *workspaces*, *applications* and *modules*.

Workspace
A *workspace* is a container for several projects.

Applications
An *application* is a project containing source files and a Makefile that builds into a single executable (`.xe`) file. By convention application directories start with the prefix `app_`. These applications appear at the top level in the Project Explorer in xTIMEcomposer.

Modules
A *module* is a directory containing source files and/or binary libraries. The source does not build to anything by itself but can be used by applications. By convention module directories start with the prefix `module_`. These modules appear at the top level in the Project Explorer in xTIMEcomposer.

[2] http://www.gnu.org/software/make/

11.1.1 Workspace structure and automatic module detection

Depending on the configuration of your workspace the Makefiles will search folders on your file system to find modules used by an application.

The simplest structure is shown below:

```
app_avb_demo1/
app_avb_demo2/
module_avb1/
module_avb2/
module_xtcp/
module_zeroconf/
module_ethernet/
```

In this case when building the applications, the build system will find the modules on the same directory level as the applications.

Sometimes applications and modules are organized in separate repositories:

```
repo1/
  app_avb_demo1/
  module_avb1/
repo2/
  module_zeroconf/
```

If the Makefiles detect that the folder containing the application is a repository then the Makefiles will search the sub-folders of all repositories at the same nesting level for modules (in this case the sub-folders of `repo1` and `repo2`). The Makefiles will detect a folder as a repository if one of the following conditions hold:

- The folder has a `.git` sub-folder.
- The folder starts with the prefix `sc_`, `ap_`, `sw_`, `tool_` or `lib_`.
- The folder contains a file called `.xcommon_repo` or `xpd.xml`.

If the folder above the application is detected as a repository but the folder above that is then the Makefiles will search at that level. So in the following case:

```
repo1/
  examples/
     app_avb_demo1/
  module_avb1/
repo2/
  module_zeroconf/
```

The sub-folders of `repo1` and `repo2` will be searched.

In addition to the automatic searching for modules, the environment variable `XMOS_MODULE_PATH` can be set to a list of paths that the Makefiles should search.

If you just want to solely use the user specified search path then the automatic searching for modules can be disabled by setting the environment variable XCOMMON_DISABLE_AUTO_MODULE_SEARCH to 1.

11.2 The Application Makefile

Every application directory should contain a file named Makefile that includes the common XMOS Makefile. The common Makefile controls the build, by default including all source files within the application directory and its sub-directories. The application Makefile supports the following variable assignments.

XCC_FLAGS[_config]
: Specifies the flags passed to xcc during the build. This option sets the flags for the particular build configuration *config*. If no suffix is given, it sets the flags for the default build configuration.

XCC_C_FLAGS[_config]
: If set, these flags are passed to xcc instead of XCC_FLAGS for all .c files. This option sets the flags for the particular build configuration *config*. If no suffix is given, it sets the flags for the default build configuration.

XCC_ASM_FLAGS[_config]
: If set, these flags are passed to xcc instead of XCC_FLAGS for all .s or .S files. This option sets the flags for the particular build configuration *config*. If no suffix is given, it sets the flags for the default build configuration.

XCC_MAP_FLAGS[_config]
: If set, these flags are passed to xcc for the final link stage instead of XCC_FLAGS. This option sets the flags for the particular build configuration *config*. If no suffix is given, it sets the flags for the default build configuration.

XCC_FLAGS_filename
: Overrides the flags passed to xcc for the filename specified. This option overrides the flags for all build configurations.

VERBOSE
: If set to 1, enables verbose output from the make system.

SOURCE_DIRS
: Specifies the list of directories, relative to the application directory, that have their contents compiled. By default all directories are included.

INCLUDE_DIRS
: Specifies the directories to look for include files during the build. By default all directories are included.

LIB_DIRS
: Specifies the directories to look for libraries to link into the application during the build. By default all directories are included.

`EXCLUDE_FILES`
: Specifies a space-separated list of source file names (not including their path) that are not compiled into the application.

`USED_MODULES`
: Specifies a space-separated list of module directories that are compiled into the application. The module directories should always be given without their full path irrespective of which project they come from, for example `USED_MODULES = module_xtcp module_ethernet`.

`MODULE_LIBRARIES`
: This option specifies a list of preferred libraries to use from modules that specify more than one. See §12 for details.

11.2.1 Combined xCORE/ARM applications

`XCORE_ARM_PROJECT`
: This option should be set to 1 for applications that wish to build for dual architecture chips such as the XA-series. In this case the build will compile the source files of the project into two binaries - one for xCORE and one for ARM.

`XCORE_ONLY_DIRS`
: This option specifies the directories within the project that should only be used as part of the xCORE build. If not set it will default to `src/xcore`.

`ARM_ONLY_DIRS`
: This option specifies the directories within the project that should only be used as part of the ARM build. If not set it will default to `src/arm`.

`ARM_GCC_FLAGS[_config]`
: Specifies the flags passed to gcc during the build. This option sets the flags for the particular build configuration *config*. If no suffix is given, it sets the flags for the default build configuration.

`ARM_GCC_C_FLAGS[_config]`
: If set, these flags are passed to gcc instead of `ARM_GCC_FLAGS` for all `.c` files. This option sets the flags for the particular build configuration *config*. If no suffix is given, it sets the flags for the default build configuration.

`ARM_GCC_ASM_FLAGS[_config]`
: If set, these flags are passed to gcc instead of `ARM_GCC_FLAGS` for all `.s` or `.S` files. This option sets the flags for the particular build configuration *config*. If no suffix is given, it sets the flags for the default build configuration.

`ARM_GCC_LD_FLAGS[_config]`
: If set, these flags are passed to gcc for the final link stage instead of `ARM_GCC_FLAGS`. This option sets the flags for the particular build configuration *config*. If no suffix is given, it sets the flags for the default build configuration.

`ARM_GCC_FLAGS_filename`
: Overrides the flags passed to arm_gcc for the filename specified. This option overrides the flags for all build configurations.

`EMBED_XCORE_IMAGE_IN_ARM_IMAGE`
: If this variable is set to 1 then the xCORE image will be embedded into the ARM image so that ARM can boot the xCORE.

11.3 The module_build_info file

Each module directory should contain a file named `module_build_info`. This file informs an application how to build the files within the module if the application includes the module in its build. It can optionally contain several of the following variable assignments.

`DEPENDENT_MODULES`
: Specifies the dependencies of the module. When an application includes a module it will also include all its dependencies.

`MODULE_XCC_FLAGS`
: Specifies the options to pass to xcc when compiling source files from within the current module. The definition can reference the `XCC_FLAGS` variable from the application Makefile, for example `MODULE_XCC_FLAGS = $(XCC_FLAGS) -O3`.

`MODULE_XCC_XC_FLAGS`
: If set, these flags are passed to xcc instead of `MODULE_XCC_FLAGS` for all `.xc` files within the module.

`MODULE_XCC_C_FLAGS`
: If set, these flags are passed to xcc instead of `MODULE_XCC_FLAGS` for all `.c` files within the module.

`MODULE_XCC_ASM_FLAGS`
: If set, these flags are passed to xcc instead of `MODULE_XCC_FLAGS` for all `.s` or `.S` files within the module.

`OPTIONAL_HEADERS`
: Specifies a particular header file to be an optional configuration header. This header file does not exist in the module but is provided by the application using the module. The build system will pass the a special macro `__filename_h_exists__` to xcc if the application has provided this file. This allows the module to provide default configuration values if the file is not provided.

11.3.1 Modules for combined xCORE/ARM applications

`XCORE_ARM_PROJECT`
: This option should be set to 1 for modules that wish to build for dual architecture chips such as the XA-series. This will allow modules to participate in applications with this flag set and compile into the ARM part of the build.

`XCORE_ONLY_DIRS`
: This option specifies the directories within the module that should only be used as part of the xCORE build. If not set it will default to `src/xcore`.

`ARM_ONLY_DIRS`
: This option specifies the directories within the module that should only be used as part of the ARM build. If not set it will default to `src/arm`.

`MODULE_ARM_GCC_FLAGS`
: Specifies the options to pass to gcc when compiling source files from within the current module. The definition can reference the `ARM_GCC_FLAGS` variable from the application Makefile, for example `MODULE_ARM_GCC_FLAGS = $(ARM_GCC_FLAGS) -O3`.

`MODULE_ARM_GCC_XC_FLAGS`
: If set, these flags are passed to gcc instead of `MODULE_ARM_GCC_FLAGS` for all `.xc` files within the module.

`MODULE_ARM_GCC_C_FLAGS`
: If set, these flags are passed to gcc instead of `MODULE_ARM_GCC_FLAGS` for all `.c` files within the module.

`MODULE_ARM_GCC_ASM_FLAGS`
: If set, these flags are passed to gcc instead of `MODULE_ARM_GCC_FLAGS` for all `.s` or `.S` files within the module.

12 Using XMOS Makefiles to create binary libraries

IN THIS CHAPTER

- The module_build_info file
- The module Makefile
- Using the module

The default module system used by XMOS application makefiles includes common modules at the source code level. However, it is possible to build a module into a binary library for distribution without the source.

A module that is to be built into a library needs to be split into source that is used to build the library and source/includes that are to be distributed with the library. For example, you could specify the following structure.

```
module_my_library/
       Makefile
       module_build_info
       libsrc/
          my_library.xc
       src/
          support_fns.xc
       include/
          my_library.h
```

The intention with this structure is that the source file `my_library.xc` is compiled into a library and that library will be distributed along with the `src` and `include` directories (but not the `libsrc` directory).

12.1 The module_build_info file

To build a binary library some extra variables need to be set in the `module_build_info` file. One of the `LIBRARY` or `LIBRARIES` variables must be set.

`LIBRARY` This variable specifies the name of the library to be created, for example:

```
LIBRARY = my_library
```

`LIBRARIES` This variable can be set instead of the `LIBRARY` variable to specify that several libraries should be built (with different build flags), for example:

```
LIBRARY = my_library my_library_debug
```

The first library in this list is the default library that will be linked in when an application includes this module. The application can specify one of the other libraries by adding its name to its MODULE_LIBRARIES list.

LIB_XCC_FLAGS_libname
: This variable can be set to the flags passed to xcc when compiling the library libname. This option can be used to pass different compilation flags to different variants of the library.

EXPORT_SOURCE_DIRS
: This variable should contain a space separated list of directories that are *not* to be compiled into the library and distributed as source instead, for example:

```
EXPORT_SOURCE_DIRS = src include
```

12.2 The module Makefile

Modules that build to a library can have a Makefile (unlike normal, source-only modules). The contents of this Makefile just needs to be:

```
XMOS_MAKE_PATH ?= ../..
include $(XMOS_MAKE_PATH)/xcommon/module_xcommon/build/Makefile.library
```

This Makefile has two targets. Running `make all` will build the libraries. Calling the target `make export` will create a copy of the module in a directory called `export` which does not contain the library source. For the above example, the exported module would look like the following.

```
export/
  module_my_library/
     module_build_info
     lib/
       xs1b/
         libmy_library.a
       src/
         support_fns.xc
       include/
         my_library.h
```

12.3 Using the module

An application can use a library module in the same way as a source module (including the module name in the USED_MODULES list). Either the module with the library source or the exported module can be used with the same end result.

Part E

Timing

CONTENTS

- Use xTIMEcomposer to time a program
- Use the XTA from the command line
- XTA command-line manual

13 Use xTIMEcomposer to time a program

IN THIS CHAPTER

- Launch the timing analyzer
- Time a section of code
- Specify timing requirements
- Add program execution information
- Validate timing requirements during compilation

The xCORE Timing Analyzer lets you determine the time taken to execute code on your target platform. Due to the deterministic nature of the xCORE architecture, the tools can measure the shortest and longest time required to execute a section of code. When combined with user-specified requirements, the tools can determine at compile-time whether all timing-critical sections of code are guaranteed to execute within their deadlines.

13.1 Launch the timing analyzer

To load a program under control of the timing analyzer, follow these steps:

1. Select a project in the **Project Explorer**.
2. Choose **Run ▶ Time Configurations**.
3. In the left panel, double-click **XCore Application**. xTIMEcomposer creates a new configuration and displays the default settings in the right panel.
4. xTIMEcomposer tries to identify the target project and executable for you. To select one yourself, click **Browse** to the right of the **Project** text box and select your project in the **Project Selection** dialog box. Then click **Search Project** and select the executable file in the **Program Selection** dialog box.

You must have previously compiled your program without any errors for the executable to be available for selection.

5. In the **Name** text box, enter a name for the configuration.
6. To save the configuration and launch the timing analyzer, click **Time**.

xTIMEcomposer loads your program in the timing analyzer and opens it in the **Timing** perspective. In this perspective the editor is read-only, to ensure the relationship between the binary and source code remains consistent.

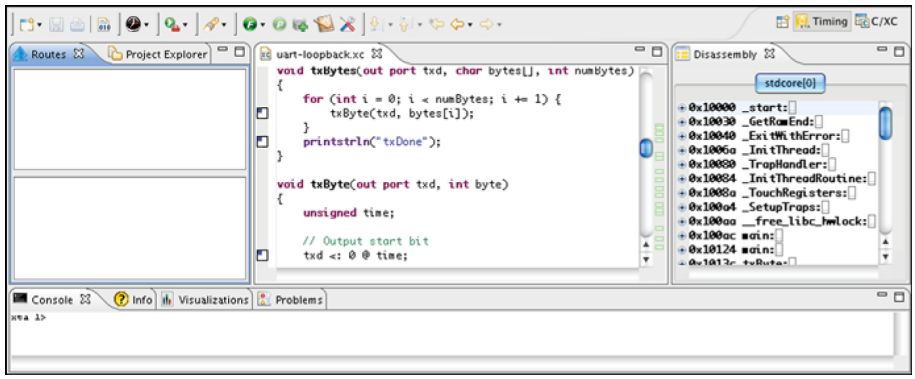


Figure 12: Timing perspective

xTIMEcomposer remembers the configuration last used to load your program. To load XTA the program later using the same settings, just click the **XTA** button. To use a different configuration, click the arrow to the right of the **XTA** button and select a configuration from the drop-down list.

13.2 Time a section of code

A *route* consists of the set of all paths through which control can flow between two points (or *endpoints*) in a program. Each route has a best-case time, in which branches always follow the path that takes the shortest time to execute, and a corresponding worst-case time.

To specify a route and analyze it, follow these steps:

1. Right-click on an endpoint marker in the editor margin and choose **Set from endpoint**. xTIMEcomposer displays a green dot in the top-right quarter of the marker.

2. Right-click on an endpoint marker and choose **Set to endpoint**. xTIMEcomposer displays a red dot in the bottom-right quarter of the marker.

   You can specify a start point above an end point. You can also specify a start point at or below an end point, defining a route whose paths flow out and then back into the function. This is typical of functions called multiple times or from within a loop.

3. Click the **Analyze Endpoints** button in the main toolbar. xTIMEcomposer analyzes all the paths in the specified route, displaying a tree-like representation in the lower panel of the **Routes** view and a graph-like representation in the **Structure** tab of the **Visualizations** view.

   Alternatively, to analyze the time taken to execute a function, just click the **Analyze Function** button in the main toolbar and select a function from the drop-down list.

xTIMEcomposer provides endpoint markers for all statements whose order is guaranteed to be preserved during compilation. These statements include I/O operations and function calls.

13.2.1 Visualize a route

The **Routes** view displays a structural representation of the route. Each time you analyze a route, an entry is added to the top panel. Click on a route to view it in the bottom panel. It is represented using the following nodes:

- A source-level function.
- A list of nodes that are executed in sequence.
- A set of nodes that are executed conditionally.
- A loop consisting of a sequence of nodes in which the last node can branch back to the first node.
- A block containing a straight-line sequence of instructions.
- A single machine instruction.

13.2.2 The Visualizations view

The **Visualizations** view provides graphical representations of the route. The Structure tab represents the route as a line that flows from left to right, as shown in the example below. The route forks into multiple paths whenever the code branches, and all paths join at its end. The best-case timing path is highlighted in green, the worst-case path in red, and all other paths are colored gray.

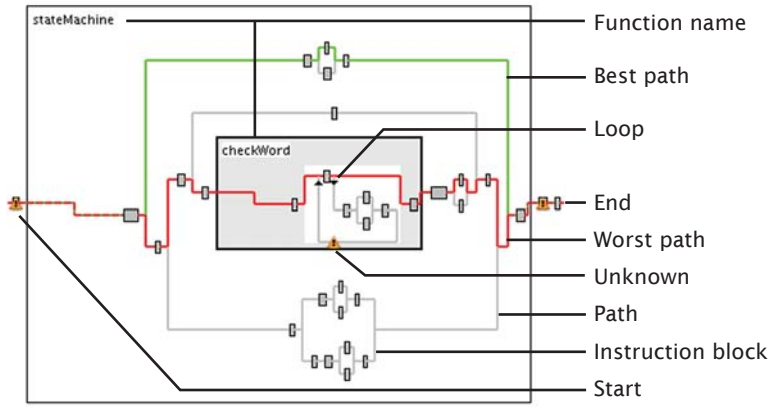


Figure 13: Visualizations view

In both the **Route** view and **Structure** view, you can hover over a node to display a summary of its timing properties. Click on a node to highlight its source code in

the editor, or double-click to go to the line at the start of the node. In the **Structure** view, double-click on a function name to expand or collapse it.

13.3 Specify timing requirements

A *timing requirement* specifies how long the paths in a route may take to execute for the program to behave correctly. In the top panel of the **Routes** view, the status of each route is indicated by an icon to the left of its name:

No timing requirement is specified.

A timing requirement is specified and met.

A timing requirement is specified and met, subject to all I/O instructions being ready to execute.

A timing requirement is specified and not met.

To specify a timing requirement, right-click on a route and choose **Set timing requirements**. A dialog box opens. Enter the maximum time in which the paths must execute in either *ns*, *cycles* or *MHz* and click **OK**. xTIMEcomposer updates the status of the route.

13.4 Add program execution information

Under some conditions the timing analyzer is unable to prove timing without additional information. Examples of common conditions include:

- The route contains an I/O instruction that can pause for an unknown length of time.
- The route contains a loop with a data-dependent exit condition.
- A path fails to meet timing, but the path is only executed as a result of an error condition and is not therefore timing critical.

In these cases you can provide the timing analyzer additional information about the execution of your program. Armed with this additional information, the analyzer may then be able to prove that a route's timing requirement is met. Information you can provide includes:

- **The number of loop iterations:** Right-click on a loop node and choose **Set loop iterations** to display a dialog box. Enter a maximum loop count and click **OK**.
- **The maximum pause time for an I/O instruction:** Right-click on an instruction node and choose **Set instruction time** to display a dialog box. Enter a value, select a unit of time/rate (such as nanoseconds or MHz) and click **OK**.
- **Exclude a path from the route:** Right-click on a node and choose **Exclude**.

13.4.1 Refine the worst-case analysis

By default, the timing analyzer assumes that a route always follows branches that take the longest time to execute. If you know that this is not the case, for example through inspection during simulation or a formal analysis of your program, you can refine the parameters used by the analyzer. Refinements you can make include:

- **Specifying an absolute execution time for a function call:** Right-click on a function node and choose **Set function time** to open a dialog box. Enter a time and click **OK**.
- **Specifying an absolute time for a path:** Select a path by holding down Ctrl (Windows, Linux) or ⌘ (Mac) and clicking on two instruction nodes, then right-click and choose **Set path time** to open a dialog box. Enter a time and click **OK**.
- **Specifying the number of times a node is executed:** By default, the analyzer assumes that the number of times a node is executed is the multiplication of each loop count in its scope. To change the iteration count to be an absolute value, right-click on a node and choose **Set loop scope** to open a dialog box. Select **Make scope absolute** and click **OK**.
- **Specifying the number of times a conditional is executed in a loop:** By default, the analyzer assumes that a conditional node always follows the path that takes the longest time to execute. To specify the number of times a conditional target is executed, right-click on the target node and choose **Set loop path** iterations to open a dialog box. Enter the number of iterations and click **OK**.

13.5 Validate timing requirements during compilation

Once you've specified the timing requirements for your program, including any refinements about its execution, you can generate a script that checks these requirements at compile-time.

To create a script that checks all timing requirements specified in the **Routes** view, follow these steps:

1. Click the **Generate Script** button.
2. In the **Script location** text box, enter a filename for the script. The filename must have a `.xta` extension.
3. To change the names of the pragmas added to the source file, modify their values in the **Pragma name** fields.
4. Click **OK** to save the script and update your source code. xTIMEcomposer adds the script to your project and opens it in the editor. It also updates your source files with any pragmas required by the script.

The next time you compile your program, the timing requirements are checked and any failures are reported as compilation errors. Double-click on a timing error to view the failing requirement in the script.

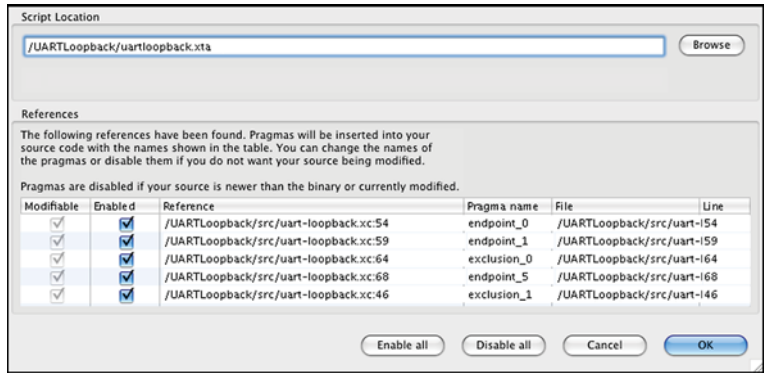


Figure 14: Script Options dialog box

14 Use the XTA from the command line

IN THIS CHAPTER

- Frequently used commands
- Viewing results
- Refining timing results
- Program structure
- Automating the process
- Scripting XTA via the Jython interface

The XTA tool can be used interactively on the command-line or the console in xTIMEcomposer Studio.

14.1 Frequently used commands

This section summarizes a number of frequently used commands that can be run from the command line.

14.1.1 Loading a binary

To load a binary type:

```
load <FILE NAME>
```

14.1.2 Routes

A *route* is a timing-critical section of code. It consists of the set of all paths through which control can flow between two points in a program (*endpoints*). A route can be created by timing a function, timing a loop or by timing between endpoints.

14.1.3 Endpoints

An *endpoint* is any source line that, during the compilation process, must be preserved, and its order with respect to other endpoints must be maintained.

To show a list of all endpoints type:

```
list allendpoints
```

If specifying a route with respect to assembly code then any valid label/program counter (PC) can be used as an endpoint. However, program counters are classed

as non-portable endpoints as they are likely to change between compilations and their use in scripts is therefore discouraged.

14.1.4 Adding endpoints to source

Source lines can be labeled with endpoint pragmas to ensure that the endpoints are portable. For example, Figure 15 shows a function that has been annotated with endpoint pragmas called `start` and `stop`.

```
int g(in port p) {
    int x, y;

    # pragma xta endpoint " start "
    p :> x;

    # pragma xta endpoint " stop "
    p :> y;

    return (y - x);
}
```

Figure 15: Putting an endpoint pragma into the souce

To show a list of endpoints type:

```
list endpoints
```

14.1.5 Timing between endpoints

To time between endpoints type:

```
analyze endpoints <from ENDPOINT> <to ENDPOINT>
```

The XTA does not time code across multiple xCORE tiles so both endpoints must be on the same tile.

One analysis can result in multiple routes being generated.

14.1.6 Timing functions

Type the function name on the console:

```
analyze function <FUNCTION>
```

This will create a route which describes the set of all possible paths from the function entry point to all the function return points.

14.1.7 Timing loops

To time a loop type:

```
analyze loop <ANY>
```

This creates a route that describes all possible paths through the loop. It it effectively a shortcut for timing between endpoints where the start and stop endpoint is the same, the point is within a loop and an exclusion has been placed such that everything outside the loop is excluded.

One analysis can result in multiple routes being generated.

14.1.8 Setting timing requirements

To define the timing requirements for a route type:

```
set required <route id> <value> <MODE>
```

The supported timing modes are defined in §15.3.

The route IDs can be found by typing:

```
print summary
```

Alternatively, the - character can be used on the command-line or in a script to refer to the last route analyzed.

14.2 Viewing results

14.2.1 Route IDs

All analyzed routes are given a unique route ID. However, when referring to routes in a script, using the route ID may not always result in portable or robust scripts. In many cases, the only route that needs to be referenced is the one that was last analyzed. This can be achieved by using the '-' character as the route ID. If the last command created multiple routes then the '-' character refers to all of the routes created.

14.2.2 Node IDs

Within a single route, all nodes are assigned a unique ID number. This is required as input for some of the console commands. The '-' character can be used in this context to refer to the top level node of the route.

14.2.3 Summary

To show a list of all routes type:

```
print summary
```

Details for a specific route are shown using the command:

```
print routeinfo <route id>
```

14.2.3.1 Violation

When a timing requirement has been set for a route and the route takes more time to execute than required, the time difference is called a *violation*. This value specifies how much faster the route needs to be executed in order to meet the timing requirement.

14.2.3.2 Slack

When a timing requirement has been set for a route and the route takes less time to execute than required, the time difference is called *slack*. This value specifies how much slower the route could be executed and still meet the timing requirement.

14.2.4 Structure

To display the the structure of a route in xTIMEcomposer Studio type:

```
print structure <route id>
```

The structure used by the XTA is described in §14.4.2.

14.2.5 Source code annotation

To display the source code which is executed by a route type:

```
print src <route id>
```

If only a part of a route should be used then the node ID can be specified:

```
print src <route id> <node id>
```

14.2.6 Instruction traces

To help developers understand the execution flow of a route, the XTA can create representative instruction traces. Type:

```
print trace <route id>
```

As a result of loops being unrolled when tracing, it is possible for the traces to get very large. The trace operation can be cancelled at any time by pressing *CTRL+C* in the command-line tool.

A trace can be redirected to a file by typing:

```
print trace <route id> > <file>
```

By default, the trace for worst-case path is printed. This can be changed to print the best-case path instead by typing:

```
config case best
```

14.2.7 Fetch no-ops

The xCORE device may need to pause at certain times while more instructions are fetched from memory. This results in the issue of fetch no-op instructions. These are shown in the traces as `FNOP` at the points they will happen on the hardware.

In xTIMEcomposer Studio they are inserted into the disassembly at the points they occur.

14.2.8 Scaling Results

By default, the XTA scales all timing results. This means that the appropriate unit (ms, us, ns) will be used to print time values. This can be changed so that all times are printed in ns by typing:

```
config scale false
```

14.2.9 Unknowns

The XTA may not always be able to determine the exact timing of a section of code if it is unable to determine loop iteration counts or the execution time of instructions. These unknown conditions can be displayed on the console by typing:

```
list unknowns <route id>
```

§14.3 describes how to address these warnings.

14.3 Refining timing results

There are cases where the XTA is unable to fully determine the timing of a section of code, due to, for example, not being able to determine a loop count. This can be addressed by adding *defines*. Defines can be added in two ways, to a *global list*, or to a *route-specific list*. Those added to the *global list* get applied to every route when upon creation.

The use of the global list can result in more concise scripts. However, It is important to be careful with defines added to the *global list* since they are ignored if they fail to get applied to a route. This allows a full set of defines to be created before any routes, but does mean that errors in these defines might be missed. Route specific defines (added post route creation) will always flag an error if there is one.

14.3.1 Exclusions

Not all paths of execution in a route may be timing-critical. The route may contain cases to handle errors where the timing of the code is not important. These paths can be ignored in the timing script by adding exclusions. *Exclusions* tell the XTA to ignore all paths which pass through that code point. Exclusions can be added to the global list or applied to a specific route.

To set an exclusion on an existing route type:

```
set exclusion <route id> <ANY>
```

To add an exclusion to the list of exclusions to be taken into account during route creation type:

```
add exclusion <ANY>
```

To list the global list of exclusions type:

```
list exclusions
```

To remove an exclusion from the global list type:

```
remove exclusion <ANY|*>
```

For example, consider the code in Figure 16.

```
int calculate ( int a, int b) {
    if ( willOverflow (a, b) {
        # pragma xta label " overflow "
        return processOverflow ();
    }
    return a + b;
}
```

Figure 16: Excluding an invalid path

To time the `calculate` function ignoring the error case:

- Using route-specific defines:
  - `analyze function calculate`
  - `set exclusion - overflow`

- Using global defines:
  - `add exclusion overflow`
  - `analyze function calculate`

Although functionally equivalent, exclusion via the global defines mechanism can result in faster, and more memory efficient, route creation. This is because the global exclusions can be taken into account during route creation, so the search space can be reduced. For post route creation exclusions, the complete route is created before any pruning occurs.

14.3.2 Loop Iterations

Loop iteration counts can be unknown. Whenever possible, the compiler tells the XTA about loop iteration counts. However, some loop counts are not known statically. In these cases developers must specify worst-case values.

The compiler does not emit any loop iteration counts unless optimizations have been enabled (-O1 or greater).

Some loops are self loops (loops whose body is the same as the header) and therefore have a minimum iteration count of 1.

To set loop iterations on an existing route type:

```
set loop <route id> <ANY> <iterations>
```

To add an iteration count to the list of iteration counts to be used during route creation type:

```
add loop <ANY> <iterations>
```

To list the current global loop iteration counts type:

```
list loops
```

To remove a loop iteration count from the global list type:

```
remove loop <ANY|*>
```

For example, consider the code in Figure 17.

To time the `test` function:

- Using route-specific defines:
  - `analyze function test`
  - `set loop - delay_loop 10`
- Using global defines:

```
void delay ( int j) {
    for ( unsigned int i = 0; i < j; ++i) {
        # pragma xta label " delay_loop "
        delay_us (1);
    }
}

int test () {
    delay (10);
}
```

Figure 17: Setting loop iterations.

- `add loop delay_loop 10`
- `analyze function test`

14.3.3 Loop path iterations

A loop may contain multiple paths through it. When a loop iteration count has been set the tools assumes that all iterations will take the worst-case path of execution through the loop. This is not always the case, and a more realistic worst-case can be established by specifying the number of iterations on individual paths through the loop.

To set loop path iterations on an existing route type:

```
set looppath <route id> <ANY> <iterations>
```

To add a loop path count to the list of loop path counts to be used during route creation type:

```
add looppath <ANY> <iterations>
```

To display the current list of global loop path counts type:

```
list looppaths
```

To remove a loop path count from the global list type:

```
remove looppath <ANY|*>
```

There are some rules that need to be followed when setting loop path iterations:

- In a nested loop, the outer loop iterations need to be set first.
- The loop path iterations set must be less than or equal to the loop iterations set on the enclosing loop.

- If the loop path iterations set are less than that of the enclosing loop, then there must exist another path within the loop without its iterations set to which the remaining iterations can be allocated.

For example, consider the code in Figure 18:

```
void f( int j) {
    for ( unsigned int i = 0; i < j; ++i) {
        # pragma xta label " f_loop "
        if ((i & 1) == 0) {
            # pragma xta label " f_if "
            g ();
        }
    }
}

int test () {
    f (10);
}
```

Figure 18: Setting loop path iterations.

To time the `test` function:

- Using route-specific defines:
  - `analyze function test`
  - `set loop - f_loop 10`
  - `set looppath - f_if 5`
- Using global defines:
  - `add loop f_loop 10`
  - `add looppath f_if 5`
  - `analyze function test`

14.3.4 Loop scope

By default, the XTA assumes that the iterations for loops are *relative*—the iterations for an inner loop will be multiplied by the iterations of enclosing loops. However this is not sufficient to describe all loop structures. If this assumption is not correct a loop count can be set to *absolute*. The iteration count set on an absolute loop is not multiplied up by the iterations set on enclosing loops.

To set loop scope on an existing route type:

```
set loopscope <route id> <ANY> <absolute|relative>
```

To add a loop scope to the list of loop scopes to be used during route creation type:

```
add loopscope <ANY> <absolute|relative>
```

To display the current list of global loop scopes, type:

```
list loopscopes
```

To remove a loop scope from the global list, type:

```
remove loopscope <ANY|*>
```

For example, consider the code in Figure 19

```
void f( int l) {
    for ( unsigned int i = 0; i < l; ++i) {
        # pragma xta label " outer_loop "
        for ( unsigned int j = 0; j < i; ++j) {
            # pragma xta label " inner_loop "
            g ();
        }
    }
}

void test () {
    f (10);
}
```

Figure 19: Setting loop scope.

To time the `test` function:

- Using route-specific defines:
  - `analyze function test`
  - `set loop - outer_loop 10`
  - `set loop - inner_loop 45`
  - `set loopscope - inner_loop absolute`
- Using global defines:
  - `add loop outer_loop 10`
  - `add loop inner_loop 45`
  - `add loopscope inner_loop absolute`
  - `analyze function test`

14.3.5 Instruction times

Some instructions can pause the processor. By default, the XTA reports timing assuming that no instructions pause, but flags them as warnings. Developers must specify what the worst-case execution time of instructions are.

To set an instruction time in an existing route, type:

```
set instructiontime <route id> <ENDPOINT> <value> <MODE>
```

To add an instruction time to the list of instruction times to be used during route creation, type:

```
add instructiontime <ENDPOINT> <value> <MODE>
```

To display the current list of global instruction times, type:

```
list instructiontimes
```

To remove an instruction time from the global list, type:

```
remove instructiontime <ANY|*>
```

For example, consider the code in Figure 20.

Figure 20: Setting an instruction time.

```
void f( port p) {
    # pragma endpoint " instr "
    p :> value ;
}
```

To time the `f` function:

- Using route-specific defines:
  - `analyze function f`
  - `set instructiontime - instr 100.0 ns`
- Using global defines:
  - `add instructiontime instr 100.0 ns`
  - `analyze function f`

14.3.6 Function times

In some cases it is necessary to define the time it takes to execute an entire function. The XTA supports defining a function time. Once a function time is

defined, all the unknowns within it are ignored and any routes which span this function will use the defined time instead of calculating it.

To set a function time on an existing route, type:

```
set functiontime <route id> <FUNCTION> <value> <MODE>
```

To add a function time to the list of function times to be used during route creation, type:

```
add functiontime <FUNCTION> <value> <MODE>
```

To display the current list of global function times, type:

```
list functiontimes
```

To remove an function time from the global list, type:

```
remove functiontime <FUNCTION|*>
```

For example, consider the code in Figure 21.

```
void delayOneSecond () {
    g ();
    }

void test () {
    delayOneSecond ();
}
```

Figure 21: Setting a function time.

To time the `test` function:

- Using route-specific defines:
  - `analyze function test`
  - `set functiontime - delayOneSecond 1000.0 ms`
- Using global defines:
  - `add functiontime delayOneSecond 1000.0 ms`
  - `analyze function test`

14.3.7 Path times

In some cases it is necessary to define the time it takes to execute a particular section of code. The XTA supports defining a path time for this case. Once a path

time is defined all the unknowns within it are ignored, and any routes which span this section of code will use the defined time instead of calculating it.

To set a path time on an existing route, type:

```
set pathtime <route id> <from ENDPOINT> <to ENDPOINT> <value> <MODE>
```

To add a path time to the list of path times to be used during route creation, type:

```
add pathtime <from ENDPOINT> <to ENDPOINT> <value> <MODE>
```

To display the current list of global path times, type:

```
list pathtimes
```

To remove an path time from the global list, type:

```
remove pathtime <from ENDPOINT|*> <to ENDPOINT|*>
```

For example, consider the code in Figure 22.

```
int f() {
    int time ;
    timer t;
    # pragma xta endpoint " start "
    t :> time ;
    # pragma xta endpoint " stop "
    t when timerafter ( time + 100) :> time ;
}

void test () {
    f ();
}
```

Figure 22: Setting a path time.

To time the `test` function:

- Using route-specific defines:
  - `analyze function test`
  - `set pathtime - start stop 1000.0 ns`
- Using global defines:
  - `add pathtime start stop 1000.0 ns`
  - `analyze function test`

14.3.8 Active tiles

By default the XTA finds routes on all tiles within a program. However, it is possible to restrict the XTA to work only on a subset of the tiles in the program. The set of tiles all commands apply to is called the *active* tiles.

To select which tiles are active, type:

- `add tile <tile id>`
- `remove tile <tile id|*>`
- `list tiles`

14.3.9 Node frequency

An xCORE device consists of a number of nodes, each one composed of a number of xCORE tiles. The frequency at which a node runs is defined in the binary and the XTA reads this and configures the node frequencies when it loads the binary. It is possible to experiment to determine what will happen at different frequencies if desired.

To change the frequency for the node, type:

```
config freq <node id> <tile frequency>
```

14.3.10 Number Of logical cores

The maximum number of logical cores run on a tile is known at compile time and the XTA extracts this information from the binary for each tile. It is possible to experiment to determine what will happen if running with a different number of cores if desired.

To change the number of cores for the node/tile, type:

```
config cores <tile id> <num cores>
```

14.4 Program structure

Programs are written in multiple source files, each containing functions. Each function will contain sequences of statements, loops (e.g. `for` / `while` / `do`), conditionals (e.g. `if` / `switch`) and function calls.

14.4.1 Compiling for the XTA

The compiler outputs information which allows the XTA to make associations between source and instructions. This information is on by default but can be disabled by adding the following flag to the compiler options:

```
-fno-xta-info
```

The compiler also supports adding debug information without affecting optimizations. Debug information is not required for the XTA to analyze code, but the mapping between instructions and source code is not available without the debug information. In order to add debug information compile with:

```
-g
```

14.4.2 Structural nodes

The compiler tools create a binary file with one program per xCORE tile. The XTA uses the binary file to produce accurate timing results.

When a route is created, the XTA analyzes the binary to create a structure which closely represents the high-level program structure. It decomposes the program into structural nodes which can be displayed as a tree.

The worst and best case time is then calculated for each of the structural nodes. The way this is calculated depends on the type of structural node. The worst and best case times for the overall route is built up from the worst and best case times of the sub nodes.

The structural nodes can be of the following types:

- Instruction: the most basic building block of the program is the instruction.
- Block: a list of instruction nodes with no conditional branching which is therefore executed in sequence. The worst/best case time for a block is the sum of its component instructions.
- Sequence: a list of structural nodes which are executed in order. The worst/best case time for a sequence is the sum of the worst/best case times of its sub nodes.
- Conditional: a set of structural nodes out of which at most one node is executed. If this is within a loop then on each iteration a different node might be chosen. In some cases the entire conditional is optional. In those cases the best case time is for none of the options to be taken. The worst/best case time for a conditional is determined by the worst/best case time of each of its sub nodes.
- Loop: consists of a header and a body (both of which are structural nodes). The header corresponds to the conditional test part of the loop, and the body corresponds to the code that is executed if the loop is taken. This roughly corresponds to high level code structures such as `while` or `for` loops.

  The body is executed once per iteration. The header always executes once more than the number of iterations. The worst/best case times for a loop is the worst/best case time of its header multiplied by (number of iterations + 1) plus the worst/best case time of the body multiplied by the number of iterations.

- Self-loop: a loop where the header and body are the same. It is therefore considered to have a minimum loop count of 1. This roughly corresponds to high level code structures such as `do` loops. The worst/best case time for a self-loop is determined by the worst/best case time of its body multiplied up buy the number of iterations.

- Function: is the high-level construct of the function and consists of a list of other structural nodes. The worst/best case time for a function is calculated in the same way as that of a sequence.

14.4.3 Identifying nodes: code references

A `code reference` is the way to specify a particular location in an application. A code reference is made up of a base and an optional backtrail. The base consists of a `reference type` and the backtrail consists of a comma separated list of `reference types`.

There are a number of different reference types, all of which map to one or more instruction program counters (PCs). This will usually be one PC, but can be more than one due to compiler optimizations or because the user has explicitly named multiple instructions with the same reference. Compiler optimizations such as inlining or unrolling will result in the same reference mapping to multiple PCs.

The different reference types are detailed below. The commands to list the instances of them for the currently loaded executable in the console are detailed with each type.

- **Source file-line** references are valid for source lines which the compiler has defined as belonging to a source-level basic block. The valid lines can be listed in the console using:

  ```
  list allsrclabels
  ```

- **Source labels** are added to source code using the `#pragma xta label`. To list the source labels in the console, type:

  ```
  list srclabels
  ```

- **Call file-line** references are valid for source lines which map to function calls. To list the valid source lines in the console, type:

  ```
  list allcalls
  ```

- **Call labels** are added to source code using the `#pragma xta call`. To list the source labels in the console, type:

  ```
  list calls
  ```

- **Endpoint file-line** references are available for source lines which map to a valid *endpoint*. To list the endpoints in the console type:

  ```
  list allendpoints
  ```

- **Endpoint labels** are added to the source using `#pragma xta endpoint`. They must be on the line before an input/output operation. To list the labeled endpoints in the console, type:

  ```
  list endpoints
  ```

- **Labels** are arbitrary text strings referring to any source or assembly label. To list the labels in the console, type:

  ```
  list labels
  ```

  Labels in assembly must be within an executable section.

- **Functions** are the functions contained within the binary. To list the labels in the console, type:

  ```
  list functions
  ```

  Functions in assembler must be labeled as functions with the `.type` directive to be correctly detected by the XTA (see xTIMEcomposer Studio User Guide). They must also be within an executable section.

- **Program counters** (PC) are the lowest-level reference, giving a hexadecimal program counter value starting with `0x`. They must map to the PC of an instruction within the executable section of the program.

14.4.4 Reference Classes

Particular console commands only work on particular types of references. The sets of reference types that are defined for a particular command are know as reference classes.

- **ENDPOINT**: A reference that can be used for timing. This means any reference in assembler (PC/label) and only source references which map to lines which can be reliably used for timing. Compiler optimizations cannot remove them or re-order them with respect to each other. In XC code these correspond to source lines with I/O operations. The following console command lists the types available in the class:

  ```
  help ENDPOINT
  ```

- **CALL**: References that map to function calls. These are used in *back trails* to identify unique instances of a code reference. The following console command lists the types available in the class:

  ```
  help CALL
  ```

- **FUNCTION**: References that map to functions. The following console command lists the types available in the class:

  ```
  help FUNCTION
  ```

- **LABEL**: The following console command lists the types available in the class:

  ```
  help LABEL
  ```

- **PC**: The following console command lists the types available in the class:

```
help PC
```

It is possible to have a code reference which could map to multiple types. For example there could be an endpoint which has been given the same name as a function in the program. The way a reference in a backtrail is matched can depend upon the type of the reference. To resolve this potential ambiguity, it is possible to force the code reference to a certain type by prefixing with its type.

14.4.5 Back trails

A code reference's base may occur multiple times within a program. For example, a function can be called from multiple places. The *back trail* for a reference is a way of restricting a reference to specific instances. Consider the example file shown in Figure 23.

```
1  void delay_n_seconds ( int j) {
2      for ( unsigned int i = 0; i < j; ++i) {
3          # pragma xta label " delay_loop "
4          delay_1_second ();
5      }
6  }
7
8  int test () {
9      # pragma xta call " delay_1 "
10     delay_n_seconds (10);
11     # pragma xta call " delay_2 "
12     delay_n_seconds (20);
13     return 0;
14 }
```

Figure 23: Using backtrails.

The following commands could be used to time the `test` function:

- `analyze function test`
- `set loop - delay_loop 10`

That would have the effect of setting the number of loop iterations for the loop in both instances of the `delay_n_seconds` to 10. However, as the number of iterations are passed as a parameter to delay_n_seconds, the value is different for each call.

To time test correctly the loop iterations for each instance needs to be specified differently. This can be achieved by the use of the call references and backtrails. For example:

- `analyze function test`
- `set loop - delay_1,delay_loop 10`
- `set loop - delay_2,delay_loop 20`

This tells the tool to set `delay_loop` to 10 iterations when called from `delay_1`, and to 20 iterations when called from `delay_2`. The references used in the above case are composed of a base reference of type source label, and a backtrial or size one, of type call label. The above can also be achieved using the file-line equivalents. For example:

- `analyze function test`
- `set loop - source.xc:10,source.xc:3 10`
- `set loop - source.xc:12,source.xc:3 20`

However, this would not result in a portable and robust script implementation, so using file-line references in this way from a script is not encouraged.

When the compiler inlines some code (for example the `delay_n_seconds` function above) then some references will no longer be valid. In this case the following reference would not exist because the call no longer exists:

```
source.xc:10,source.xc:3
```

However, if the call has been labeled with a call label, the compiler ensures that the reference is still valid even if the code is inlined. So, in the above case, the following reference will still be valid;

```
delay_1,delay_loop
```

14.4.6 Scope of references

References can have either *global* or *local* scope. Globally scoped references are those which apply to (or get resolved on) the global tree. The global tree is the notional structural representation of the whole program, prior to any route analysis taking place. Locally scoped references are those which apply to (or get resolved on) a user created route tree. Whether a particular reference is globally of locally scoped depends on the command being executed. The following commands used globally scoped references:

- `analyze path`
- `analyze function`
- `analyze loop`
- `add exclusion`
- `add branch`

The following commands used locally scoped references:

- `set/add loop`
- `set/add looppath`

- `set/add loopscope`
- `set/add instructiontime`
- `set/add pathtime`
- `set/add functiontime`

In general, globally scoped references can lead to multiple route creation.

14.5 Automating the process

The XTA can be automated to ensure that new versions of an application meet timing requirements using a script.

14.5.1 Writing a script

The script file is a sequence of XTA console commands. Each one on a separate line. Any line starting with the # symbol is considered a comment.

Developers must insert `pragmas` into the source code where required to make the script portable. If the script creation process modifies the source (e.g. by inserting pragmas) the relevant binary must be rebuilt before the script can be successfully executed.

It is recommended not to put a `load` or `exit` command in the script. These commands should be done at the time of calling the script.

XTA scripts must use the `.xta` extension in order to be used by the compiler and understood correctly by xTIMEcomposer Studio.

14.5.2 Running a script

Scripts can be run in different ways, either in xTIMEcomposer Studio or on the command-line.

- **During compilation**: On the command line the `.xta` scripts must be passed to the compiler manually. By default, timing failures are treated as warnings and syntax errors in the script as errors.

  To treat timing failures as errors, add the following to the compiler arguments:

  ```
  -Werror=timing
  ```

  To treat script syntax errors as warnings, add the following to the compiler arguments:

  ```
  -Wno-error=timing-syntax
  ```

- **Batch mode**: In batch mode the XTA takes command-line arguments and interprets them as XTA commands. For example, to run an XTA script (`script.xta`) on a binary (`test.xe`) use:

  ```
  xta -load test.xe -source script.xta -exit
  ```

Note: the '-' character is used as a separator between commands.

14.5.3 Embedding commands into source

The XTA can embed commands into source code using a `command` pragma. For example,

```
#pragma xta command "print summary"
```

All commands embedded into the source are run every time the binary is loaded into the XTA. Commands are executed in the order they occur in the file, but the order between commands in different source files is not defined.

Pragmas are only supported in XC code.

14.6 Scripting XTA via the Jython interface

The XTA supports scripts written using the Jython language (an implementation of Python running on the Java virtual machine). XTA Jython scripts must have the extension `.py`. They can be executed in the same way as command based XTA scripts. From within Jython, XTA features are made available though the globally accessible *xta* object. See Figure 24 for an example script. This scripts loads the binary `test.xe` into the XTA and analyzes the function `functionName`. It then sets a loop count on each of the resulting routes and finally, prints the best and worst case times for each.

```
import sys
import java

try :
    xta . load (" test .xe");

except java . lang . Exception , e:
    print e. getMessage ()

try :
    ids = xta . analyzeFunction (" functionName ");

    for id in ids :
        xta . setLoop (id , " loopReference ", 10)

    for id in ids :
        print xta . getRouteDescription (id),
        print xta . getWorstCase (id , "ns"),
        print xta . getBestCase (id , "ns")

except java . lang . Exception , e:
    print e. getMessage ()
```

Figure 24: Example of an XTA Jython script.

15 XTA command-line manual

IN THIS CHAPTER

- Commands
- Pragmas
- Timing Modes
- Loop Scopes
- Reference Classes
- XTA Jython interface
- Code reference grammar

This chapter lists all the commands and options supported by the XTA, reference classes, and a reference to the grammar.

15.1 Commands

15.1.1 add

Function	**add branch <from BRANCH> [<to INSTRUCTION>]+**
Description	Adds the given from/to references to the branches list
Type	`add branch <from BRANCH> [<to INSTRUCTION>]+`

Function	**add tile <tile id|*>**
Description	Add xCORE tile to active set
Type	`add tile <tile id|*>`

Function	**add exclusion <ANY>**
Description	Adds the given reference to the list of exclusions
Type	`add exclusion <ANY>`

Function	add functiontime <FUNCTION> <value> <MODE>
Description	Adds the given function time to the list of defines
Type	`add functiontime <FUNCTION> <value> <MODE>`

Function	add instructiontime <ENDPOINT> <value> <MODE>
Description	Adds the given instruction time to the list of defines
Type	`add instructiontime <ENDPOINT> <value> <MODE>`

Function	add loop <ANY> <iterations>
Description	Adds the given loop count define to the list of defines
Type	`add loop <ANY> <iterations>`

Function	add looppath <ANY> <iterations>
Description	Adds the given loop path count define to the list of defines
Type	`add looppath <ANY> <iterations>`

Function	add loopscope <ANY> <SCOPE>
Description	Adds the given loop scope define to the list of defines
Type	`add loopscope <ANY> <SCOPE>`

Function	add pathtime <from ENDPOINT> <to ENDPOINT> <value> <MODE>
Description	Adds the given path time to the list of defines
Type	`add pathtime <from ENDPOINT> <to ENDPOINT> <value> <MODE>`

15.1.2 analyze

Function	analyze endpoints <from ENDPOINT> <to ENDPOINT>
Description	Analyzes between the specified endpoints
Type	`analyze endpoints <from ENDPOINT> <to ENDPOINT>`

Function	analyze function <FUNCTION>
Description	Analyzes the given function
Type	`analyze function <FUNCTION>`

Function	analyze loop <ANY>
Description	Analyzes the given loop
Type	`analyze loop <ANY>`

15.1.3 config

Function	config case <best/worst>
Description	Sets the case (currently: worst)
Type	`config case <best/worst>`

Function	config Ewarning <on/off>
Description	Treats errors as warnings or not (currently: off)
Type	`config Ewarning <on/off>`

Function	config freq <node id> <tile frequency>
Description	Sets the operating frequency in MHz for the given node
Type	`config freq <node id> <tile frequency>`

Function	config from <ENDPOINT>
Description	Sets the from endpoint
Type	`config from <ENDPOINT>`

Function	config looppoint <ANY>
Description	Sets the loop point
Type	`config looppoint <ANY>`

Function	config scale <true/false>
Description	Configures whether results are scales (currently: true)
Type	`config scale <true/false>`

Function	config srcpaths <paths>
Description	Sets the (semicolon separated) source search path
Type	`config srcpaths <paths>`

Function	config cores <tile id> <num cores>
Description	Sets number of cores currently executing for the given tile
Type	`config cores <tile id> <num cores>`

Function	config timeout <seconds>
Description	Sets the tools timeout on load
Type	`config timeout <seconds>`

Function	config Terror <on/off>
Description	Treat timing failures as errors or not (currently: on)
Type	`config Terror <on/off>`

Function	config to <ENDPOINT>
Description	Sets the to endpoint
Type	`config to <ENDPOINT>`

Function	config verbosity <level>
Description	Sets the tool verbosity level (range: -10 -> +10, default: 0)
Type	`config verbosity <level>`

Function	config Werror <on/off>
Description	Treats warnings as errors or not (currently: off)
Type	`config Werror <on/off>`

15.1.4 clear

Function	clear
Description	Clears the screen (GUI mode only)
Type	`clear()`

15.1.5 debug

Function	dumpactiveexclusions
Description	Dumps a list of PCs that the exclusions have resolved to
Type	`debug dumpactiveexclusions()`

Function	debug dumpcachedfunction <FUNCTION>
Description	Dumps the cached function structure
Type	`debug dumpcachedfunction <FUNCTION>`

Function	dumpcallgraph
Description	Dumps the call graph for all tiles in dot (graphviz) format
Type	`debug dumpcallgraph()`

Function	debug dumpcontrolflow <FUNCTION>
Description	Dumps the control flow graph for the given function in dot (graphviz) format
Type	`debug dumpcontrolflow <FUNCTION>`

Function	dumpmanual
Description	Dumps the console reference chapter of the manual in tex format
Type	`debug dumpmanual()`

Function	debug dumpstacknodes <REFERENCE>
Description	Dumps the stack nodes for the given reference
Type	`debug dumpstacknodes <REFERENCE>`

Function	dumpunresolvedinstructions
Description	Dumps a list of instructions that are unresolved
Type	`debug dumpunresolvedinstructions()`

Function	debug verifyreference <ANY>
Description	Verifies the existance of the given reference
Type	`debug verifyreference <ANY>`

Function	frompoints
Description	Displays the from endpoints currently configured
Type	`debug frompoints()`

Function	topoints
Description	Displays the to endpoints currently configured
Type	`debug topoints()`

Function	debug instructiontime <route id> <node id>
Description	Displays the instruction time set for the given node in the given route
Type	`debug instructiontime <route id> <node id>`

Function	debug loop <route id> <node id>
Description	Displays the loop iterations set for the given node in the given route
Type	`debug loop <route id> <node id>`

Function	debug looppath <route id> <node id>
Description	Displays the loop path iterations set for the given node in the given route
Type	`debug looppath <route id> <node id>`

Function	debug loopscope <route id> <node id>
Description	Displays the loop scope set for the given node in the given route
Type	`debug loopscope <route id> <node id>`

Function	debug listglobalreferences <ANY>
Description	Lists all the matching references for the given reference on the global tree
Type	`debug listglobalreferences <ANY>`

Function	debug listroutereferences <route id> <ANY>
Description	Lists all the matching references for the given reference on the given route
Type	`debug listroutereferences <route id> <ANY>`

Function	memusage
Description	Displays the current memory usage for the JVM
Type	`debug memusage()`

Function	getmemthreshold
Description	Displays the current memory usage threshold
Type	`debug getmemthreshold()`

Function	debug setmemthreshold <threshold>
Description	Sets the memory threshold to the given value (0.0 - 1.0)
Type	`debug setmemthreshold <threshold>`

15.1.6 echo

Function	echo "text"
Description	Prints the text to the console
Type	`echo "text"`

15.1.7 exit

Function	exit
Description	Quits the application
Type	`exit()`

15.1.8 help

Function	help [command|command subcommand|option]
Description	Displays help message for the given arguments
Type	`help [command|command subcommand|option]`

15.1.9 history

Function	history
Description	Displays the command history
Type	`history()`

15.1.10 load

Function	load <xe file>
Description	Loads the given XMOS executable file
Type	`load <xe file>`

15.1.11 list

Function	allcalls
Description	Lists all the possible locations for calls
Type	`list allcalls()`

Function	allendpoints
Description	Lists all the possible locations for endpoints
Type	`list allendpoints()`

Function	list branches [route id]
Description	Lists the branches - optionally for the specified route
Type	`list branches [route id]`

Function	calls
Description	Lists the calls
Type	`list calls()`

Function	tiles
Description	Lists the active xCORE tiles

Continued on next page

Type	`list tiles()`

Function	**endpoints**
Description	Lists the endpoints
Type	`list endpoints()`

Function	**list exclusions [route id]**
Description	Displays the exclusions - optionally for the specified route
Type	`list exclusions [route id]`

Function	**functions**
Description	Lists the functions in the loaded application
Type	`list functions()`

Function	**functiontimes**
Description	Displays the function time defines
Type	`list functiontimes()`

Function	**instructiontimes**
Description	Displays the instruction time defines
Type	`list instructiontimes()`

Function	list knowns <route id>
Description	Displays the list of knowns set for the given route
Type	`list knowns <route id>`

Function	labels
Description	Lists the labels
Type	`list labels()`

Function	loops
Description	Displays the loop defines
Type	`list loops()`

Function	looppaths
Description	Displays the loop path defines
Type	`list looppaths()`

Function	loopscopes
Description	Displays the loop scope defines
Type	`list loopscopes()`

Function	pathtimes
Description	Displays the path time defines
Type	`list pathtimes()`

Function	sources
Description	Lists the source files
Type	`list sources()`

Function	srccommands
Description	Displays the command list embedded in the loaded executable
Type	`list srccommands()`

Function	srcloops
Description	Displays the loop counts embedded in the loaded executable
Type	`list srcloops()`

Function	srclabels
Description	Lists the source labels
Type	`list srclabels()`

Function	allsrclabels
Description	Lists all the possible locations for source labels
Type	`list allsrclabels()`

Function	corestartpoints
Description	Lists the logical core start points
Type	`list corestartpoints()`

Function	corestoppoints
Description	Lists the logical core stop points
Type	`list corestoppoints()`

Function	list unknowns <route id>
Description	Displays the list of unknowns for the given route
Type	`list unknowns <route id>`

15.1.12 print

Function	summary
Description	Shows routes summary (verbosity -2|-1|0)
Type	`print summary()`

Function	print structure <route id> [node id]
Description	Displays the structure for given route/node (verbosity 0|1)
Type	`print structure <route id> [node id]`

Function	print asm <route id> [node id]
Description	Displays annotated assembly for the given route/node
Type	`print asm <route id> [node id]`

Function	print src <route id> [node id]
Description	Displays annotated source file(s) for given route/node
Type	`print src <route id> [node id]`

Function	print trace <route id> [node id]
Description	Displays instruction trace for the worst case path of the given route/node
Type	`print trace <route id> [node id]`

Function	print routeinfo <route id>
Description	Shows detailed information for the given route
Type	`print routeinfo <route id>`

Function	print nodeinfo <route id> <node id>
Description	Shows detailed information for the given node in the given route
Type	`print nodeinfo <route id> <node id>`

Function	warnings
Description	Prints all timing warnings
Type	`print warnings()`

Function	print distribution <route id> [node id]
Description	Displays time distribution for the given route/node
Type	`print distribution <route id> [node id]`

15.1.13 pwd

Function	pwd
Description	Displays the current working directory
Type	`pwd()`

15.1.14 remove

Function	**remove branch <from BRANCH|*> [<to INSTRUCTION|*>]+**
Description	Removes the given from/to references from the branches list
Type	`remove branch <from BRANCH|*> [<to INSTRUCTION|*>]+`

Function	**remove tile <tile id|*>**
Description	Removes xCORE tile from active set
Type	`remove tile <tile id|*>`

Function	**remove exclusion <ANY|*>**
Description	Removes the given reference (or all if '*') from the list of exclusions
Type	`remove exclusion <ANY|*>`

Function	**remove functiontime <FUNCTION|*>**
Description	Removes the given functon time from the list of defines
Type	`remove functiontime <FUNCTION|*>`

Function	**remove instructiontime <ENDPOINT|*>**
Description	Removes the given instruction time from the list of defines
Type	`remove instructiontime <ENDPOINT|*>`

Function	**remove loop <ANY|*>**
Description	Removes the given loop count define to the list of defines
Type	`remove loop <ANY|*>`

Function	**remove looppath <ANY|*>**
Description	Removes the given loop path count define to the list of defines
Type	`remove looppath <ANY|*>`

Function	**remove loopscope <ANY|*>**
Description	Removes the given loop scope define to the list of defines
Type	`remove loopscope <ANY|*>`

Function	**remove pathtime <from ENDPOINT|*> <to ENDPOINT|*>**
Description	Removes the given path time from the list of defines
Type	`remove pathtime <from ENDPOINT|*> <to ENDPOINT|*>`

Function	**remove route <route id>**
Description	Removes the route with the given id from the current analysis
Type	`remove route <route id>`

15.1.15 scripter

Function	**scripter disable <ANY>**
Description	Disables a mapping
Type	`scripter disable <ANY>`

Function	**dump**
Description	Dumps script which represents the current state - also embeds active pragmas into source

Continued on next page

Type	`scripter dump()`

Function	**scripter embed <filename>**
Description	Embeds the script into the designated file - also embed active pragmas into source
Type	`scripter embed <filename>`

Function	**scripter enable <ANY>**
Description	Enables a mapping
Type	`scripter enable <ANY>`

Function	**listrefs**
Description	Lists all references which will be used in the script
Type	`scripter listrefs()`

Function	**scripter rename <ANY> <TO_NAME>**
Description	Renames a mapping
Type	`scripter rename <ANY> <TO_NAME>`

15.1.16 set

Function	**set exclusion <route id> <ANY>**
Description	Sets an exclusion on the given reference
Type	`set exclusion <route id> <ANY>`

Function	set functiontime <route id> <FUNCTION> <value> <MODE>
Description	Sets timing requirement for the given function on the given route
Type	`set functiontime <route id> <FUNCTION> <value> <MODE>`

Function	set instructiontime <route id> <ENDPOINT> <value> <MODE>
Description	Sets the time taken for the instruction at the given pc
Type	`set instructiontime <route id> <ENDPOINT> <value> <MODE>`

Function	set loop <route id> <ANY> <iterations>
Description	Sets the number of iterations for the loop identified
Type	`set loop <route id> <ANY> <iterations>`

Function	set looppath <route id> <ANY> <iterations>
Description	Sets the number of iterations for the path identified
Type	`set looppath <route id> <ANY> <iterations>`

Function	set loopscope <route id> <ANY> <SCOPE>
Description	Sets the scope of the referenced loop
Type	`set loopscope <route id> <ANY> <SCOPE>`

Function	set pathtime <route id> <from ENDPOINT> <to ENDPOINT> <value> <MODE>
Description	Sets timing requirement for the given path on the given route
Type	`set pathtime <route id> <from ENDPOINT> <to ENDPOINT> <value> <MODE>`

Function	set required <route id> <value> <MODE>
Description	Sets the maximum allowed time taken for the given route
Type	`set required <route id> <value> <MODE>`

15.1.17 source

Function	source <file name> [args]
Description	Sources the given script file
Type	`source <file name> [args]`

15.1.18 status

Function	status
Description	Displays current status
Type	`status()`

15.1.19 version

Function	version
Description	Displays the version information
Type	`version()`

15.2 Pragmas

Function	#pragma xta label "name"
Description	Provides a label that can be used to specify timing constraints.
Type	`#pragma xta label "name"`

Function	#pragma xta endpoint "name"
Description	Specifies an endpoint. It may appear before an input or output statement.
Type	`#pragma xta endpoint "name"`

Function	#pragma xta call "name"
Description	Defines a label for a (function) call point. Use to specify a particular called instance of a function. For example, if a function contains a loop, the iterations for this loop can be set to a different value depending on which call point the function was called from.
Type	`#pragma xta call "name"`

Function	#pragma xta command "command"
Description	Allows XTA commands to be embedded into source code. All commands are run every time the binary is loaded into the XTA. Commands are executed in the order they occur in the file, but the order between commands in different source files is not defined.
Type	`#pragma xta command "command"`

Function	#pragma xta loop "integer"
Description	Applies the given loop XTA iterations to the loop containing the pragma.
Type	`#pragma xta loop "integer"`

15.3 Timing Modes

The available timing modes are:

Function	ns
Description	nanoseconds
Type	`ns()`

Function	us
Description	microseconds
Type	`us()`

Function	ms
Description	milliseconds
Type	`ms()`

Function	MHz
Description	megahertz
Type	`MHz()`

Function	KHz
Description	kilohertz
Type	`KHz()`

Function	Hz
Description	hertz
Type	`Hz()`

Function	cycles
Description	The core cycle count is the number of scheduled slots that the logical core required to perform the sequence. The relationship between core cycles and time is a function of the number of cores currently running and the xCORE tile frequency.

Continued on next page

Type	`cycles()`

15.4 Loop Scopes

Supported values for scope are:

Function	**r**
Description	Iteration number propagates to the enclosing path (Default)
Type	`relative/r()`

Function	**a**
Description	Absolute number of iterations
Type	`absolute/a()`

15.5 Reference Classes

15.5.1 FUNCTION

Function	**FunctionPc**
Description	Raw program counter specified in the format: 0x*
Type	`FunctionPc()`

Function	**Function**
Description	Any function
Type	`Function()`

15.5.2 BRANCH

Function	EndpointPC
Description	Raw program counter specified in the format: 0x*
Type	`EndpointPC()`

Function	CallPc
Description	Call specified in the format: 0x*
Type	`CallPc()`

Function	CallFileLine
Description	Call specified in the format: 'file name:line number'
Type	`CallFileLine()`

Function	Call
Description	Call specified using the source level pragma mechanism
Type	`Call()`

Function	Label
Description	Any source or assembly level symbol defined with respect to an executable section
Type	`Label()`

Function	CallLabel
Description	Any source or assembly level symbol defined with respect to an executable section
Type	`CallLabel()`

15.5.3 INSTRUCTION

Function	EndpointPC
Description	Raw program counter specified in the format: 0x*
Type	`EndpointPC()`

Function	FunctionPc
Description	Raw program counter specified in the format: 0x*
Type	`FunctionPc()`

Function	Function
Description	Any function
Type	`Function()`

Function	Label
Description	Any source or assembly level symbol defined with respect to an executable section
Type	`Label()`

15.5.4 ENDPOINT

Function	EndpointPC
Description	Raw program counter specified in the format: 0x*
Type	`EndpointPC()`

Function	EndpointFileLine
Description	Endpoint specified in the format: 'file name:line number'
Type	`EndpointFileLine()`

Function	Endpoint
Description	Endpoint specified using the source level pragma mechanism
Type	`Endpoint()`

Function	CallPc
Description	Call specified in the format: 0x*
Type	`CallPc()`

Function	CallFileLine
Description	Call specified in the format: 'file name:line number'
Type	`CallFileLine()`

Function	Call
Description	Call specified using the source level pragma mechanism
Type	`Call()`

Function	Label
Description	Any source or assembly level symbol defined with respect to an executable section
Type	`Label()`

Function	CallLabel
Description	Any source or assembly level symbol defined with respect to an executable section
Type	`CallLabel()`

15.5.5 ANY

Function	SrcLabelPc
Description	Raw program counter specified in the format: 0x*
Type	`SrcLabelPc()`

Function	EndpointPC
Description	Raw program counter specified in the format: 0x*
Type	`EndpointPC()`

Function	EndpointFileLine
Description	Endpoint specified in the format: 'file name:line number'
Type	`EndpointFileLine()`

Function	Endpoint
Description	Endpoint specified using the source level pragma mechanism
Type	`Endpoint()`

Function	CallPc
Description	Call specified in the format: 0x*

Continued on next page

Type	`CallPc()`

Function	**CallFileLine**
Description	Call specified in the format: ‘file name:line number’
Type	`CallFileLine()`

Function	**Call**
Description	Call specified using the source level pragma mechanism
Type	`Call()`

Function	**SrcLabelFileLine**
Description	Source label specified in the format: ‘file name:line number’
Type	`SrcLabelFileLine()`

Function	**SrcLabel**
Description	Source label specified using the source level pragma mechanism
Type	`SrcLabel()`

Function	**Label**
Description	Any source or assembly level symbol defined with respect to an executable section
Type	`Label()`

Function	CallLabel
Description	Any source or assembly level symbol defined with respect to an executable section
Type	`CallLabel()`

15.5.6 FUNCTION_WITH_EVERYTHING

Function	EverythingReference
Description	Matches everything: '*'
Type	`EverythingReference()`

Function	FunctionPc
Description	Raw program counter specified in the format: 0x*
Type	`FunctionPc()`

Function	Function
Description	Any function
Type	`Function()`

15.5.7 BRANCH_WITH_EVERYTHING

Function	EverythingReference
Description	Matches everything: '*'
Type	`EverythingReference()`

Function	EndpointPC
Description	Raw program counter specified in the format: 0x*
Type	`EndpointPC()`

Function	CallPc
Description	Call specified in the format: 0x*
Type	`CallPc()`

Function	CallFileLine
Description	Call specified in the format: 'file name:line number'
Type	`CallFileLine()`

Function	Call
Description	Call specified using the source level pragma mechanism
Type	`Call()`

Function	Label
Description	Any source or assembly level symbol defined with respect to an executable section
Type	`Label()`

Function	CallLabel
Description	Any source or assembly level symbol defined with respect to an executable section
Type	`CallLabel()`

15.5.8 INSTRUCTION_WITH_EVERYTHING

Function	**EverythingReference**
Description	Matches everything: '*'
Type	`EverythingReference()`

Function	**EndpointPC**
Description	Raw program counter specified in the format: 0x*
Type	`EndpointPC()`

Function	**FunctionPc**
Description	Raw program counter specified in the format: 0x*
Type	`FunctionPc()`

Function	**Function**
Description	Any function
Type	`Function()`

Function	**Label**
Description	Any source or assembly level symbol defined with respect to an executable section
Type	`Label()`

15.5.9 ENDPOINT_WITH_EVERYTHING

Function	EverythingReference
Description	Matches everything: '*'
Type	`EverythingReference()`

Function	EndpointPC
Description	Raw program counter specified in the format: 0x*
Type	`EndpointPC()`

Function	EndpointFileLine
Description	Endpoint specified in the format: 'file name:line number'
Type	`EndpointFileLine()`

Function	Endpoint
Description	Endpoint specified using the source level pragma mechanism
Type	`Endpoint()`

Function	CallPc
Description	Call specified in the format: 0x*
Type	`CallPc()`

Function	CallFileLine
Description	Call specified in the format: 'file name:line number'
Type	`CallFileLine()`

Function	Call
Description	Call specified using the source level pragma mechanism
Type	`Call()`

Function	Label
Description	Any source or assembly level symbol defined with respect to an executable section
Type	`Label()`

Function	CallLabel
Description	Any source or assembly level symbol defined with respect to an executable section
Type	`CallLabel()`

15.5.10 ANY_WITH_EVERYTHING

Function	EverythingReference
Description	Matches everything: '*'
Type	`EverythingReference()`

Function	SrcLabelPc
Description	Raw program counter specified in the format: 0x*
Type	`SrcLabelPc()`

Function	EndpointPC
Description	Raw program counter specified in the format: 0x*
Type	`EndpointPC()`

Function	EndpointFileLine
Description	Endpoint specified in the format: 'file name:line number'
Type	`EndpointFileLine()`

Function	Endpoint
Description	Endpoint specified using the source level pragma mechanism
Type	`Endpoint()`

Function	CallPc
Description	Call specified in the format: 0x*
Type	`CallPc()`

Function	CallFileLine
Description	Call specified in the format: 'file name:line number'
Type	`CallFileLine()`

Function	Call
Description	Call specified using the source level pragma mechanism
Type	`Call()`

Function	SrcLabelFileLine
Description	Source label specified in the format: 'file name:line number'
Type	`SrcLabelFileLine()`

Function	SrcLabel
Description	Source label specified using the source level pragma mechanism
Type	`SrcLabel()`

Function	Label
Description	Any source or assembly level symbol defined with respect to an executable section
Type	`Label()`

Function	CallLabel
Description	Any source or assembly level symbol defined with respect to an executable section
Type	`CallLabel()`

15.6 XTA Jython interface

The Jython interface to the global xta object is as follows:

15.6.1 Load methods

```
void load(String fileName) throws Exception
```

15.6.2 Route creation/deletion methods

```
List<Integer> analyzeFunction(String functionName) throws Exception
List<Integer> analyzeEndpoints(String fromRef, String toRef) throws Exception
List<Integer> analyzeLoop(String loopRef) throws Exception
void removeRoute(int routeId) throws Exception
```

15.6.3 Add/remove methods

```
void addTile(String tileReference) throws Exception
void removeTile(String tileReference) throws Exception
Collection<String> getTiles() throws Exception

void addExclusion(String ref) throws Exception
void removeExclusion(String ref) throws Exception
Collection<String> getExclusions() throws Exception

void addBranch(String fromRefString, Collection<String> toRefStrings)
throws Exception
void removeBranch(String fromRefString, Collection<String> toRefStrings)
throws Exception
Collection<String> getBranches() throws Exception
Collection<String> getBranchTargets(String branch) throws Exception

void addLoop(String ref, long iterations) throws Exception
void removeLoop(String ref) throws Exception
Collection<String> getLoops() throws Exception

void addLoopPath(String ref, long iterations) throws Exception
void removeLoopPath(String ref) throws Exception
Collection<String> getLoopPaths() throws Exception

void addLoopScope(String ref, boolean absolute) throws Exception
void removeLoopScope(String ref) throws Exception
Collection<String> getLoopScopes() throws Exception

void addInstructionTime(String ref, double value, String units)
throws Exception
void removeInstructionTime(String ref) throws Exception
Collection<String> getInstructionTimes() throws Exception

void addFunctionTime(String ref, double value, String units)
throws Exception
void removeFunctionTime(String ref) throws Exception
Collection<String> getFunctionTimes() throws Exception

void addPathTime(String fromRef, String toRef, double value, String units)
throws Exception
void removePathTime(String fromRef, String toRef) throws Exception
Collection<String> getPathTimes() throws Exception
```

15.6.4 Set methods

```
void setRequired(int routeId, double value, String units) throws Exception
void setFunctionTime(int routeId, String refString, double value,
String units) throws Exception
void setPathTime(int routeId, String fromRef, String toRef, double value,
String units) throws Exception
void setInstructionTime(int routeId, String refString, double value,
```

```
String units) throws Exception
void setLoop(int routeId, String refString, long iterations)
throws Exception
void setLoopPath(int routeId, String refString, long iterations)
throws Exception
void setLoopScope(int routeId, String refString, boolean absolute)
throws Exception
void setExclusion(int routeId, String refString) throws Exception
```

15.6.5 Get methods

```
double getRequired(int routeId, String units) throws Exception
double getWorstCase(int routeId, String units) throws Exception
double getBestCase(int routeId, String units) throws Exception
List<String> getWarnings(int routeId) throws Exception
List<String> getErrors(int routeId) throws Exception
List<Integer> getRouteIds() String getRouteDescription(int routeId)
throws Exception
```

15.6.6 Config methods

```
void configCores(String tileReference, int numCores) throws Exception
void configFreq(String nodeId, double tileFrequency) throws Exception
```

15.7 Code reference grammar

A code reference constructed of a *back trail* and a base reference of the form:

code-ref ::= *back-trail base-ref*

back-trail ::= *base-ref*
| *base-ref , back-trail*

base-ref ::= *pc-ref*
| *label-ref*
| *function-ref*
| *endpoint-ref*
| *srclabel-ref*
| *call-ref*

pc-ref ::= *pc-class hex-constant*

label-ref ::= *label-class label-string*

function-ref ::= *function-class function-name*
| *function-class hex-constant*

endpoint-ref ::= *endpoint-class file-line*
| *endpoint-class endpoint-label*
| *endpoint-class hex-constant*

srclabel-ref ::= *srclabel-class label-string*

call-ref ::= *call-class label-string*
| *call-class hex-constant*

pc-class ::=
| *PC:*

label-class ::=
| *LABEL:*

functionclass ::=
| *FUNCTION:*

endpointclass ::=
| *ENDPOINT:*

srclabelclass ::=
| *SRCLABEL:*

call-class ::=
| *CALL:*

file-line ::= *file-name : integer-constant*

Part F

Run on Hardware

CONTENTS

- Use xTIMEcomposer to run a program
- XRUN Command-Line Manual

16 Use xTIMEcomposer to run a program

IN THIS CHAPTER

- Create a Run Configuration
- Re-run a program

xTIMEcomposer uses **Run Configurations** to determine the settings used to run a program. Run Configurations are specific to the project and target platform.

16.1 Create a Run Configuration

To create a Run Configuration, follow these steps:

1. Select a project in the **Project Explorer**.
2. Choose **Run ▶ Run Configurations**.
3. In the left panel, double-click **XCore Application**.

   xTIMEcomposer creates a new configuration and displays the default settings in the right panel, as shown in Figure 25.
4. In **Name**, enter a name for the configuration.
5. xTIMEcomposer tries to identify the target project and executable for you. To select one yourself, click **Browse** to the right of the **Project** text box and select your project in the **Project Selection** dialog box. Then click **Search Project** and select the executable file in the **Program Selection** dialog box.

   You must have previously compiled your program without any errors for the executable to be available for selection.
6. If you have a development board connected to your system, check the **hardware** option and select your debug adapter from the **Target** list. Alternatively, check the **simulator** option to run your program on the XMOS simulator.
7. Click **Run**.

xTIMEcomposer loads your executable, displaying any output generated by your program in the **Console**.

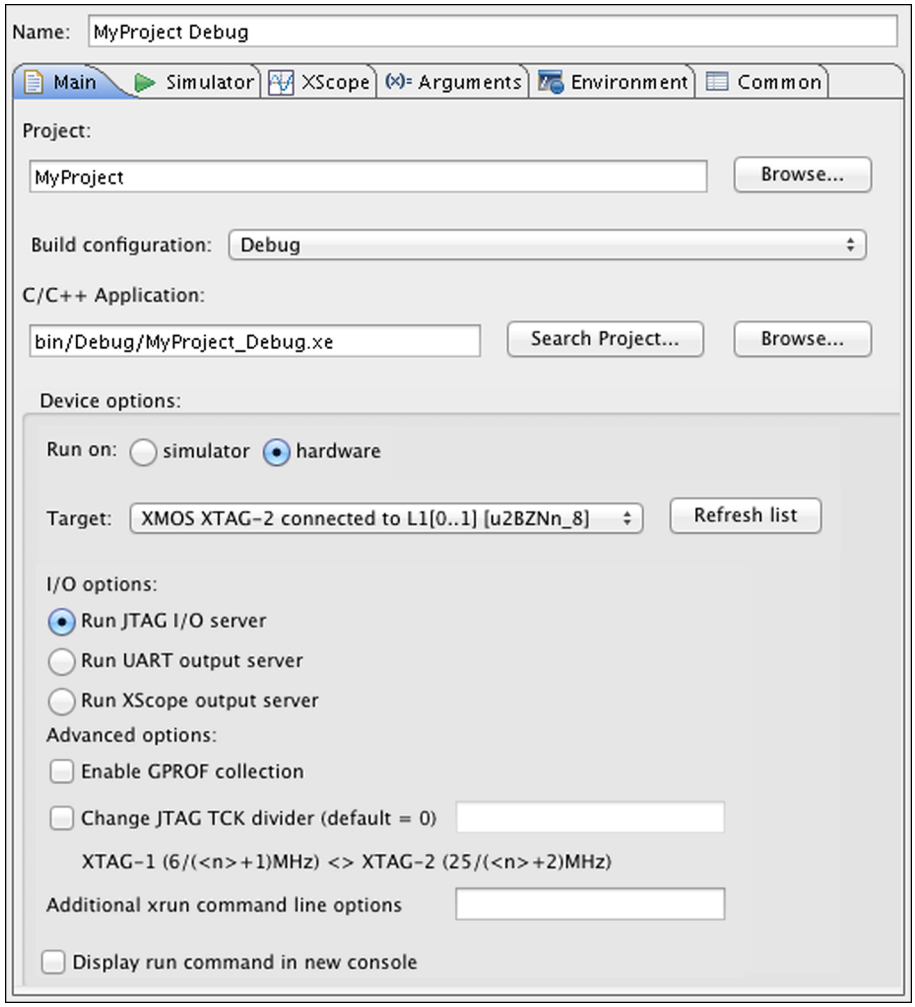


Figure 25: Run Configuration window

16.2 Re-run a program

xTIMEcomposer remembers the configuration last used to run your program. To run it again using the same configuration, just click the **Run** button. To use a different configuration, click the arrow to the right of the **Run** button and select a configuration from the drop-down list.

17 XRUN Command-Line Manual

IN THIS CHAPTER

- Overall Options
- Target Options
- Debugging Options
- xSCOPE Options

XRUN loads and runs XMOS Executable (XE) files on target hardware. It requires either the XMOS or FTDI USB-to-JTAG drivers to be installed, depending on the adapter used with the target hardware (see §2).

17.1 Overall Options

The following options are used to specify an executable to run and, optionally, an xCORE tile on which to run the program.

`xe-file` Specifies an XE file to load and run.

`--verbose` Prints information about the program loaded onto the target devices.

`--help` Prints a description of the supported command line options.

`--version` Displays the version number and copyrights.

17.2 Target Options

The following options are used to specify a target hardware platform.

`--list-devices`
`-l` Prints an enumerated list of all JTAG adapters connected to the host and the devices on each JTAG chain, in the form:

```
    ID        Name        Adapter ID        Devices

   --        ----        ----------        -------
```

The adapters are ordered by their serial numbers.

`--list-board-info`
`-lb` Displays information about the connected target board.

`--id ID` Specifies the adapter connected to the target hardware.

`--adapter-id ADAPTER-SERIAL-NUMBER`
Specifies the serial number of the adapter connected to the target hardware.

`--jtag-speed n`
: Sets the divider for the JTAG clock to *n*. If unspecified, the default value is 0. The maximum value is 70.

  For FTDI-based debug adapters, the JTAG clock speed is set to 6/(*n*+1)MHz.

  For XMOS-based debug adapters, the JTAG clock speed is set to 25/(*n*+1)MHz.

`--noreset`
: Does not reset the XMOS devices on the JTAG scan chain before loading the program. This is not default.

17.3 Debugging Options

The following options are used to enable debugging capabilities.

`--io`
: Causes XRUN to remain attached to the JTAG adapter after loading the program, enabling system calls with the host. XRUN terminates when the program calls `exit`.

  By default, XRUN disconnects from the JTAG adapter once the program is loaded.

`--uart`
: Enables a UART server that interfaces with the UART-to-USB converter on the XMOS USB-to-JTAG adapters. The converter operates at a rate of 115200 bits/sec.

  The USB-to-UART converter on XMOS adapter interfaces with two pins on the XSYS connector that, on XMOS development boards, are connected to ports on an XMOS device. The ports are named in the XN files as PORT_UART_TX and PORT_UART_RX.

  This option is not supported for adapters based on FTDI chips.

`--attach`
: Attaches to a JTAG adapter (of a running program), enabling system calls with the host. XRUN terminates when the program performs a call to `exit`.

  An XE file must be specified with this option.

`--dump-state`
: Prints the core, register and stack contents of all xCORE Tiles in JTAG scan chain.

17.4 xSCOPE Options

The following options are used to enable xSCOPE capabilities.

`--xscope` Enables an xSCOPE server with the target.

`--xscope-realtime`
Enables an xSCOPE server with the target using a socket connection.

`--xscope-file filename`
Specifies the filename for xSCOPE data collection.

`--xscope-port ip:port`
Specifies the IP address and port for realtime data capture.

`--xscope-limit limit`
Specifies the record limit for xSCOPE data collection.

Part G

Application Instrumentation and Tuning

CONTENTS

- Use xTIMEcomposer and xSCOPE to trace data in real-time
- xSCOPE performance figures
- xSCOPE Library API

18 Use xTIMEcomposer and xSCOPE to trace data in real-time

IN THIS CHAPTER

- XN File Configuration
- Instrument a program
- Configure and run a program with tracing enabled
- Analyze data offline
- Analyze data in real-time
- Trace using the UART interface

xTIMEcomposer and the xSCOPE library let you instrument your program with probes that collect application data in real-time. This data can be sent over an XTAG-2 debug adapter to xTIMEcomposer for real-time display or written to a file for offline analysis.

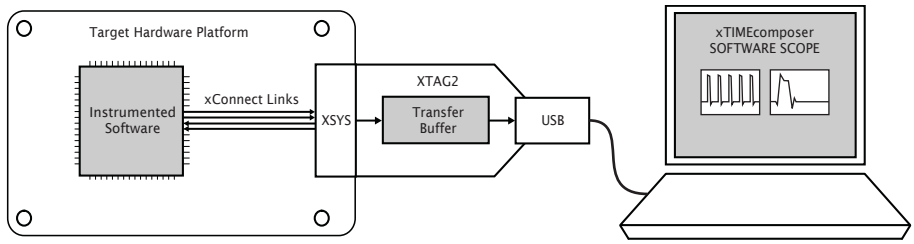


Figure 26: xSCOPE connectivity

If you are using a legacy FTDI or XTAG-1 debug adapter, or if the XSYS connector on your target hardware does not provide an xCONNECT Link, you can configure the probes to output trace data over your adapter's UART interface instead (see XM-000957-PC). Note that the UART interface is supported on a single tile only and offers significantly reduced performance.

If your hardware requires an xTAG adapter board (XA-SK-XTAG2) to connect the xTAG, the XMOS LINK switch on the adapter board must be set to ON to use xSCOPE.

18.1 XN File Configuration

To allow the tools to configure the xCONNECT link required for high speed data collection using xscope, the XN file for a board must be modified to expose the connection to the XTAG-2 device. The following information must be added to the links section of an XN file for a board to set up the link used by the target device to communicate with the XTAG-2 and the xscope channel.

```
<Link Encoding="2wire" Delays="4,4" Flags="XSCOPE">
  <LinkEndpoint NodeId="0" Link="X0LD"/>
  <LinkEndpoint RoutingId="0x8000" Chanend="1"/>
</Link>
```

Note that when the link is set to 2 wire, the minimum delay is set to 4 and the flags specify that this link is to be used for streaming debug. Setting the delay higher results in the output of packets used by xscope being less frequent. The RoutingId is also important as the value 0x8000 specifies to the tools that this is a special link used for xscope.

When used in a multi-tile system the NodeId of the package which is connected to the XSYS connector must be specified. The tools set up the links with the other tiles but they need to know which specific device has the external link to be connected to the XTAG-2.

18.2 Instrument a program

The example program in Figure 27 uses the xSCOPE instrumentation functions to trace the input levels to a microphone.

```
#include <xscope.h>

port micL;
port micR;

void xscope_user_init(void) {
  xscope_register(2,
    XSCOPE_CONTINUOUS, "Microphone Left",  XSCOPE_UINT, "mV",
    XSCOPE_CONTINUOUS, "Microphone Right", XSCOPE_UINT, "mV"
  );
}

int main() {
  while (1) {
    int sample;
    micL :> sample;
    xscope_uint(0, sample);
    micR :> sample;
    xscope_uint(1, sample);
  }
}
```

Figure 27: Program that traces input levels to a microphone

The constructor `xscope_user_init` registers two probes for tracing the left and right inputs to a microphone. The probes are defined as **continuous**, which means xTIMEcomposer can interpolate values between two subsequent measurements. The probes are defined to take values of type `unsigned int`.

In `main`, the program calls the probe function `xscope_uint` each time it samples data from the microphone. This function creates a trace record and sends it to the PC.

Figure 28 summarizes the different types of probes that can be configured. Only **continous** probes can be displayed real-time.

Probe Type	Data Type	Scope View	Example

`XSCOPE_CONTINUOUS`	`XSCOPE_UINT`
	`XSCOPE_INT`
	`XSCOPE_FLOAT`
		Line graph. May be interpolated	Voltage levels of a motor controller

`XSCOPE_DISCRETE`	`XSCOPE_INT`	Horizontal lines	Buffer levels of audio CODEC
`XSCOPE_STATEMACHINE`	`XSCOPE_UINT`	State machine	Progression of protocol

`XSCOPE_STARTSTOP`	`XSCOPE_NONE`
	`XSCOPE_UINT`
	`XSCOPE_INT`
	`XSCOPE_FLOAT`
		Start/stop bars	Recorded function entry and exit, with optional label value

Figure 28: Supported probe types

18.3 Configure and run a program with tracing enabled

Once you have instrumented your program, you must compile and link it with the xSCOPE library, and run it in either offline or real-time mode.

To link with the xSCOPE library and run xSCOPE, follow these steps:

1. Open the Makefile for your project.
2. Locate the `XCC_FLAGS_`*config* variable for your build configuration, for example `XCC_FLAGS_Release`.
3. Add the option `-fxscope`.
4. Create a Run Configuration for your target device (see §16.1).
5. Click the ** xSCOPE** tab and select **Offline Mode** to save data to a file for offline analysis, or **Real-Time Mode** to output the data to the real-time viewer.
   - In offline mode, xTIMEcomposer logs trace data until program termination and saves the traced data to the file `xscope.xmt`. To change, enter a filename in the **Output file** text box. To limit the size of the trace file, enter a number in the **Limit records to** text box.
   - In real-time mode, xTIMEcomposer opens the Scope view and displays an animated view of the traced data as the program executes.
6. Click **Run** to save and run the configuration.

18.4 Analyze data offline

Double-click a trace file in *Project Explorer* to open it in the **Scope view**, as shown in Figure 29.

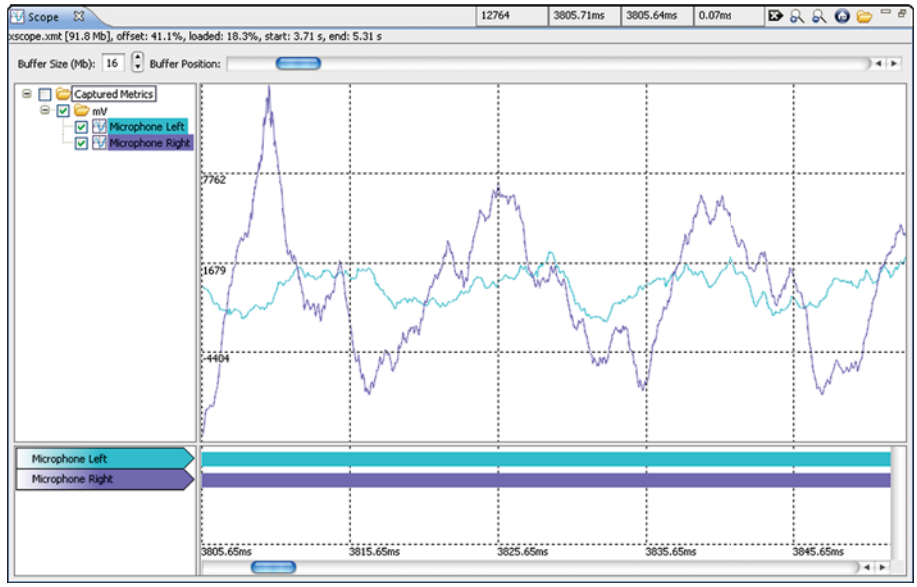


Figure 29: Offline Scope view

The top panel of the **Scope** view displays a graph of the data values for each selected probe: the *x*-axis represents time (as per the timeline in the bottom panel) and the *y*-axis represents the traced data values. The probes are grouped by their assigned units, and multiple probes with the same unit can be overlaid onto a single graph.

Moving the cursor over the scope data displays the current data (*y*-value) and time (*x*-value) in the first two of the four numeric boxes at the top of the window. Left-click on the view to display a marker as a red line - the associated time is displayed in the third numeric box. The fourth numeric box displays the difference between the marker time and the current cursor position.

If the cursor changes to a pointing finger, double-click to locate the statement in the source code responsible for generating the trace point.

The bottom panel of this view displays a timeline for each probe: vertical lines on a probe's timeline indicate times at which the probe created a record.

Drag the **Buffer Position** slider left or right to move through the timeline. To show more information in the window, increase the value in the **Buffer Size** field.

Use the **Scope** view toolbar at the top of the window to perform additional tasks:

To show data points for interpolated continuous signals, click the **Continuous**

points button.

To view all data points, click the **Zoom Fit** button.

Load a trace file that is not part of your project, click the **Open** button and browse to the file.

18.5 Analyze data in real-time

The *Scope* view can display trace data streamed from hardware in real-time. The left panel displays the signal information and controls and the right panel displays the screen view for the signals.

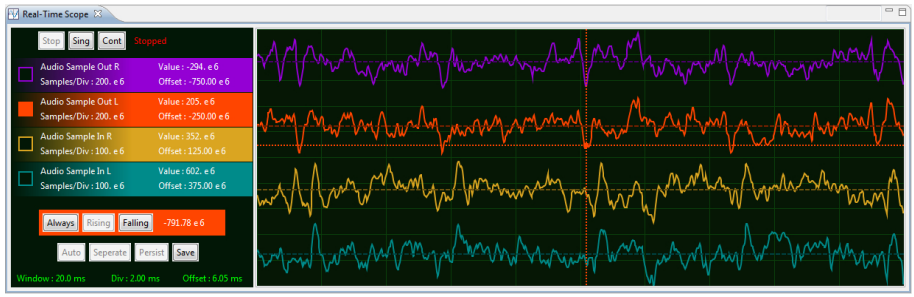


Figure 30: Real-Time Scope view

The left panel displays a list of the *continous* probes registered by the application (see §18.2). Each named probe is assigned a color that is used to draw events on the display, and which is used to identify the probe in the screen panel.

The *Scope* view is based around a traditional oscilloscope, and data is *captured* around a *trigger*, and then *displayed*. The capture mode, display mode, trigger and timebase are all controlled in the left panel. The right panel has 10 horizontal and vertical divisions, and the scales are all shown as units per division.

Numeric controls can all be modified by using the mouse: click the left button to increase the value or the right button to decrease the value. The scroll wheel can be used if your platform supports it (Mac OS/X, Linux, and some but not all versions of Windows).

18.5.1 Capture control

There are three capture-modes: *continous*, *single capture* or *stopped*. The default mode on start-up is for the system to capture and display continuously. The label associated with the capture controls shows the current state of the xSCOPE system.

Figure 31: Capture controls

Stop Display

Stops the screen panel from triggering and capturing, no more updates will be applied to the screen whilst this mode is set. The mode can be used to inspect the captured data. The mouse can be used to change signal and time base scales and offsets as described below to inspect the signals in detail. When stopped, you can zoom in on the time base and view the signal in more detail: the displayed signals are subsampled when the timebase is large, and zooming in on the timebase will reveal all data.

Single Capture

Selest single shot mode to capture one screen of data and return to the stopped state. If a trigger is enabled (see Figure 33) the system will wait for this trigger condition to be met before updating the screen and returning to the stopped state.

Continuous Capture

Select free running mode to update the screen as frequently as possible. If triggers are enabled, the screen will update only when the trigger is met.

18.5.2 Signal Control

The signal controls are available for each registered probe on the coloured label displayed in the left panel (see Figure 32)

Figure 32: Signal Controls

Enable / Disable Signal

Toggle the visibility of the signal by double clicking on the name.

Signal Samples/Div

Change the *Samples per Division* of this probe with the mouse buttons; this affects the vertical scale of the signal.

Signal Screen Offset

Change the vertical *Offset* of this probe with the mouse buttons; this affects the vertical position of the signal.

Signal Trigger

The signal can be used as a trigger (see Figure 33) by clicking in trigger box to the left of the probe label. Only one signal can be used for triggering.

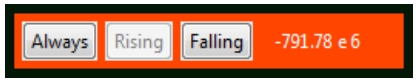
18.5.3 Trigger Control

A trigger can be used to restrict the system so that data is only captured when a condition is met. By default all triggers are disabled, causing data to be captured unconditionally. To enable triggering, a trigger must be selected by clicking on the box to the left of the probe label.

When triggering is enabled, a cross appears on the screen showing the trigger level (relative to the signal on which the trigger is selected) and the trigger offset on the timebase. The center of the cross is the time and value where the trigger happens/happened; to the left of this are the signals that lead up to the trigger; to the right are the signals after the trigger.

The trigger level and offset can be set directly by clicking in the right-hand pane. Changes only take effect if the scope is not stopped, and either running continuously, or set for a single trigger.

Figure 33: Trigger Controls



Always

Disables the trigger and captures data unconditionally.

Rising

Trigger on a rising edge of the signal. This is the default mode when selecting a signal to be used for triggering.

Falling

Trigger on a falling edge of the signal.

The value label associated with the enabled trigger shows the current trigger value set for the signal. This can be changed by using the mouse buttons.

18.5.4 Timebase Control

The timebase controls are used to set the time range for the signal capture window, allowing you to scale and shift the horizontal axis.

Figure 34: Timebase Controls

Time Window

The current size to the time window. Scales all signals in time and affects the time per division.

Time per Division

The time units per division. Scales all signals in time and affects the time window.

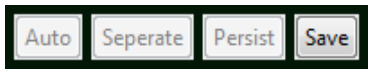
Time Window Offset

The position of the trigger in the time window. Shifts all signals left and right. Note that the trigger may not be visible, and could be to the left or right of the time window. The signals can be shifted right only a limited value.

18.5.5 Screen Control

Several commands are available that operate on all signals.

Figure 35: Screen Controls



Auto Range Signals

Automatically arranges all current signals to fit on the screen. The signals are measured for a short while, and each signal scaled and offset to fit the screen. All signals are displayed across each other.

Separate Signals

Similar to Auto Range, but all signals are scaled to fit in a small part of the screen. All signals are offset so that they are visible separately.

Persistant Display

Disabled.

Save Data

Saves the current scope view to a PNG file in a user-defined location.

18.6 Trace using the UART interface

If you are using a legacy FTDI or XTAG-1 debug adapter, or if the XSYS connector on your target hardware does not provide an xCONNECT Link, you can output data over the UART interface provided by your adapter.

To use the UART interface, you must provide the xSCOPE library with a 1-bit UART TX port that has been initialized with the pin connected to the UART-TX pin on your debug adapter. An example initialization is shown below.

```
#include <platform.h>
#include <xscope.h>

port uart_tx = PORT_UART_TX;

void xscope_user_init(void) {
  xscope_register(2,
    XSCOPE_CONTINUOUS, "Microphone Left",  XSCOPE_UINT, "mV",
    XSCOPE_CONTINUOUS, "Microphone Right", XSCOPE_UINT, "mV"
  );
  xscope_config_uart(uart_tx);
}
```

Because the UART interface uses a port instead of an xCONNECT Link, the probe functions can be called on a single tile only.

19 xSCOPE performance figures

IN THIS CHAPTER

- Transfer rates between the xCORE Tile and XTAG-2
- Transfer rates between the XTAG-2 and Host PC

Data transferred from the xCORE device to the debug adapter is lossless, but data transferred from the debug adapter to your host PC may be lossy, depending on the speed of your PC.

19.1 Transfer rates between the xCORE Tile and XTAG-2

The recommended xCONNECT Link speed for most target hardware is 10ns between transitions (10MByte/sec). This can be achieved by setting the link interbit gap to 5 cycles (see §54.5). The latencies and maximum call rates for the probe functions using an xCONNECT Link at this speed are given in Figure 36.

Figure 36: xSCOPE performance figures for xCONNECT Link with 5-cycle interbit gap

Probe function	Latency (core cycles)	Max calls/sec
`xscope_probe_data_pred`	15 (always)	666,000
`xscope_probe`	20 (with no contention)	999,000
`xscope_probe_cpu`	27 (with no contention)	666,000
`xscope_probe_data`	22 (with no contention)	666,000
`xscope_probe_cpu_data`	28 (with no contention)	555,000

If two subsequent calls are made, the second call may be delayed in line with the maximum frequency. For example, if `xscope_probe_data_pred` is called twice, the second call is delayed by approximately 1.5µs.

The maximum call rates can be increased by speeding up the link and reducing the interbit gap (see §54.5). A small interbit gap requires careful layout of the link, since it increases link frequency.

The UART interface executes at a rate of 2MB/s.

19.2 Transfer rates between the XTAG-2 and Host PC

Many PCs are limited to inputting trace data from the XTAG-2 at a rate of 500,000 trace records/sec or less. If your PC is unable to keep up it will drop records, reducing the granularity of the trace data. The XDE Scope view marks the loss of data on the timeline.

20 xSCOPE Library API

IN THIS CHAPTER

- Functions
- Enumerations

20.1 Functions

Function	xscope_bytes
Description	Send a trace event for the specified XSCOPE probe with a byte array.

Type	`void xscope_bytes(unsigned char id,`
	`unsigned int size,`
	`const unsigned char data[])`

Parameters	`id` XSCOPE probe id. `size` User data size. `data` User data bytes (char[]).

Function	xscope_char
Description	Send a trace event for the specified XSCOPE probe of type char.
Type	`void xscope_char(unsigned char id, unsigned char data)`
Parameters	`id` XSCOPE probe id. `data` User data value (char).

Function	xscope_config_io
Description	Configures XScope I/O redirection.
Type	`void xscope_config_io(unsigned int mode)`

Continued on next page

Parameters	`mode`	I/O redirection mode.

Function	**xscope_core_bytes**
Description	Send a trace event for the specified XSCOPE probe with a byte array with logical core info.
Type	`void xscope_core_bytes(unsigned char id, unsigned int size, const unsigned char data[])`
Parameters	`id` XSCOPE probe id. `size` User data size. `data` User data bytes (char[]).

Function	**xscope_core_char**
Description	Send a trace event for the specified XSCOPE probe of type char with logical core info.
Type	`void xscope_core_char(unsigned char id, unsigned char data)`
Parameters	`id` XSCOPE probe id. `data` User data value (char).

Function	**xscope_core_double**
Description	Send a trace event for the specified XSCOPE probe of type double with logical core info.
Type	`void xscope_core_double(unsigned char id, double data)`
Parameters	`id` XSCOPE probe id. `data` User data value (double).

Function	xscope_core_float
Description	Send a trace event for the specified XSCOPE probe of type float with logical core info.
Type	`void xscope_core_float(unsigned char id, float data)`
Parameters	`id` XSCOPE probe id. `data` User data value (float).

Function	xscope_core_int
Description	Send a trace event for the specified XSCOPE probe of type int with logical core info.
Type	`void xscope_core_int(unsigned char id, unsigned int data)`
Parameters	`id` XSCOPE probe id. `data` User data value (int).

Function	xscope_core_longlong
Description	Send a trace event for the specified XSCOPE probe of type long long with logical core info.
Type	`void xscope_core_longlong(unsigned char id, unsigned long long data)`
Parameters	`id` XSCOPE probe id. `data` User data value (long long).

Function	xscope_core_short
Description	Send a trace event for the specified XSCOPE probe of type short with logical core info.
Type	`void xscope_core_short(unsigned char id, unsigned short data)`

Continued on next page

Parameters	`id`	XSCOPE probe id.
	`data`	User data value (short).

Function	**xscope_core_start**
Description	Start a trace block for the specified XSCOPE probe with logical core info.
Type	`void xscope_core_start(unsigned char id)`
Parameters	`id` XSCOPE probe id.

Function	**xscope_core_start_int**
Description	Start a trace block for the specified XSCOPE probe with logical core info and capture a value of type int.
Type	`void xscope_core_start_int(unsigned char id, unsigned int data)`
Parameters	`id` XSCOPE probe id. `data` User data value (int).

Function	**xscope_core_stop**
Description	Stop a trace block for the specified XSCOPE probe with logical core info.
Type	`void xscope_core_stop(unsigned char id)`
Parameters	`id` XSCOPE probe id.

Function	**xscope_core_stop_int**
Description	Stop a trace block for the specified XSCOPE probe with logical core info and capture a value of type int.

Continued on next page

Type	`void xscope_core_stop_int(unsigned char id, unsigned int data)`
Parameters	`id` XSCOPE probe id. `data` User data value (int).

Function	**xscope_disable**
Description	Disable the XSCOPE event capture on the local xCORE tile.
Type	`void xscope_disable()`

Function	**xscope_double**
Description	Send a trace event for the specified XSCOPE probe of type double.
Type	`void xscope_double(unsigned char id, double data)`
Parameters	`id` XSCOPE probe id. `data` User data value (double).

Function	**xscope_enable**
Description	Enable the XSCOPE event capture on the local xCORE tile.
Type	`void xscope_enable()`

Function	**xscope_float**
Description	Send a trace event for the specified XSCOPE probe of type float.
Type	`void xscope_float(unsigned char id, float data)`
Parameters	`id` XSCOPE probe id. `data` User data value (float).

Function	**xscope_int**
Description	Send a trace event for the specified XSCOPE probe of type int.
Type	`void xscope_int(unsigned char id, unsigned int data)`
Parameters	`id` XSCOPE probe id. `data` User data value (int).

Function	**xscope_longlong**
Description	Send a trace event for the specified XSCOPE probe of type long long.
Type	`void xscope_longlong(unsigned char id, unsigned long long data)`
Parameters	`id` XSCOPE probe id. `data` User data value (long long).

Function	**xscope_ping**
Description	Generate an XSCOPE ping system timestamp event.
Type	`void xscope_ping()`

Function	**xscope_register**
Description	Registers the trace probes with the host system. First parameter is the number of probes that will be registered. Further parameters are in groups of four. **Examples:** `xscope_register(1, XSCOPE_DISCRETE, "A probe", XSCOPE_UINT, "value"); ``` `xscope_register(2, XSCOPE_CONTINUOUS, "Probe", XSCOPE_FLOAT, "Level", XSCOPE_STATEMACHINE, "State machine", XSCOPE_NONE, "no name");`
Type	`void xscope_register(int num_probes, ...)`
Parameters	`num_probes` Number of probes that will be specified.

Function	**xscope_short**	
Description	Send a trace event for the specified XSCOPE probe of type short.	
Type	`void xscope_short(unsigned char id, unsigned short data)`	
Parameters	`id`	XSCOPE probe id.
	`data`	User data value (short).

Function	**xscope_start**	
Description	Start a trace block for the specified XSCOPE probe.	
Type	`void xscope_start(unsigned char id)`	
Parameters	`id`	XSCOPE probe id.

Function	**xscope_start_int**	
Description	Start a trace block for the specified XSCOPE probe and capture a value of type int.	
Type	`void xscope_start_int(unsigned char id, unsigned int data)`	
Parameters	`id`	XSCOPE probe id.
	`data`	User data value (int).

Function	**xscope_stop**	
Description	Stop a trace block for the specified XSCOPE probe.	
Type	`void xscope_stop(unsigned char id)`	
Parameters	`id`	XSCOPE probe id.

Function	**xscope_stop_int**
Description	Stop a trace block for the specified XSCOPE probe and capture a value of type int.
Type	`void xscope_stop_int(unsigned char id, unsigned int data)`
Parameters	`id` XSCOPE probe id. `data` User data value (int).

20.2 Enumerations

Type	**xscope_IORedirectionMode**
Description	Enum of all I/O redirection modes.

Values	`XSCOPE_IO_NONE`
	I/O is not redirected.
	
	`XSCOPE_IO_BASIC`
	Basic I/O redirection.
	
	`XSCOPE_IO_TIMED`
	Timed I/O redirection.
	
	`XSCOPE_IO_NONE`
	I/O is not redirected.
	
	`XSCOPE_IO_BASIC`
	Basic I/O redirection.
	
	`XSCOPE_IO_TIMED`
	Timed I/O redirection.

Type	**xscope_UserDataType**
Description	Enum for all user data types.
Values	`XSCOPE_NONE` No user data. `XSCOPE_UINT` Unsigned int user data. `XSCOPE_INT` Signed int user data. `XSCOPE_FLOAT` Floating point user data. `XSCOPE_NONE` No user data. `XSCOPE_UINT` Unsigned int user data. `XSCOPE_INT` Signed int user data. `XSCOPE_FLOAT` Floating point user data.

Type	**xscope_EventType**
Description	Enum for all types of xscope events.

Values	`XSCOPE_STARTSTOP`
	Start/Stop - Event gets a start and stop value representing a block of execution.
	
	`XSCOPE_CONTINUOUS`
	Continuous - Only gets an event start, single timestamped “ping”.
	
	`XSCOPE_DISCRETE`
	Discrete - Event generates a discrete block following on from the previous event.
	
	`XSCOPE_STATEMACHINE`
	State Machine - Create a new event state for every new data value.
	
	`XSCOPE_HISTOGRAM`
	
	`XSCOPE_STARTSTOP`
	Start/Stop - Event gets a start and stop value representing a block of execution.
	
	`XSCOPE_CONTINUOUS`
	Continuous - Only gets an event start, single timestamped “ping”.
	
	`XSCOPE_DISCRETE`
	Discrete - Event generates a discrete block following on from the previous event.
	
	`XSCOPE_STATEMACHINE`
	State Machine - Create a new event state for every new data value.
	
	`XSCOPE_HISTOGRAM`

Part H

Simulation

CONTENTS

- Use xTIMEcomposer to simulate a program
- xSIM command-line manual
- XSIM Testbench and Plugin Interfaces

21 Use xTIMEcomposer to simulate a program

IN THIS CHAPTER

- Configure the simulator
- Trace a signal
- Set up a loopback
- Configure a simulator plugin

The xCORE simulator provides a near cycle-accurate model of systems built from one or more xCORE devices. Using the simulator, you can view a processor's instruction trace, visualize machine state and configure loopbacks to model the behavior of components connected to XMOS ports and links.

21.1 Configure the simulator

To configure the simulator, follow these steps:

1. Select a project in the **Project Explorer**.
2. Choose **Run ▶ Run Configurations**.
3. In the left panel, double-click **XCore Application**. xTIMEcomposer creates a new configuration and displays the default settings in the right panel.
4. In the right panel, in **Name**, enter a name for the configuration.
5. xTIMEcomposer tries to identify the target project and executable for you. To select one yourself, click **Browse** to the right of the **Project** text box and select your project in the **Project Selection** dialog box. Then click **Search Project** and select the executable file in the **Program Selection** dialog box.

   You must have previously compiled your program without any errors for the executable to be available for selection.
6. Select the **simulator** option and click the **Simulator** tab to configure additional options, as shown in Figure 37.

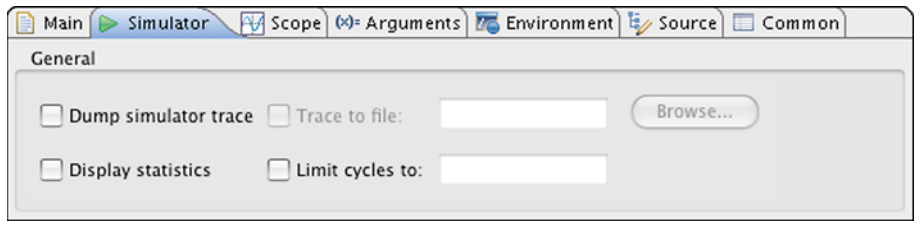


Figure 37: Simulator configuration options

- To output the processor instruction trace during simulation, select **Dump simulator trace**.
  By default, the instruction trace is displayed in the **Console**. To write the trace to a file instead, select **Trace to file** and enter a filename. The filename must be different from all other files in your project.
  The format of the instruction trace is described Figure 41.
- To view a summary of the program's execution once the program terminates, select **Display statistics**. The summary includes the instruction count for each logical core, and the number of data and control tokens sent through the switches.
- To limit the number of cycles executed by the simulator, enter a value in the **Limit cycles to** text box. Leave blank if you want the program to run from start to finish. This is useful for simulating programs with infinite loops.

7. To save and run the configuration, click **Run**.

xTIMEcomposer loads your executable, displaying any output generated by your program in the **Console**.

xTIMEcomposer remembers the configuration last used to run your program. To run it again using the same configuration, just click the **Run** button button. To use a different configuration, click the arrow to the right of the **Run** button and select a configuration from the drop-down list.

21.2 Trace a signal

The simulator can output signal tracing data to a VCD file which you can visualize with the xTIMEcomposer waveform viewer.

21.2.1 Enable signal tracing

To enable signal tracing during simulation, follow these steps:

1. Create a simulator **Run Configuration** (see §21.1).
2. In the **Simulator** tab, in the **Signal Tracing** panel, select **Enable signal tracing**.
   - To trace all I/O pins, in the **System Trace Options** group, select **Pins**.
   - To trace machine state on a specific core, in the **Core Trace Options** group, click **Add** to display a set of configurable drop-down lists and checkboxes. Then select the core and machine state you wish to trace. You can trace process cycles, ports, cores, clock blocks, pads and processor instructions.
3. Click **Run**.

xTIMEcomposer loads your program into the simulator and, on termination, adds the generated VCD file to your project.

21.2.2 View a trace file

In the **Project Explorer**, double-click on a VCD file to open it in the **Signals** view, as shown in Figure 38.

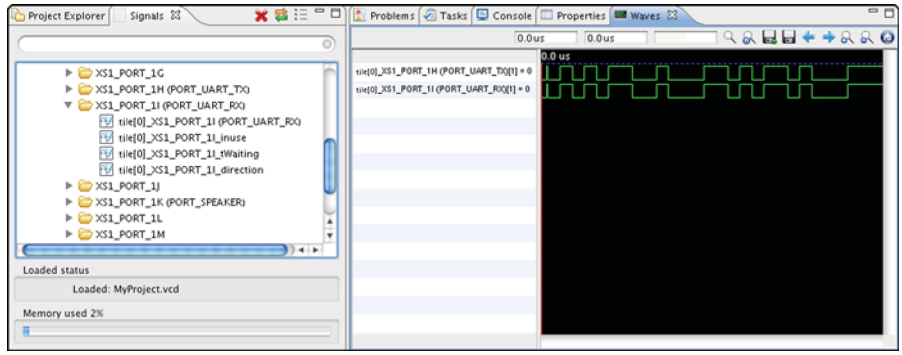


Figure 38: Signals and Waves views

In the **Signals View**, click the **plus sign** (Windows) or the **disclosure triangle** (Mac) to expand a folder and display its collection of signals or subfolders. Double-click on a signal or an entire folder to display in the **Waves** view.

To switch between a hierarchical and flat view of the signals, click the **Display** button.

21.2.3 View a signal

In the **Waves** view, move the cursor over a signal to view the time since the start of the simulation in the right numerical control at the top of the Waves view. If the cursor changes to a pointing finger, you can double-click to locate the output statement in the source code responsible for driving the signal. Use the **Waves** view toolbar to perform the following operations:

To view the entire waveform, click the **Zoom Fit** button.

To move between transitions of the selected signal, click the **Next** and **Previous** buttons. The output statement responsible for the transition is highlighted in the editor.

To search for a specific transition, click the **Search Transition** button to open a dialog box. Enter a value and click **Find**.

To save the configuration, click the **Write Session File** button and enter a filename for the file. Your settings are saved for use whenever you load the VCD file in the **Waves** view. Click the **Read Session File** button to load a recently saved settings file.

You can control how signals are displayed in the **Waves** view as follows:

- **Display signal values in ASCII:** Right-click on a signal in the **Waves** view to bring up a contextual menu and choose **Data Format** ▶ **ASCII**.
- **Add a separator between signals:** Right-click on a signal in the Waves view to bring up a menu and choose **Add Separator**.
- **Name a separator:** Right-click on a separator to bring up a menu and choose **Name Separator**. Enter a name for the separator in the **Name Separator** dialog box and click **OK**.
- **Move a separator:** Click-and-drag a separator to the desired position.

21.3 Set up a loopback

You can connect any two ports or pins together in your simulation, to model connections between the pins. To configure a loopback, follow these steps:

1. Create a simulator **Run Configuration** (see §21.1).
2. Click the **Simulator** tab to display the simulator configuration options.
3. Click the **Loopback** tab in the **Plugins** panel and select **Enable pin connections**.
4. In the **Pin Connections** panel, click **Add**. An empty loopback configuration is displayed. The loopback consists of two sets of options that you can configure for two different ports.

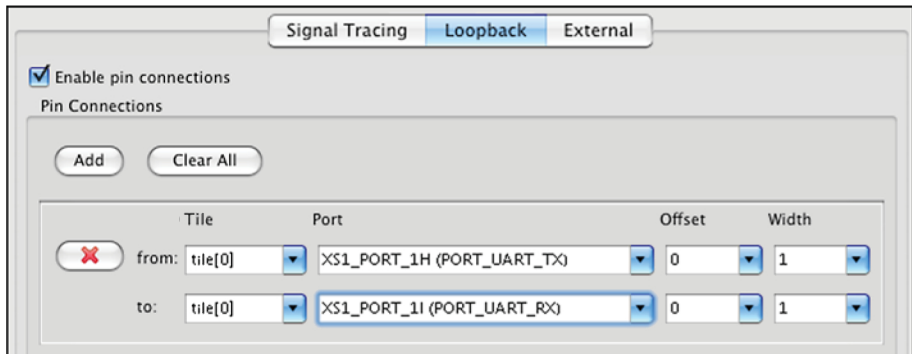


Figure 39: Setting up a loopback connection

5. In the drop-down lists for each end of the connection, select a value for **Tile** and **Port**. If you leave the tile unspecified, the list of ports is taken from the project's XN file and the tile is determined automatically. If you specify tile, the list of ports is taken from the header file `<xs1.h>`. To specify that only a subset of the pins are connected to the port, change the values for **Offset** and **Width**.

6. Click **Run**.

21.4 Configure a simulator plugin

You can connect the simulator to any external plugin that has been compiled on your host PC using the XMOS simulator plugin interface. To configure an external plugin, follow these steps:

1. Create a simulator **Run Configuration** (see §21.1).

2. Click the **Simulator** tab to display the simulator configuration options.

3. In the **Plugins** panel, click the **External** tab.

4. Click **Add** to open the plugin configuration dialog.

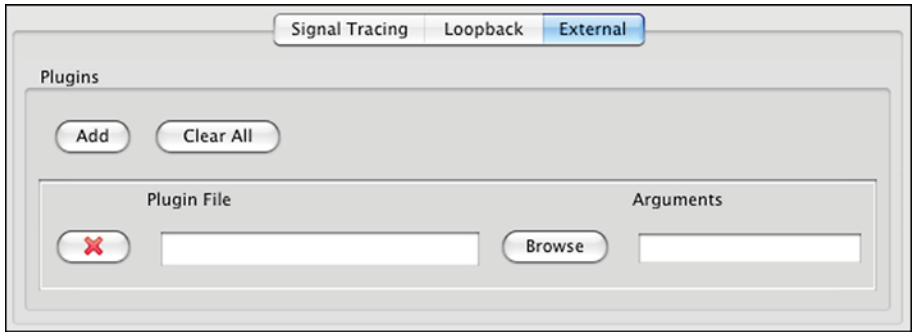


Figure 40: Setting up an external plugin

5. Select the plugin DLL and specify an optional command-line argument string.

6. Click **Run** to save your settings and run your program on the simulator with the specified plugins.

22 xSIM command-line manual

IN THIS CHAPTER

- Overall Options
- Warning Options
- Tracing Options
- Loopback Plugin Options
- xSCOPE Options

xSIM performs a cycle-based simulation of an XMOS Executable (XE) file. The XE file contains a description of the target hardware.

22.1 Overall Options

`xe-file` Specifies an XE file to simulate.

`--max-cycles n`
Exits when *n* system cycles is reached.

`--plugin name args`
Loads a plugin DLL. The format of *args* is determined by the plugin; if *args* contains any spaces, it must be enclosed in quotes.

`--stats` On exit, prints the following:

- A breakdown of the instruction counts for each logical core.
- The number of data and control tokens sent through the switches.

`--help` Prints a description of the supported command line options.

`--version` Displays the version number and copyrights.

22.2 Warning Options

`--warn-resources`
Prints (on standard error) warning messages for the following:

- A timed input or output operation specifies a time in the past.
- The data in a buffered port's transfer register is overwritten before it is input by the processor.

`--warn-stack`
: Turns on warnings about possible stack corruption.

  xSIM prints a warning if one XC task attempts to read or write to another task's workspace. This can happen if the stack space for a task is specified using either `#pragma stackfunction` (see §9) or `#pragma stackcalls` (see §9).

`--no-warn-registers`
: Don't warn when a register is read before being written.

22.3 Tracing Options

`--trace`
`-t`
: Turns on instruction tracing for all tiles (see Figure 41).

`--trace-to file`
: Turns on instruction tracing for all tiles. The trace is output to *file*.

`--disable-rom-tracing`
: Turns off tracing for all instructions executed from ROM.

`--enable-fnop-tracing`
: Turns on tracing of FNOP instructions.

`--vcd-tracing args`
: Enables signal tracing. The trace data is output in the standard VCD file format.

Figure 41: Trace output for XS1 processors

Tile	Core State											Address		Instruction		Mem	Cycle
Name from XN	I_0	I_1	I_2	$S_0S_1(T_0)$ .. $S_0S_1(T_n)$			.	M	S	K	N	PC	(sym + offset) :	name	operands	address	@val
	- D	* P	- d	- a A i I p m s w	- b i e	*n* status pairs		- m	- s	- k	- n				val rn(val) res[id]	L[adr] S[adr]	

I_0: -	No debug interrupt	S_1: -	Interrupts and events disabled
I_0: D	Instruction caused debug interrupt	S_1: b	Interrupts and events enabled
I_1: *	Instruction excepted	S_1: i	Interrupts enabled and events disabled
I_1: P	Instruction paused	S_1: e	Interrupts disabled and events enabled
I_2: -	Not in debug mode	M: -	MSYNC not set
I_2: d	Tile in debug mode	M: m	MSYNC set
S_0: -	Core not in use	S: -	SSYNC not set
S_0: a	Core active	S: s	SSYNC set
S_0: A	Core active (the instruction being traced belongs to this core)	K: -	INK not set
S_0: i	Core active with ININT bit set	K: k	INK set
S_0: I	Core active with ININT bit set (belongs to this core)	N: -	INENB not set
S_0: p	Core paused due to instruction fetch	N: n	INENB set
S_0: m	Core paused with MSYNC bit set	rn(val)	Value of register *n*
S_0: s	Core paused withSSYNC bit set	res[id]	Resource identifier
S_0: w	Core paused with WAITING bit set	L/S[adr]	Load from/Store to address

If *args* contains any spaces, it must be enclosed in quotes. Its format is:

global-options$_{opt}$⟨`-tile name` ⟨trace-options⟩*⟩*

The global options are:

`-pads` Turns on pad tracing.

`-o file` Places output in *file*.

The trace options are specific to the tile associated with the XN core declaration name, for example `tile[0]`.

The trace options are:

`-ports` Turns on port tracing.

`-ports-detailed`
Turns on more detailed port tracing.

`-cycles` Turns on clock cycle tracing.

`-clock-blocks`
Turns on clock block tracing.

`-cores` Turns on logical core tracing.

`-instructions`
Turns on instruction tracing.

To output traces from different nodes, tiles or logical cores to different files, this option can be specified multiple times.

For example, the following command configures the simulator to trace the ports on tile[0] to the file trace.vcd.

```
▶ xsim a.xe --vcd-tracing "-o trace.vcd -start-disabled -tile tile[0]
  -ports"
```

Tracing by the VCD plugin can be enabled and disabled using the `_traceStart()` and `_traceStop()` syscalls. The `-start-disabled` argument disables the vcd tracing from the start, allowing the user to enable/disable only those sections of code where tracing is desired. For example:

```
#include <xs1.h>
#include <syscall.h>

port p1 = XS1_PORT_1A;

int main() {
   p1 <: 1;
   p1 <: 0;

   _traceStart();
   p1 <: 1;
   p1 <: 0;
   _traceStop();

   p1 <: 1;
   p1 <: 0;

   return 0;
}
```

22.4 Loopback Plugin Options

The XMOS Loopback plugin configures any two ports on the target platform to be connected together. The format of the arguments to the plugin are:

`-pin package pin`
: Specifies the pin by its name on a package datasheet. The value of *package* must match the `Id` attribute of a `Package` node (see §54.3) in the XN file used to compile the program.

`-port name n offset`
: Specifies *n* pins that correspond to a named port.

  The value of *name* must match the `Name` attribute of a `Port` node (see §54.4.2) in the XN file used to compile the program.

  Setting *offset* to a non-zero value specifies a subset of the available pins.

`-port tile p n offset`
: Specifies *n* pins that are connected to the port *p* on a *tile*.

  The value of *tile* must match the `Reference` attribute of a `Tile` node (see §54.4.1) in the XN file used to compile the program.

  p can be any of the port identifiers defined in `<xs1.h>`. Setting *offset* to a non-zero value specifies a subset of the available pins.

The plugin options are specified in pairs, one for each end of the connection. For example, the following command configures the simulator to loopback the pin connected to port `XS1_PORT_1A` on `tile[0]` to the pin defined by the port `UART_TX` in the program.

▶ `xsim uart.xe --plugin LoopbackPort.dll '-port tile[0] XS1_PORT_1A 1 0 -port UART_TX 1 0'`

22.5 xSCOPE Options

`--xscope args`

Enables xSCOPE. file format.

If *args* contains any spaces, it must be enclosed in quotes. One of the following 2 options is mandatory:

`-offline <filename>`

Runs with xSCOPE in offline mode, placing the xSCOPE output in the given file.

`-realtime <URL:port>`

Runs with xSCOPE in realtime mode, sending the xSCOPE output in the given URL:port.

The following argument is optional:

`-limit <num records>`

Limts the xSCOPE output records to the given number.

For example, the following will run xSIM with xSCOPE enabled in offline mode:

```
▶ xsim app.xe --xscope "-offline xscope.xmt"
```

For example, the following will run xSIM with xSCOPE enabled in reatime mode:

```
▶ xsim app.xe --xscope "-realtime localhost:12345"
```

23XSIM Testbench and Plugin Interfaces

IN THIS CHAPTER

- Implementing a Plugin
- Plugin Notifications
- Implementing a testbench
- Plugin API
- Testbench API

The XMOS simulator provides two interfaces that you can use to simulate your program within the context of a larger system.

- The *plugin* interface allows you to connect one or more external devices to the simulator, with all external devices clocked by the simulator, as shown in Figure 42.
- The *testbench* interface allows you to connect one or more simulators and external devices together, all clocked under control of the testbench, as shown in Figure 43.

Figure 42: Plugins Interface

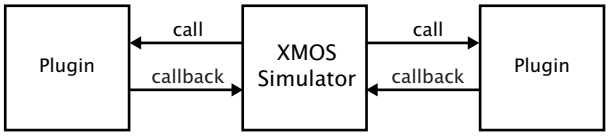
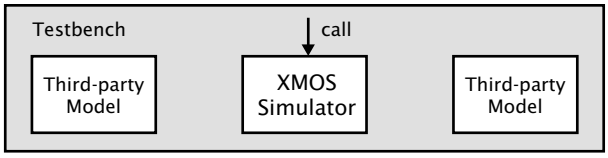


Figure 43: Testbench Interface



23.1 Implementing a Plugin

The plugin interface allows you to write models of external devices that interface with the XMOS simulator. The example plugin shown below loops back two pins in the simulator.

```
#include "xsiplugin.h"
#include <assert.h>

XsiCallbacks \*g_xsi;

XsiStatus plugin_create(void \**instance, XsiCallbacks \*xsi,
                        const char \*arguments) {
  assert(CHECK_INTERFACE_VERSION(xsi));
  g_xsi = xsi;
  return XSI_STATUS_OK;
}

XsiStatus plugin_clock(void \*instance) {
  unsigned value   = 0;
  XsiStatus status = g_xsi->sample_pin("0", "X1D52", &value);
  \*instance = (void \*)0;
  if (status == XSI_STATUS_OK)
    status = g_xsi->drive_pin("0", "X1D53", value);
  return status;
}
```

The simulator calls `plugin_create` at startup. The call to `CHECK_INTERFACE_VERSION` checks whether the version of the interface supported by the simulator is compatible with the version of the interface library used to compile the plugin. The parameter `xsi` is a structure of function pointers that can be used to implement callbacks into the simulator. It is assigned to the global variable `g_xsi` for later use.

The simulator calls `plugin_clock` on each tick of the processor clock. On each call, the plugin samples the value on [package 0, pin X1D52] and drives it on [package 0, pin X1D53], effectively looping back the two pins. The package name is taken from the XN file for the program running on the simulator, and the pin name from the package datasheet.

An example loopback plugin is distributed with the tools. It can be found in `src/plugins/ExamplePlugin/`. An example xc program that uses the plugin can be found in `examples/ExamplePluginTest`.

23.2 Plugin Notifications

The notification mechanism allows you to communicate between a plugin and code running on an xCORE tile directly via a system call. An example is shown below:

```
#include <syscall.h>

#define TYPE_DISPLAY 0

int main() {
    _plugins(TYPE_DISPLAY, 42, 99);
    return 0;
}
```

```
#include <stdio.h>

#define TYPE_DISPLAY 0

XsiStatus plugin_notify(void \*instance,
  int type, unsigned arg1, unsigned arg2) {

  if (type == TYPE_DISPLAY)
    printf("Args: %\%d%, %\%d%, %\%d%\n", arg1, arg2);

  return XSI_STATUS_OK;
}
```

In the example above, the program running on the xCORE simulator calls the function `_plugins` with a user-defined notification type and two arguments. This call causes the simulator to call `plugin_notify` with these parameters. In this case, the notification function simply outputs the parameters to the console.

23.3 Implementing a testbench

The testbench interface allows you to instantiate an XMOS simulator within a larger system environment. The testbench is responsible for clocking the entire system. The example testbench below instantiates the XMOS system and clocks it until completion.

```
#include "xsidevice.h"
#include <assert.h>

void *xsim = 0;

int main(int argc, char **argv) {

  XsiStatus status = xsi_create(&xsim, "test.xe");
  assert(status == XSI_STATUS_OK);

  while (status != XSI_STATUS_DONE) {
    status = xsi_clock(xsim);
    assert(status == XSI_STATUS_OK
       || status == XSI_STATUS_DONE);
  }

  status = xsi_terminate(xsim);
  assert(status == XSI_STATUS_OK);
  return 0;
}
```

The testbench calls `xsi_create` to create an instance of an XMOS simulator for the platform defined by the given binary. On each iteration of the `while` loop, the testbench calls `xsi_clock`, which causes the simulator to advance one or more logical cores by a single clock tick.

An example testbench is distributed with the tools. It can be found in the tools installation directory `src/testbenches/ExampleTestbench/`. An example xc program that runs on the device is provided in `examples/ExampleTestbenchTest`.

A C89 compiler is required to compile a plugin on your host PC. On Windows the library to link against is `lib/xsidevice.dll`, and on Mac and Linux the library is `lib/libxsidevice.so`.

23.4 Plugin API

All plugins should include the header file `xsiplugin.h`. Except where otherwise stated, functions return `XSI_STATUS_OK` on success.

You should provide implementations of the following functions to create, clock and terminate your plugin.

Function	**plugin_create**

Continued on next page

Description	The simulator calls this function on startup. If multiple instances of the plugin are required, `instance` can be used as a handle to enable multiple instantiations (for example, by defining as an index into a global array of structures containing the state for each instance of the plugin). `xsi` contains a structure of callback function pointers, which the plugin can use to interface with the simulator. `arguments` contains the command-line argument string passed to the plugin. On error, you may return `XSI_STATUS_INVALID_ARGS`.
Type	`XsiStatus plugin_create(void **instance, XsiCallbacks *xsi, const char *arguments)`

Function	**plugin_clock**
Description	By default, the simulator calls this function each time it increments the processor clock. The clock rate can be changed by calling the function `set_mhz`.
Type	`XsiStatus plugin_clock(void *instance)`

Function	**plugin_notify**
Description	The simulator calls this function whenever a system call of type `OS_PLUGINS` is executed by a program. This allows the target code to communicate with the plugin.
Type	`XsiStatus plugin_notify(void *instance, int type, unsigned arg1, unsigned arg2)`

Function	**plugin_terminate**
Description	The simulator calls this function immediately prior to terminating.
Type	`XsiStatus plugin_terminate(void *instance)`

The callback functions provided by the structure passed to `plugin_create` are used to configure the simulator.

23.4.1 Interfacing with the Simulator

During the call to `plugin_create`, the simulator provides a structure of callback functions (an argument of type `XsiCallbacks`) that can be used to interface with the simulator.

The following callback functions are provided to configure the simulator.

Function	**CHECK_INTERFACE_VERSION**
Description	`CHECK_INTERFACE_VERSION` checks whether the version of the interface library running on the simulator is compatible with the version of the interface used to compile the plugin.
Type	`XsiStatus CHECK_INTERFACE_VERSION(XsiCallbacks *)`

Function	**(*set_mhz)**
Description	`set_mhz` sets the rate at which the plugin is clocked in MHz. The simulator has a mechanism that ensures that all processors and external plugins are clocked, over time and on average, at the correct rate relative to the fastest component (and hence one another).
Type	`XsiStatus (*set_mhz)(double mhz)`

Function	**(*reset)**
Description	`reset` performs a device reset operation. The only supported value for `type` is `XSI_RESET_HARD`, which performs a hard reset of all devices on the target platform. This effect is the same as driving the package RESET pin.
Type	`XsiStatus (*reset)(XsiResetType type)`

Function	**(*save_state)**
Description	`save_state` writes the simulator state to a file. If the file cannot be opened for writing, the simulator returns `XSI_STATUS_INVALID_FILE`.
Type	`XsiStatus (*save_state)(const char *file)`

Function	(*restore_state)
Description	restore_state reads a simulator state file and continues execution from this point. If the file cannot be opened, the simulator returns XSI_STATUS_INVALID_FILE.
Type	XsiStatus (*restore_state)(const char *file)

In the following functions:

- The value of the parameter package must match the Id attribute of a Package node in the XN file used to compile the running program.
- The value of pin must match the name of an I/O pin specified on the corresponding package datasheet. On error, the simulator returns XSI_STATUS_INVALID_PACKAGE.
- The value of parameter tile must match the Reference attribute of a Tileref node in the XN file used to compile the running program. On error, the simulator returns XSI_STATUS_INVALID_TILE.
- If parameters for both tile and port are provided, tile may be null, in which case port must match the Name attribute of a Port node in the XN file used to compile the running program. Otherwise, port must be a valid port identifier of the form XS1_PORT_xy. If the port is incorrectly specified, the simulator returns XSI_STATUS_INVALID_PORT.
- If a parameter used to store a sampled value is null, the simulator returns XSI_STATUS_NULL_ARG.

The following callback functions are used to interface with the ports.

Function	(*sample_pin)
Description	sample_pin reads the value on the specified pin and assigns it to a variable: if the plugin was previously driving the pin, it stops driving it.
Type	XsiStatus (*sample_pin)(const char *package, const char *pin, unsigned *var)

Function	(*sample_port_pins)
Description	sample_port_pins reads the values on zero or more pins as specified by a mask, and assigns to a variable: if bit *n* of the mask is 1, the pin connected to bit *n* of the port is sampled, and if the plugin was previously driving the pin, it stops driving it; otherwise no action is performed on the pin.

Continued on next page

Type	`XsiStatus`
	`(*sample_port_pins)(const char *tile,`
	`const char *port,`
	`XsiPWord mask,`
	`XsiPWord *var)`

Function	**(*is_pin_driving)**
Description	`is_pin_driving` assigns a variable a value of 1 if the xCORE processor is currently driving the pin, and 0 otherwise.

Type	`XsiStatus`
	`(*is_pin_driving)(const char *package,`
	`const char *pin,`
	`int *var)`

Function	**(*is_port_pins_driving)**
Description	`is_port_pins_driving` assigns a variable a value such that if bit *n* of the value is 1, the xCORE processor is currently driving the pin connected to bit *n* of the specified port; a bit value of 0 signifies that the pin is not currently being driven.

Type	`XsiStatus`
	`(*is_port_pins_driving)(const char *tile,`
	`const char *port,`
	`XsiPWord *var)`

Function	**(*drive_pin)**
Description	`drive_pin` writes a value on the specified pin. The value continues to be held on the pin until a subsequent call to a function that samples or drives the pin.

Type	`XsiStatus (*drive_pin)(const char *package,`
	`const char *pin,`
	`unsigned value)`

Function	(*drive_port_pins)
Description	`drive_port_pins` writes to zero or more pins as specified by a mask: if bit *n* of the mask is 1, bit *n* of the value is written the pin connected to bit *n* of the specified port, otherwise no action is performed on the pin. The value written to each pin continues to be held on the pin until a subsequent call to a function that samples or drives the pin.
Type	`XsiStatus (*drive_port_pins)(const char *tile, const char *port, XsiPWord mask, XsiPWord value)`

The following callback functions are used to interface with the memory and debug registers.

Function	(*read_mem)
Description	`read_mem` reads a number of bytes from xCORE tile memory, starting at the specified address, and assigns them to a variable. The size of the data pointed to by `data` must be at least as large as `num_bytes`. If the memory address is invalid, the simulator returns `XSI_STATUS_MEMORY_ERROR`.
Type	`XsiStatus (*read_mem)(const char *tile, XsiWord32 address, unsigned num_bytes, unsigned char *data)`

Function	(*write_mem)
Description	`write_mem` writes a number of bytes of data to xCORE tile memory, starting at the specified address. The size of the data pointed to by `data` must be at least as large as `num_bytes`. If the memory address is invalid, the simulator returns `XSI_STATUS_MEMORY_ERROR`.
Type	`XsiStatus (*write_mem)(const char *tile, XsiWord32 address, unsigned num_bytes, const unsigned char *data)`

Function	(*read_pswitch_reg)
Description	read_pswitch_reg reads the specified processor-switch register and writes it to a variable. If the register number address is invalid, the simulator returns XSI_STATUS_PSWITCH_ERROR.
Type	XsiStatus (*read_pswitch_reg)(const char *tile, unsigned reg_num, unsigned *var)

Function	(*write_pswitch_reg)
Description	write_pswitch_reg writes a value to the specified processor-switch register. If the register number address is invalid, the simulator returns XSI_STATUS_PSWITCH_ERROR.
Type	XsiStatus (*write_pswitch_reg)(const char *tile, unsigned reg_num, unsigned value)

23.5 Testbench API

A testbench should include the header file xsidevice.h. Except where otherwise stated, functions return XSI_STATUS_OK on success.

The following functions are used to create, configure, clock and terminate an instance of an XMOS simulator.

Function	xsi_create
Description	xsi_create creates an instance of a simulator configured with the specified command-line arguments. The error return code is XSI_STATUS_INVALID_ARGS.
Type	XsiStatus xsi_create(void **instance, const char *arguments)

Function	xsi_clock

Continued on next page

Description	`xsi_clock` steps the fastest processor in the system by one clock cycle, and may step one or more slower processors by one clock cycle too. The simulator has a mechanism that ensures that all processors are clocked, over time and on average, at the correct rate relative to one another. A return value of `XSI_STATUS_OK` indicates that the clocking operation completed, and a value of `XSI_STATUS_DONE` indicates that the entire simulation has completed.
Type	`XsiStatus xsi_clock(void *instance)`

Function	**xsi_reset**
Description	`xsi_reset` performs a device reset operation. The only currently supported value for `type` is `XSI_RESET_HARD`, which performs a hard reset of all devices on the target platform. This effect is the same as driving the package RESET pin.
Type	`XsiStatus xsi_reset(void *instance, XsiResetType type)`

Function	**xsi_save_state**
Description	`xsi_save_state` writes the simulator state to a file. If the file cannot be opened for writing, the function returns `XSI_STATUS_INVALID_FILE`.
Type	`XsiStatus xsi_save_state(const char *file)`

Function	**restore_state**
Description	`xsi_restore_state` reads a simulator state file and continues execution from this point. If the file cannot be opened, the function returns `XSI_STATUS_INVALID_FILE`.
Type	`XsiStatus restore_state(const char *file)`

Function	**xsi_terminate**
Description	`xsi_terminate` terminates the specified simulation instance.
Type	`XsiStatus xsi_terminate(void *instance)`

23.5.1 Interfacing with a Simulator

The following functions can be used to interface with devices in a simulation. In the following functions:

- If the simulator instance is invalid, the function returns `XSI_STATUS_INVALID_INSTANCE`.
- The value of the parameter `package` must match the `Id` attribute of a `Package` node in the XN file used to compile the running program.
- The value of `pin` must match the name of an I/O pin specified on the corresponding package datasheet. On error, the function returns `XSI_STATUS_INVALID_PACKAGE`.
- The value of parameter `tile` must match the `Reference` attribute of a `Tileref` node in the XN file used to compile the running program. On error, the function returns `XSI_STATUS_INVALID_TILE`.
- If parameters for both `tilee` and `port` are provided, `tile` may be null, in which case `port` must match the `Name` attribute of a `Port` node in the XN file used to compile the running program. Otherwise, `port` must be a valid port identifier of the form `XS1_PORT_xy`. If the port is incorrectly specified, the function `XSI_STATUS_INVALID_PORT`.
- If a parameter used to store a sampled value is null, the function `XSI_STATUS_NULL_ARG`.

The following functions are used to interface with the ports.

Function	**xsi_sample_pin**
Description	`xsi_sample_pin` reads the value on the specified pin and writes it to a variable. If the testbench was previously driving the pin, it stops driving it.
Type	`XsiStatus *xsi_sample_pin(void *instance, const char *package, const char *pin, unsigned *var)`

Function	**xsi_sample_port_pins**
Description	`xsi_sample_port_pins` reads the values on zero or more pins as specified by a mask, and assigns to a variable: if bit *n* of the mask is 1, the pin connected to bit *n* of the port is sampled, and if the testbench was previously driving the pin, it stops driving it; otherwise no action is performed on the pin.
Type	`XsiStatus *xsi_sample_port_pins(void *instance, const char *tile, const char *port, XsiPWord mask, XsiPWord *var)`

Function	xsi_is_pin_driving
Description	`xsi_is_pin_driving` assigns a variable a value of 1 if the xCORE tile is currently driving the pin, and 0 otherwise.
Type	`XsiStatus xsi_is_pin_driving(void *instance, const char *package, const char *pin, int *var)`

Function	xsi_is_port_pins_driving
Description	`xsi_is_port_pins_driving` assigns a variable a value such that if bit *n* of the value is 1, the xCORE tile is currently driving the pin connected to bit *n* of the specified port; a bit value of 0 signifies that the pin is not currently being driven.
Type	`XsiStatus * xsi_is_port_pins_driving(void *instance, const char *tile, const char *port, XsiPWord mask, XsiPWord *var)`

Function	xsi_drive_pin
Description	`xsi_drive_pin` writes a value on the specified pin. The value continues to be held on the pin until a subsequent call to a function that samples or drives the pin.
Type	`XsiStatus *xsi_drive_pin(void *instance, const char *package, const char *pin, unsigned value)`

Function	xsi_drive_port_pins

Continued on next page

Description	`xsi_drive_port_pins` writes to zero or more pins as specified by a mask: if bit *n* of the mask is 1, bit *n* of the value is written the pin connected to bit *n* of the specified port, otherwise no action is performed on the pin. The value written to each pin continues to be held on the pin until a subsequent call to a function that samples or drives the pin. The error return codes are `XSI_STATUS_INVALID_TILE` and `XSI_INVALID_PORT`.

Type	`XsiStatus xsi_drive_port_pins(void *instance,`
	`const char *tile,`
	`const char *port,`
	`XsiPWord mask,`
	`XsiPWord value)`

The following functions are used to interface with the memory and debug registers.

Function	**xsi_read_mem**
Description	`xsi_read_mem` reads a number of bytes from xCORE tile memory, starting at the specified address, and assigns them a variable. The size of the data pointed to by `data` must be at least as large as `num_bytes`. If the memory address is invalid, the function returns `XSI_STATUS_MEMORY_ERROR`.

Type	`XsiStatus xsi_read_mem(void *instance,`
	`const char *tile,`
	`XsiWord32 address,`
	`unsigned num_bytes,`
	`unsigned char *data)`

Function	**xsi_write_mem**
Description	`xsi_write_mem` writes a number of bytes of data to xCORE tile memory, starting at the specified address. The size of the data pointed to by `data` must be at least as large as `num_bytes`. If the memory address is invalid, the function returns `XSI_STATUS_MEMORY_ERROR`.

Type	`XsiStatus xsi_write_mem(void *instance,`
	`char *tile,`
	`XsiWord32 address,`
	`unsigned num_bytes,`
	`const unsigned char *data)`

Function	xsi_read_pswitch_reg
Description	`xsi_read_pswitch_reg` reads the specified processor-switch register and writes it to a variable. If the register number address is invalid, the function returns `XSI_STATUS_PSWITCH_ERROR`.
Type	`XsiStatus xsi_read_pswitch_reg(void *instance, const char *tile, unsigned reg_num, unsigned *var)`

Function	xsi_write_pswitch_reg
Description	`xsi_write_pswitch_reg` writes a value to the specified processor-switch register. If the register number address is invalid, the function returns `XSI_STATUS_PSWITCH_ERROR`.
Type	`XsiStatus xsi_write_pswitch_reg(void *instance, const char *tile, unsigned reg_num, unsigned value)`

Part I

Debugging

CONTENTS

- Use xTIMEcomposer to debug a program
- Debug with printf in real-time

24 Use xTIMEcomposer to debug a program

IN THIS CHAPTER

- Launch the debugger
- Control program execution
- Examine a suspended program
- Set a breakpoint
- View disassembled code

The xCORE Debugger lets you see what's going on "inside" your program while it executes on hardware or on the simulator. It can help you identify the cause of any erroneous behavior.

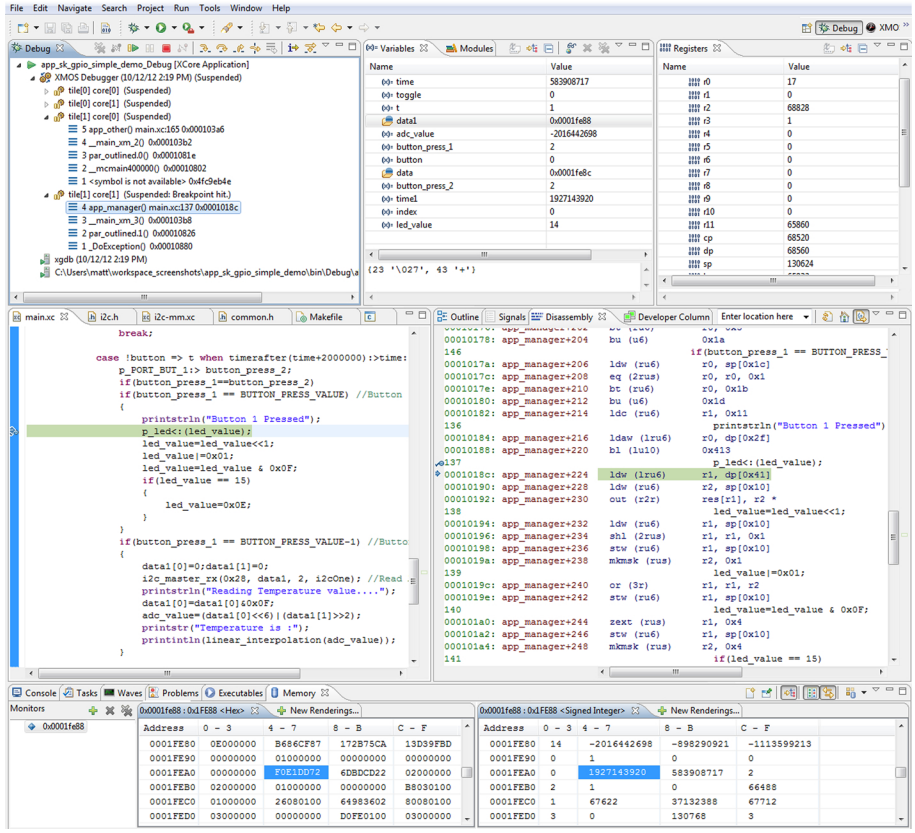


Figure 44: Debug perspective

For full visibility of your program, you must compile it with debugging enabled (see §10.3). This causes the compiler to add symbols to the executable that let the debugger make direct associations back to the source code. Note that compiling with optimizations enabled (see §10.4) can also make debugging more difficult.

24.1 Launch the debugger

To load a program under control of the debugger, follow these steps:

1. Select a project in the **Project Explorer**.
2. Choose **Run ▶ Debug Configurations**.
3. In the left panel, double-click **XCore Application**. xTIMEcomposer creates a new configuration and displays the default settings in the right panel.
4. In the **Name** text box, enter a name for the configuration.
5. xTIMEcomposer tries to identify the target project and executable for you. To select one yourself, click **Browse** to the right of the **Project** text box and select your project in the **Project Selection** dialog box. Then click **Search Project** and select the executable file in the **Program Selection** dialog box.

You must have previously compiled your program without any errors for the executable to be available for selection.

6. If you have a development board connected to your system, in the the **Device options** panel check the **hardware** option and select your debug adapter from the **Adapter** list. Alternatively, check the **simulator** option to run your program on the simulator.
7. To save the configuration and launch the debugger, click **Debug**. If you are asked whether to open the **Debug** perspective, check **Remember my decision** and click **Yes**.

xTIMEcomposer loads your program in the debugger and opens it in the **Debug** perspective.

xTIMEcomposer remembers the configuration last used to load your program. To debug the program later using the same settings, just click the **Debug** button. To use a different configuration, click the arrow to the right of the **Debug** button and select a configuration from the drop-down list.

24.2 Control program execution

Once launched, the debugger runs the program until either an exception is raised or you suspend execution by clicking the **Suspend** button .

Click the **Resume** button to continue executing a suspended program, or use one of the step controls to advance the core selected in the **Debug** view incrementally:

- **Step Into:** Executes a single line of source code on the core selected in the

Debug view. If the next line of code is a function call, the debugger suspends at the first statement in the called function. All other cores are resumed.

- **Step over:** Executes a single line of source code on the core selected in the **Debug** view. All other cores are resumed.

- **Step return:** Steps the core selected in the **Debug** view until the current function returns. If the next line of code is a function call, the debugger executes the entire function. All other cores are resumed.

- **Step through:** Switches the debugger context to the corresponding input core of a channel output statement. This is useful for following the path of data as it flows between cores. No cores are resumed.

When debugging optimized code, a step operation is not guaranteed to advance to the next line in the source code, since the compiler may have reordered instruction execution to improve performance.

24.3 Examine a suspended program

Once a program is suspended, you can query the state of each core and can inspect the values held in registers and memory.

- **Examine a core's call stack:** The **Debug view** displays a list of software tasks, each of which can be expanded to show its call stack, as shown in Figure 45.

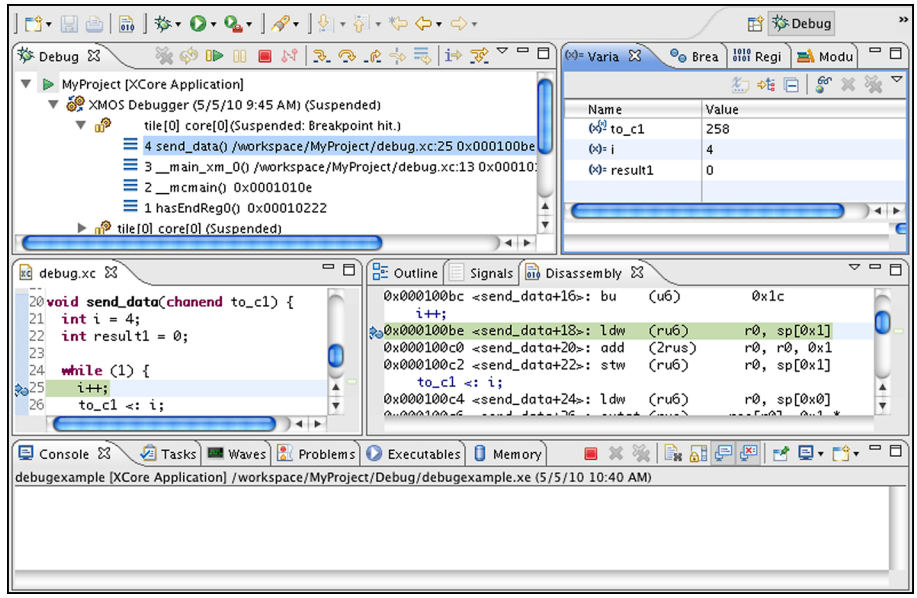


Figure 45: Debug view

In the example above, the tile `tile[0]` is suspended at a breakpoint in the function `send_data` on line 25 of the file `debug.xc`.

- **Examine Variables:** The **Variables** view displays variables and their values. In the **Debug** view click on any function in a core's call stack to view its variables, as shown in Figure 24.3.

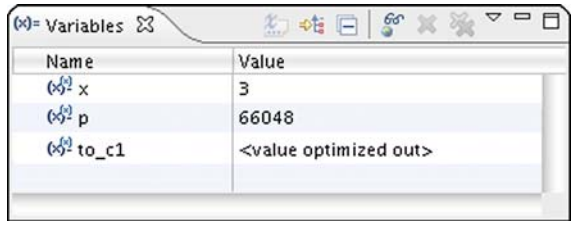


Figure 46: Variables view

To view a global variable, right-click in the **Variables** view, select **Add Global Variables** from the pop-up menu to open a dialog box and select the global variable to add to the view.

Compiling a program without optimizations guarantees that every variable is held in memory for the duration of its scope so that its value can always be displayed. If optimizations are enabled, a variable may not be available to be examined, resulting in the message `<value optimized out>`.

You can do the following with variables:

- **Display a variable's value in hexadecimal format:** Right-click on a variable to bring up a menu and choose **Format ▸ Hexadecimal**. You can also choose binary, decimal or normal. The normal format is determined by the type of the variable.
- **Change a variable's value:** Click on a value to highlight it, enter a new value and press **Enter**. The table entry is highlighted yellow to indicate its value has changed. This allows you to test what happens under what-if scenarios.
- **Prevent the debugger from reading a variable:** Right-click on a variable and choose **Disable** from the contextual menu. This is useful if the variable's type is qualified with `volatile`. To apply settings to multiple variables at once, press Ctrl (Windows, Linux) or ⌘ (Mac) while you click on multiple variables, then right-click and select an option from the contextual menu.
- **Examine Memory:** The **Memory** view provides a list of memory monitors, each representing a section of memory. To open the Memory view, choose **Window ▸ Show View ▸ Memory**. In the **Debug** view click on any core to view the contents of its memory, as shown in Figure 24.3.

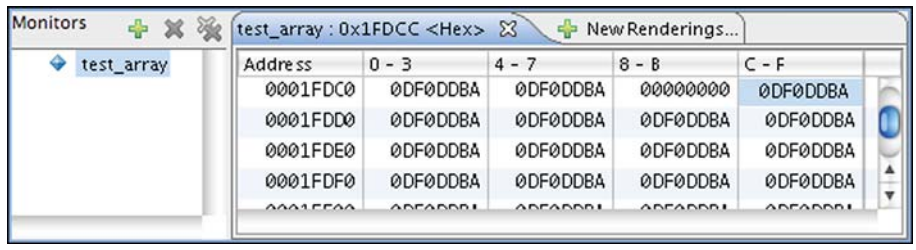


Figure 47: Memory view

To specify a memory location to view, click the **Add** button to open the **Memory Monitor** dialog box, enter a memory location and click **OK**. You can enter either an absolute address or a C/XC expression. To view the contents of an array just enter its name.

To display the memory contents in a different format such as Hex or ASCII, click the **New Renderings** tab, select a format and click **Add Renderings**. xTIMEcomposer adds new tabs in the panel to the right of the **Memory** view, each showing a different interpretation of the values in memory.

24.4 Set a breakpoint

A *breakpoint* is a marker in the program that instructs the debugger to interrupt execution so that you can investigate the state of the program. You can add a breakpoint to any executable line of code, causing execution to suspend before that line of code executes.

To add a breakpoint, double-click the marker bar in the left margin of the code editor next to the line at which you wish to suspend execution. A blue dot is displayed to indicate the presence of the breakpoint. Note that the breakpoint applies to every core that executes the function.

Breakpoints are also displayed in the **Breakpoints** view. To open the Breakpoints view, choose **Window ▶ Show ▶ View ▶ Breakpoints**. Double-click on a breakpoint to locate the corresponding line in the source code editor.

Here are some other things can do with breakpoints:

- **Set a conditional breakpoint:** Right-click on a breakpoint marker to bring up a contextual menu, and choose **Breakpoint Properties** to display a properties dialog box. Click the **Common** option in the left panel and enter a C/XC conditional expression in the **Condition** text box in the right panel. The expression can contain any variables in the scope of the breakpoint.
- **Set a conditional breakpoint:** Right-click on a breakpoint marker to bring up a contextual menu, and choose **Breakpoint Properties** to display a properties dialog box. Click the **Common** option in the left panel and enter a C/XC conditional expression in the **Condition** text box in the right panel. The expression can contain any variables in the scope of the breakpoint.
- **Set a watchpoint on a global variable:** A watchpoint is a special breakpoint that suspends execution whenever the value of an expression changes (without specifying where it might happen). Right-click anywhere in the **Breakpoints** view and choose **Add Watchpoint C/XC** from the contextual menu. Enter a C/XC expression in the dialog box, for example `a[MAX]`. Select **Write** to break when the expression is written, and **Read** to break when the expression is read.
- **Disable a breakpoint:** In the **Breakpoints** view, clear the checkbox next to a breakpoint. Enable the checkbox to re-enable the breakpoint.
- **Remove a breakpoint:** Double-click on a breakpoint marker in the code editor to remove it. Alternatively, right-click a breakpoint in the **Breakpoints** view and

select **Remove** from the contextual menu; to remove all breakpoints, select **Remove All**.

24.5 View disassembled code

The **Disassembly** view displays the assembly instructions that are executed on the target platform. To open the **Disassembly** view, choose **Window ▶ Show View ▶ Disassembly**.

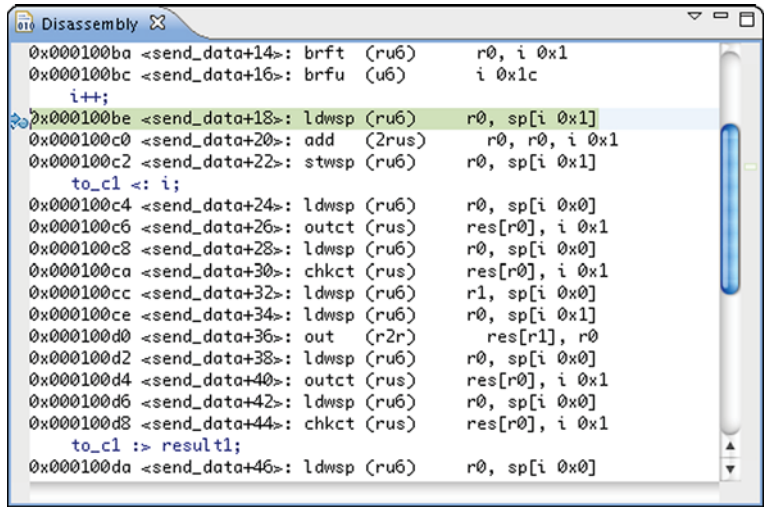


Figure 48: Disassembly view

xTIMEcomposer automatically enables instruction stepping mode whenever the **Disassembly** view has focus. Alternatively, click the **Instruction Stepping Mode** button to enable. Once enabled, click the **Step** button to advance the program by a single assembly instruction.

25 Debug with printf in real-time

IN THIS CHAPTER

- Redirect `stdout` and `stderr` to the xTAG
- Run a program with xTAG output enabled
- Output using the UART interface

The xCORE debugger lets you suspend execution of a program in order to analyze its internal state. However, if your program contains timing-critical behavior, for example due to it implementing a real-time communication protocol, the act of suspending the program may cause other system components to fail, preventing further debugging.

An alternative approach to debugging is to add trace statements to your program that are used to observe its internal behavior at run-time (sometimes referred to as `printf` debugging). By printing the results of intermediate calculations, you can quickly isolate where errors occur in your program.

In a traditional debugging environment, outputting data using a standard such as JTAG results in interrupts that block core execution, slowing your program down considerably. xTIMEcomposer lets you redirect the standard streams `stdout` and `stderr` to an xTAG debug adapter, where the data is buffered until it can be output to the host.

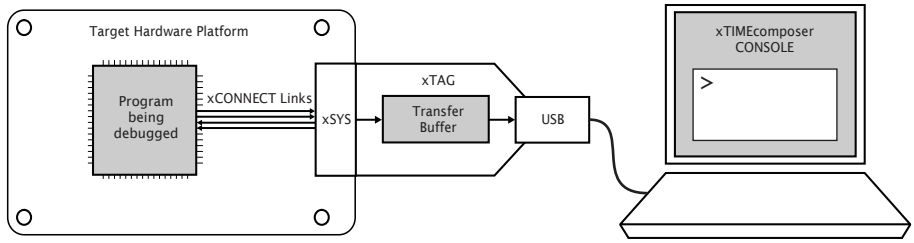


Figure 49: xTAG debug configuration with I/O redirection

In this configuration, calls to output routines such as `printf` complete as soon as the data has been output on an xCONNECT Link, minimizing the effect on the program's timing characteristics. This allows debugging statements to be added to many timing-critical code sections and viewed in a console during execution. In the case of a program crash, all remaining contents in the xTAG buffer is forwarded to the PC, ensuring important information is not lost.

If you are using a legacy FTDI or xTAG-1 debug adapter, or if the XSYS connector on your target hardware does not provide an xCONNECT Link, you can output data over your adapter's UART interface instead (see §25.3). Note that the UART interface offers significantly reduced performance.

25.1 Redirect `stdout` and `stderr` to the xTAG

The program below redirects standard output to the xTAG.

```
#include <stdio.h>
#include <xscope.h>

port receive;
port transmit;

int process(int);

void xscope_user_init(void) {
  xscope_register(0);
  xscope_config_io(XSCOPE_IO_BASIC);
}

int main() {

  while (1) {
    int dataIn, dataOut;

    receive :> dataIn;
    dataOut = process(dataIn);

    /* Debug Information */
    if (dataOut < 0)
      printf("%d %d", dataIn, dataOut);

    transmit <: dataOut;
  }
}
```

In the constructor `xscope_user_init`, the call to `xscope_register` initializes the xTAG interface, and the call to `xscope_config_io` redirects the streams `stdout` and `stderr` to this interface.

The main program inputs data from a port, performs a computation on it and outputs the result to another port. It uses the standard output function `printf` to log instances where the computed result is less than zero.

You can use the C standard I/O functions on any core at the same time. This usage results in a single channel end being allocated on each tile on which data is output.

You can timestamp the output data by calling `xscope_config_io` with the option `XSCOPE_IO_TIMED`. This causes the output timestamp to be displayed with the data in the console. Note that this also reduces the amount of data that can be buffered at any time.

25.2 Run a program with xTAG output enabled

To redirect standard output to the xTAG and display it in the console, you must build and run your program with the xSCOPE instrumentation library. To build and run your program, follow these steps:

1. Open the Makefile for your project.
2. Locate the `XCC_FLAGS_`*config* variable for your build configuration, for example `XCC_FLAGS_Release`.
3. Add the option `-fxscope`.
4. If you are developing using xTIMEcomposer Studio, create a Run Configuration for your target device (see §16.1). In the **xSCOPE** tab, select **Offline mode**. Click **Run** to save and run the configuration.

   xTIMEcomposer loads your program, displaying data received from the xTAG in the console.
5. If you are developing using the command-line tools, pass the option `--xscope` to XRUN, for example:

   ▶ `xrun --xscope myprog.xe`

   XRUN loads your program and remains attached to the xTAG adapter, displaying data received from it in the terminal. XRUN terminates when the program performs a call to `exit`.

25.3 Output using the UART interface

If you are using a legacy FTDI or XTAG-1 debug adapter, or if the XSYS connector on your target hardware does not provide an xCONNECT Link, you can output data over the UART interface provided by your adapter.

To use the UART interface, you must provide the xSCOPE library with a 1-bit UART TX port that has been initialized with the pin connected to the UART-TX pin on your debug adapter. An example initialization is shown below.

```
#include <platform.h>
#include <xscope.h>

port uart_tx = PORT_UART_TX;

void xscope_user_init(void) {
  xscope_register(0);
  xscope_config_uart(uart_tx);
  xscope_config_io(XSCOPE_IO_BASIC);
}
```

To run your program in xTIMEcomposer Studio, create a Run Configuration for your target device (see §16.1) and select the option **Run UART Server**.

To run your program using the command-line tools, pass the option `--uart` to XRUN, for example:

```
▶ xrun --uart --xscope myprog.xe
```

Because the UART interface uses a port instead of an xCONNECT Link, you can use the C standard I/O functions on a single tile only.

Part J

Flash Programming

CONTENTS

- Design and manufacture systems with flash memory
- libflash API
- libquadflash API
- List of devices natively supported by libflash
- List of devices natively supported by libquadflash
- Add support for a new flash device
- XFLASH Command-Line Manual

26 Design and manufacture systems with flash memory

IN THIS CHAPTER

- Boot a program from flash memory
- Generate a flash image for manufacture
- Perform an in-field upgrade
- Customize the flash loader

xTIMEcomposer can be used to target xCORE devices that use SPI flash memory for booting and persistent storage. The xCORE flash format is shown in Figure 50.

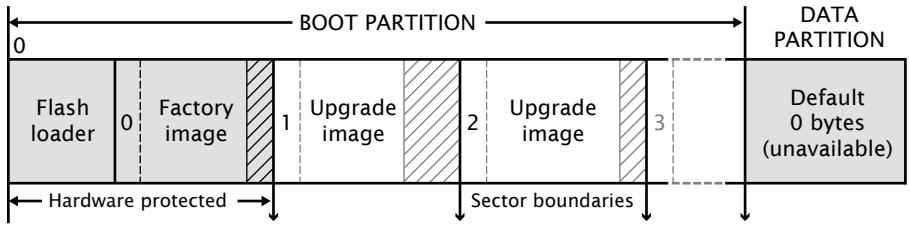


Figure 50: Flash format diagram

The flash memory is logically split between a boot and data partition. The boot partition consists of a flash loader followed by a "factory image" and zero or more optional "upgrade images." Each image starts with a descriptor that contains a unique version number, a header that contains a table of code/data segments for each tile used by the program and a CRC. By default, the flash loader boots the image with the highest version with a valid CRC.

26.1 Boot a program from flash memory

To load a program into an SPI flash memory device on your development board, start the command-line tools (see §3.2) and enter the following commands:

1. `xflash -l`

   XFLASH prints an enumerated list of all JTAG adapters connected to your PC and the devices on each JTAG chain, in the form:

   ```
   ID        Name        Adapter ID        Devices
   --        ----        ----------        -------
   ```

2. `xflash --id ID` *`program`*`.xe`

   XFLASH generates an image in the xCORE flash format that contains a first stage loader and factory image comprising the binary and data segments from your

compiled program. It then writes this image to flash memory using the xCORE device.

The XN file used to compile your program must define an SPI flash device and specify the four ports of the xCORE device to which it is connected (see XM-000929-PC).

26.2 Generate a flash image for manufacture

In manufacturing environments, the same program is typically programmed into multiple flash devices.

To generate an image file in the xCORE flash format, which can be subsequently programmed into flash devices, start the command-line tools (see §3.2) and enter the following command:

▶ `xflash` *program*`.xe -o image-file`

XFLASH generates an image comprising a first stage loader and your program as the factory image, which it writes to the specified file.

26.3 Perform an in-field upgrade

xTimeComposer and the libflash library let you manage multiple firmware upgrades over the life cycle of your product. You can use XFLASH to create an upgrade image and, from within your program, use libflash to write this image to the boot partition. Using libflash, updates are robust against partially complete writes, for example due to power failure: if the CRC of the upgrade image fails during boot, the previous image is loaded instead.

26.3.1 Write a program that upgrades itself

The example program in Figure 51 uses the libflash library to upgrade itself.

The call to `fl_connect` opens a connection between the xCORE and SPI devices, and the call to `fl_getPageSize` determines the SPI device's page size. All read and write operations occur at the page level.

The first upgrade image is located by calling `fl_getFactoryImage` and then `getNextBootImage`. Once located, `fl_startImageReplace` prepares this image for replacement by a new image with the specified (maximum) size. `fl_startImageReplace` must be called until it returns 0, signifying that the preparation is complete.

The function `fl_writeImagePage` writes the next page of data to the SPI device. Calls to this function return after the data is output to the device but may return before the device has written the data to its flash memory. This increases the amount of time available to the processor to fetch the next page of data. The function `fl_endWriteImage` waits for the SPI device to write the last page of data to its flash memory. To simplify the writing operation, XFLASH adds padding to the upgrade image to ensure that its size is a multiple of the page size.

```
#include <platform.h>
#include <flash.h>

#define MAX_PSIZE 256

/* initializers defined in XN file
 * and available via platform.h */

fl_SPIPorts SPI = { PORT_SPI_MISO,
                    PORT_SPI_SS,
                    PORT_SPI_CLK,
                    PORT_SPI_MOSI,
                    XS1_CLKBLK_1 };

int upgrade(chanend c, int usize) {

  /* obtain an upgrade image and write
   * it to flash memory
   * error checking omitted */

  fl_BootImageInfo b;
  int page[MAX_PSIZE];
  int psize;

  fl_connect(SPI);

  psize = fl_getPageSize();
  fl_getFactoryImage(b);
  fl_getNextBootImage(b);

  while(fl_startImageReplace(b, usize))
    ;
  for (int i=0; i page[j];)
    fl_writeImagePage(page);

  fl_endWriteImage();

  fl_disconnect();

      return 0;
}

int main() {
  /* main application - calls upgrade
   * to perform an in-field upgrade */
}
```

Figure 51: C program that uses libflash to upgrade itself

The call `fl_disconnect` closes the connection between the xCORE and SPI devices.

26.3.2 Build and deploy the upgrader

To build and deploy the first release of your program, start the command-line tools (see §3.2) and enter the following commands:

1. `xcc` *file*`.xc -target=`*boardname* `-lflash -o first-release.xe`

   XCC compiles your program and links it against libflash. Alternatively add the option -lflash to your Makefile.

2. `xflash first-release.xe -o manufacture-image`

   XFLASH generates an image in the xCORE flash format that contains a first stage loader and the first release of your program as the factory image.

To build and deploy an upgraded version of your program, enter the following commands:

1. `xcc` *file*`.xc -target=`*boardname* `-lflash -o latest-release.xe`

   XCC compiles your program and links it against libflash.

2. `xflash --upgrade` *version* `latest-release.xe --factory-version` *tools-version* `-o upgrade-image`

   XFLASH generates an upgrade image with the specified version number, which must be greater than 0. Your program should obtain this image to upgrade itself.

If the upgrade operation succeeds, upon resetting the device the loader boots the upgrade image, otherwise it boots the factory image.

26.4 Customize the flash loader

xTIMEcomposer lets you customize the mechanism for choosing which image is loaded from flash. The example program in Figure 52 determines which image to load based on the value at the start of the data partition.

The xCORE loader first calls the function `init`, and then iterates over each image in the boot partition. For each image, it calls `checkCandidateImageVersion` with the image version number and, if this function returns non-zero and its CRC is validated, it calls `recordCandidateImage` with the image version number and address. Finally, the loader calls `reportSelectedImage` to obtain the address of the selected image.

To produce a custom loader, you are required to define the functions `init`, `checkCandidateImageVersion`, `recordCandidateImage` and `reportSelectedImage`.

The loader provides the function `readFlashDataPage`.

```
extern void *readFlashDataPage(unsigned addr);

int   dpVersion;
void *imgAdr;

void init(void) {
  void *ptr = readFlashDataPage(0);
  dpVersion = *(int *)ptr;
}

int checkCandidateImageVersion(int v) {
  return v == dpVersion;
}

void recordCandidateImage(int v, unsigned adr) {
  imgAdr = adr;
}

unsigned reportSelectedImage(void) {
  return imgAdr;
}
```

Figure 52: C functions that customize the flash loader

26.4.1 Build the loader

To create a flash image that contains a custom flash loader and factory image, start the command-line tools (see §3.2) and enter the following commands:

1. `xcc -c` *file*`.xc -o loader.o`

   XCC compiles your functions for image selection, producing a binary object.

2. `xflash` *bin*`.xe --loader loader.o`

   XFLASH writes a flash image containing the custom loader and factory image to the specified file.

26.4.2 Add additional images

The following command builds a flash image that contains a custom flash loader, a factory image and two additional images:

- `xflash` *factory*`.xe --loader loader.o --upgrade 1 usb.xe 0x20000 --upgrade 2 avb.xe`

The arguments to `--upgrade` include the version number, executable file and an optional size in bytes. XFLASH writes each upgrade image on the next sector boundary. The size argument is used to add padding to an image, allowing it to be field-upgraded in the future by a larger image.

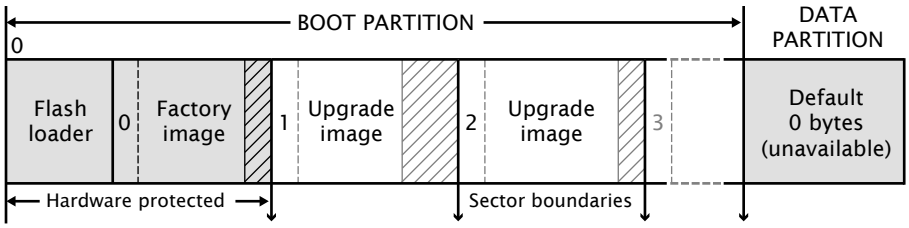
27 libflash API

IN THIS CHAPTER

- General Operations
- Boot Partition Functions
- Data Partition Functions

The libflash library provides functions for reading and writing data to SPI flash devices that use the xCORE format shown in the diagram below.

Figure 53: Flash format diagram



All functions are prototyped in the header file <flash.h>. Except where otherwise stated, functions return 0 on success and non-zero on failure.

27.1 General Operations

The program must explicitly open a connection to the SPI device before attempting to use it, and must disconnect once finished accessing the device.

The functions fl_connect and fl_connectToDevice require an argument of type fl_SPIPorts, which defines the four ports and clock block used to connect to the device.

```
typedef struct {
  in buffered port:8 spiMISO;
  out port spiSS;
  out port spiCLK;
  out buffered port:8 spiMOSI;
  clock spiClkblk;
} fl_SPIPorts;
```

Function	fl_connect
Description	`fl_connect` opens a connection to the specified SPI device.
Type	`int fl_connect(fl_SPIPorts *SPI)`

Function	fl_connectToDevice
Description	`fl_connectToDevice` opens a connection to an SPI device. It iterates through an array of *n* SPI device specifications, attempting to connect using each specification until one succeeds.
Type	`int fl_connectToDevice(fl_SPIPorts *SPI, fl_DeviceSpec spec[], unsigned n)`

Function	fl_getFlashType
Description	`fl_getFlashType` returns an `enum` value for the flash device. The enumeration of devices known to libflash is given below.

```
typedef enum {
  UNKNOWN = 0,
  ALTERA_EPCS1,
  ATMEL_AT25DF041A,
  ATMEL_AT25FS010,
  ST_M25PE10,
  ST_M25PE20,
  WINBOND_W25X40
} fl_FlashId;
```

If the function call `fl_connectToDevice(p, spec, n)` is used to connect to a flash device, `fl_getFlashType` returns the parameter value `spec[`*i*`].flashId` where *i* is the index of the connected device.

Type	`int fl_getFlashType(void)`

Function	fl_getFlashSize
Description	`fl_getFlashSize` returns the capacity of the SPI device in bytes.
Type	`unsigned fl_getFlashSize(void)`

Function	fl_disconnect
Description	`fl_disconnect` closes the connection to the SPI device.
Type	`int fl_disconnect(void)`

27.2 Boot Partition Functions

By default, the size of the boot partition is set to the size of the flash device. Access to boot images is provided through an iterator interface.

Function	fl_getFactoryImage
Description	`fl_getFactoryImage` provides information about the factory boot image.
Type	`int fl_getFactoryImage(fl_BootImageInfo *bootImageInfo)`

Function	fl_getNextBootImage
Description	`fl_getNextBootImage` provides information about the next upgrade image. Once located, an image can be upgraded. Functions are also provided for reading the contents of an upgrade image.
Type	`int fl_getNextBootImage(fl_BootImageInfo *bootImageInfo)`

Function	fl_getImageVersion
Description	`fl_getImageVersion` returns the version number of the specified image.
Type	`unsigned fl_getImageVersion(fl_BootImageInfo *bootImageInfo)`

Function	fl_startImageReplace
Description	`fl_startImageReplace` prepares the SPI device for replacing an image. The old image can no longer be assumed to exist after this call. Attempting to write into the data partition or the space of another upgrade image is invalid. A non-zero return value signifies that the preparation is not yet complete and that the function should be called again. This behavior allows the latency of a sector erase to be masked by the program.
Type	`int` `fl_startImageReplace(fl_BootImageInfo *, unsigned maxsize)`

Function	fl_startImageAdd
Description	`fl_startImageAdd` prepares the SPI device for adding an image after the specified image. The start of the new image is at least padding bytes after the previous image. Attempting to write into the data partition or the space of another upgrade image is invalid. A non-zero return value signifies that the preparation is not yet complete and that the function must be called again. This behavior allows the latency of a sector erase to be masked by the program.

Type	`int fl_startImageAdd(fl_BootImageInfo*,`
	`unsigned maxsize,`
	`unsigned padding)`

Function	fl_startImageAddAt
Description	`fl_startImageAddAt` prepares the SPI device for adding an image at the specified address offset from the base of the first sector after the factory image. Attempting to write into the data partition or the space of another upgrade image is invalid. A non-zero return value signifies that the preparation is not yet complete and that the function must be called again.
Type	`int fl_startImageAddAt(unsigned offset, unsigned maxsize)`

Function	fl_writeImagePage
Description	`fl_writeImagePage` waits until the SPI device is able to accept a request and then outputs the next page of data to the device. Attempting to write past the maximum size passed to `fl_startImageReplace`, `fl_startImageAdd` or `fl_startImageAddAt` is invalid.

Continued on next page

Type	`int` `fl_writeImagePage(const unsigned char page[])`

Function	**fl_writeImageEnd**
Description	`fl_writeImageEnd` waits until the SPI device has written the last page of data to its memory.
Type	`int fl_writeImageEnd(void)`

Function	**fl_startImageRead**
Description	`fl_startImageRead` prepares the SPI device for reading the contents of the specified upgrade image.
Type	`int` `fl_startImageRead(fl_BootImageInfo *b)`

Function	**fl_readImagePage**
Description	`fl_readImagePage` inputs the next page of data from the SPI device and writes it to the array page.
Type	`int` `fl_readImagePage(unsigned char page[])`

Function	**fl_deleteImage**
Description	`fl_deleteImage` erases the upgrade image with the specified image.
Type	`int fl_deleteImage(fl_BootImageInfo* b)`

27.3 Data Partition Functions

All flash devices are assumed to have uniform page sizes but are not assumed to have uniform sector sizes. Read and write operations occur at the page level, and erase operations occur at the sector level. This means that to write part of a sector, a buffer size of at least one sector is required to preserve other data.

In the following functions, writes to the data partition and erasures from the data partition are not fail-safe. If the operation is interrupted, for example due to a power failure, the data in the page or sector is undefined.

Function	**fl_getDataPartitionSize**
Description	`fl_getDataPartitionSize` returns the size of the data partition in bytes.
Type	`unsigned fl_getDataPartitionSize(void)`

Function	**fl_readData**
Description	`fl_readData` reads a number of bytes from an offset into the data partition and writes them to the array dst.
Type	`int fl_readData(unsigned offset, unsigned size, unsigned char dst[])`

Function	**fl_getWriteScratchSize**
Description	`fl_getWriteScratchSize` returns the buffer size needed by `fl_writeData` for the given parameters.
Type	`unsigned fl_getWriteScratchSize(unsigned offset, unsigned size)`

Function	**fl_writeData**
Description	`fl_writeData` writes the array `src` to the specified offset in the data partition. It uses the array `buffer` to preserve page data that must be re-written.
Type	`int fl_writeData(unsigned offset, unsigned size, const unsigned char src[], unsigned char buffer[])`

27.3.1 Page-Level Functions

Function	fl_getPageSize
Description	`fl_getPageSize` returns the page size in bytes.
Type	`unsigned fl_getPageSize(void)`

Function	fl_getNumDataPages
Description	`fl_getNumDataPages` returns the number of pages in the data partition.
Type	`unsigned fl_getNumDataPages(void)`

Function	fl_writeDataPage
Description	`fl_writeDataPage` writes the array data to the *n*-th page in the data partition. The data array must be at least as big as the page size; if larger, the highest elements are ignored.
Type	`unsigned fl_writeDataPage(unsigned n, const unsigned char data[])`

Function	fl_readDataPage
Description	`fl_readDataPage` reads the *n*-th page in the data partition and writes it to the array data. The size of data must be at least as large as the page size.
Type	`unsigned fl_readDataPage(unsigned n, unsigned char data[])`

27.3.2 Sector-Level Functions

Function	fl_getNumDataSectors
Description	`fl_getNumDataSectors` returns the number of sectors in the data partition.
Type	`unsigned fl_getNumDataSectors(void)`

Function	fl_getDataSectorSize
Description	`fl_getDataSectorSize` returns the size of the *n*-th sector in the data partition in bytes.
Type	`unsigned fl_getDataSectorSize(unsigned n)`

Function	fl_eraseDataSector
Description	`fl_eraseDataSector` erases the *n*-th sector in the data partition.
Type	`unsigned fl_eraseDataSector(unsigned n)`

Function	fl_eraseAllDataSectors
Description	`fl_eraseAllDataSectors` erases all sectors in the data partition.
Type	`unsigned fl_eraseAllDataSectors(void)`

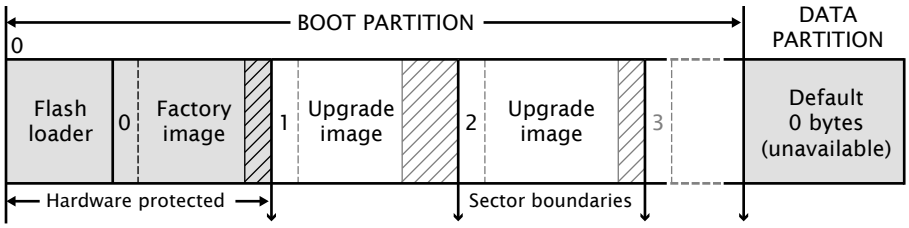
28libquadflash API

IN THIS CHAPTER

- General Operations
- Boot Partition Functions
- Data Partition Functions

The libquadflash library provides functions for reading and writing data to Quad-SPI flash devices that use the xCORE format shown in the diagram below.

Figure 54: Flash format diagram



All functions are prototyped in the header file <quadflash.h>. Except where otherwise stated, functions return 0 on success and non-zero on failure.

28.1 General Operations

The program must explicitly open a connection to the Quad-SPI device before attempting to use it, and must disconnect once finished accessing the device.

The functions `fl_connect` and `fl_connectToDevice` require an argument of type `fl_QSPIPorts`, which defines the four ports and clock block used to connect to the device.

```
typedef struct {
  out port qspiCS;
  out port qspiSCLK;
  out buffered port:32 qspiSIO;
  clock qspiClkblk;
} fl_QSPIPorts;
```

Function	fl_connect
Description	`fl_connect` opens a connection to the specified Quad-SPI device.
Type	`int fl_connect(fl_QSPIPorts *SPI)`

Function	fl_connectToDevice
Description	`fl_connectToDevice` opens a connection to an Quad-SPI device. It iterates through an array of *n* Quad-SPI device specifications, attempting to connect using each specification until one succeeds.
Type	`int fl_connectToDevice(fl_QSPIPorts *SPI, fl_DeviceSpec spec[], unsigned n)`

Function	fl_getFlashType
Description	`fl_getFlashType` returns an `enum` value for the flash device. The enumeration of devices known to libquadflash is given below.

```
typedef enum {
  UNKNOWN = 0,
  ISSI_IS25LQ016B,
  ISSI_IS25LQ032B,
  ISSI_IS25LQ080B,
  SPANSION_S25FL116K,
  SPANSION_S25FL132K,
  SPANSION_S25FL164K,
} fl_QuadFlashId;
```

	If the function call `fl_connectToDevice(p, spec, n)` is used to connect to a flash device, `fl_getFlashType` returns the parameter value `spec[`*i*`].flashId` where *i* is the index of the connected device.
Type	`int fl_getFlashType(void)`

Function	fl_getFlashSize
Description	`fl_getFlashSize` returns the capacity of the Quad-SPI device in bytes.
Type	`unsigned fl_getFlashSize(void)`

Function	fl_disconnect
Description	`fl_disconnect` closes the connection to the Quad-SPI device.
Type	`int fl_disconnect(void)`

28.2 Boot Partition Functions

By default, the size of the boot partition is set to the size of the flash device. Access to boot images is provided through an iterator interface.

Function	fl_getFactoryImage
Description	`fl_getFactoryImage` provides information about the factory boot image.
Type	`int fl_getFactoryImage(fl_BootImageInfo *bootImageInfo)`

Function	fl_getNextBootImage
Description	`fl_getNextBootImage` provides information about the next upgrade image. Once located, an image can be upgraded. Functions are also provided for reading the contents of an upgrade image.
Type	`int fl_getNextBootImage(fl_BootImageInfo *bootImageInfo)`

Function	fl_getImageVersion
Description	`fl_getImageVersion` returns the version number of the specified image.
Type	`unsigned fl_getImageVersion(fl_BootImageInfo *bootImageInfo)`

Function	fl_startImageReplace
Description	`fl_startImageReplace` prepares the Quad-SPI device for replacing an image. The old image can no longer be assumed to exist after this call. Attempting to write into the data partition or the space of another upgrade image is invalid. A non-zero return value signifies that the preparation is not yet complete and that the function should be called again. This behavior allows the latency of a sector erase to be masked by the program.
Type	`int` `fl_startImageReplace(fl_BootImageInfo *, unsigned maxsize)`

Function	fl_startImageAdd
Description	`fl_startImageAdd` prepares the Quad-SPI device for adding an image after the specified image. The start of the new image is at least padding bytes after the previous image. Attempting to write into the data partition or the space of another upgrade image is invalid. A non-zero return value signifies that the preparation is not yet complete and that the function must be called again. This behavior allows the latency of a sector erase to be masked by the program.

Type	`int fl_startImageAdd(fl_BootImageInfo*,`
	`unsigned maxsize,`
	`unsigned padding)`

Function	fl_startImageAddAt
Description	`fl_startImageAddAt` prepares the Quad-SPI device for adding an image at the specified address offset from the base of the first sector after the factory image. Attempting to write into the data partition or the space of another upgrade image is invalid. A non-zero return value signifies that the preparation is not yet complete and that the function must be called again.
Type	`int fl_startImageAddAt(unsigned offset, unsigned maxsize)`

Function	fl_writeImagePage
Description	`fl_writeImagePage` waits until the Quad-SPI device is able to accept a request and then outputs the next page of data to the device. Attempting to write past the maximum size passed to `fl_startImageReplace`, `fl_startImageAdd` or `fl_startImageAddAt` is invalid.

Continued on next page

Type	`int` `fl_writeImagePage(const unsigned char page[])`

Function	**fl_writeImageEnd**
Description	`fl_writeImageEnd` waits until the Quad-SPI device has written the last page of data to its memory.
Type	`int fl_writeImageEnd(void)`

Function	**fl_startImageRead**
Description	`fl_startImageRead` prepares the Quad-SPI device for reading the contents of the specified upgrade image.
Type	`int` `fl_startImageRead(fl_BootImageInfo *b)`

Function	**fl_readImagePage**
Description	`fl_readImagePage` inputs the next page of data from the Quad-SPI device and writes it to the array page.
Type	`int` `fl_readImagePage(unsigned char page[])`

Function	**fl_deleteImage**
Description	`fl_deleteImage` erases the upgrade image with the specified image.
Type	`int fl_deleteImage(fl_BootImageInfo* b)`

28.3 Data Partition Functions

All flash devices are assumed to have uniform page sizes but are not assumed to have uniform sector sizes. Read and write operations occur at the page level, and erase operations occur at the sector level. This means that to write part of a sector, a buffer size of at least one sector is required to preserve other data.

In the following functions, writes to the data partition and erasures from the data partition are not fail-safe. If the operation is interrupted, for example due to a power failure, the data in the page or sector is undefined.

Function	**fl_getDataPartitionSize**
Description	`fl_getDataPartitionSize` returns the size of the data partition in bytes.
Type	`unsigned fl_getDataPartitionSize(void)`

Function	**fl_readData**
Description	`fl_readData` reads a number of bytes from an offset into the data partition and writes them to the array dst.
Type	`int fl_readData(unsigned offset, unsigned size, unsigned char dst[])`

Function	**fl_getWriteScratchSize**
Description	`fl_getWriteScratchSize` returns the buffer size needed by `fl_writeData` for the given parameters.
Type	`unsigned fl_getWriteScratchSize(unsigned offset, unsigned size)`

Function	**fl_writeData**
Description	`fl_writeData` writes the array `src` to the specified offset in the data partition. It uses the array `buffer` to preserve page data that must be re-written.
Type	`int fl_writeData(unsigned offset, unsigned size, const unsigned char src[], unsigned char buffer[])`

28.3.1 Page-Level Functions

Function	fl_getPageSize
Description	`fl_getPageSize` returns the page size in bytes.
Type	`unsigned fl_getPageSize(void)`

Function	fl_getNumDataPages
Description	`fl_getNumDataPages` returns the number of pages in the data partition.
Type	`unsigned fl_getNumDataPages(void)`

Function	fl_writeDataPage
Description	`fl_writeDataPage` writes the array data to the *n*-th page in the data partition. The data array must be at least as big as the page size; if larger, the highest elements are ignored.
Type	`unsigned fl_writeDataPage(unsigned n, const unsigned char data[])`

Function	fl_readDataPage
Description	`fl_readDataPage` reads the *n*-th page in the data partition and writes it to the array data. The size of data must be at least as large as the page size.
Type	`unsigned fl_readDataPage(unsigned n, unsigned char data[])`

28.3.2 Sector-Level Functions

Function	fl_getNumDataSectors
Description	`fl_getNumDataSectors` returns the number of sectors in the data partition.
Type	`unsigned fl_getNumDataSectors(void)`

Function	fl_getDataSectorSize
Description	`fl_getDataSectorSize` returns the size of the *n*-th sector in the data partition in bytes.
Type	`unsigned fl_getDataSectorSize(unsigned n)`

Function	fl_eraseDataSector
Description	`fl_eraseDataSector` erases the *n*-th sector in the data partition.
Type	`unsigned fl_eraseDataSector(unsigned n)`

Function	fl_eraseAllDataSectors
Description	`fl_eraseAllDataSectors` erases all sectors in the data partition.
Type	`unsigned fl_eraseAllDataSectors(void)`

29 List of devices natively supported by libflash

libflash supports a wide range of flash devices available in the market. Each flash device is described using a SPI specification file. The table in Figure 56 lists the flash devices for which SPI spec files are included with xTIMEcomposer.

Manufacturer	Part Number	Enabled in libflash by default
Altera	EPCS1	Y
AMIC	A25L016	N
	A25L40P	N
	A25L40PT	N
	A25L40PUM	N
	A25L80P	N
Atmel	AT25DF021	N
	AT25DF041A	Y
	AT25F512	N
	AT25FS010	Y
ESMT	F25L004A	N
Macronix	MX25L1005C	N
NUMONYX	M25P10	N
	M25P16	N
	M25P40	N
	M45P10E	N
SPANSION	S25FL204K	N
SST	SST25VF010	N
	SST25VF016	N
	SST25VF040	N
ST Microelectronics	M25PE10	Y
	M25PE20	Y
Winbond	W25X10	N
	W25X20	N
	W25X40	Y

Figure 55: List of flash devices supported natively by libflash

30 List of devices natively supported by libquadflash

libquadflash supports a limited range of flash devices available in the market. Each flash device is described using a Quad-SPI specification file. The table in ??? lists the flash devices for which Quad-SPI spec files are included with xTIMEcomposer.

Figure 56: List of flash devices supported natively by libquadflash

Manufacturer	Part Number	Enabled in libquadflash by default
ISSI	IS25LQ080B	Y
	IS25LQ016B	Y
	IS25LQ032B	Y
SPANSION	S25FL116K	Y
	S25FL132K	Y
	S25FL164K	Y

31 Add support for a new flash device

IN THIS CHAPTER

- Libflash Device ID
- Page Size and Number of Pages
- Address Size
- Clock Rate
- Read Device ID
- Sector Erase
- Write Enable/Disable
- Memory Protection
- Programming Command
- Read Data
- Sector Information
- Status Register Bits
- Add Support to xTimeComposer
- Select a Flash Device

To support a new flash device, a configuration file must be written that describes the device characteristics, such as page size, number of pages and commands for reading, writing and erasing data. This information can be found in the datasheet for the flash device. Many devices available in the market can be described using these configuration parameters; those that cannot are unsupported.

The configuration file for the Numonyx M25P10-A[3] is shown below. The device is described as an initializer for a C structure, the values of which are described in the following sections.

[3] http://www.xmos.com/references/m25p10a

```
10,                     /* 1.  libflash device ID */
256,                    /* 2.  Page size */
512,                    /* 3.  Number of pages */
3,                      /* 4.  Address size */
4,                      /* 5.  Clock divider */
0x9f,                   /* 6.  RDID cmd */
0,                      /* 7.  RDID dummy bytes */
3,                      /* 8.  RDID data size in bytes */
0x202011,               /* 9.  RDID manufacturer ID */
0xD8,                   /* 10. SE cmd */
0,                      /* 11. SE full sector erase */
0x06,                   /* 12. WREN cmd */
0x04,                   /* 13. WRDI cmd */
PROT_TYPE_SR,           /* 14. Protection type */
{{0x0c,0x0},{0,0}},     /* 15. SR protect and unprotect cmds */
0x02,                   /* 16. PP cmd */
0x0b,                   /* 17. READ cmd */
1,                      /* 18. READ dummy bytes*/
SECTOR_LAYOUT_REGULAR,  /* 19. Sector layout */
{32768,{0,{0}}},        /* 20. Sector sizes */
0x05,                   /* 21. RDSR cmd*/
0x01,                   /* 22. WRSR cmd */
0x01,                   /* 23. WIP bit mask */
```

31.1 Libflash Device ID

```
10, /* 1. libflash device ID */
```

This value is returned by libflash on a call to the function `fl_getFlashType` so that the application can identify the connected flash device.

31.2 Page Size and Number of Pages

```
10,                     /* 1.  libflash device ID */
256,                    /* 2.  Page size */
```

These values specify the size of each page in bytes and the total number of pages across all available sectors. On the M25P10-A datasheet, these can be found from the following paragraph on page 6:

> *The memory is organized as 4 sectors, each containing 128 pages. Each page is 256 bytes wide. Thus, the whole memory can be viewed as consisting of 512 pages, or 131,072 bytes.*

31.3 Address Size

```
3,                          /* 4.  Address size */
```

This value specifies the number of bytes used to represent an address. Figure 57 reproduces the part of the M25P10-A datasheet that provides this information. In the table, all instructions that require an address take three bytes.

Instruction	Description	One-byte instruction code		Address byte	Dummy bytes	Data bytes
WREN	Write Enable	0000 0110	06h	0	0	0
WRDI	Write Disable	0000 0100	04h	0	0	0
RDID	Read	1001 1111	9Fh	0	0	1 to 3
RDSR	Read Status Register	0000 0101	05h	0	0	1 to ∞
WRSR	Write Status Register	0000 0001	01h	0	0	1
READ	Read Data Bytes	0000 0011	03h	3	0	1 to ∞
FAST_READ	Read Data Bytes at Higher Speed	0000 1011	0Bh	3	1	1 to ∞
PP	Page Program	0000 0010	02h	3	0	1 to 256
SE	Sector Erase	1101 1000	D8h	3	0	0
BE	Bulk Erase	1100 0111	C7h	0	0	0
DP	Deep Power-down	1011 1001	B9h	0	0	0
RES	Release from Deep Power-down, and Read Electronic Signature	1010 1011	ABh	0	3	1 to ∞
	Release from Deep Power-down			0	0	0

Figure 57: Table 4 on page 17 of M25P10-A datasheet

31.4 Clock Rate

```
4,                          /* 5.  Clock divider */
```

This value is used to determine the clock rate for interfacing with the SPI device. For a value of *n*, the SPI clock rate used is 100/2**n* MHz. libflash supports a maximum of 12.5MHz.

Figure 58 reproduces the part of the M25P10-A datasheet that provides this information. The AC characteristics table shows that all instructions used in the configuration file, as discussed throughout this document, can operate at up to 25MHz. This is faster than libflash can support, so the value 4 is provided to generate a 12.5MHz clock.

In general, if the SPI device supports different clock rates for different commands used by libflash, the lowest value must be specified.

Figure 58: Table 18 on page 40 of M25P10-A datasheet (first four entries only).

Symbol	Alt.	Parameter	Min	Typ	Max	Unit
f_C	f_C	Clock frequency for the following instructions: FAST_READ, PP, SE, BE, DP, RES, WREN, WRDI, RDSR, WRSR	D.C.		25	MHz
f_R		Clock frequency for READ instructions	D.C.		20	MHz
t_{CH}	t_{CLH}	Clock High time	18			ns
t_{CL}	t_{CLL}	Clock Low time	18			ns

31.5 Read Device ID

```
0x9f,                   /* 6.  RDID cmd */
0,                      /* 7.  RDID dummy bytes */
3,                      /* 8.  RDID data size in bytes */
0x202011,               /* 9.  RDID manufacturer ID */
```

Most flash devices have a hardware identifier that can be used to identify the device. This is used by libflash when one or more flash devices are supported by the application to determine which type of device is connected. The sequence for reading a device ID is typically to issue an RDID (read ID) command, wait for zero or more dummy bytes, and then read one or more bytes of data.

Figure 57 reproduces the part of the M25P10-A datasheet that provides this information. The row for the instruction RDID shows that the command value is `0x9f`, that there are no dummy bytes, and one to three data bytes. As shown in Figure 59 and Figure 60, the amount of data read depends on whether just the manufacturer ID (first byte) is required, or whether both the manufacturer ID and the device ID (second and third bytes) are required. All three bytes are needed to uniquely identify the device, so the manufacturer ID is specified as the three-byte value `0x202011`.

Figure 59: Table 5 on page 19 of M25P10-A datasheet

Manufacturer identification	Device identification	
	Memory type	Memory capacity
20h	20h	11h

In general, if there is a choice of RDID commands then the JEDEC compliant one should be preferred. Otherwise, the one returning the longest ID should be used.

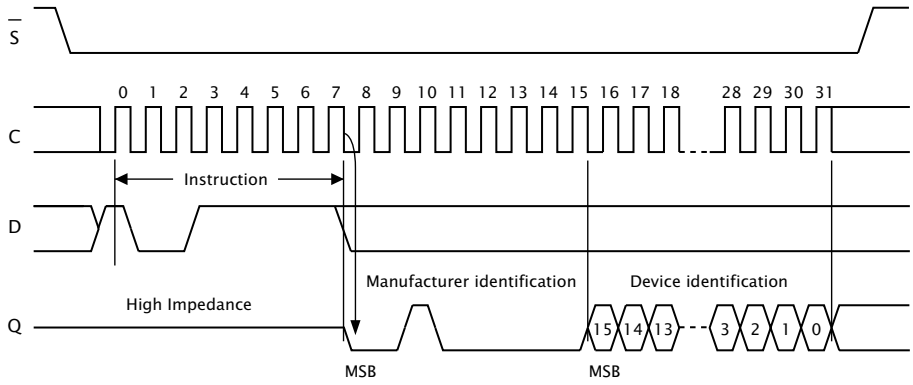


Figure 60: Figure 9 on page 19 of M25P10-A datasheet

31.6 Sector Erase

```
0xD8,                   /* 10. SE cmd */
0,                      /* 11. SE full sector erase */
```

Most flash devices provide an instruction to erase all or part of a sector.

Figure 57 reproduces the part of the M25P10-A datasheet that provides this information. The row for the instruction SE shows that the command value is 0xd8. On the M25P10-A datasheet, the amount of data erased can be found from the first paragraph on page 28:

> *The Sector Erase (SE) instruction sets to '1' (FFh) all bits inside the chosen sector.*

In this example the SE command erases all of the sector, so the SE data value is set to 0. If the number of bytes erased is less than a full sector, this value should be set to the number of bytes erased.

31.7 Write Enable/Disable

```
0x06,                   /* 12. WREN cmd */
0x04,                   /* 13. WRDI cmd */
```

Most flash devices provide instructions to enable and disable writes to memory. Figure 57 reproduces the part of the M25P10-A datasheet that provides this information. The row for the instruction WREN shows that the command value is 0x06, and the row for the instruction WRDI shows that the command value is 0x04.

31.8 Memory Protection

```
PROT_TYPE_SR,            /* 14. Protection type */
{{0x0c,0x0},{0,0}},      /* 15. SR protect and unprotect cmds */
```

Some flash devices provide additional protection of sectors when writes are enabled. For devices that support this capability, libflash attempts to protect the flash image from being accidentally corrupted by the application. The supported values for *protection* type are:

`PROT_TYPE_NONE`
: The device does not provide protection

`PROT_TYPE_SR`
: The device provides protection by writing the status register

`PROT_TYPE_SECS`
: The device provides commands to protect individual sectors

The protection details are specified as part of a construction of the form:

```
{{a,b},{c,d}}
```

If the device does not provide protection, all values should be set to 0. If the device provides SR protection, *a* and *b* should be set to the values to write to the SR to protect and unprotect the device, and *c* and *d* to 0. Otherwise, *c* and *d* should be set to the values to write to commands to protect and unprotect the device, and *a* and *b* to 0.

Figure 61 and Figure 62 reproduce the parts of the M25P10-A datasheet that provide this information. The first table shows that BP0 and BP1 of the status register should be set to 1 to protect all sectors, and both to 0 to disable protection. The second table shows that these are bits 2 and 3 of the SR.

Figure 61: Table 2 on page 13 of M25P10-A datasheet

Status Register Content		Memory content	
BP1 bit	**BP0 bit**	**Protected area**	**Unprotected area**
0	0	none	All sectors (four sectors: 0, 1, 2 and 3)
0	1	Upper quarter (sector 3)	Lower three-quarters (three sectors: 0 to 2)
1	0	Upper half (two sectors: 2 and 3)	Lower half (sectors 0 and 1)
1	1	All sectors (four sectors: 0, 1, 2 and 3)	none

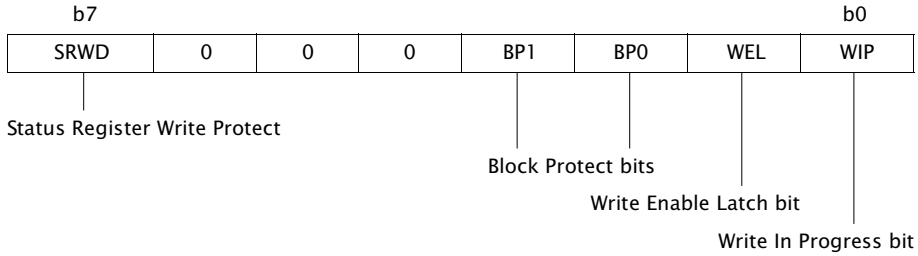


Figure 62: Table 6 on page 20 of M25P10-A datasheet

31.9 Programming Command

```
0x02,                    /* 16. PP cmd */
```

Devices are programmed either a page at a time or a small number of bytes at a time. If page programming is available it should be used, as it minimizes the amount of data transmitted over the SPI interface.

Figure 57 reproduces the part of the M25P10-A datasheet that provides this information. In the table, a page program command is provided and has the value 0x02.

If page programming is not supported, this value is a concatenation of three separate values. Bits 0..7 must be set to 0. Bits 8..15 should contain the program command. Bits 16..23 should contain the number of bytes per command. The libflash library requires that the first program command accepts a three byte address but subsequent program command use auto address increment (AAI).

An example of a device without a PP command is the ESMT F25L004A[4]. Figure 63 reproduces the part of the F25L004A datasheet that provides this information. In the timing diagram, the AAI command has a value 0xad, followed by a three-byte address and two bytes of data.

Figure 63: Table 7 on page 12 of F25L004A datasheet.

Symbol	Parameter	Minimum	Units
$T_{PU-READ}$	V_{DD} Min to Read Operation	10	µs
$T_{PU-WRITE}$	V_{DD} Min to Write Operation	10	µs

The corresponding entry in the specification file is:

```
0x00|(0xad<<8)|(2<<16), /* No PP, have AAI for 2 bytes */
```

[4] http://www.xmos.com/references/f25l004

31.10 Read Data

```
0x0b,                       /* 17. READ cmd */
1,                          /* 18. READ dummy bytes*/
```

The sequence for reading data from a device is typically to issue a READ command, wait for zero or more dummy bytes, and then read one or more bytes of data.

Figure 57 reproduces the part of the M25P10-A datasheet that provides this information. There are two commands that can be used to read data: READ and FAST_READ. The row for the instruction FAST_READ shows that the command value is 0x0b, followed by one dummy byte.

31.11 Sector Information

```
SECTOR_LAYOUT_REGULAR,  /* 19. Sector layout */
{32768,{0,{0}}},        /* 20. Sector sizes */
```

The first value specifies whether all sectors are the same size. The supported values are:

SECTOR_LAYOUT_REGULAR
: The sectors all have the same size

SECTOR_LAYOUT_IRREGULAR
: The sectors have different sizes

On the M25P10-A datasheet, this can be found from the following paragraph on page 15:

> *The memory is organized as:*
>
> - 131,072 bytes (8 bits each)
> - 4 sectors (256 Kbits, 32768 bytes each)
> - 512 pages (256 bytes each).

The sector sizes is specified as part of a construction: {a, {b, {c}}}. For regular sector sizes, the size is specified in *a*. The values of *b* and *c* should be 0.

For irregular sector sizes, the size number of sectors is specified in *b*. The log base 2 of the number of pages in each sector is specified in c. The value of *a* should be 0. An example of a device with irregular sectors is the AMIC A25L80P[5]. Figure 64 reproduces the part of this datasheet that provides the sector information.

[5] http://www.xmos.com/references/a25l80p

Sector	Sector Size (Kb)	Address Range	
15	64	F0000h	FFFFFh
14	64	E0000h	EFFFFh
13	64	D0000h	DFFFFh
12	64	C0000h	CFFFFh
11	64	B0000h	BFFFFh
10	64	A0000h	AFFFFh
9	64	90000h	9FFFFh
8	64	80000h	8FFFFh
7	64	70000h	7FFFFh
6	64	60000h	6FFFFh
5	64	50000h	5FFFFh
4	64	40000h	4FFFFh
3	64	30000h	3FFFFh
2	64	20000h	2FFFFh
1	64	10000h	1FFFFh
0-4	32	08000h	0FFFFh
0-3	16	04000h	07FFFh
0-2	8	02000h	03FFFh
0-1	4	01000h	01FFFh
0-0	4	00000h	00FFFh

Figure 64: Table 2 on page 7 of A25L80P datasheet

The corresponding entry in the specification file is:

```
SECTOR_LAYOUT_IRREGULAR,
    {0,{20,{4,4,5,6,7,8,8,8,8,8,8,8,8,8,8,8,8,8,8,8}}},
```

31.12 Status Register Bits

```
0x05,                   /* 21. RDSR cmd*/
0x01,                   /* 22. WRSR cmd */
0x01,                   /* 23. WIP bit mask */
```

Most flash devices provide instructions to read and write a status register, including a write-in-progress bit mask.

Figure 57 reproduces the part of the M25P10-A datasheet that documents the RDSR and WRSR commands. The diagram in Figure 62 shows that the WIP bit is in bit position 0 of the SR, resulting in a bit mask of `0x01`.

31.13 Add Support to xTimeComposer

A configuration file can be used with libflash or xflash. The example program below uses libflash to connect to a M25P10-A device, the configuration parameters which are specified in `m25p10a`.

```
#include "platform.h"
#include "flash.h"
#include "flashlib.h"
#include "stdio.h"
#include "stdlib.h"

fl_PortHolderStruct SPI = {PORT_SPI_MISO,
                           PORT_SPI_SS,
                           PORT_SPI_CLK,
                           PORT_SPI_MOSI,
                           XS1_CLKBLK_1};

fl_DeviceSpec myFlashDevices[] = {
 {
   #include "m25p10a"
 }
};

int flash_access() {
  if (fl_connectToDevice(SPI, myFlashDevices,
      sizeof(myFlashDevices)/sizeof(fl_DeviceSpec)) != 0) {
    printf("No supported flash devices found.\n"); exit(1);
  } else {
  printf("Found custom flash device m25p10a.\n"); exit(0);
  }
  return 0;
}

int main() {
  // multicore main is required for xscope
  par {
    on stdcore[0] : flash_access();
  }
}
```

The custom flash device must be specified in the XN file as follows:

```
<ExternalDevices>
  <Device NodeId="0" Tile="0" Name="bootFlash"
          Class="SPIFlash" Type="M25P10A">
    <Attribute Name="PORT_SPI_MISO" Value="PORT_SPI_MISO" />
    <Attribute Name="PORT_SPI_SS" Value="PORT_SPI_SS" />
    <Attribute Name="PORT_SPI_CLK" Value="PORT_SPI_CLK" />
    <Attribute Name="PORT_SPI_MOSI" Value="PORT_SPI_MOSI" />
  </Device>
</ExternalDevices>
```

To compile an image file that links to the lib flash library, start the command-line tools (see §3.2) and enter the following command:

```
▶ xcc main.xc -o prog.xe -target=target_with_custom_flash.xn -lflash
```

To generate an image file in the xCORE flash format, which can be subsequently programmed into the above flash device, enter the following command:

```
▶ xflash prog.xe -o imgfile --spi-spec m25p10a
```

XFLASH generates an image for the custom flash device, which it writes to the specified image file.

31.14 Select a Flash Device

When selecting a flash device for use with an xCORE device, the following guidelines are recommended:

- If access to the data partition is required, select a device with fine-grained erase granularity, as this will minimize the gaps between the factory and upgrade images, and will also minimize the amount of data that libflash needs to buffer when writing data.
- Select a device with sector protection if possible, to ensure that the bootloader and factory image are protected from accidental corruption post-deployment.
- Select a flash speed grade suitable for the application. Boot times are minimal even at low speeds.

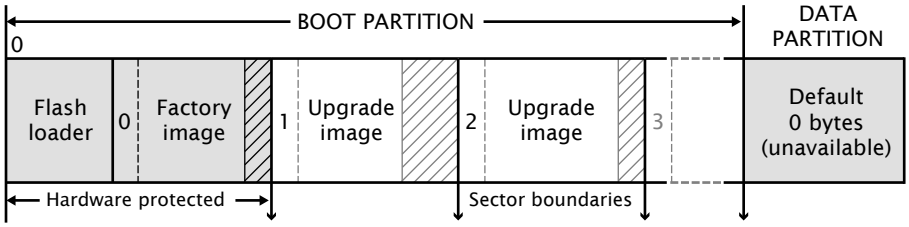
32XFLASH Command-Line Manual

IN THIS CHAPTER

- Overall Options
- Target Options
- Security Options
- Programming Options

XFLASH creates binary files in the xCORE flash format, as illustrated in the diagram below. It can also program these files onto flash devices used to boot XMOS systems.

Figure 65: Flash format diagram



32.1 Overall Options

The following options are used to specify the program images and data that makes up the binary and its layout. Padding is inserted when required to ensure that images are aligned on sector boundaries.

`xe-file [size]`

`--factory xe-file [size]`

Specifies *xe-file* as the factory image. If *size* is specified, padding is inserted to make the space between the start of this image and the next image at least the specified size. The default unit of *size* is "bytes;" the size can be postfixed with `k` to specify a unit of kilobytes.

At most one factory image may by specified.

`--upgrade id xe-file [size]`

Specifies *xe-file* as an upgrade image with version *id*. Each version number must be a unique number greater than 0. If *size* is specified, padding is inserted to make the space between the start of this image and the next image at least the specified size. The default unit of size is "bytes;" the size can be postfixed with `k` to specify a unit of kilobytes.

Multiple upgrade images are inserted into the boot partition in the order specified on the command line.

If no factory image is specified, a single upgrade image may be specified and written to a file with the option `-o`.

`--factory-version version`
Specifies version as the tools release master version that was used to create the factory image. Accepted values are: 13.0, 13.1, 13.2 and 14.0. This option need only be specified when `--upgrade` is provided but `--factory` is not. This option will ensure that the produced flash upgrade image is of the correct format for the installed factory image.

`--boot-partition-size n`
Specifies the size of the boot partition to be *n* bytes. If left unspecified, the default size used is the total size of the flash device. *n* must be greater than or equal to the minimum size required to store the boot loader, factory image and any upgrade images.

`--data file` Specifies the contents of *file* to be written to the data partition.

`--loader file`
Specifies custom flash loader functions in *file* (see §26.4). The file may be either an object (`.o`) or archive (`.a`).

By default, the xCORE flash loader loads the image with the highest version number that validates against its CRC.

`--verbose` Prints additional information about the program when loaded onto the target system.

`--help` Prints a description of the supported command line options.

`--version` Displays the version number and copyrights.

32.2 Target Options

The following options are used to specify which flash device the binary is to be programmed on. The type of flash device used determines the values for the SPI divider, sector size and memory capacity.

`--list-devices`
`-l` Prints an enumerated list of all JTAG adapters connected to the PC and the devices on each JTAG chain, in the form:

```
ID          Name          Adapter ID          Devices
--          ----          ----------          -------
```

The adapters are ordered by their serial numbers.

`--id ID` Specifies the adapter connected to the target hardware.

XFLASH connects to the target platform and determines the type of flash device connected to it.

`--adapter-id ADAPTER-SERIAL-NUMBER`
Specifies the serial number of the adapter connected to the target hardware. XFLASH connects to the target hardware and determines the type of flash device connected to it.

`--jtag-speed n`
Sets the divider for the JTAG clock to *n*. The corresponding JTAG clock speed is 6/(*n*+1)MHz. The default value of the divider for the JTAG clock is 0, representing 6MHz.

`--spi-spec file`
Enables support for the flash device specified in *file* (see §31).

`--spi-div n` Sets the divider for the SPI clock to *n*, producing an SPI clock speed of 100/2*n*MHz. By default, if no target is specified, the divider value is set to 3 (16.7MHz).

`--spi-read-id cmd`
Reads the spi manufacturer's id from the attached device. The *cmd* can be obtained from the spi manufacturer's datasheet. If there is more than one device in a network then all id's will be returned.

`--noinq` Does not run the device inquisitor program, which checks that images are aligned on sector boundaries. If `--noinq` is omitted XFLASH expects to be able to connect to the device via JTAG.

`--force-pll-reset`
Will force an xCORE-200 device to reset when the PLL register is written during the boot process. By default the xCORE-200 device will not reset when the PLL register is written allowing for faster boot times.

32.3 Security Options

The following options are used in conjunction with the AES Module (see §33.1).
`--key keyfile`
Encrypts the images in the boot partition using the keys in *keyfile*.

`--disable-otp`
Causes the flash loader to disable access to OTP memory after the program is booted. This is default if the option `--key` is used.

`--enable-otp`
Causes the flash loader to enable access to OTP memory after the program is booted. This is default unless the option `--key` is used.

32.4 Programming Options

By default, XFLASH programs the generated binary file to the target flash device.
`-o file` Places output in *file*, disabling programming.

If the target platform is booted from more than one flash device, multiple output files are created, one for each device. The name of each output file is `file_`*node*, where *node* is the value of the `Id` attribute (see §54.4) of the corresponding node.

`--arm-binary file`
Places XFLASH output into the reserved flash memory area of the ARM binary *file*.

The given ARM binary file will not be directly modified, but a copy of the file will be created. If the option -o *output file* is given then *output file* will contain the modified ARM binary. If the -o option is not supplied then a temporary file will be created and programmed directly onto the attached ARM device.

The following options perform generic read, write and erase operations on the target flash device. A target XN file must be specified, which provides ports used to communicate with the SPI device on the hardware platform.

`--target-file xn-file [node]`
Specifies *xn-file* as the target platform.

If *xn-file* specifies more than one flash device, a value for *node* must be specified. This value must correspond to the `Id` attribute (see §54.4) of the node connected to the target flash device.

`--target platform [node]`
Specifies a target platform. The platform configuration must be specified in the file `platform.xn`, which is searched for in the paths specified by the `XCC_DEVICE_PATH` environment variable (see §10.8).

If *xn-file* specifies more than one flash device, a value for *node* must be specified. This value must correspond to the `Id` attribute (see §54.4) of the node connected to the target flash device.

`--erase-all` Erases all memory on the flash device.

`--read-all` Reads the contents of all memory on the flash device and writes it to a file on the host. Must be used with `-o`.

`--write-all file`
Writes the bytes in *file* to the flash device.

`--no-reset-on-write`
Prevents XFLASH from resetting the xCORE after programming the device.

`--no-compression`
Prevents XFLASH from compressing all binary data.

`--no-compression-validation`
Prevents XFLASH from validating the decompression algorithm within XSIM.

Part K

Security and OTP Programming

CONTENTS

- Safeguard IP and device authenticity
- XBURN Command-Line Manual

33 Safeguard IP and device authenticity

IN THIS CHAPTER

- The xCORE AES module
- Develop with the AES module enabled
- Production flash programming flow
- Production OTP programming flow

xCORE devices contain on-chip one-time programmable (OTP) memory that can be blown during or after device manufacture testing. You can program the xCORE AES Module into the OTP of a device, allowing programs to be stored encrypted on flash memory. This helps provide:

- **Secrecy**

  Encrypted programs are hard to reverse engineer.

- **Program Authenticity**

  The AES loader will not load programs that have been tampered with or other third-party programs.

- **Device Authenticity**

  Programs encrypted with your secret keys cannot be cloned using xCORE devices provided by third parties.

Once the AES Module is programmed, the OTP security bits are blown, transforming each tile into a “secure island” in which all computation, memory access, I/O and communication are under exclusive control of the code running on the tile. When set, these bits:

- force boot from OTP to prevent bypassing,
- disable JTAG access to the tile to prevent the keys being read, and
- stop further writes to OTP to prevent updates.

The AES module provides a strong level of protection from casual hackers. It is important to realize, however, that there is no such thing as unbreakable security and there is nothing you can do to completely prevent a determined and resourceful attacker from extracting your keys.

33.1 The xCORE AES module

The xCORE AES Module authenticates and decrypts programs from SPI flash devices. When programmed into a device, it enables the following secure boot procedure, as illustrated in Figure 66.

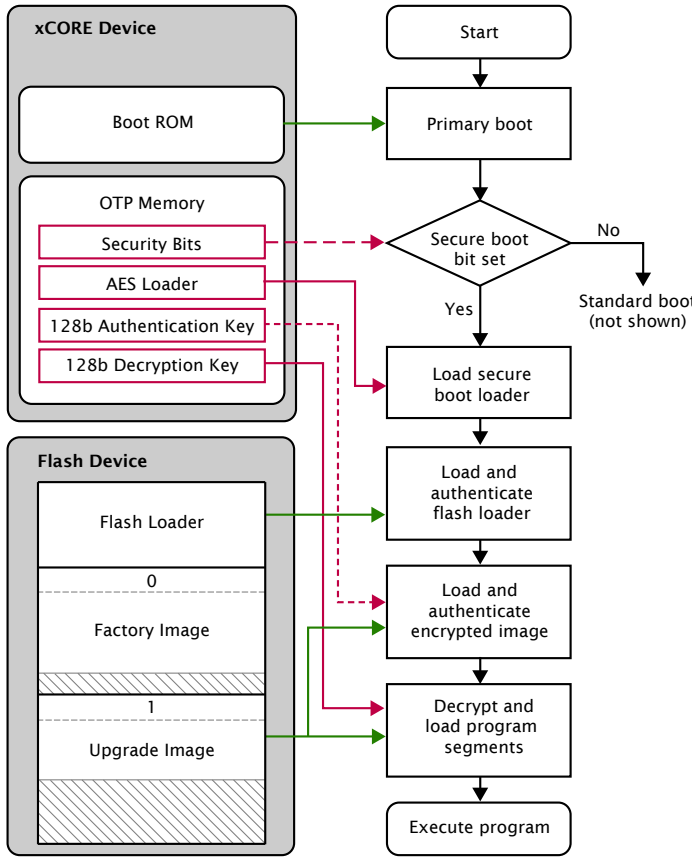


Figure 66: Secure boot procedure used with the AES Module

1. The device loads the primary bootloader from its ROM, which detects that the secure boot bit is set in the OTP and then loads and executes the AES Module from OTP.
2. The AES Module loads the flash loader into RAM over SPI.
3. The AES Module authenticates the flash loader using the CMAC-AES-128 algorithm and the 128-bit authentication key. If authentication fails, boot is halted.
4. The AES Module places the authentication key and decryption key in registers and jumps to the flash loader.

The flash loader performs the following operations:

1. Selects the image with the highest number that validates against its CRC.
2. Authenticates the selected image header using its CMAC tag and authentication key. If the authentication fails, boot is halted.
3. Authenticates, decrypts and loads the table of program/data segments into memory. If any images fail authentication, the boot halts.
4. Starts executing the program.

For multi-node systems, the AES Module is written to the OTP of one tile, and a secure boot-from-xCONNECT Link protocol is programmed into all other tiles.

33.2 Develop with the AES module enabled

You can activate the AES Module at any time during development or device manufacture. In a development environment, you can activate the module but leave the security bits unset, enabling:

- XFLASH to use the device to load programs onto flash memory,
- XGDB to debug programs running on the device, and
- XBURN to later write additional OTP bits to protect the device.

In a production environment, you must protect the device to prevent the keys from being read out of OTP by the end user.

To program the AES Module into the xCORE device on your development board, start the command-line tools (see §3.2) and enter the following commands:

1. `xburn --genkey` *keyfile*

   XBURN writes two random 128-bit keys to *keyfile*. The first line is the authentication key, the second line the decryption key.

   The keys are generated using the open-source library crypto++. If you prefer, you can create this file and provide your own keys.

2. `xburn -l`

   XBURN prints an enumerated list of all JTAG adapters connected to your PC and the devices on each JTAG chain, in the form:

   ```
   ID - NAME (ADAPTER-SERIAL-NUMBER)
   ```

3. `xburn --id` *ID* `--lock` *keyfile* `--target-file` *target*`.xn --enable-jtag --disable-master-lock`

   XBURN writes the AES Module and security keys to the OTP memory of the target device and sets its secure boot bit. The SPI ports used for booting are taken from the XN file (see XM-000929-PC).

To encrypt your program and write it to flash memory, enter the command:

- `xflash --id` *ID* *bin*`.xe --key` *keyfile*

To protect the xCORE device, preventing any further development, enter the command:

- `xburn --id` *ID* `--target-file` *target*`.xn --disable-jtag --enable-master-lock`

33.3 Production flash programming flow

In production manufacturing environments, the same program is typically programmed into multiple SPI devices.

To generate an encrypted image in the xCORE flash format, start the command-line tools (see §3.2) and enter the following command:

- `xflash` *prog*`.xe -key` *keyfile* `-o` *image-file*

This image can be programmed directly into flash memory using a third-party flash programmer, or it can be programmed using XFLASH (via an xCORE device). To program using XFLASH, enter the following commands:

1. `xflash -l`

   XFLASH prints an enumerated list of all JTAG adapters connected to your PC and the devices on each JTAG chain, in the form:

   `ID - NAME (ADAPTER-SERIAL-NUMBER)`

2. `xflash --id` *ID* `--target-file` *platform*`.xn --write-all` *image-file*

   XFLASH generates an image in the xCORE flash format that contains a first stage loader and factory image comprising the binary and data segments from your compiled program. It then writes this image to flash memory using the xCORE device.

The XN file must define an SPI flash device and specify the four ports of the xCORE device to which it is connected (see XM-000929-PC).

33.4 Production OTP programming flow

In production manufacturing environments, the same keys are typically programmed into multiple xCORE devices.

To generate an image that contains the AES Module and security keys to be written to the OTP, start the command-line tools (see §3.2) and enter the following commands:

1. `xburn --genkey` *keyfile*

   XBURN writes two random 128-bit keys to keyfile. The first line is the authentication key, the second line the decryption key.

   The keys are generated using the open-source library crypto++. If you prefer, you can create this file and provide your own keys.

2. `xburn --target-file` *target*`.xn --lock` *keyfile* `-o` *aes-image*`.otp`

   XBURN generates an image that contains the AES Module, security keys and the values for the security bits.

The image contains the keys and must be kept secret.

To write the AES Module and security bits to a device in a production environment, enter the following commands:

1. `xburn -l`

   XBURN prints an enumerated list of all JTAG adapters connected to the host and the devices on each JTAG chain, in the form:

   ```
   ID - NAME (ADAPTER-SERIAL-NUMBER)
   ```

2. `xburn --id` *ID* `--target-file` *target*`.xn` *aes-image*`.otp`

   XBURN loads a program onto the device that writes the AES Module and security keys to the OTP, and sets its secure boot bits. XBURN returns 0 for success or non-zero for failure.

34XBURN Command-Line Manual

IN THIS CHAPTER

- Overall Options
- Target Options
- Security Options
- Security Register Options
- Programming Options

XBURN creates OTP images, and programs images into the OTP memory of xCORE devices.

34.1 Overall Options

The following options are used to specify the OTP image and security register contents.

`xe-file` Specifies bootable images to be constructed from the loadable segments from *xe-file* and a default set of security bits (see Figure 67).

`otp-file` Specifies the OTP segments from *otp-file* which includes the security register value.

`--version`
`-v` Displays the version number and copyrights.

`--help`
`-h` Prints a description of the supported command-line options.

34.2 Target Options

The following options are used to specify the target hardware platform.

`--list-devices`
`-l` Prints an enumerated list of all JTAG adapters connected to the host and the devices on each JTAG chain, in the form:

```
ID - NAME    (ADAPTER-SERIAL-NUMBER)
```

The adapters are ordered by their serial numbers.

`--id ID` Specifies the adapter connected to the target hardware.

`--adapter-id ADAPTER-SERIAL-NUMBER`
Specifies the serial number of the adapter connected to the target hardware.

`--jtag-speed n`
: Sets the divider for the JTAG clock to n. The corresponding JTAG clock speed is 6/(*n*+1)MHz. The default value is 0 (6MHz).

`--spi-div n`
: Sets the divider used in the AES Module for the SPI clock to *n*. The corresponding SPI clock speed is set to 100/(2n) MHz. The deafult value is 20 (2.5MHz).

  This option is only valid with `--lock`.

34.3 Security Options

`--genkey keyfile`
: Outputs to *keyfile* two 128-bit keys used for authentication and decrpytion. The keys are generated using the open-source library `crypto++`.

  This option is not valid with `--burn` or `--lock`.

`--lock keyfile`
: Specifies the XCORE AES boot module (see §33.1) and a default set of security bits (see Figure 67).

34.4 Security Register Options

The following options are used to specify the contents of the OTP security register, overriding the default options for burning XE images, OTP images and the AES module, as given in Figure 67.

Figure 67: Default security bits written by XBURN

Security Bit	XE Image	OTP Image	AES Module (`--lock`)
OTP Boot	Enabled	As per OTP image file	Enabled
JTAG Access	Enabled		Disabled
Plink Access	Enabled		Enabled
Global Debug	Enabled		Disabled
Master Lock	Disabled		Enabled
Secure Config Access	Enabled		Enabled

`--enable-otp-boot`
: Enables boot from OTP.

`--disable-jtag`
: Disables JTAG access. Once disabled, it is not possible to gain debug access to the device or to read the OTP.

  This option does not disable boundary scan.

`--disable-plink-access`
: Disables access to the plink registers from other tiles. Disabling plink access restricts all access of the registers of each plinks to the tile local to that plink.

`--disable-global-debug`
: Prevents the device from participating in global debug. Disabling global debug prevents the tiles from entering debug using the global debug pin.

`--enable-master-lock`
: Enables the OTP master lock. No further modification of the OTP is permitted.

`--disable-secure-config-access`
: Disables access to the security config register.

34.5 Programming Options

By default, XBURN writes the specified OTP images to the target platform.

`--force`
`-f`
: Do not prompt before writing the OTP. This is not default.

`--outfile`
`-o otp-file`
: Place output in *otp-file*, disabling programming.

`--make-exec xe-file`
: Place an executable in *xe-file* that when run on an xCORE device performs the specified OTP burning operation; disables programming.

  The XE file can be run later using XRUN.

`--target-file xn-file`
: Specifies *xn-file* as the target platform.

`--target platform`
: Specifies a target platform. The platform configuration must be specified in the file *platform*`.xn`, which is searched for in the paths specified by the `XCC_DEVICE_PATH` environment variable (see §10.8).

`--read`
: Prints the entire contents of the OTP.

`--size-limit n`
: Limits the amounts of OTP memory written to the first *n* bytes of the OTP. If the image doesn't fit within the specified limit an error will be given.

`--mac-address mac`
: Writes a MAC address to the end of the OTP. The MAC address should be specified in the form 12:34:56:78:9A:BC. Multiple MAC addresses can be written by specifying the `--mac-address` option multiple times. MAC addresses are written to the OTP in the order the options appear.

`--serial-number serial`
: Writes a 32-bit serial number to the end of the OTP.

`--no-compression`
: Prevents XBURN from compressing all binary data.

`--no-compression-validation`
: Prevents XBURN from validating the decompression algorithm within XSIM.

Part L

Programming xCORE-XA devices

CONTENTS

- xTIMEcomposer support for xCORE-XA devices and boards
- xCORE ARM bridge library
- xCORE ARM boot library

35 xTIMEcomposer support for xCORE-XA devices and boards

IN THIS CHAPTER

- 3rd-party ARM tool chain
- 3rd-party ARM libraries
- Additional software requirements
- Library support for communication and device boot
- Multi architecture project support
- xTIMEcomposer support
- Setting up the ARM GDB Server
- Running applications on hardware
- Debugging applications on hardware
- Using xTIMEcomposer launch groups
- Deploying applications into xCORE-XA flash memory

The xTIMEcomposer tools support application development for xCORE-XA devices and boards. Extensions have been made to xTIMEcomposer to support multi-architecture projects; the tools now include a full command line toolchain for supporting the ARM Cortex-M series. This and the additional support added to the build infrastructure for XMOS applications allows single project development of applications within the xTIMEcomposer design environment.

35.1 3rd-party ARM tool chain

To develop for the xCORE-XA series of devices a toolchain has been re-distributed with the xTIMEcomposer SDK, which allows for development of ARM based applications.

This tool chain is freely available, and support and development is provided from the following website:

https://launchpad.net/gcc-arm-embedded

Whilst this toolchain is included for convenience, there is no requirement to use xTIMEcomposer to develop for the ARM core contained within the xCORE-XA device if you have an existing toolchain which supports development of the ARM Cortex-M series of devices.

35.2 3rd-party ARM libraries

There are two libraries provided with the ARM toolchain in order to aid development of applications targeting the ARM Cortex-M3.

- **EMLIB**

The energy micro peripheral support library which is used to access the hardware peripherals of the ARM Cortex-M3 device and forms the base layer for software.

- **CMSIS**

The Cortex microcontroller software interface standard is a vendor-independent hardware abstraction layer for the Cortex-M processor series. The CMSIS enables consistent and simple software interfaces to the processor and the peripherals, simplifying software re-use, reducing the learning curve for microcontroller developers, and reducing the time to market for new devices.

35.3 Additional software requirements

The ARM support within xTIMEcomposer has been developed using the SEGGER J-Link series of development adapters. In order for the runtime and debugging tools provided to connect to hardware you need to install the SEGGER GDB J-Link support tools. This will provide the SEGGER developed GDB server functionality that allows connection to the xCORE-XA based hardware device from the development tools.

This additional software is available to download and install from:

http://www.segger.com/jlink-software.html

35.4 Library support for communication and device boot

To allow development of applications for xCORE-XA devices, a set of libraries is provided in xTIMEcomposer that provide the following functionality:

35.4.1 xCORE-ARM bridge library

This library provides the interface for the the ARM core and xCORE (inside the xCORE-XA device) to communicate. There are several modes in which this can operate and these are documented in more detail in the *xCORE ARM bridge library documentation* - §36.

35.4.2 xCORE-ARM boot library

This library provides the support required for booting the xCORE at runtime from the flash memory contained within the ARM core. This allows xCORE-XA products to boot from a single internal flash on the device. See the *xCORE ARM Boot library* for further information - §37.

35.5 Multi architecture project support

Building applications for the xCORE-XA device requires two tool chains to compile code for both the ARM core and the xCORE present in the xCORE-XA device. To simplify this process the XMOS build system has been extended to support projects of this type. This allows a single makefile to build the multiple executables required for using xCORE-XA devices.

35.5.1 Project structure

The project structure for an xCORE-XA based design differs slightly from a standard xTIMEcomposer project. There are three folders in the *src* directory of the project: *arm*, *xcore* and *common*. Any source code placed in the *arm* folder will be compiled and built using the ARM tool chain provided. Likewise any source code within the *xcore* directory will be built using the xCORE tool chain. Anything in the *common* directory can be used by both tool chains; this is useful for shared header files for example.

35.5.2 Makefile extensions

To mark a makefile as being part of an xCORE-XA device project the `XCORE_ARM_PROJECT` variable has been added, which is understood by the build system. When set to 1, the build system operates as described above in the project structure.

To pass compiler options to the tool chain below there are additional variables. Anything specified in the variable `XCC_FLAGS` is passed to the xCORE tool chain, and using the variable `ARM_GCC_FLAGS` passes options to the ARM tools.

35.5.3 Building projects

Once a project is in the form described above, the build system takes care of the generation of binaries for each of the architectures. These projects are built in the standard way using the XMAKE tool.

35.6 xTIMEcomposer support

xTIMEcomposer supports the xCORE-XA project structure, as described above. If you select an xCORE-XA target in the new project wizard, xTIMEcomposer automatically generates a project in the correct structure, and sets up the makefile with default settings required in the variable for an xCORE-XA project.

35.7 Setting up the ARM GDB Server

The xCORE-XA device has additional software requirements on the SEGGER GDB server in order to operate. This is not provided with the tools and must be downloaded from:

http://www.segger.com/jlink-software.html

Once downloaded and installed it should be started in the following way

On Windows:

On Windows there is a command line tool and a graphical version which provides more information about the device executing.

The graphical version can be started from your **Start** menu by running the J-Link GDB server application. The target interface required to be set to communicate with the xCORE-XA device should be set to SWD. Once this is done and your development board or JLINK debug adapter is connected to your PC, select OK to connect the GDB server to your target hardware.

Once a connection is established the J-Link connected and CPU status should be green and target voltage detected and displayed.

The command line version can be started from a Windows command prompt by locating the SEGGER J-Link GDB server command line application, which will be installed under the Program Files directory.

This should be started with the following command line:

```
JLinkGDBServerCL.exe -if SWD
```

This will then startup the command line gdb server for the ARM device. Correct operation will be displayed by printing out a target voltage and that it has successfully connected to the target.

On Linux:

For Linux users there you need to install the SEGGER J-Link support software for the appropriate linux version you have from the SEGGER site.

http://www.segger.com/jlink-software.html

After installation you need to set up the USB device rules for the the SEGGER J-Link in order to allow the tools to access the device. This may be done automatically depending on the Linux software package type you installed. This can be checked by looking in the directory /etc/udev/rules.d and checking for a rules file for SEGGER products. If this is not contained within this directory you can copy the rules file from the your JLINK gdb server installation into the above directory and restart the udev service. After unplugging and reconnecting the debug adapter from the host machine you should now be able to use the SEGGER J-Link.

Provided with the installation is a command line version of the SEGGER GDB server application. This can be executed with the following command line,

```
./JLinkGDBServer -if SWD
```

This will then startup the command line gdb server for the ARM device. Correct operation will be displayed by printing out a target voltage and that it has successfully connected to the target.

On OSX:

On OSX you need to install the SEGGER J-Link support software for the OSX version you have from the SEGGER download site.

http://www.segger.com/jlink-software.html

Once this is installed you will have access to the GDB server application to allow communication with the ARM target of the xCORE-XA.

Once installation is complete you can run the following command:

```
./JLinkGDBServer -if SWD
```

This will then startup the command line gdb server for the ARM device. Correct operation will be displayed by printing out a target voltage and that it has successfully connected to the target.

35.8 Running applications on hardware

xCORE-XA applications have two binaries associated with each project - the xCORE binary has an XE suffix and the ARM binary has no suffix. The binaries are loaded via the associated debug adapter for the ARM or the xCORE part of the device in order to execute code. Unless you boot from the on-chip flash of the xCORE-XA, the following extensions have been made to allow the XMOS tools to program the device.

35.8.1 Command line

The XRUN tool contains support for booting loading ARM code into an xCORE-XA device from the command line. To do this the command passes the ARM architecture binary and it will be loaded onto the device. This requires the ARM GDB server to be running on the host machine to allow *xrun* to interface to the hardware. When booting the ARM executable no other command line options are supported as they are with an xCORE application as described in the xrun user guide - §17.

The ARM GDB server must be started before the xTIMEcomposer tools can be connected to the hardware.

35.8.2 xTIMEcomposer Studio

There is support for running ARM applications inside xTIMEcomposer. A new architecture target has been added to the *Run Configuration* menu, which supports running on an ARM target. The use of *Run configurations* inside xTIMEcomposer is described in the *Use xTIMEcomposer to run an application* chapter - §16.

35.9 Debugging applications on hardware

You can also debug xCORE-XA applications using xTIMEcomposer, which includes a version of the GNU debugger (GDB) for ARM architectures as well as for the xCORE architecture. As with running an application the debugger for each of the associated architecture binaries must be started.

35.9.1 Command line

From the command line, you must launch a debugger for the ARM application code running on an xCORE-XA. The ARM gdb server needs to be running before attempting to connect a debugger. There is a GDB script in the tools installation which executes the setup commands required for debugging an xCORE-XA ARM application from the command line. See the following subdirectory in the tools installation:

```
arm_toolchain\arm-none-eabi\share\gdb\scripts\arm_gdb_break_at_main.gdb
```

Use the following command to set up a debug session to debug the ARM application (you need to specify the path to the .gdb script if you are not in the subdirectory):

```
arm-none-eabi-gdb -x arm_gdb_break_at_main.gdb arm_application_binary
```

This will start up a command line debugging session which will stop at the `main()` function in the ARM application.

The script can be copied and modified if you want to stop at another location within the application.

35.9.2 xTIMEcomposer Studio

There is an ARM architecture target within the xTIMEcomposer *Debug configurations*, which lets you debug your code within the xTIMEcomposer graphical debugger. Setting up this debug configuration is done in the standard way selecting the ARM target for the ARM binary generated by the build system. *Debug configurations* are described in the *Using xTIMEcomposer to debug applications* chapter - §24.

The ARM gdb server must be started for xTIMEcomposer to be able to connect to the hardware.

35.10 Using xTIMEcomposer launch groups

xTIMEcomposer Studio supports the launching of more that one *run/debug configuration* at the same time. This is useful for xCORE-XA development as you often need to launch both the ARM and the xCORE code to run a complete application.

Launch groups can be created from within a *Run Configuration* or *Debug configuration* by double clicking on the *Launch group* target type in the configuration list. This creates a new launch configuration to which you can add existing run or debug configurations. Configurations in a launch group are executed in sequence.

Once you have created a launch group and configured it with your run/debug configurations, you can select the launch group using **Run ▶ Run Configurations** or **Run ▶ Debug Configurations**.

35.11 Deploying applications into xCORE-XA flash memory

Application code for an xCORE-XA device can be deployed into the internal flash. This allows both the ARM application and the xCORE application to boot from the same flash device. To achieve this you need to modify the project makefile to allow xTIMEcomposer to combine the applications together into a single flash image.

The makefile variable `EMBED_XCORE_IMAGE_IN_ARM_IMAGE` is used to select if the build system will automatically generate a binary containing both the ARM and the xCORE application code.

With this the size of the flash partition required for the xCORE application to be embedded must be specified. This is done by adding the command:

```
-Wl,--defsym=XCORE_FLASH_SIZE=0x40000
```

to the ARM_GCC_FLAGS variable in the project makefile. The size is specified in hex bytes and this space will be reserved in the binary for embedding the xCORE image.

Once these options are specified in the project makefile, the build system will generate a third binary with the extension `-with-xcore-image` applied to the end of the project name. This image only needs to be programmed into the ARM part of the xCORE-XA device. The binary must be run on the ARM device to program the flash.

Once this is done, the application can boot the xCORE from the ARM flash providing the mode pins are set to *Boot xCORE tile from xCONNECT Link* (Boot xCORE tile from ARM overloads the boot from xCONNECT link).

Additional options have been added to the XFLASH tool to allow embedding of xCORE binaries into an ARM application. These are described in the *XFLASH user manual* - §32.

36 xCORE ARM bridge library

IN THIS CHAPTER

- xCORE ARM bridge
- Functional overview
- ARM API
- xCORE API

36.1 xCORE ARM bridge

The xCORE ARM bridge component allows transfer of data between the code running on the xCORE XA-series microcontroller and the attached ARM core. Note the this library *only* applies to the XA series of devices.

The component uses ports internal to the xCORE-XA device but does not use any external GPIO ports.

The xCORE-XA device has a separate core reserved for the bridge, so the xCORE ARM bridge component does not consume any of the advertised cores of the device.

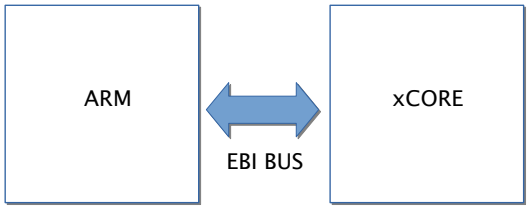
36.1.1 Features

- Event based API for within xCORE code
- C API for supporting ARM code
- Variable data transfer size
- Either side can initiate transfers
- ARM can transfer data via DMA
- Transfer speed up to 20MBit/s

36.2 Functional overview

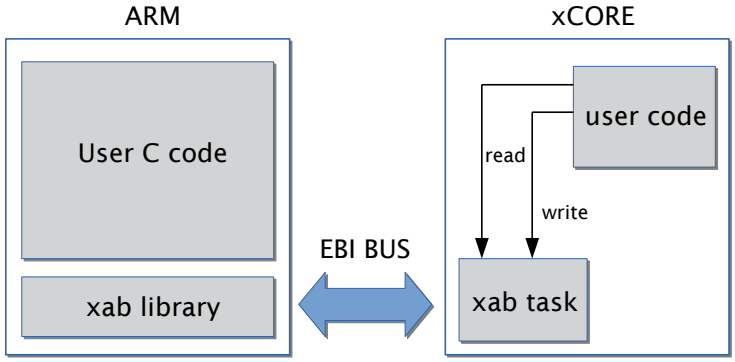
The XMOS XA-series devices consist of an ARM device and xCORE device in one package. They communicate via the EBI bus connected within the device:

Figure 68: xCORE + ARM + EBI



The xCORE ARM bridge library provides a software layer to control this communication. On the ARM, this is accessed via a C-api which manages the EBI hardware and interrupts on the device. It also has capacity to use the DMA unit of the device. On the xCORE, the bridge code runs in a separate task that uses a logical core. The application can connect to this task to send and receive data.

Figure 69: xCORE ARM bridge library architecture



The API for the ARM library can be found in §36.3 and the API for the xCORE library can found in §36.4.

This following snippet shows a minimal program using the library on the ARM that sends a single packet:

```
#include <XA/xab.h>

int main() {
  char buf[100];
  CMU_ClockEnable(cmuClock_GPIO, true);
  CMU_ClockEnable(cmuClock_EBI, true);
  xab_connect();
  for (i = 0; i < 100; i++)
    buf[i] = i;
  xab_write(buf, 100);
  xab_disconnect();
  return 0;
}
```

Here is the corresponding minimal program for the xCORE that receives a single packet:

```
#include <XA/xab.h>

void application(client xab_read_if xab_rx, client xab_write_if xab_tx) {
  char buf[100];
  int len;
  len = xab_rx.read(buf, 100);
}

xab_ports_t xab_ports = XAB_PORTS_INITIALIZER(XS1_CLKBLK_1);

int main() {
  interface xab_read_if i_xab_rx;
  interface xab_write_if i_xab_tx;
  par {
    xcore_arm_bridge(xab_ports, i_xab_rx, i_xab_tx, 1024);
    application(i_xab_rx, i_xab_tx);
  }
  return 0;
}
```

36.3 ARM API

All functions can be accessed via the `xab.h` header supplied with the tools:

```
#include <XA/xab.h>
```

The ARM side of the bridge needs to be started with an explicit call to xab_connect() at the start of your program:

```
int main() {
  CMU_ClockEnable(cmuClock_GPIO, true);
  CMU_ClockEnable(cmuClock_EBI, true);
  xab_connect();
  ...
```

Note that the GPIO clock and the EBI clock of the ARM need to be enabled for the XAB library functions to function correctly.

Function	**xab_connect**
Description	Connect to the xCORE ARM bridge. Connect to the xCORE via the xCORE ARM bridge. After connecting it will set up the EBI bus to communicate with the xCORE. Both the GPIO clock and the EBI clock must be enabled for this function to work.
Type	`void xab_connect(void)`

Function	**xab_disconnect**
Description	Disconnect from the xCORE ARM bridge. This function can be used to disconnect from the bridge prior to resetting the xCORE. During this function, the EBI bus will be disabled. A subsequent call to xab_connect() can be used to re-connect to the bridge.
Type	`void xab_disconnect(void)`

Function	**xab_write**
Description	Write a packet to the xCORE. This function attempts to send a packet of data to the xCORE. The function will block until buffer space becomes available on the xCORE.
Type	`void xab_write(uint8_t *data, size_t length)`
Parameters	`data` the array of data to transmit `length` the number of bytes to transmit

Function	**xab_read**
Description	Read a packet from the xCORE. This function reads a packet of data from the xCORE. It will block until data is available from the xCORE.
Type	`size_t xab_read(uint8_t *data, size_t length)`
Parameters	`data` the array to fill with the data from the xCORE `n` the maximum number of bytes to fill
Returns	The size of the packet read (if this is greater than the size of the array passed in then the data passed to the application will be truncated).

Function	**xab_get_status**
Description	Retun the current status of the connection. You can check the returned flags by perfoming a logical *and* operation (`&`) with the relevant flag. The flags set are: ▶ `XAB_DATA_READY` - set when data is available from the xCORE ▶ `XAB_TX_READY` - set when the connection has buffer space avaialbe to send a packet.
Type	`xab_status_t xab_get_status(void)`
Returns	A status word of flags that show the current status of the connection.

36.3.1 Interrupt usage

The xCORE can optionally trigger an interrupt on the ARM to indicate a change of status i.e. when buffer space becomes available to write to the xCORE or when data is available to transfer to the xCORE. This can enable your application (or OS) to handle other tasks and react to a change in state.

Function	**xab_enable_irq**
Description	Enable the XAB interrupt. This code will configure the interrupt to fire when the xCORE changes status. It should be called *after* xab_connect().
Type	`void xab_enable_irq(void)`

Function	**xab_disable_irq**
Description	Disable the XAB interrupt. This code will configure the interrupt to not fire when the xCORE changes status.
Type	`void xab_disable_irq(void)`

Function	**xab_irq_handler**

Description	The xCORE ARM bridge interrupt handler.
	This function implements the interrupt handler for the xCORE ARM bridge. If the ARM enables the XAB interrupt then yhis function then needs to be called as part of the GPIO_ODD_IRQHandler function i.e. within your application you will need to define:
	`void GPIO_ODD_IRQHandler(void) {`
	`  int is_xcore_interrupt = xab_irq_handler();`
	`  // any other interrupt handler code for you application goes`
	`  // here...`
	`}`
	The interrupt handler will deal with any communication between the ARM and the xCORE and will set the global status accordingly. The application can check the status using the xab_get_status() function.
	The GPIO_ODD_IRQHandler may be called due to an interrupt for a reason unrelated to the xCORE. In this case this function will return 0.

Type	`int xab_irq_handler()`

Continued on next page

Returns	Non-zero if the xCORE has signalled the interrupt

36.3.2 DMA usage

Function	**xab_configure_dma**
Description	Configure DMA channels for use by the xCORE ARM bridge. This function configures two DMA channels (one for read and one for write) to be used by the xCORE arm bridge. It should be called after the DMA unit is initialized.

Type	`void`
	`xab_configure_dma(unsigned int read_channel,`
	`  DMA_CB_TypeDef *read_callback,`
	`  DMA_ArbiterConfig_TypeDef read_arbRate,`
	`  unsigned int write_channel,`
	`  DMA_CB_TypeDef *write_callback,`
	`  DMA_ArbiterConfig_TypeDef write_arbRate)`

Parameters	`read_channel`
	the number of the DMA channel to be used for transferring data from the xCORE to the ARM
	
	`read_callback`
	the DMA callback structure describing the callback to make when a DMA transfer is complete. See the EnergyMicro DMA library documentation for details. Note that the structure pointed to must live for the duration of all DMA transfers.
	
	`read_arbRate`
	the arbitration rate of this channel. See the EnergyMicro DMA library documentation for details.
	
	`write_channel`
	the number of the DMA channel to be used for transferring data from the ARM to the xCORE
	
	`write_callback`
	the DMA callback structure describing the callback to make when a DMA transfer is complete. See the EnergyMicro DMA library documentation for details. Note that the structure pointed to must live for the duration of all DMA transfers.
	
	`write_arbRate`
	the arbitration rate of this channel. See the EnergyMicro DMA library documentation for details.

Function	**xab_init_dma_write**
Description	Start writing a packet to the xCORE via DMA. This function sends a packet of data to the xCORE via the ARM's DMA unit. The data transmission will occur asynchronously to the CPU. At the end of the transfer the relevant callback function registered by xab_configure_dma() will be called.
Type	`void xab_init_dma_write(uint8_t *buffer, size_t length)`
Parameters	`data` the array of data to transmit `length` the number of bytes to transmit

Function	**xab_init_dma_read**
Description	Start reading a packet from the xCORE via DMA. This function initialization reads a packet of data from the xCORE via the ARM's DMA unit. It will block until data is available from the xCORE. The data transmission will occur asynchronously to the CPU. At the end of the transfer the relevant callback function registered by xab_configure_dma() will be called.
Type	`size_t xab_init_dma_read(uint8_t *buffer, size_t length)`
Parameters	`data` the array to fill with the data from the xCORE `n` the maximum number of bytes to fill
Returns	The size of the packet read (if this is greater than the size of the array passed in then the data passed to the application will be truncated).

36.4 xCORE API

All functions can be accessed via the `xab.h` header supplied with the xTIMEcomposer tools:

```
#include <XA/xab.h>
```

To communicate between the xCORE and ARM, the connection needs to be present in the top level `par` construct of your program. It connects to your application over an two interface connections - one for reading and one for writing:

```
int main(){
  interface xab_read_if i_xab_rx;
  interface xab_write_if i_xab_tx;
  par {
    xcore_arm_bridge(xab_ports, i_xab_rx, i_xab_tx, 1024);
    application(i_xab);
    ...
```

You instantiate the bridge with the following function:

Function	**xcore_arm_bridge**
Description	The xCORE ARM bridge. This function creates an instance of the connection between the xCORE and ARM in XA-series devices.

Type	`void`
	`xcore_arm_bridge(xab_ports_t &xab_ports,`
	`  server interface xab_read_if i_xab_rx,`
	`  server interface xab_write_if i_xab_tx,`
	`  static const size_t max_transfer_size)`

Parameters	`xab_ports` The internal ports used by the bridge, you can declare these using the `XAB_PORTS_INITIALIZER` define as intializer. `i_xab_rx` The interface connection to the bridge used for receiving data. `i_xab_tx` The interface connection to the bridge used for transmitting data. `max_transfer_size` The maximum packet size that can be sent/received. This value can be at most (2^16 - 1) bytes.

Macro	XAB_PORTS_INITIALIZER
Description	

36.4.1 The interface APIs

<table>
<tr><td>Type</td><td>xab_read_if</td></tr>
<tr><td>Description</td><td>This interface allows your program read from the xCORE ARM bridge.</td></tr>
<tr><td>Functions</td><td>
<table>
<tr><td>Function</td><td>read</td></tr>
<tr><td>Description</td><td>Read a data packet from the ARM core.
This function reads a packet of data from the ARM core. It will block until there is data to read.</td></tr>
<tr><td>Type</td><td><pre>[[clears_notification, guarded]]
size_t read(uint8_t data[n], size_t n)</pre></td></tr>
<tr><td>Parameters</td><td><code>data</code> the array to fill with the data from the ARM core

<code>n</code> the maximum number of bytes to fill</td></tr>
<tr><td>Returns</td><td>The size of the packet read (if this is greater than the size of the array passed in then the data passed to the application will be truncated).</td></tr>
</table>
<table>
<tr><td>Function</td><td>data_ready</td></tr>
<tr><td>Description</td><td>Notification that there is some data waiting from the ARM. This notification function can be selected upon and will event when a data is ready to be received from the ARM.</td></tr>
<tr><td>Type</td><td><pre>[[notification]]
slave void data_ready(void)</pre></td></tr>
</table>
</td></tr>
</table>

Type	**xab_write_if**
Description	This interface allows your program write to the xCORE ARM bridge.
Functions	

Function	**write**
Description	Transfer a data packet to the ARM core. This function sends a packet of data to the ARM. It will block until there is buffer space in the bridge for the packet.
Type	`[[clears_notification, guarded]]` `void write(uint8_t data[n], size_t n)`
Parameters	`data` the array of data to transmit `n` the number of bytes to transmit

Function	**write_completed**
Description	Notification that a previous write has completed. This notification function can be selected upon and will event when a transmit to the ARM core is complete (in which case more data can be sent).
Type	`[[notification]]` `slave void write_completed(void)`

37xCORE ARM boot library

IN THIS CHAPTER

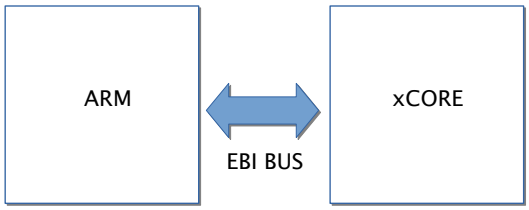
- Functional overview
- Booting an xCORE from the ARM core
- ARM API

The xCORE ARM boot library is used to boot the xCORE processor inside an xCORE-XA device from the ARM processor. The xCORE factory and upgrade images can be stored within the ARM core's internal flash memory rather than an external SPI device. Note the this library *only* applies to the xCORE-XA series of devices.

37.1 Functional overview

The XMOS xCORE-XA series devices consist of an ARM device and xCORE device in one package. They communicate via the EBI bus connected within the device:

Figure 70: xCORE + ARM + EBI



The API for the ARM library can be found in §37.3.

37.2 Booting an xCORE from the ARM core

This following snippet shows a minimal program using the library on the ARM that boots the attached xCORE device:

```
#include <XA/xaboot.h>

int main() {
  CMU_ClockEnable(cmuClock_GPIO, true);
  CMU_ClockEnable(cmuClock_EBI, true);
  xaboot_boot_xcore();
  ...
  return 0;
}
```

The ARM core must be running at its default clock speed of 13MHz or less. If the ARM core clock speed has been increased before the *xaboot_boot_xcore()* function is called, the xCORE-XA device may not boot correctly.

37.3 ARM API

All functions can be accessed via the `xaboot.h` header supplied with the xTIMEcomposer:

```
#include <XA/xaboot.h>
```

The ARM core boots the xCORE with an explicit call to xaboot_boot_xcore() at the start of your program:

```
int main() {
  CMU_ClockEnable(cmuClock_GPIO, true);
  CMU_ClockEnable(cmuClock_EBI, true);
  xaboot_boot_xcore();
  ...
```

The GPIO clock and the EBI clock of the ARM need to be enabled for the xaboot library functions to function correctly.

37.3.1 ARM API

Function	**xaboot_boot_xcore**
Description	Boot the attached xCORE. This function boots the xCORE device from the ARM flash
Type	`void xaboot_boot_xcore(void)`

Function	**xaboot_power_off_xcore**
Description	Power off the attached xCORE. This function turns off the power for the xCORE device

Type	`void`
	`xaboot_power_off_xcore(GPIO_Port_TypeDef port_1v0,`
	`  unsigned int pin_1v0,`
	`  unsigned int active_high_1v0,`
	`  GPIO_Port_TypeDef port_3v3,`
	`  unsigned int pin_3v3,`
	`  unsigned int active_high_3v3)`

Parameters	`port_1v0` the GPIO port used to control the xCORE 1v0 power supply `pin_1v0` the GPIO port pin using to control the xCORE 1v0 power supply `active_high_1v0` set to 0 if 1v0 is active low, 1 if it is active high `port_3v3` the GPIO port used to control the xCORE 3v3 power supply `pin_3v3` the GPIO port pin using to control the xCORE 3v3 power supply `active_high_3v3` set to 0 if 3v3 is active low, 1 if it is active high

Function	**xaboot_power_on_xcore**
Description	Power on the attached xCORE. This function turns on the power for the xCORE device

Type	`void`
	`xaboot_power_on_xcore(GPIO_Port_TypeDef port_1v0,`
	`  unsigned int pin_1v0,`
	`  unsigned int active_high_1v0,`
	`  GPIO_Port_TypeDef port_3v3,`
	`  unsigned int pin_3v3,`
	`  unsigned int active_high_3v3)`

Continued on next page

Parameters	`port_1v0`	the GPIO port used to control the xCORE 1v0 power supply
	`pin_1v0`	the GPIO port pin using to control the xCORE 1v0 power supply
	`active_high_1v0`	set to 0 if 1v0 is active low, 1 if it is active high
	`port_3v3`	the GPIO port used to control the xCORE 3v3 power supply
	`pin_3v3`	the GPIO port pin using to control the xCORE 3v3 power supply
	`active_high_3v3`	set to 0 if 3v3 is active low, 1 if it is active high

Part M

Programming in C/XC

CONTENTS

- Calling between C/C++ and XC
- XC Implementation-Defined Behavior
- C Implementation-Defined Behavior
- C and C++ Language Reference
- XC Library

38 Calling between C/C++ and XC

IN THIS CHAPTER

- Passing arguments from XC to C/C++
- Passing arguments from C/C++ to XC

In certain cases, it is possible to pass arguments of one type in XC to function parameters that have different types in C/C++, and vice versa.

To help simplify the task of declaring common functions between C/C++ and XC, the system header file `xccompat.h` contains various type definitions and macro defines. See the header file for documentation.

38.1 Passing arguments from XC to C/C++

A function defined in C/C++ with a parameter of type `unsigned int` can be declared in XC as taking a parameter of type `port`, `chanend` or `timer`.

A function defined in C/C++ with a parameter of type "pointer to `T`" can be declared in XC as taking a parameter of type "reference to `T`" or "nullable reference to `T`."

A function defined in C/C++ with a parameter of type "pointer to `T`" can be declared in XC as taking a parameter of type "array of `T`."

38.2 Passing arguments from C/C++ to XC

A function defined in XC with a parameter of type `port`, `chanend` or `timer` can be declared in C/C++ as taking a parameter of type `unsigned int`.

A function defined in XC with a parameter of type "reference to `T`" or "nullable reference to `T`" can be declared in C/C++ as taking a parameter of type "pointer to `T`."

A function defined in XC with a parameter of type "array of `T`" can be declared in C/C++ as taking a parameter of type "pointer to type `T`." In this case, the xCORE linker inserts code to add an implicit array bound parameter equal to the maximum value of the `unsigned int` type.

39XC Implementation-Defined Behavior

A conforming XC implementation is required to document its choice of behavior for all parts of the language specification that are designated implementation-defined. In the following section, all choices that depend on an externally determined application binary interface are listed as "determined by ABI," and are documented in the Application Binary Interface Specification (see §51).

- **The value of a multi-character constant (§1.5.2).**

  The value of a multi-character constant is the same as the value of its first character; all other characters are ignored.

- **Whether identical string literals are distinct (§1.6).**

  Identical string literals are not distinct; they are implemented in a single location in memory.

- **The available range of values that may be stored into a `char` and whether the value is signed (§3.2).**

  The size of `char` is 8 bits. Whether values stored in a `char` variable are signed or not is determined by the ABI.

- **The number of pins interfaced to a port and used for communicating with the environment; and the value of a port or clock not declared `extern` and not explicitly initialized (§3.2, §7.7).**

  The number of pins connected to a port for communicating with the environment is defined either by the explicit initializer for its declarator. If no initializer is provided, the compiler produces an error message.

- **The notional transfer type of a port, the notional counter type of a port, and the notional counter type of a timer (§3.2).**

  The notional types are determined by the ABI.

- **The value of an integer converted to a signed type such that its value cannot be represented in the new type (§5.2).**

  When any integer is converted to a signed type and its value cannot be represented in the new type, its value is truncated to the width of the new type and sign extended.

- **The handing of overflow, divide check, and other exceptions in expression evaluation (§6).**

  All conditions (divide by zero, mod zero and overflow of signed divide / mod) result in undefined behaviour.

- **The notion of alignment (§6.3.4).**

  An alignment of 2^n guarantees that the least significant n bits of the address in memory are 0. The specific alignment of the types is determined by the ABI.

- **The value and the type of the result of `sizeof` (§6.4.6).**

  The value of the result of the `sizeof` operator is determined by the ABI. The type of the result is `unsigned int`.

- **Attempted run-time division by zero (§6.6).**

  Attempted run-time division by zero results in a trap.

- **The extent to which suggestions made by using the `inline` function specifier are effective (§7.3).**

  The `inline` function specifier is taken as a hint to inline the function. The compiler tries to inline the function at all optimization levels above `-O0`.

- **The extent to which suggestions made by using the `register` storage class specifier are effective (§7.7.4).**

  The `register` storage class specifier causes the register allocator to try and place the variable in a register within the function. However, the allocator is not guaranteed to place it in a register.

- **The supported predicate functions for input operations (§8.3).**

  The set of supported predicate functions is documented in §50.8.

- **The meaning of inputs and outputs on ports (§8.3.2).**

  The inputs and outputs on ports map to in and out instructions on port resources, the behaviour of which is defined in the XS1 Ports Specification (see X1373).

- **The extent to which the underlying communication protocols are optimized for transaction communications (§8.9).**

  The communication protocols are determined by the ABI.

- **Whether a transaction is invalidated if a communication occurs such that the number of bytes output is unequal to the number of byte input, and the value communicated (§11).**

  This is determined by the ABI.

- **The behavior of an invalid operation (§12).**

  Except as described below, all invalid operations are either reported as compilation errors or cause a trap at run-time.

  - The behavior of an invalid master transaction statement is undefined; an invalid slave transaction always traps.
  - The `unsafe` pragma (see §9) can be used to disable specific safety checks, resulting in undefined behavior for invalid operations.

40 C Implementation-Defined Behavior

IN THIS CHAPTER

- Environment
- Identifiers
- Characters
- Floating point
- Hints
- Preprocessing directives
- Library functions
- Locale-Specific Behavior

A conforming C99 implementation is required to document its choice of behavior for all parts of the language specification that are designated *implementation-defined*. xTIMEcomposer implementation-defined behavior matches that of GCC 4.2.1[6] except for the choices listed below.

The following section headings refer to sections in the C99 specification (see §41.1) and all choices that depend on an externally determined application binary interface are listed as "determined by ABI," and are documented in the Application Binary Interface Specification (see §51).

Only the supported C99 features are documented.

40.1 Environment

- **The name and type of the function called at program startup in a freestanding environment (5.1.2.1).**

  A hosted environment is provided.

- **An alternative manner in which the `main` function may be defined (5.1.2.2.1).**

  There is no alternative manner in which `main` may be defined.

- **The values given to the strings pointed to by `argv` argument to `main` (5.1.2.2.1).**

  The value of `argc` is equal to zero. `argv[0]` is a null pointer. There are no other array members.

- **What constitutes an interactive device (5.1.2.3).**

  All streams are refer to interactive devices.

- **Signal values other than `SIGFPE`, `SIGILL`, and `SIGSEGV` that correspond to a computational exception (7.14.1.1).**

[6] http://www.xmos.com/references/gcc-4.2.1-c-implementation

No other signal values correspond to a computational exception.

- **Signal values for which is equivalent of `signal(sig, SIG_IGN);` is executed at program startup (7.14.1.1).**

  At program startup the equivalent of `signal(sig, SIG_DFL);` is executed for all signals.

- **The set of environment names and the method for altering the environment list used by the `getenv` function (7.20.4.5).**

  The set of environment names is empty. There is no method for altering the environment list used by the `getenv` function.

- **The manner of execution of the string by the `system` function used by the `getenv` function (7.20.4.6).**

  This is determined by the execution environment.

40.2 Identifiers

- **The number of significant initial characters in an identifier (5.2.4.1, 6.4.1).**

  All characters in identifiers (with or without external linkage) are significant.

40.3 Characters

- **The value of the members of the execution character set (5.2.1).**

  This is determined by the ASCII character set.

- **The unique value of the member of the execution character set produced for each of the standard alphabetic escape sequences (5.2.2).**

  This is determined by the ASCII character set.

- **The value of a `char` object into which has been stored any character other than a member of the basic execution set (6.2.5).**

  The value of any character other than a member of the basic execution set is determined by the ASCII character set.

- **The mapping of members of the source character set (in character constants and string literals) to members of the execution character set (6.4.4.4, 5.1.1.2).**

  The source character set is required to be the ASCII character set. Each character in the source character set is mapped to the same character in the execution character set.

- **The value of an integer character constant containing more than one character or containing a character or escape sequence that does not map to a single-byte execution character (6.4.4.4).**

  The value of an integer character constant containing more than one character is equal to the value of the last character in the character constant. The value of an integer character constant containing a character or escape sequence that does not map to a single-byte execution character is equal to the value reduced modulo 2^n to be within range of the `char` type, where n is the number of bits in a `char`.

- **The value of a wide character constant containing more than one multibyte character, or containing a multibyte character or escape multibyte character, or containing a multibyte character or escape sequence not represented in the extended execution character set (6.4.4.4).**

  Wide character constants must not contain multibyte characters.

- **The current locale used to convert a wide character constant consisting of a single multibyte character that maps to a member of the extended execution character set into a corresponding wide character code (6.4.4.4).**

  Wide character constants must not contain multibyte characters.

- **The value of a string literal containing a multibyte character or escape sequence not represented in the execution character set (6.4.5).**

  String literals must not contain multibyte characters. If an escape sequence not represented in the execution character set is used in a string literal, the value of the corresponding character in the string is the same as the value that would be given to an integer character constant consisting of that escape sequence.

40.4 Floating point

- **The accuracy of the floating-point operations and of the library functions in `<math.h>` and `<complex.h>` that return floating-point results (5.2.4.2.2).**

  This is intentionally left undocumented.

- **Additional floating-point exceptions, rounding modes, environments, and classifications, and their macros names (7.6, 7.12).**

  No additional floating-point exceptions, rounding modes, environments or classifications are defined.

40.5 Hints

- **The extent to which suggestions made by using the `register` storage-class specifier are effective (6.7.1).**

  The `register` specifier is ignored except when used as part of the register variable extension.

40.6 Preprocessing directives

- **The behavior on each recognized non-STDC `#pragma` directive (6.10.6).**

  This is documented in §9.

40.7 Library functions

- **Any library facilities available to a freestanding program, other than the minimal set required by clause 4 (5.1.2.1).**

  A hosted environment is provided.

- **The format of the diagnostic printed by the `assert` macro (7.2.1.1).**

  The `assert` macro uses the format "Assertion failed: *expression*, file *filename*, line *line number*, function: *function*." where *expression* is the text of the argument, *filename* is the value of `__FILE__`, *line number* is the value of `__LINE__` and *function* is the name of the current function. If the name of the current function cannot be determined, this part of the message is omitted.

- **The representation of the floating-point status flags stored by the `fegetexceptflag` function (7.6.2.2).**

  The function `fegetexceptflag` is not supported.

- **Whether the `feraiseexcept` function raises the "inexact" floating-point exception in addition to the "overflow" and "underflow" floating-point exception (7.6.2.3).**

  The function `feraiseexcept` is not supported.

- **Strings other than `"C"` and `""` that may be passed as the second argument to the `setlocale` function (7.11.1.1).**

  No other strings may be passed as the second argument to the `setlocale` function.

- **The types defined for `float_t` and `double_t` when the value of the `FLT_EVAL_METHOD` macro is less than 0 or greater than 2 (7.12).**

  No other values of the `FLT_EVAL_METHOD` macro are supported.

- **Domain errors for the mathematics functions, other than those required by this International Standard (7.12.1).**

  This is intentionally left undocumented.

- **The values returned by the mathematics functions on domain errors (7.12.1).**

  This is intentionally left undocumented.

- **The values returned by the mathematics functions on underflow range errors, whether `errno` is set to the value of the macro `ERANGE` when the integer expression `math_errhandling & MATH_ERRNO` is nonzero, and whether the "underflow" floating-point exception is raised when the integer expression `math_errhandling & MATH_ERREXCEPT` is nonzero (7.12.1).**

  This is intentionally left undocumented.

- **Whether a domain error occurs or zero is returned when an `fmod` function has a second argument of zero (7.12.10.1).**

  A domain error occurs when an `fmod` function has a second argument of zero.

- **The base-2 logarithm of the modulus used by the `remquo` functions in reducing the quotient (7.12.10.3).**

  The quotient is reduced modulo 2^7.

- **Whether the equivalent of `signal(sig, SIG_DFL);` is executed prior to the call of a signal handler, and, if not, the blocking of signals that is performed (7.14.1.1).**

  The equivalent of `signal(sig, SIG_DFL);` is executed prior to the call of a signal handler.

- **The null pointer constant to which the macro `NULL` expands (7.17).**

  `NULL` is defined as `((void *)0)`.

- **Whether the last line of a text stream requires a terminating new-line character (7.19.2).**

  This is determined by the execution environment.

- **Whether space characters that are written out to a text stream immediately before a newline character appear when read in (7.19.2).**

  This is determined by the execution environment.

- **The number of null characters that may be appended to data written to a binary stream (7.19.2).**

  This is determined by the execution environment.

- **Whether the file position indicator of an append-mode stream is initially positioned at the beginning or end of a file (7.19.3).**

  This is determined by the execution environment.

- **Whether a write on a text stream causes the associated file to be truncated beyond that point (7.19.3).**

  This is determined by the execution environment.

- **The characteristics of file buffering (7.19.3).**

  A buffered output stream saves characters until the buffer is full and then writes the characters as a block. A line buffered output stream saves characters until the line is complete or the buffer is full and then writes the characters as a block. An unbuffered output stream writes characters to the destination file immediately.

- **Whether a zero-length file actually exists (7.19.3).**

  This is determined by the execution environment.

- **The rules for composing valid file names (7.19.3).**

  This is determined by the execution environment.

- **Whether the same file can be simultaneously opened multiple times (7.19.3).**

  This is determined by the execution environment.

- **The nature and choice of encodings used for multibyte characters in files (7.19.3).**

  The execution character set must not contain multibyte characters.

- **The effect of the `remove` function on an open file (7.19.4.1).**

  This is determined by the execution environment.

- **The effect if a file with the new name exists prior to a call to the `rename` function (7.19.4.1).**

  This is determined by the execution environment.

- **Whether an open temporary file is removed upon abnormal program termination (7.19.4.3).**

  Temporary files are not removed on abnormal program termination.

- **Which changes of mode are permitted (if any), and under what circumstances (7.19.5.4).**

  The file cannot be given a more permissive access mode (for example, a mode of "w" will fail on a read-only file descriptor), but can change status such as append or binary mode. If modification is not possible, failure occurs.

- **The style used to print an infinity or *NaN*, and the meaning of any n-char or n-wchar sequence printed for a *NaN* (7.19.6.1, 7.24.2.1).**

  A `double` argument representing infinity is converted in the style `[-]inf`. A `double` argument representing a *NaN* is converted in the style as `nan`.

- **The output for `%p` conversion in the `fprintf` or `fwprintf` function (7.19.6.1, 7.24.2.1).**

  The value of the pointer is converted to unsigned hexadecimal notation in the style *dddd*; the letters `abcdef` are used for the conversion. The precision specifies the minimum number of digits to appear; if the value being converted can be represented in fewer digits, it is expanded with leading zeros. The default precision is 1. The characters `0x` are prepended to the output.

  The `fwprintf` function is unsupported.

- **The interpretation of a `-` character that is neither the first nor the last character, nor the second where a `^` character is the first, in the scanlist for `%[` conversion in the `fscanf` or `fwscanf` function (7.19.6.2, 7.24.2.1).**

  The `-` character is considered to define a range if the character following it is numerically greater than the character before it. Otherwise the `-` character itself is added to the scanset.

  The `fwscanf` function is unsupported.

- **The set of sequences matched by a `%p` conversion and the interpretation of the corresponding input item in the `fscanf` or `fwscanf` function (7.19.6.2, 7.24.2.2).**

  `%p` matches the same format as `%x`. The corresponding input item is converted to a pointer.

  The `fwscanf` function is unsupported.

- **The meaning of any n-char or n-wchar sequence in a string representing *NaN* that is converted by the `strtod`, `strtof`, `strtold`, `wcstod`, `wcstof` or `wcstold` function (7.20.1.3, 7.24.4.1.1).**

  The functions `wcstod`, `wcstof` and `wcstold` are not supported. A n-char sequence in a string representing *NaN* is scanned in hexadecimal form. Any characters which are not hexadecimal digits are ignored.

- **Whether or not the `strtod`, `strtof`, `strtold`, `wcstod`, `wcstof` or `wcstold` function sets `errno` to `ERANGE` when underflow occurs (7.20.1.3, 7.24.4.1.1).**

  The functions `wcstod`, `wcstof` and `wcstold` are not supported. The functions `strtod`, `strtof` and `strtold` do not set `errno` to `ERANGE` when and return 0 when underflow occurs.

- **Whether the `calloc`, `malloc`, and `realloc` functions return a null pointer or a pointer to an allocated object when the size requested is zero (7.20.3).**

  The functions `calloc`, `malloc` and `realloc` functions all return a pointer to an allocated object when the size requested is zero.

- **Whether open streams with unwritten buffered data are flushed, open streams are closed, or temporary files are removed when the `abort` or `_Exit` function is called (7.20.4.1, 7.20.4.3, 7.20.4.4).**

  When the `abort` function or `_Exit` function is called, temporary files are not removed, buffered files are not flushed and open streams are left open.

- **The termination status returned to the host environment by the `abort`, `exit` or `_Exit` function (7.20.3).**

  The function `abort` causes a software exception to be raised. The termination status returned to the host environment by the functions `exit` and `_Exit` is determined by the execution environment.

- **The value returned by the `system` function when its argument is not a null pointer (7.20.4.6).**

  This is determined by the execution environment.

- **The range and precision of times representable in `clock_t` and `time_t` (7.23.1).**

  The precision of times representable in `time_t` is defined by the execution environment. `time_t` designates an `unsigned long`. The actual range of times representable by `time_t` is defined by the execution environment.

  The macro `CLOCKS_PER_SEC` is defined as `1000`. `clock_t` designates an unsigned long.

- **The era for the `clock` function (7.23.2.1).**

  The `clock` function always returns the value `(clock_t)(-1)` to indicate that the processor time used is not available.

- **The replacement string for the `%Z` specifier to the `strftime` and `wcsftime` functions in the `"C"` locale (7.23.3.5, 7.24.5.1).**

  The `%Z` specifier is replaced with the string “GMT”.

40.8 Locale-Specific Behavior

- **Additional members of the source and execution character sets beyond the basic character set (5.2.1).**

  Both the source and execution character sets include all members of the ASCII character set.

- **The presence, meaning, and representation of additional multibyte characters in the execution character set beyond the basic character set (5.2.1.2).**

  The execution character set does not contain multibyte characters.

- **The shift states used for the encoding of multibyte characters (5.2.1.2).**

  The source and execution character sets does not contain multibyte characters.

- **The direction of writing of successive printing characters (5.2.2).**

  Characters are printed from left to right.

- **The decimal-point character (7.1.1).**

  The decimal-point character is '.'.

- **The set of printing characters (7.4, 7.25.2).**

  This is determined by the ASCII character set.

- **The set of control characters (7.4, 7.25.2).**

  This is determined by the ASCII character set.

- **The set of characters tested for by the `isalpha`, `isblank`, `islower`, `ispunct`, `isspace`, `isupper`, `iswalpha`, `iswblank`, `iswlower`, `iswpunct`, `iswspace`, or `iswupper` functions (7.4.1.2, 7.4.1.3, 7.4.1.7, 7.4.1.9, 7.4.1.10, 7.4.1.11, 7.25.2.1.2, 7.25.2.1.3, 7.25.2.1.7, 7.25.2.1.9, 7.25.2.1.10, 7.25.2.1.11).**

  The functions `isblank`, `iswalpha`, `iswblank`, `iswlower`, `iswpunct`, `iswspace` and `iswupper` are not supported.

  `islower` tests for the characters 'a' to 'z'. `isupper` tests for the characters 'A' to 'Z'. `isspace` tests for the characters ' ', '\f', '\n', '\r', '\t' and '\v'. `isalpha` tests for upper and lower case characters. `ispunct` tests for all printable characters except space and alphanumeric characters.

- **The native environment (7.11.1.1).**

  The native environment is the same as the minimal environment for C translation.

- **Additional subject sequences accepted by the numerical conversion functions (7.20.1, 7.24.4.1).**

  No additional subject sequences are accepted by the numerical conversion functions.

- **The collation sequence of the execution character set (7.21.4.3, 7.24.4.4.2).**

  The comparison carried out by the function `strcoll` is identical to the comparison carried out by the function `strcmp`.

- **The contents of the error message strings set up by the `strerror` function (7.21.4.3, 7.24.4.4.2).**

  The contents of the error message strings are given in Figure 71.

- **Character classifications that are supported by the `iswctype` function (7.25.1).**

  The character classifications supported by `iswctype` are given in Figure 72.

Figure 71: Error message strings

Value	String
EPERM	Not owner
ENOENT	No such file or directory
EINTR	Interrupted system call
EIO	I/O error
ENXIO	No such device or address
EBADF	Bad file number
EAGAIN	No more processes
ENOMEM	Not enough space
EACCES	Permission denied
EFAULT	Bad address
EBUSY	Device or resource busy
EEXIST	File exists
EXDEV	Cross-device link
ENODEV	No such device
ENOTDIR	Not a directory
EISDIR	Is a directory
EINVAL	Invalid argument
ENFILE	Too many open files in system
EMFILE	Too many open files
ETXTBSY	Text file busy
EFBIG	File too large
ENOSPC	No space left on device
ESPIPE	Illegal seek
EROFS	Read-only file system
EMLINK	Too many links
EPIPE	Broken pipe
EDOM	Math argument
ERANGE	Result too large
ENAMETOOLONG	File or path name too long
ENOSYS	Function not implemented
ENOTEMPTY	Directory not empty
ELOOP	Too many symbolic links

Figure 72: Wide character mappings

Value	Description
WCT_TOLOWER	Convert to lower case
WCT_TOUPPER	Convert to upper case

41 C and C++ Language Reference

IN THIS CHAPTER

- Standards
- Books
- Online

XMOS does not produce documentation for C and C++ standard language features as high quality documentation is readily available.

41.1 Standards

- ISO/IEC 9899:1989: Programming Languages — C. (C89). International Organization for Standardization.
- ISO/IEC 9899:1999: Programming Languages — C. (C99). International Organization for Standardization.
- ISO/IEC 14882:2011: Programming Languages — C++ (C++ Standard). International Organization for Standardization.

41.2 Books

- The C Programming Language (second edition), by Brian W. Kernighan and Dennis M. Ritchie, published by Prentice-Hall, Upper Saddle River, NJ, USA, 1988. ISBN-10: 0131103628

41.3 Online

- comp.lang.c Frequently Asked Questions: http://c-faq.com/

42 XC Library

IN THIS CHAPTER

- print.h functions
- safestring.h functions
- xccompat.h typedefs

The XC library provides a set of supporting functions and typedefs that make it easier to write code that can be called from C, C++ and XC.

42.1 print.h functions

The `print` library provides support for formatted output.

Function	**printchar**
Description	Prints a character.
Type	`int printchar(char value)`
Parameters	`value` The character to print.
Returns	The number of characters printed, or -1 on error.

Function	**printcharln**
Description	Prints a character followed by a new line.
Type	`int printcharln(char value)`
Parameters	`value` The character to print.
Returns	The number of characters printed, or -1 on error.

Function	printint
Description	Prints a value as a signed decimal.
Type	`int printint(int value)`
Parameters	`value` The value to print.
Returns	The number of characters printed, or -1 on error.

Function	printintln
Description	Prints a value as a signed decimal followed by a newline.
Type	`int printintln(int value)`
Parameters	`value` The value to print.
Returns	The number of characters printed, or -1 on error.

Function	printuint
Description	Prints a value as a unsigned decimal.
Type	`int printuint(unsigned value)`
Parameters	`value` The value to print.
Returns	The number of characters printed, or -1 on error.

Function	printuintln
Description	Prints a value as a unsigned decimal followed by a newline.
Type	`int printuintln(unsigned value)`
Parameters	`value` The value to print.
Returns	The number of characters printed, or -1 on error.

Function	printllong
Description	Prints a long long value as a signed decimal.
Type	`int printllong(long long value)`
Parameters	`value` The value to print.
Returns	The number of characters printed, or -1 on error.

Function	printllongln
Description	Prints a long long value as a signed decimal followed by a newline.
Type	`int printllongln(long long value)`
Parameters	`value` The value to print.
Returns	The number of characters printed, or -1 on error.

Function	printullong
Description	Prints a long long value as a unsigned decimal.
Type	`int printullong(unsigned long long value)`
Parameters	`value` The value to print.
Returns	The number of characters printed, or -1 on error.

Function	printullongln
Description	Prints a long long value as a unsigned decimal followed by a newline.
Type	`int printullongln(unsigned long long value)`
Parameters	`value` The value to print.

Continued on next page

Returns	The number of characters printed, or -1 on error.

Function	**printhex**
Description	Prints a value as a unsigned hexadecimal. The upper-case letters are used for the conversion.
Type	`int printhex(unsigned value)`
Parameters	`value` The value to print.
Returns	The number of characters printed, or -1 on error.

Function	**printhexln**
Description	Prints a value as a unsigned hexadecimal followed by a newline. The upper-case letters are used for the conversion.
Type	`int printhexln(unsigned value)`
Parameters	`value` The value to print.
Returns	The number of characters printed, or -1 on error.

Function	**printllonghex**
Description	Prints a long long value as a unsigned hexadecimal. The upper-case letters are used for the conversion.
Type	`int printllonghex(unsigned long long value)`
Parameters	`value` The value to print.
Returns	The number of characters printed, or -1 on error.

Function	printllonghexln
Description	Prints a long long value as a unsigned hexadecimal followed by a newline. The upper-case letters are used for the conversion.
Type	`int printllonghexln(unsigned long long value)`
Parameters	`value` The value to print.
Returns	The number of characters printed, or -1 on error.

Function	printstr
Description	Prints a null terminated string.
Type	`int printstr(const char(&alias s)[])`
Parameters	`s` The string to print.
Returns	The number of characters printed, or -1 on error.

Function	printstrln
Description	Prints a null terminated string followed by a newline.
Type	`int printstrln(const char(&alias s)[])`
Parameters	`s` The string to print.
Returns	The number of characters printed, or -1 on error.

42.2 safestring.h functions

The `safestring` library provides safe versions of the string functions found in `string.h` of the C standard library. All functions are callable from XC. When called from XC any attempt to perform an out of bounds array access will cause an exception to be raised.

Function	**safestrcpy**	
Description	Copies a string (including the terminating null character) to an array.	
Type	`void safestrcpy(char s1[], const char s2[])`	
Parameters	`s1`	The array to copy to.
	`s2`	The string to copy.

Function	**safestrncpy**	
Description	Copies no more than `n` characters of the string `s1` to the array `s2`. If the length of `s2` is less than `n` then null characters will be appended to the copied characters until `n` bytes are written.	
Type	`void safestrncpy(char s1[], const char s2[], unsigned n)`	
Parameters	`s1`	The array to copy to.
	`s2`	The string to copy.
	`n`	The number of characters to copy.

Function	**safestrcat**	
Description	Appends a copy of a string (including the terminating null character) to the end of another string.	
Type	`void safestrcat(char s1[], const char s2[])`	
Parameters	`s1`	The string to append to.
	`s2`	The string to append.

Function	safestrncat	
Description	Appends no more than `n` characters of the string `s2` to the string `s1`. The null characters at the end of `s1` is overwritten by the first character of `s2`. A terminating null character is always appended to the result.	
Type	`void safestrncat(char s1[], const char s2[], unsigned n)`	
Parameters	`s1`	The string to append to.
	`s2`	The string to append.
	`n`	The number of characters to append.

Function	safestrcmp	
Description	Compares two strings. If the strings are equal 0 is returned. If the strings are not equal a non-zero value is returned, the sign of which is determined by the sign of the difference between the first pair of characters which differ in the strings being compared.	
Type	`int safestrcmp(const char s1[], const char s2[])`	
Parameters	`s1`	The first string to compare.
	`s2`	The second string to compare.
Returns	A integer greater than, equal to, or less than 0, if `s1` is respectively greater than, equal to, or less than `s2`.	

Function	safestrncmp
Description	Compares up to the first `n` character of two strings. If the strings are equal up to the first `n` characters, 0 is returned. Otherwise a non-zero value is returned, the sign of which is determined by the sign of the difference between the first pair of characters which differ.
Type	`int safestrncmp(const char s1[], const char s2[], unsigned n)`

Continued on next page

Parameters	`s1`	The first string to compare.
	`s2`	The second string to compare.
	`n`	The maximum number of characters to compare.
Returns	A integer greater than, equal to, or less than 0, if `s1` is respectively greater than, equal to, or less than `s2`.	

Function	**safestrlen**
Description	Returns the length of a string. The length is given by the number of characters in the string not including the terminating null character.
Type	`int safestrlen(const char s[])`
Parameters	`s` The string.
Returns	The length of the string.

Function	**safestrchr**
Description	Returns the index of the first occurrence of `c` (converted to a `char`) in `s`. If `c` does not occur in `s`, `-1` is returned. The terminating null character is considered to be part of `s`.
Type	`int safestrchr(const char s[], int c)`
Parameters	`s` The string to scan. `c` The character to scan for.
Returns	The index of `c`.

Function	**safestrrchr**
Description	Returns the index of the last occurrence of `c` (converted to a `char`) in `s`, or `-1` if `c` does not occur in `s`. The terminating null character is considered to be part of `s`.

Continued on next page

Type	`int safestrrchr(const char s[], int c)`
Parameters	`s` The string to scan. `c` The character to scan for.
Returns	The index of `c`.

Function	**safestrspn**
Description	Returns the length of the longest initial segment of `s1` which consists entirely of characters from `s2`.
Type	`unsigned safestrspn(const char s1[], const char s2[])`
Parameters	`s1` The string to scan. `s2` The string containing characters to scan for.
Returns	The length of the initial segment.

Function	**safestrcspn**
Description	Returns the length of the longest initial segment of `s1` which consists entirely of characters not from `s2`.
Type	`unsigned safestrcspn(const char s1[], const char s2[])`
Parameters	`s1` The string to scan. `s2` The string containing characters to scan for.
Returns	The length of the initial segment.

Function	**safestrpbrk**
Description	Returns the index of the first occurrence in `s1` of any character in `s2`. If no character in `s2` occurs in `s1`, -1 is returned.
Type	`int safestrpbrk(const char s1[], const char s2[])`

Continued on next page

Parameters	`s1`	The string to scan.
	`s2`	The string containing characters to scan for.
Returns	The index of first matching character.	

Function	**safestrstr**
Description	Returns the index of the first occurrence of `s1` as a sequence of characters (excluding the terminating null character) in `s2`. If `s1` is not contained in `s2`, -1 is returned. If `s2` is a zero length string then 0 is returned.
Type	`int safestrstr(const char s1[], const char s2[])`
Parameters	`s1` The string to scan. `s2` The string to scan for.
Returns	The index of first matching subsequence.

Function	**safememcpy**
Description	Copies `length` bytes from the array `src` to the array `dst`.

Type	`void`
	`safememcpy(unsigned char dst[length],`
	`const unsigned char src[length],`
	`unsigned length)`

Parameters	`dst` The destination array. `src` The source array. `length` The number of bytes to copy.

Function	**safememmove**

Continued on next page

Description	Copies `length` bytes from offset `src` of array `data` to offset `dst` of array `data`. If the source and destination areas overlap then copying takes place as if the bytes are first copied from the source into a temporary array and then copied to the destination.
Type	`void safememmove(unsigned char data[], unsigned dst, unsigned src, unsigned length)`
Parameters	`data` The array to move data in. `dst` The offset of the destination area. `src` The offset of the source area. `length` The number of bytes to copy.

Function	**safememset**
Description	Copies `value` (converted to an `unsigned char`) to each of the first `length` bytes of the array `dst`.
Type	`void safememset(unsigned char dst[length], int value, unsigned length)`
Parameters	`dst` The destination array. `value` The value to copy. `length` The number of bytes to copy.

Function	**safememcmp**
Description	Compares the first `length` bytes of the arrays `s1` and `s2`. If there is no difference 0 is returned, otherwise a non-zero value is returned, the sign of which is determined by the sign of the difference between the first pair of bytes which differ.

Continued on next page

Type	`int safememcmp(const unsigned char s1[], const unsigned char s2[], unsigned length)`
Parameters	`s1` The first array. `s2` The second array. `length` The number of bytes to compare.
Returns	A integer greater than, equal to, or less than 0, if the first `length` bytes of `s1` are respectively greater than, equal to, or less than the first `length` bytes of `s2`.

Function	**safememchr**
Description	Returns the index of the first occurrence of `c` (converted to an `unsigned char`) in the first `length` bytes of `s`. If `c` does not occur in `s`, -1 is returned.
Type	`int safememchr(const unsigned char s[], int c, unsigned length)`
Parameters	`s` The array to scan. `c` The character to scan for. `length` The number of bytes to scan.
Returns	The index of `c`.

42.3 xccompat.h typedefs

The `xccommpat` header file provides type definitions that simplify the task of writing functions that may be called from both C/C++ and XC.

Type	**chanend**
Description	chanend typedef for use in C/C++ code. This typedef is only supplied if xccompat.h is included from C/C++ code. This enables a XC function prototyped as taking a parameter of type chanend to be called from C and vice versa.

Type	**timer**
Description	timer typedef for use in C/C++ code. This typedef is only supplied if xccompat.h is included from C/C++ code. This enables a XC function prototyped as taking a parameter of type timer to be called from C and vice versa.

Type	**port**
Description	port typedef for use in C/C++ code. This typedef is only supplied if xccompat.h is included from C/C++ code. This enables a XC function prototyped as taking a parameter of type port to be called from C and vice versa.

Part N

Programming in Assembly

CONTENTS

- Inline Assembly
- Make assembly programs compatible with the XMOS XS1 ABI
- Using the XTA With Assembly
- Assembly Programming Manual

43 Inline Assembly

The asm statement can be used to embed code written in assembly inside a C or XC function. For example, the add instruction can be written as follows:

```
asm("add %0, %1, %2" : "=r"(result) : "r"(a), "r"(b));
```

Colons separate the assembler template, the output operands and the input operands. Commas separate operands within a group. Each operand is described by an operand constraint string followed by an expression in parentheses. The "r" in the operand constraint string indicates that the operand must be located in a register. The "=" in the operand constraint string indicates that the operand is written.

Each output operand expression must be an lvalue and must have "=" in its constraint.

The location of an operand may be referred to in the assembler template using an escape sequence of the form %num where num is the operand number. The escape sequence "%=" can be used to emit a number that is unique to each expansion of an asm statement. This can be useful for making local labels. To produce a literal "%" you must write "%%".

If code overwrites specific registers this can be described by using a third colon after the input operands, followed by the names of the clobbered registers as a comma-separated list of strings. For example:

```
asm ("get r11, id\n\tmov %0, r11"
     : "=r"(result)
     : /* no inputs */
     : "r11");
```

The compiler ensures none of input or output operands are placed in clobbered registers.

If an asm statement has output operands, the compiler assumes the statement has no side effects apart from writing to the output operands. The compiler may remove the asm statement if the values written by the asm statement are unused. To mark an asm statement as having side effects add the volatile keyword after asm. For example:

```
asm volatile("in %0, res[%1]" : "=r"(result) : "r"(lock));
```

If the asm statement accesses memory, add "memory" to the list of clobber registers. For example:

```
asm volatile("stw %0, dp[0]"
     : /* no outputs */
     : "r"(value));
```

This prevents the compiler caching memory values in registers around the `asm` statement.

The *earlyclobber* constraint modifier "`&`" can be used to specify that an output operand is modified before all input operands are consumed. This prevents the compiler from placing the operand in the same register as any of the input operands. For example:

```
asm("or %0, %1, %2\n"
    "or %0, %0, %3\n"
    : "=&r"(result)
    : "r"(a), "r"(b), "r"(c));
```

Jumps from one `asm` statement to another are not supported. `asm` statements must not be used to modify the event enabled status of any resource.

An input operand can be tied to an output operand by specifying the number of the output operand in the input operand's constraint. For example:

```
::
```

asm("zext %0, 8n"
: "=r"(result) : "0"(x));

Operands that are tied together will be allocated to the same register.

44 Make assembly programs compatible with the XMOS XS1 ABI

IN THIS CHAPTER

- Symbols
- Alignment
- Sections
- Functions
- Elimination blocks
- Typestrings
- Example

The XMOS XS1 Application Binary Interface (ABI) defines the linking interface for objects compiled from C/C++, XC and assembly code. This tutorial explains how to write functions in assembly code that can be linked against objects generated by the XMOS compiler.

44.1 Symbols

As the assembler parses an assembly file, it maintains a current address which it increments every time it allocates storage.

Symbols are used to associate names to addresses. Symbols may be referenced in directives and instructions, and the linker patches the corresponding address onces its value is calculated.

The program below defines a symbol with name `f` that refers to the value of the current address. It also makes the symbol globally visible from other files, which can reference the symbol by its name.

```
# Give the symbol f the value of the current address.
f:
# Mark the symbol f as global.
  .globl f
```

The symbol is defined by writing its name followed by a colon. The `.globl` directive makes the symbol visible from outside of the file.

44.2 Alignment

The XS1 ABI specifies minimum alignment requirements for code and data. The start of a function must be 2-byte aligned, and data must be word-aligned. An address is aligned by placing the `.align` directive before the definition of a symbol.

The program below defines a symbol `f` that is defined to be the next 2-byte aligned address.

```
# Force 2 byte alignment of the next address.
  .align 2
f:
```

44.3 Sections

Each object file may contain multiple sections. When combined by the linker, sections with the same name in each object file are placed together at consecutive addresses. This allows different types of code or data to be grouped together in the final executable.

The XS1 ABI requires functions to be placed in the `.text` section, read-only data in the `.cp.rodata` section and writable data in the `.dp.data` section. The default section is the `.text` section, and the current section can be changed using one of the following directives.

Figure 73: Sections supported by the XMOS linker

Section	Used For	Directive
`.text`	Executable code	`.text`
`.dp.data`	Writable data	`.section .dp.data, "awd", @progbits`
`.cp.data`	Read only data	`.section .cp.rodata, "ac", @progbits`

44.3.1 Data

The example program below defines a 4-byte writeable object, initialized with the value 5, and aligned on a 4-byte boundary.

```
  .section .dp.data, "awd", @progbits
  .align 4
x:
  .word 5
```

You can use the following directives to emit different types of data.

Figure 74: Directives for emitting different types of data

Directive	Description
`.byte`	Emits a 1 byte value
`.short`	Emits a 2 byte value
`.word`	Emits a 4 byte value
`.space`	Emits an *n*-byte array of zero-initialized storage, where *n* is the argument to the directive
`.asciiz`	Emits a null terminated ASCII string
`.ascii`	Emits an ASCII string (no implicit terminating character)

44.3.2 Arrays

The program below defines a global array that is 42 bytes in size.

```
  .section .dp.data, "awd", @progbits
  .globl a
  .align 4
a:
  .space 42
  .globl a.globound
  .set a.globound, 42
```

The XS1 ABI requires that for each global array `f` there is a corresponding global symbol `f.globound` which is initialized with the number of elements of the first dimension of the array. You can use the `.set` directive to perform the initialization. Note that this value is used for array bounds checking if the variable is used by an XC function.

44.4 Functions

The XS1 ABI specifies rules for passing parameters and return values between functions. It also defines symbols for specifying the amount of hardware resources required by the function.

44.4.1 Parameters and return values

Scalar values of up to 32 bits are passed as 32 bit values. The first four parameters are passed in registers *r0*, *r1*, *r2* and *r3*, and any additional parameters are passed on the stack. Similarly, the first four return values are returned in the registers *r0*, *r1*, *r2* and *r3*, and any additional values are returned on the stack.

In the XC function prototype below, the parameters a and b are passed in registers *r0* and *r1*, as are the return values.

```
{int, int} swap(int a, int b);
```

An assembly implementation of this function is shown below.

```
  .globl swap
  .align 2
swap:
  mov r2, r0
  mov r0, r1
  mov r1, r2
  retsp 0
```

44.4.2 Caller and callee save registers

The XS1 ABI specifices that the registers *r0*, *r1*, *r2*, *r3* and *r11* are *caller-save*, and all other registers are *callee-save*.

Before a function is called, the contents of all caller-save registers whose values are required after the call must be saved. Upon returning from a function, the contents of all callee-save registers must be the same as on entry to the function.

The following example shows the prologue and epilogue for a function that uses the callee-save registers *r4*, *r5* and *r6*. The prologue copies the register values to the stack, and the epilogue restores the values from the stack back to the registers.

```
# Prologue
  entsp 4
  stw r4, sp[1]
  stw r5, sp[2]
  stw r6, sp[3]

# Main body of function goes here
# ...

# Epilogue
  ldw r4, sp[1]
  ldw r5, sp[2]
  ldw r6, sp[3]
  retsp 4
```

44.4.3 Resource usage

The linker attempts to calculate the amount of resources required by each function, including its memory requirements, and the number of threads, channel ends and timers it uses. This allows the linker to check that the resource usage of the final executable does not exceed that available on the target device.

For a function `f`, the resource usage symbols defined by the XS1 ABI are as follows.

Figure 75: Resource usage symbols defined by the XS1 ABI

Symbol	Description
`f.nstackwords`	Stack size (in words)
`f.maxthreads`	Maximum number of threads allocated, including the current thread
`f.maxchanends`	Maximum number of channel ends allocated
`f.maxtimers`	Maximum number of timer allocated

You can define resource usage symbols using the `.linkset` directive. If a function is global, you should also make the resource usage symbols global.

The example program below defines resource usage symbols for a function `f` that uses 4 words of stack, 2 threads, 0 timers and 2 channel ends.

```
.globl f
.globl f.nstackwords
.linkset f.nstackwords, 5
.globl f.maxthreads
.linkset f.maxthreads, 2
.globl f.maxtimers
.linkset f.maxtimers, 0
.globl f.maxchanends
.linkset f.maxchanends, 2
```

In more complex cases, you can use the maximum (`$M`) and addition (+) operators to build expressions for the resource usage that are evaluated by the linker. If two functions are called in sequence, you should compute the maximum for the two functions, and if called in parallel you should compute the sum for the two functions.

The example program below defines resource usage symbols for a function `f` that extends the stack by 10 words, allocates two timers and calls functions `g` and `h` in sequence before freeing the timer and returning.

```
.globl f
.globl f.nstackwords
.linkset f.nstackwords, 10 + (g.nstackwords $M h.nstackwords)
.globl f.maxthreads
.linkset f.maxthreads, 1 + ((g.maxthreads-1) $M (h.maxthreads-1))
.globl f.maxtimers
.linkset f.maxtimers, 2 + (g.maxtimers $M h.maxtimers)
.globl f.maxchanends
.linkset f.maxchanends, 0 + (g. maxchanends $M h.maxchanends)
```

You can omit the definition of a resource usage symbol if its value is unknown, for example if the function makes an indirect call through a function pointer. If the value of the symbol is required to satisfy a relocation in the program, however, the program will fail to link.

44.4.4 Side effects

The XC language requires that functions used as boolean guards in `select` statements have no side effects. It also specifies that functions called from within a transaction statement do not declare channels. By default, a function `f` is assumed to be side-effecting and to declare channels unless you explicitly set the following symbols to zero.

Figure 76: Symbols for denoting side-effects

Symbol	Description
`f.locnoside`	Specifies whether the function is side effecting
`f.locnochandec`	Specifies whether the function allocates a channel end

44.5 Elimination blocks

The linker can eliminate unused code and data. Code and data must be placed in elimination blocks for it to be a candidate for elimination. At final link time, if all of the symbols inside an elimination block are unreferenced, the block is removed from the final image.

The example program below declares a symbol within an elimination block.

```
  .cc_top f.function, f
f:
  .cc_bottom f.function
```

The first argument to the `.cc_top` directive and the `.cc_bottom` directive is the name of the elimination block. The `.cc_top` directive takes an additional argument, which is a symbol on which the elimination of the block is predicated on. If the symbol is referenced, the block is not eliminated.

Each elimination block must be given a name which is unique within the assembly file.

44.6 Typestrings

A typestring is a string used to describe the type of a function or variable. The encoding of type information into a typestring is specified by the XS1 ABI. The following directives are used to associate a typestring with a symbol.

Figure 77: Typestring directives

Binding	Directive
Global	`.globl name, "typestring"`
External	`.extern name, "typestring"`
Local	`.locl name, "typestring"`

When a symbol from one object file is matched with a symbol with the same name in another object, the linker checks whether the typestrings are compatible. If the typestrings are compatible linking continues as normal. If the typestrings are function types which differ only in the presence of array bound parameters the linker generates a thunk and replaces uses of the symbol with this thunk to account for the difference in arguments. The linker errors on all other typestring mismatches. This ensures that programs that are compiled from multiple files are as robust as those compiled from a single file.

If you fail to emit a typestring for a symbol, comparisons against this symbol are assumed to be compatible. If you are implementing a function which takes an array of unknown size, you should emit a typestring to allow it to be called from both C and XC. In other cases, typestrings can be omitted, but error checking is not performed.

44.7 Example

The program below prints the words “Hello world” to standard output.

```
const char str[] = "Hello world";

int main() {
  printf(str);
  return 0;
}
```

The assembly implementation below complies with the XS1 ABI.

```
  .extern printf, "f{si}(p(c:uc),va)"
  .section .cp.rodata, "ac", @progbits
  .globl str, "a(12:c:uc)"
  .cc_top str.data, str
  .align 4
str :
  .asciiz "Hello world"
.cc_bottom str.data
  .globl str.globound
  .set str.globound, 12

  .text
  .globl main, "f{si}(0)"
.cc_top main.function, main
  .align 2
main:
  entsp 1
  ldaw r11, cp[str]
  mov r0, r11
  bl printf
  ldc r0, 0
  retsp 0
.cc_bottom main.function
  .globl main.nstackwords
  .linkset main .nstackwords, 1 + printf.nstackwords
  .globl main.maxthreads
  .linkset main.maxthreads, printf.maxthreads
  .globl main.maxtimers
  .linkset main.maxtimers, 0 + printf.maxtimers
  .globl main.maxchanends
  .linkset main.maxchanends, 0 + printf.maxchanends
  .linkset main.locnochandec, 1
  .linkset main.locnoside, 1
```

By defining symbols for resource usage, the linker can check whether the program fits on a target device. By providing typestrings, the linker can check type compatibilty when different object files are linked. The linker can eliminate unused code and data since it is placed in elimination blocks.

45 Using the XTA With Assembly

IN THIS CHAPTER

- Assembly Directives
- Branch Table Example
- Core Start/Stop Example

When writing programs in assembly it is still possible to label code to make it portable using assembler directives.

45.1 Assembly Directives

The XMOS Timing Analyzer directives add timing metadata to ELF sections.

- `xtabranch` specifies a comma-separated list of locations that may be branched to from the current location.
- `xtacall` marks the current location as a function call with the specified label.
- `xtaendpoint` marks the current location as an endpoint with the specified label.
- `xtalabel` marks the current location using the specified label.
- `xtacorestart` specifies that a logical core may be initialized to start executing at the current location.
- `xtacorestop` specifies that a logical core executing the instruction at the current location will not execute any further instructions.

The `xtacall`, `xtaendpoint`, `xtalabel` directives are intended for use by the compiler only. They are used to link lines of source code with assembly instructions. All other XTA functionality provided by these directives (timing, exclusions) should be possible through the use of labels in the assembly code.

Strings used by the XTA for `xtacall`, `xtaendpoint` and `xtalabel` must not contain spaces.

45.2 Branch Table Example

If a branch table is written in assembly, branch target information must be added for the XTA to be able to analyze the assembly properly . This information is given in the form of a `.xtabranch` directive. For example, consider the code in Figure 78.

```
. type f, @function
. globl f
f:
    entsp 1
. xtabranch Ltarget1 , Ltarget2 , Ltarget3
    bru r0
Ltarget1 :
    bl taskA
    retsp 1
Ltarget2 :
    bl taskB
    retsp 1
Ltarget3 :
    bl taskC
    retsp 1
```

Figure 78: Setting branch targets

The XTA is not able to determine where the `bru` instruction will branch to because it is branching off a register value which is an argument to `main`. With the directive the XTA can consider the `bru` instruction to have the three targets (`Ltarget1`, `Ltarget2`, `Ltarget3`) and the XTA can successfully time the function.

45.3 Core Start/Stop Example

By default the XTA, assumes that the initial logical core starts executing at the RAM base. However, if developers add another core in assembly, they also need to add `.xtacorestart` and `.xtacorestop` directives for the XTA to know that the code is reachable. For example, consider the code in Figure 79.

```
. type main , @function
. globl main
main :
    getr r1 , XS1 \ _RES \ _TYPE \ _CORE
    ldap r11 , secondCore
    init t[r1 ]:pc , r11
    start t[r1]
    ldc r1 , 0
loop :
    bf r1 , loop
    retsp 0

secondCore :
. xtacorestart
    ldc r0 , 1
    tsetmr r1 , r0
. xtacorestop
    freet
```

Figure 79: Setting core start and stop points.

With the `xtacorestart` and `xtacorestop` directives the XTA knows that the code after the label `secondCore` is reachable and hence can be analyzed.

46 Assembly Programming Manual

IN THIS CHAPTER

- Lexical Conventions
- Sections and Relocations
- Symbols
- Labels
- Expressions
- Directives
- Instructions
- Assembly Program

The XMOS assembly language supports the formation of objects in the Executable and Linkable Format (ELF)[7] with DWARF 3[8] debugging information. Extensions to the ELF format are documented in the XMOS Application Binary Interface (see §51).

46.1 Lexical Conventions

There are six classes of tokens: symbol names, directives, constants, operators, instruction mnemonics and other separators. Blanks, tabs, formfeeds and comments are ignored except as they separate tokens.

46.1.1 Comments

The character # introduces a comment, which terminates with a newline. Comments do not occur within string literals.

46.1.2 Symbol Names

A symbol name begins with a letter or with one of the characters '.' or '_', followed by an optional sequence of letters, digits, periods, underscores and dollar signs. Upper and lower case letters are different.

46.1.3 Directives

A directive begins with '.' followed by one or more letters. Directives instruct the assembler to perform some action (see §46.6).

[7] http://www.xmos.com/references/elf
[8] http://www.xmos.com/references/dwarf3

46.1.4 Constants

A constant is either an integer number, a character constant or a string literal.

- A binary integer is `0b` or `0B` followed by zero or more of the digits `01`.
- An octal integer is `0` followed by zero or more of the digits `01234567`.
- A decimal integer is a non-zero digit followed by zero or more of the digits `0123456789`.
- A hexadecimal integer is `0x` or `0X` followed by one or more of the digits and letters `0123456789abcdefABCDEF`.
- A character constant is a sequence of characters surrounded by single quotes.
- A string literal is a sequence of characters surrounded by double quotes.

The C escape sequences may be used to specify certain characters.

46.2 Sections and Relocations

Named ELF sections are specified using directives (see §46.6.13). In addition, there is a unique unnamed "absolute" section and a unique unnamed "undefined" section. The notation {*secname* X} refers to an "offset X into section *secname*."

The values of symbols in the absolute section are unaffected by relocations. For example, address {absolute 0} is "relocated" to run-time address 0. The values of symbols in the undefined section are not set.

The assembler keeps track of the current section. Initially the current section is set to the text section. Directives can be used to change the current section. Assembly instructions and directives which allocate storage are emitted in the current section. For each section, the assembler maintains a location counter which holds the current offset in the section. The *active location counter* refers to the location counter for the current section.

46.3 Symbols

Each symbol has exactly one name; each name in an assembly program refers to exactly one symbol. A local symbol is any symbol beginning with the characters ".L". A local symbol may be discarded by the linker when no longer required for linking.

46.3.1 Attributes

Each symbol has a *value*, an associated section and a *binding*. A symbol is assigned a value using the `set` or `linkset` directives (see §46.6.15), or through its use in a label (see §46.4). The default binding of symbols in the undefined section is *global*; for all other symbols the default binding is *local*.

46.4 Labels

A label is a symbol name immediately followed by a colon (:). The symbol's value is set to the current value of the active location counter. The symbol's section is set to the current section. A symbol name must not appear in more than one label.

46.5 Expressions

An expression specifies an address or value. The result of an expression must be an absolute number or an offset into a particular section. An expression is a *constant expression* if all of its symbols are defined and it evaluates to a constant. An expression is a simple expression if it is one of a constant expression, a symbol, or a symbol ± a constant. An expression may be encoded in the ELF-extended expression section and its value evaluated by the linker (see §46.6.15); the encoding scheme is determined by the ABI. The syntax of an expression is:

expression ::= *unary-exp*
| *expression infix-op unary-exp*
| *unary-exp* `?` *unary-exp* `$:` *unary-exp*
| *function-exp*

unary-exp ::= *argument*
| *prefix-op unary-exp*

argument ::= *symbol*
| *constant*
| `(` *expression* `)`

function-exp ::= `$overlay_region_ptr (` *symbol* `)`
| `$overlay_index (` *symbol* `)`
| `$overlay_physical_addr (` *symbol* `)`
| `$overlay_virtual_addr (` *symbol* `)`
| `$overlay_num_bytes (` *symbol* `)`

infix-op ::= *one of*
`+ - < > <= >= || << >> * $M $A & /`

prefix-op ::= *one of*
`- ~ $D`

Symbols are evaluated to {*section x*} where *section* is one of a named section, the absolute section or the undefined section, and *x* is a signed 2's complement 32-bit integer.

Infix operators have the same precedence and behavior as C, and operators with equal precedence are performed left to right. In addition, the `$M` operator has lowest precedence, and the `$A` operator has the highest precedence.

For the + and - operators, the set of valid operations and results is given in Figure 80. For the `$D` operator, the argument must be a symbol; the result is 1 if the symbol is defined and 0 otherwise.

Figure 80: Valid operations for + and - operators

Op	Left Operand	Right Operand	Result
+	{*section* x}	{absolute y}	{*section* x+y}
+	{absolute x}	{*section* y}	{*section* x+y}
+	{absolute x}	{absolute y}	{absolute x+y}
-	{*section* x}	{*section* y}	{absolute x-y}
-	{*section* x}	{absolute y}	{*section* x-y}
-	{absolute x}	{absolute y}	{absolute x-y}

The ? operator is used to select between symbols: if the first operand is non-zero then the result is the second operand, otherwise the result is the third operand.

The operators `$overlay_region_ptr`, `$overlay_index`, `$overlay_physical_addr`, `$overlay_virtual_addr` and `$overlay_num_bytes` can be used to query properties of the overlay containing the overlay roots with the specified overlay key symbol (see §46.6.21). The set of results of these operators is given in Figure 81.

Figure 81: Operators for querying properties of overlays.

Operator	Result
$overlay_region_ptr	Virtual address of the overlay region containing the overlay.
$overlay_index	Index of the overlay in the overlay region.
$overlay_physical_addr	Physical address of the overlay.
$overlay_virtual_addr	Virtual (runtime) address of the overlay.
$overlay_num_bytes	Size of the overlay in bytes.

For all other operators, both arguments must be absolute and the result is absolute. The `$M` operator returns the maximum of the two operands and the `$A` operator returns the value of the first operand aligned to the second.

Wherever an absolute expression is required, if omitted then {absolute 0} is assumed.

46.6 Directives

Directives instruct the assembler to perform some action. The supported directives are given in this section.

46.6.1 add_to_set

The `add_to_set` directive adds an expression to a set of expressions associated with a key symbol. Its syntax is:

add-to-set-directive ::= `.add_to_set` *symbol* , *expression*
| `.add_to_set` *symbol* , *expression* , *symbol*

An optional predicate symbol may be specified as the 3rd argument. If this argument is specified the expression will only be added to the set if the predicate symbol is not eliminated from the linked object.

46.6.2 max_reduce, sum_reduce

The `max_reduce` directive computes the maximum of the values of the expressions in a set. The `sum_reduce` directive computes the sum of the values of the expressions in a set.

max-reduce-directive ::= `.max_reduce` *symbol* , *symbol* , *expression*

sum-reduce-directive ::= `.sum_reduce` *symbol* , *symbol* , *expression*

The first symbol is defined using the value computed by the directive. The second symbol is the key symbol identifying the set of expressions (see §46.6.1). The expression specifies the initial value for the reduction operation.

46.6.3 align

The `align` directive pads the active location counter section to the specified storage boundary. Its syntax is:

align-directive ::= `.align` *expression*

The expression must be a constant expression; its value must be a power of 2. This value specifies the alignment required in bytes.

46.6.4 ascii, asciiz

The `ascii` directive assembles each string into consecutive addresses. The `asciiz` directive is the same, except that each string is followed by a null byte.

ascii-directive ::= `.ascii` *string-list*
| `.asciiz` *string-list*

string-list ::= *string-list* , *string*
| `.asciiz` *string*

46.6.5 byte, short, int, long, word

These directives emit, for each expression, a number that at run-time is the value of that expression. The byte order is determined by the endianness of the target architecture. The size of numbers emitted with the word directive is determined by the size of the natural word on the target architecture. The size of the numbers emitted using the other directives are determined by the sizes of corresponding types in the ABI.

value-directive ::= *value-size exp-list*

value-size ::= `.byte`
| `.short`
| `.int`
| `.long`
| `.word`

exp-list ::= *exp-list* , *expression*
| *expression*

46.6.6 file

The `file` directive has two forms.

file-directive ::= `.file` *string*
| `.file` *constant string*

When used with one argument, the `file` directive creates an ELF symbol table entry with type `STT_FILE` and the specified string value. This entry is guaranteed to be the first entry in the symbol table.

When used with two arguments the `file` directive adds an entry to the DWARF 3 `.debug_line` file names table. The first argument is a unique positive integer to use as the index of the entry in the table. The second argument is the name of the file.

46.6.7 loc

The `.loc` directive adds a row to the DWARF 3 `.debug_line` line number matrix.

loc-directive ::= *constant constant constant*$_{\text{opt}}$
| *constant constant constant* ⟨*loc-option*⟩$*$

loc-option ::= `basic_block`
| `prologue_end`
| `epilogue_begin`
| `is_stmt` *constant*
| `isa` *constant*

The address register is set to active location counter. The first two arguments set the file and line registers respectively. The optional third argument sets the column register. Additional arguments set further registers in the `.debug_line` state machine.

`basic_block`
: Sets *basic_block* to true.

`prologue_end`
: Sets *prologue_end* to true.

`epilogue_begin`
: Sets *epilogue_begin* to true.

`is_stmt`
: Sets *is_stmt* to the specified value, which must be 0 or 1.

`isa`
: Sets *isa* to the specified value.

46.6.8 weak

The `weak` directive sets the weak attribute on the specified symbol.

weak-directive ::= `.weak` *symbol*

46.6.9 globl, global, extern, locl, local

The `globl` directive makes the specified symbols visible to other objects during linking. The `extern` directive specifies that the symbol is defined in another object. The `locl` directive specifies a symbol has local binding.

visibility ::= `.globl`
| `.extern`
| `.locl`
| `.global`
| `.extern`
| `.local`

vis-directive ::= *visibility symbol*
| *visibility symbol* , *string*

If the optional string is provided, an `SHT_TYPEINFO` entry is created in the ELF-extended type section which contains the symbol and an index into the string table whose entry contains the specified string. (If the string does not already exist in the string table, it is inserted.) The meaning of this string is determined by the ABI.

The `global` and `local` directives are synonyms for the `globl` and `locl` directives. They are provided for compatibility with other assemblers.

46.6.10 globalresource

globalresource-directive ::= `.globalresource` *expression* , *string*
| `.globalresource` *expression* , *string* , *string*

The `globalresource` directive causes the assembler to add information to the binary to indicate that there was a global port or clock declaration. The first argument is the resource ID of the port. The second argument is the name of the variable. The optional third argument is the tile the port was declared on. For example:

```
.globalresource 0x10200, p, tile[0]
```

specifies that the port `p` was declared on `tile[0]` and initialized with the resource ID `0x10200`.

46.6.11 typestring

The `typestring` adds an `SHT_TYPEINFO` entry in the ELF-extended type section which contains the symbol and an index into the string table whose entry contains the specified string. (If the string does not already exist in the string table, it is inserted.) The meaning of this string is determined by the ABI.

typestring-directive ::= `.typestring` *symbol* , *string*

46.6.12 ident, core, corerev

Each of these directives creates an ELF note section named "`.xmos_note`."

info-directive ::= `.ident` *string*
| `.core` *string*
| `.corerev` *string*

The contents of this section is a (name, type, value) triplet: the name is `xmos`; the type is either `IDENT`, `CORE` or `COREREV`; and the value is the specified string.

46.6.13 section, pushsection, popsection

The section directives change the current ELF section (see §46.2).

section-directive ::= *sec-or-push name*
| *sec-or-push name* , *flags sec-type*$_{opt}$
| `.popsection`

sec-or-push ::= `.section`
| `.pushsection`

flags ::= *string*

sec-type ::= *type*
| *type* , *flag-args*

type ::= `@progbits`
| `@nobits`

flag-args ::= *string*

The code following a `section` or `pushsection` directive is assembled and appended to the named section. The optional flags may contain any combination of the following characters.

`a`	section is allocatable
`c`	section is placed in the global constant pool
`d`	section is placed in the global data region
`w`	section is writable
`x`	section is executable
`M`	section is mergeable
`S`	section contains zero terminated strings

The optional type argument `progbits` specifies that the section contains data; `nobits` specifies that it does not.

If the `M` symbol is specified as a flag, a type argument must be specified and an integer must be provided as a flag-specific argument. The flag-specific argument represents the entity size of data entries in the section. For example:

```
.section .cp.const4, "M", @progbits, 4
```

Sections with the `M` flag but not S flag must contain fixed-size constants, each *flag-args* bytes long. Sections with both the `M` and `S` flags must contain zero-terminated strings, each character *flag-args* bytes long. The linker may remove duplicates within sections with the same name, entity size and flags.

Each section with the same name must have the same type and flags. The `section` directive replaces the current section with the named section. The `pushsection` directive pushes the current section onto the top of a *section stack* and then replaces the current section with the named section. The `popsection` directive replaces the current section with the section on top of the section stack and then pops this section from the stack.

46.6.14 text

The `text` directive changes the current ELF section to the `.text` section. The section type and attributes are determined by the ABI.

text-directive ::= `.text`

46.6.15 set, linkset

A symbol is assigned a value using the `set` or `linkset` directive.

set-directive ::= *set-type symbol* , *expression*

set-type ::= `.set`
| `.linkset`

The `set` directive defines the named symbol with the value of the expression. The expression must be either a constant or a symbol: if the expression is a constant, the symbol is defined in the absolute section; if the expression is a symbol, the defined symbol inherits its section information and other attributes from this symbol.

The `linkset` directive is the same, except that the expression is not evaluated; instead one or more `SHT_EXPR` entries are created in the ELF-extended expression section which together form a tree representation of the expression.

Any symbol used in the assembly code may be a target of an `SHT_EXPR` entry, in which case its value is computed by the linker by evaluating the expression once values for all other symbols in the expression are known. This may happen at any incremental link stage; once the value is known, it is assigned to the symbol as with set and the expression entry is eliminated from the linked object.

46.6.16 cc_top, cc_bottom

The `cc_top` and `cc_bottom` directives are used to mark the beginning and end of elimination blocks.

cc-top-directive ::= `.cc_top` *name* , *predicate*
| `.cc_top` *name*

cc-directive ::= *cc-top-directive*
| `.cc_bottom` *name*

name ::= *symbol*

predicate ::= *symbol*

`cc_top` and `cc_bottom` directives with the same name refer to the same elimination block. An elimination block must have precisely one `cc_top` directive and one `cc_bottom` directive. The top and bottom of an elimination block must be in the same section. The elimination block consists of the data and labels in this section between the `cc_top` and `cc_bottom` directives. Elimination blocks must be disjoint; it is illegal for elimination blocks to overlap.

An elimination block is retained in final executable if one of the following is true:

- A label inside the elimination block is referenced from a location outside an elimination block.
- A label inside the elimination block is referenced from an elimination block which is not eliminated
- The predicate symbol is defined outside an elimination block or is contained in an elimination block which is not eliminated.

If none of these conditions are true the elimination block is removed from the final executable.

46.6.17 scheduling

The `scheduling` directive enables or disables instruction scheduling. When scheduling is enabled, the assembler may reorder instructions to minimize the number of FNOPs. The default scheduling mode is determined by the command-line option `-fschedule` (see §10.4).

scheduling-directive ::= `.scheduling` *scheduling-mode*

scheduling-mode ::= `on`
| `off`
| `default`

46.6.18 issue_mode

The `issue_mode` directive changes the current issue mode assumed by the assembler. See §46.7 for details of how the issue mode affects how instructions are assembled.

issue-mode-directive ::= `issue_mode` *issue-mode*

issue-mode ::= `single`
| `dual`

46.6.19 syntax

The `syntax` directive changes the current syntax mode. See §46.7 for details of how assembly instructions are specified in each mode.

syntax-directive ::= `.syntax` *syntax*

syntax ::= `default`
| `architectural`

46.6.20 assert

assert-directive ::= `.assert` *constant* , *symbol* , *string*

The `assert` directive requires an assertion to be tested prior to generating an executable object: the assertion fails if the symbol has a non-zero value. If the constant is 0, a failure should be reported as a warning; if the constant is 1, a failure should be reported as an error. The string is a message for an assembler or linker to emit on failure.

46.6.21 Overlay Directives

The overlay directives control how code and data is partitioned into overlays that are loaded on demand at runtime.

overlay-directive ::= `.overlay_reference` *symbol* , *symbol*
| `.overlay_root` *symbol* , *symbol*
| `.overlay_root` *symbol*
| `.overlay_subgraph_conflict` *sym-list*

sym-list ::= *sym-list* , *symbol*
| *symbol*

- The `overlay_root` directive specifies that the first symbol should be treated as an overlay root. The optional second symbols specifies a overlay key symbol. If no overlay key symbol is explictly specified the overlay root symbol is used as the key symbol. Specifying the same overlay key symbol for multiple overlay roots forces the overlay roots into the same overlay.

- The `overlay_reference` directive specifies that linker should assume that there is a reference from the first symbol to the second symbol when it partitions the program into overlays.
- The `overlay_subgraph_conflict` directive specifies that linker should not place any code or data reachable from one the symbols into an overlay that is mapped an overlay region that contains another overlay containing code or data reachable from one of the other symbols.

46.6.22 Language Directives

The language directives create entries in the ELF-extended expression section; the encoding is determined by the ABI.

xc-directive ::= *globdir symbol* , *string*
| *globdir symbol* , *symbol* , *range-args* , *string*
| `.globpassesref` *symbol* , *symbol* , *string*
| `.call` *symbol* , *symbol*
| `.par` *symbol* , *symbol* , *string*

range-args ::= *expression* , *expression*

globdir ::= `.globread`
| `.globwrite`
| `.parwrite`
| `.globpassesref`

For each directive, the string is an error message for the assembler or linker to display on encountering an error attributed to the directive.

`call`
: Both symbols must have function type. This directive sets the property that the first function may make a call to the second function.

`par`
: Both symbols must have function type. This directive sets the property that the first function is invoked in parallel with the second function.

`globread`
: The first symbol must have function type and the second directive must have object type. This directive sets the property that the function may read the object. When a range is specified, the first expression is the offset from the start of the variable in bytes of the address which is read and the second expression is the size of the read in bytes.

`globwrite`
: The first symbol must have function type and the second directive must have object type. This directive sets the property that the function may write the object. When a range is specified, the first expression is the offset from the start of the variable in bytes of the address which is written and the second expression is the size of the write in bytes.

`parwrite`
: The first symbol must have function type and the second directive must have object type. This directive set the property that the function is called in an expression which writes to the object where the order of evalulation of the write and the function call is undefined. When a range is specified, the first expression is the offset from the start of the variable in bytes of the address which is written and the second expression is the size of the write in bytes.

`globpassesref`
: The first symbol must have function type and the second directive must have object type. This directive sets the property that the object may be passed by reference to the function.

46.6.23 XMOS Timing Analyzer Directives

The XMOS Timing Analyzer directives add timing metadata to ELF sections.

xta-directive ::= `.xtabranch` *exp-list*$_{opt}$
| `.xtaendpoint` *string* , *source-location*
| `.xtacall` *string* , *source-location*
| `.xtalabel` *string* , *source-location*
| `.xtathreadstart`
| `.xtathreadstop`
| `.xtaloop` *constant*
| *.xtacommand string* , *source-location*

source-location ::= *string* , *string* , *constant*

The first string of a source location is the compilation directory. The second string is the path to the file. The path may be specified as either a relative path from the compilation directory or as an absolute path. The third argument is the line number.

- `xtabranch` specifies a comma-separated list of locations that may be branched to from the current location.
- `xtaendpoint` marks the current location as an endpoint with the specified label.
- `xtacall` marks the current location as a function call with the specified label.
- `xtalabel` marks the current location using the specified label.
- `xtathreadstart` apecifies that a thread may be initialized to start executing at the current location.
- `xtathreadstop` specifies that a thread executing the instruction at the current location will not execute any further instructions.
- `xtaloop` specifies that the innermost loop containing the current location executes the specified number of times.

- `xtacommand` specifies an XTA command to be executed when analyzing the executable.

46.6.24 uleb128, sleb128

The following directives emit, for each expression in the comma-separated list of expressions, a value that encodes either an unsigned or signed DWARF little-endian base 128 number.

leb-directive ::= `.uleb128` *exp-list*
| `.sleb128` *exp-list*

46.6.25 space, skip

The `space` directive emits a sequence of bytes, specified by the first expression, each with the fill value specified by the second expression. Both expressions must be constant expressions.

space-or-skip ::= `.space`
| `.skip`

space-directive ::= *space-or-skip expression*
| *space-or-skip expression* , *expression*

The `skip` directive is a synonym for the space directive. It is provided for compatibility with other assemblers.

46.6.26 type

The `type` directive specifies the type of a symbol to be either a function symbol or an object symbol.

type-directive ::= `.type` *symbol* , *symbol-type*

symbol-type ::= `@function`
| `@object`

46.6.27 size

The `size` directive specifies the size associated with a symbol.

size-directive ::= `.size` *symbol* , *expression*

46.6.28 jmptable, jmptable32

The `jmptable` and `jmptable32` directives generate a table of unconditional branch instructions. The target of each branch instruction is the next label in the list. The size of the each branch instruction is 16 bits for the `jmptable` directive and 32 bits for the `jmptable32` directive.

jmptable-directive ::= `.jmptable` *jmptable-list*$_{\text{opt}}$
| `.jmptable32` *jmptable-list*$_{\text{opt}}$

jmptable-list ::= *symbol*
| *jmptable-list symbol*

Each symbol must be a label. A maximum of 32 labels maybe specified. If the unconditional branch distance does not fit into a 16-bit branch instruction, a branch is made to a trampoline at the end of the table, which performs the branch to the target label.

46.7 Instructions

Assembly instructions are normally inserted into an ELF text section. The syntax of an instruction is:

instruction ::= *mnemonic instruction-args*$_{\text{opt}}$

instruction-args ::= *instruction-args* , *instruction-arg*
| *instruction-arg*

instruction-arg ::= *symbol* [*expression*]
| *symbol* [*expression*] : *symbol*
| *expression*

To target the dual issue execution mode of xCORE-200 devices, instructions may be put in bundles:

separator ::= *newline*
| ;

instruction-bundle ::= { ⟨*separator*⟩* *bundle-contents* ⟨*separator*⟩* }

bundle-contents ::= *instruction* ⟨*separator*⟩+ *instruction*
| *instruction*

The current issue mode, as specifed by the issue_mode directive (see §46.6.18), affects how the assembler assembles instructions. Initially the current issue mode is single and instruction bundles cannot be used. If the current issue mode is changed to dual then:

- Instruction bundles can be specified.
- 16-bit instructions not in an instruction bundle are implicitly placed in an instruction bundle alongside a NOP instruction.
- The encoding of some operands may change. For example the assembler applies a different scaling factor to the immediate operand of relative branch instructions to match the different scaling factor that the processor uses at runtime when the instruction is executed in dual issue mode.

The order in which instructions are listed in an instruction bundle is not significant. The assembler may reorder the instructions in the bundle to satisfy architectural constraints.

The assembly instructions are summarized below using the default assembly syntax. The architecture manual (see X7879) documents the architectural syntax of the instructions. The syntax directive is used to switch the syntax mode.

The following notation is used:

`bitp`	one of: 1, 2, 3, 4, 5, 6, 7, 8, 16, 24 and 32
`b`	register used as a base address
`c`	register used as a conditional operand
`d, e`	register used as a destination operand
`i`	register used as a index operand
`r`	register used as a resource identifier
`s`	register used as a source operand
`t`	register used as a thread identifier
u_s	small unsigned constant in the range 0...11
u_x	unsigned constant in the range 0...(2^x-1)
`v, w, x, y`	registers used for two or more source operands

A register is one of: `r0`, `r1`, `r2`, `r3`, `r4`, `r5`, `r6`, `r7`, `r8`, `r9`, `r10`, `r11`, `sp`, `dp`, `cp` and `lr`. The instruction determines which of these registers are permitted.

Where there is choice of instruction formats, the assembler chooses the format with the smallest size. To force a specific format, specify a mnemonic of the form `INSTRUCTION_format` where the instruction and format names are as described in the architecture manual. For example the `LDWCP_ru6` mnemonic specifies the ru6 format of the `LDWCP` instruction.

46.7.1 Data Access

Mnemonic	Operands	Meaning
ld16s	d, b[i]	Load signed 16 bits
ld8u	d, b[i]	Load unsigned 8 bits
lda16	d, b[i]	Add to 16-bit address
lda16	d, b[-i]	Subtract from 16-bit address
ldap	r11, u_{20}	Load pc-relative address
ldap	r11, -u_{20}	Load pc-relative address
ldaw	d, b[i]	Add to a word address
ldaw	d, b[-i]	Subtract from a word address
ldaw	d, b[u_s]	Add to a word address immediate
ldaw	d, b[-u_s]	Subtract from a word address immediate
ldaw	r11, cp[u_{16}]	Load address of word in constant pool
ldaw	d, dp[u_{16}]	Load address of word in data pool
ldaw	d, sp[u_{16}]	Load address of word on stack
ldd	e, d, b[i]	Load double word (xCORE-200 only)
ldd	e, d, b[u_s]	Load double word immediate (xCORE-200 only)
ldd	e, d, sp[u_s]	Load double from the stack (xCORE-200 only)
ldw	et, sp[4]	Load ET from the stack
ldw	sed, sp[3]	Load SED from the stack
ldw	spc, sp[1]	Load SPC from the stack
ldw	ssr, sp[2]	Load SSR from the stack
ldw	d, b[i]	Load word
ldw	d, b[u_s]	Load word immediate
ldw	d, cp[u_{16}]	Load word from constant pool
ldw	r11, cp[u_{20}]	Load word from constant pool
ldw	d, dp[u_{16}]	Load word from data pool
ldw	d, sp[u_{16}]	Load word from stack
set	cp, s	Set constant pool
set	dp, s	Set data pointer
set	sp, s	Set the stack pointer
st16	s, b[i]	16-bit store
st8	s, b[i]	8-bit store
std	e, d, b[i]	Store double word (xCORE-200 only)
std	e, d, b[u_s]	Store double word immediate (xCORE-200 only)
std	y, x, sp[u_s]	Store double word on the stack (xCORE-200 only)
stw	sed, sp[3]	Store SED on the stack
stw	et, sp[4]	Store ET on the stack
stw	spc, sp[1]	Store SPC on the stack
stw	ssr, sp[2]	Store SSR on the stack
stw	s, b[i]	Store word
stw	s, b[u_s]	Store word immediate
stw	s, dp[u_{16}]	Store word in data pool
stw	s, sp[u_{16}]	Store word on stack

46.7.2 Branching, Jumping and Calling

Mnemonic	Operands	Meaning
bau	s	Branch absolute unconditional
bf	c, u_{16}	Branch relative if false
bf	c, -u_{16}	Branch relative if false
bl	u_{20}	Branch and link relative
bl	-u_{20}	Branch and link relative
bla	s	Branch and link absolute via register
bla	cp[u_{20}]	Branch and link absolute via CP
blat	u_{16}	Branch and link absolute via table
bru	s	Branch relative unconditional via register
bt	c, u_{16}	Branch relative if true
bt	c, -u_{16}	Branch relative if true
bu	u_{16}	Branch relative unconditional
bu	-u_{16}	Branch relative unconditional
dualentsp	u_{16}	Adjust stack, save link register and enable dual issue (xCORE-200 only)
entsp	u_{16}	Adjust stack and save link register and enable single issue
extdp	u_{16}	Extend data pointer
extsp	u_{16}	Extend stack pointer
retsp	u_{16}	Return

46.7.3 Data Manipulation

Mnemonic	Operands	Meaning
add	d, x, y	Add
add	d, x, u_s	Add immediate
and	d, x, y	Bitwise and
andnot	d, s	And not
ashr	d, x, y	Arithmetic shift right
ashr	d, x, bitp	Arithmetic shift right immediate
bitrev	d, s	Bit reverse
byterev	d, s	Byte reverse
clz	d, s	Count leading zeros
crc32	d, r, p	Word CRC
crc32_inc	d, e, x, y, bitp	Word CRC with address increment (xCORE-200 only)
crc8	r, o, d, p	8-step CRC
crcn	d, x, p, n	Variable step CRC (xCORE-200 only)
divs	d, x, y	Signed division
divu	d, x, y	Unsigned division
eq	c, x, y	Equal
eq	c, x, u_s	Equal immediate
ladd	e, d, x, y, v	Long unsigned add with carry
ldc	d, u_{16}	Load constant
ldivu	d, e, v, x, y	Long unsigned divide
lextract	d, x, y, u, bitp	Bitfield extraction from register pair (xCORE-200 only)

(continued)

Mnemonic	Operands	Meaning
linsert	d, e, x, y, bitp	Inserts a bitfield into a pair of registers (xCORE-200 only)
lmul	d, e, x, y, v, w	Long multiply
lsats	d, x, y	Saturate signed (xCORE-200 only)
lss	c, x, y	Less than signed
lsu	c, x, y	Less than unsigned
lsub	e, d, x, y, v	Long unsigned subtract
maccs	d, e, x, y	Mulitply and accumulate signed
maccu	d, e, x, y	Multiply and accumulate unsigned
mkmsk	d, s	Make mask
mkmsk	d, bitp	Make mask immediate
mul	d, x, y	Multiply
neg	d, s	Two's complement negate
not	d, s	Bitwise not
or	d, x, y	Bitwise or
rems	d, x, y	Signed remainder
remu	d, x, y	Unsigned remainder
sext	d, s	Sign extend
sext	d, bitp	Sign extend immediate
shl	d, x, y	Shift left
shl	d, x, bitp	Shift left immediate
shr	d, x, y	Shift right
shr	d, x, bitp	Shift right immediate
sub	d, x, y	Subtract
sub	d, x, u_s	Subtract immediate
unzip	d, e, x	Unzips a pair of registers (xCORE-200 only)
xor	d, x, y	Bitwise exclusive or
xor4	d, e, x, y, v	Bitwise exclusive or of four words (xCORE-200 only)
zext	d, s	Zero extend
zext	s, bitp	Zero extend immediate
zip	d, e, x	Zips together a pair of registers (xCORE-200 only)

46.7.4 Concurrency and Thread Synchronization

Mnemonic	Operands	Meaning
freet		Free unsynchronized thread
get	r11, id	Get thread ID
getst	d, res[r]	Get synchronized thread
mjoin	res[r]	Master synchronize and join
msync	res[r]	Master synchronize
ssync		Slave synchronize
init	t[r]:cp, s	Initialize thread's CP
init	t[r]:dp, s	Initialize thread's DP
init	t[r]:lr, s	Initialize thread's LR
init	t[r]:pc, s	Initialize thread's PC
init	t[r]:sp, s	Initialize thread's SP
set	t[r]:d, s	Set register in thread

(continued)

Mnemonic	Operands	Meaning
start	t[r]	Start thread
tsetmr	d, s	Set register in master thread

46.7.5 Communication

Mnemonic	Operands	Meaning
chkct	res[r], s	Test for control token
chkct	res[r], u_s	Test for control token immediate
getn	d, res[r]	Get network
in	d, res[r]	Input data
inct	d, res[r]	Input control token
int	d, res[r]	Input token of data
out	res[r], s	Output data
outct	res[r], s	Output control token
outct	res[r], u_s	Output control token immediate
outt	res[r], s	Output token of data
setn	res[r], s	Set network
testlcl	d, res[r]	Test local
testct	d, res[r]	Test for control token
testwct	d, res[r]	Test for position of control token

46.7.6 Resource Operations

Mnemonic	Operands	Meaning
clrpt	res[r]	Clear port time
elate	s	Throw exception if too late (xCORE-200 only)
endin	d, res[r]	End a current input
freer	res[r]	Free a resource
getd	d, res[r]	Get resource data
getr	d, u_s	Allocate resource
gettime	d	Get the reference time (xCORE-200 only)
getts	d, res[r]	Get port timestamp
in	d, res[r]	Input data
inpw	d, res[r], bitp	Input a part word
inshr	d, res[r]	Input and shift right
out	res[r], s	Output data
outpw	res[r], s, bitp	Output a part word
outpw	res[r], s, w	Output a part word immediate (xCORE-200 only)
outshr	res[r], s	Output data and shift
peek	d, res[r]	Peek at port data
setc	res[r], s	Set resource control bits
setc	res[r], u_{16}	Set resource control bits immediate
setclk	res[r], s	Set clock for a resource
setd	res[r], s	Set data
setev	res[r], r11	Set environment vector
setpsc	res[r], s	Set the port shift count

(continued)

Mnemonic	Operands	Meaning
setpt	res[r], s	Set the port time
setrdy	res[r], s	Set ready input for a port
settw	res[r], s	Set transfer width for a port
setv	res[r], r11	Set event vector
syncr	res[r]	Synchronize a resource

46.7.7 Event Handling

Mnemonic	Operands	Meaning
clre		Clear all events
clrsr	u_{16}	Clear bits in SR
edu	res[r]	Disable events
eef	d, res[r]	Enable events if false
eet	d, res[r]	Enable events if true
eeu	res[r]	Enable events
getsr	r11, u_{16}	Get bits from SR
setsr	u_{16}	Set bits in SR
waitef	c	Wait for event if false
waitet	c	Wait for event if true
waiteu		Wait for event

46.7.8 Interrupts, Exceptions and Kernel Calls

Mnemonic	Operands	Meaning
clrsr	u_{16}	Clear bits in SR
ecallf	c	Raise exception if false
ecallt	c	Raise exception if true
get	r11, ed	Get ED into r11
get	r11, et	Get ET into r11
get	r11, kep	Get the kernel entry point
get	r11, ksp	Get the kernel stack pointer
getsr	r11, u_{16}	Get bits from SR
kcall	s	Kernel call
kcall	u_{16}	Kernel call immediate
kentsp	u_{16}	Switch to kernel stack
krestsp	u_{16}	Restore stack pointer from kernel stack
kret		Kernel return
set	kep, r11	Set the kernel entry point
setsr	u_{16}	Set bits in SR

46.7.9 Debugging

Mnemonic	Operands	Meaning
dcall		Cause a debug interrupt

(continued)

Mnemonic	Operands	Meaning
dentsp		Save and modify stack pointer for debug
dgetreg	s	Debug read of another thread's register
drestsp		Restore non debug stack pointer
dret		Return from debug interrupt
get	d, ps[r]	Get processor state
set	ps[r], s	Set processor state

46.7.10 Pseudo Instructions

In the default syntax mode, the assembler supports a small set of pseudo instructions. These instructions do not exist on the processor, but are translated by the assembler into xCORE instructions.

Mnemonic	Operands	Definition
mov	d, s	add d, s, 0
nop		r0, r0, 0

46.8 Assembly Program

An assembly program consists of a sequence of statements.

program ::= ⟨*statement*⟩*

statement ::= *label-list*$_{opt}$ *dir-or-inst*$_{opt}$ *separator*

label-list ::= *label*
| *label-list label*

dir-or-inst ::= *directive*
| *instruction*
| *instruction-bundle*

directive ::= *align-directive*
| *ascii-directive*
| *value-directive*
| *file-directive*
| *loc-directive*
| *weak-directive*
| *vis-directive*
| *text-directive*
| *set-directive*
| *cc-directive*
| *scheduling-directive*
| *syntax-directive*
| *assert-directive*
| *xc-directive*
| *xta-directive*
| *space-directive*
| *type-directive*
| *size-directive*
| *jmptable-directive*
| *globalresource-directive*

Part O

Programming for XS1 Devices

CONTENTS

- XCC Target-Dependent Behavior for XS1 Devices
- XS1 Data Types
- XS1 port-to-pin mapping
- XS1 Library
- xCORE 32-Bit Application Binary Interface

47XCC Target-Dependent Behavior for XS1 Devices

IN THIS CHAPTER

- Support for Clock Blocks
- Support for Ports
- Channel Communication

This section describes behavior of the XMOS compiler collection that is specific to the XS1 architecture.

47.1 Support for Clock Blocks

An XS1 device provides a single reference clock that ticks at a frequency derived from an external oscillior. XC requires the system designer to ensure that the reference clock ticks at 100MHz for correct operation of timers.

Each xCORE Tile provides a set of programmable clock blocks, which can be used to produce clock signals for ports. A clock block can use either a 1-bit port or a divided reference clock.

The `<xs1.h>` header file includes a `clock` type definition. A variable of type `clock`, not declared `extern`, must be initialized with an expression representing a clock block, for example:

```
clock c = XS1_CLKBLK_1;
```

The number of clock blocks available is given in the device datasheet. Their names are as the above declaration, numbered sequentially from 1.

In XC, the `clock` type is a *resource* type, with the following additional rules:

- A structure may declare members of type `clock`. Variables of a structure with type `clock` may be declared only as external declarations.
- A variable declaration prefixed with `on` may declare an object of type `clock`.
- Automatic variables may not be declared with type `clock`.

47.2 Support for Ports

The XC port declaration

```
port p;
```

declares a raw port. On XS1 devices, all ports used for inputting and outputting data are clocked by a 100MHz reference clock (see §47.1) and use a single-entry buffer, even if their declarations are not qualified with the keyword `buffered`.

The table in Figure 82 can be used to determine which I/O operations are supported on XS1 ports, depending on whether or not the corresponding XC declaration is qualified with the keyword `buffered`.

Figure 82: I/O operations supported on XS1 ports

Mode	Operation		
	Serialization	Strobing	`@ when`
Unqualified	✗	✗	✗
`buffered`	✓	✓	✓

The compiler detects and issues errors in the following cases:

- Serialization: A port not qualified with `buffered` is declared with a transfer width different from the port width.
- Strobing: A port not qualified with `buffered` is configured to use a ready-in or ready-out signal.
- An input uses both `@` and `when`: Both of these operators are used in an input statement with a port whose declaration is not qualified with `buffered`.

47.2.1 Serialization

Note that if serialization is used, the time specified by a timed input statement records the time at which the *last* bits of data are sampled. This can result in unexpected behaviour when serialization is used, since the construction

```
par {
  p @ t <: x;
  q @ t :> y;
}
```

causes the output on `p` to start at the same time as the input on `q` completes. To input and output this data in parallel, the input time should be offset in the software by an amount equal to the the transfer width divided by the port width.

47.2.2 Timestamping

The timestamp recorded by an input statement may come after the time when the data was sampled. This is because the XS1 provides separate instructions for

inputting data and inputting the timestamp, so the timestamp can be input after the next data is sampled. This issue also affects output statements, but does not affect inputs performed in the guards of a `select` statement. The compiler inputs the timestamp immediately after executing an input or output instruction, so in practice this behaviour is rarely seen.

47.2.3 Changing Direction of Buffered Ports

An attempt to change the direction of a port qualified with `buffered` results in undefined behaviour.

47.3 Channel Communication

On some revisions of the XS1 architecture, it is not possible to input data of size less than 32 bits from a streaming channel in the guard of a select statement.

- When compiling for the XS1-G architecture, the compiler disallows selecting on a channel input of less than a word-length in an XC streaming channel. The command line option `-fsubword-select` relaxes this restriction, but this can lead to cases with these functions not being taken even if data is available on the channel.
- When compiling for the XS1-G architecture, the `inuchar_byref`, `inct_byref` and `testct` functions may not be used in an XC `select` statement. The command line option `-fsubword-select` relaxes this restriction, but this can lead to cases with these functions not being taken even if data is available on the channel.

The size and alignment of C and XC's data types are not specified by the language. This allows the size of `int` to be set to the natural word size of the target device, ensuring the fastest possible performance for many computations. It also allows the alignment to be set wide enough to enable efficient memory loads and stores. Figure 83 represents the size and alignment of the data types specified by the xCORE Application Binary Interface (see §51), which provides a standard interface for linking objects compiled from both C and XC.

Data Type	Size (bits)	Align (bits)	Supported XC	Supported C	Meaning
`char`	8	8	✓	✓	Character type
`short`	16	16	✓	✓	Short integer
`int`	32	32	✓	✓	Native integer
`long`	32	32	✓	✓	Long integer
`long long`	64	32	✗	✓	Long long integer
`float`	32	32	✗	✓	32-bit IEEE float
`double`	64	32	✗	✓	64-bit IEEE float
`long double`	64	32	✗	✓	64-bit IEEE float
`void *`	32	32	✗	✓	Data pointer
`port`	32	32	✓	✗	Port
`timer`	32	32	✓	✗	Timer
`chanend`	32	32	✓	✗	Channel end

Figure 83: Size and alignment of data types on XS1 devices

In addition:

- The `char` type is by default unsigned.
- The types `char`, `short` and `int` may be specified in a bit-field's declaration.
- A zero-width bit-field forces padding until the next bit-offset aligned with the bit-field's declared type.
- The notional transfer type of a port is `unsigned int` (32 bits).
- The notional counter type of a port is `unsigned short` (16 bits).
- The notional counter type of a timer is `unsigned int` (32 bits).

49XS1 port-to-pin mapping

On XS1 devices, pins are used to interface with external components via ports and to construct links to other devices over which channels are established. The ports are multiplexed, allowing the pins to be configured for use by ports of different widths. Figure 84 gives the XS1 port-to-pin mapping, which is interpreted as follows:

- The name of each pin is given in the format X*n*D*pq* where *n* is a valid xCORE Tile number for the device and *pq* exists in the table. The physical position of the pin depends on the packaging and is given in the device datasheet.
- Each link is identified by a letter A-D. The wires of a link are identified by means of a superscripted digit 0-4.
- Each port is identified by its width (the first number 1, 4, 8, 16 or 32) and a letter that distinguishes multiple ports of the same width (A-P). These names correspond to port identifiers in the header file <xs1.h> (for example port 1A corresponds to the identifier XS1_PORT_1A). The individual bits of the port are identified by means of a superscripted digit 0-31.
- The table is divided into six rows (or *banks*). The first four banks provide a selection of 1, 4 and 8-bit ports, with the last two banks enabling the single 32-bit port. Different packaging options may export different numbers of banks; the 16-bit and 32-bit ports are not available on small devices.

The ports used by a program are determined by the set of XC port declarations. For example, the declaration

```
on tile [0] : in port p = XS1_PORT_1A
```

uses the 1-bit port 1A on xCORE Tile 0, which is connected to pin X0D00.

Usually the designer should ensure that there is no overlap between the pins of the declared ports, but the precedence has been designed so that, if required, portions of the wider ports can be used when overlapping narrower ports are used. The ports to the left of the table have precedence over ports to the right. If two ports are declared that share the same pin, the narrower port takes priority. For example:

```
on tile[2] : out port p1 = XS1_PORT_32A;
on tile[2] : out port p2 = XS1_PORT_8B;
on tile[2] : out port p3 = XS1_PORT_4C;
```

In this example:

- I/O on port p1 uses pins X2D02 to X2D09 and X2D49 to X2D70.
- I/O on port p2 uses pins X2D16 to X2D19; inputting from p2 results in undefined values in bits 0, 1, 6 and 7.
- I/O on port p3 uses pins X2D14, X2D15, X2D20 and X2D21; inputting from p1 results in undefined values in bits 28-31, and when outputting these bits are not driven.

Figure 84: Available ports and links for each pin

	⇐ highest		Precedence			lowest ⇒
Pin	link	1-bit ports	4-bit ports	8-bit ports	16-bit ports	32-bit port
XnD00		$1A$				
XnD01	A^4 out	$1B$				
XnD02	A^3 out		$4A^0$	$8A^0$	$16A^0$	$32A^{20}$
XnD03	A^2 out		$4A^1$	$8A^1$	$16A^1$	$32A^{21}$
XnD04	A^1 out		$4B^0$	$8A^2$	$16A^2$	$32A^{22}$
XnD05	A^0 out		$4B^1$	$8A^3$	$16A^3$	$32A^{23}$
XnD06	A^0 in		$4B^2$	$8A^4$	$16A^4$	$32A^{24}$
XnD07	A^1 in		$4B^3$	$8A^5$	$16A^5$	$32A^{25}$
XnD08	A^2 in		$4A^2$	$8A^6$	$16A^6$	$32A^{26}$
XnD09	A^3 in		$4A^3$	$8A^7$	$16A^7$	$32A^{27}$
XnD10	A^4 in	$1C$				
XnD11		$1D$				
XnD12		$1E$				
XnD13	B^4 out	$1F$				
XnD14	B^3 out		$4C^0$	$8B^0$	$16A^8$	$32A^{28}$
XnD15	B^2 out		$4C^1$	$8B^1$	$16A^9$	$32A^{29}$
XnD16	B^1 out		$4D^0$	$8B^2$	$16A^{10}$	
XnD17	B^0 out		$4D^1$	$8B^3$	$16A^{11}$	
XnD18	B^0 in		$4D^2$	$8B^4$	$16A^{12}$	
XnD19	B^1 in		$4D^3$	$8B^5$	$16A^{13}$	
XnD20	B^2 in		$4C^2$	$8B^6$	$16A^{14}$	$32A^{30}$
XnD21	B^3 in		$4C^3$	$8B^7$	$16A^{15}$	$32A^{31}$
XnD22	B^4 in	$1G$				
XnD23		$1H$				
XnD24		$1I$				
XnD25		$1J$				
XnD26			$4E^0$	$8C^0$	$16B^0$	
XnD27			$4E^1$	$8C^1$	$16B^1$	
XnD28			$4F^0$	$8C^2$	$16B^2$	
XnD29			$4F^1$	$8C^3$	$16B^3$	
XnD30			$4F^2$	$8C^4$	$16B^4$	
XnD31			$4F^3$	$8C^5$	$16B^5$	
XnD32			$4E^2$	$8C^6$	$16B^6$	
XnD33			$4E^3$	$8C^7$	$16B^7$	
XnD34		$1K$				
XnD35		$1L$				
XnD36		$1M$		$8D^0$	$16B^8$	
XnD37		$1N$		$8D^1$	$16B^9$	
XnD38		$1O$		$8D^2$	$16B^{10}$	
XnD39		$1P$		$8D^3$	$16B^{11}$	
XnD40				$8D^4$	$16B^{12}$	
XnD41				$8D^5$	$16B^{13}$	
XnD42				$8D^6$	$16B^{14}$	
XnD43				$8D^7$	$16B^{15}$	
XnD49	C^4 out					$32A^0$
XnD50	C^3 out					$32A^1$
XnD51	C^2 out					$32A^2$
XnD52	C^1 out					$32A^3$
XnD53	C^0 out					$32A^4$
XnD54	C^0 in					$32A^5$
XnD55	C^1 in					$32A^6$
XnD56	C^2 in					$32A^7$
XnD57	C^3 in					$32A^8$
XnD58	C^4 in					$32A^9$
XnD61	D^4 out					$32A^{10}$
XnD62	D^3 out					$32A^{11}$
XnD63	D^2 out					$32A^{12}$
XnD64	D^1 out					$32A^{13}$
XnD65	D^0 out					$32A^{14}$
XnD66	D^0 in					$32A^{15}$
XnD67	D^1 in					$32A^{16}$
XnD68	D^2 in					$32A^{17}$
XnD69	D^3 in					$32A^{18}$
XnD70	D^4 in					$32A^{19}$

50XS1 Library

IN THIS CHAPTER

- Data types
- Port Configuration Functions
- Clock Configuration Functions
- Port Manipulation Functions
- Clock Manipulation Functions
- Logical Core/Tile Control Functions
- Channel Functions
- Predicate Functions
- XS1-S Functions
- Miscellaneous Functions

50.1 Data types

Macro	clock
Description	Clock resource type. Clocks are declared as global variables and are initialized with the resource identifier of a clock block. When in a running state a clock provides rising and falling edges to ports configured using that clock.

50.2 Port Configuration Functions

Function	**configure_in_port_handshake**	
Description	Configures a buffered port to be a clocked input port in handshake mode. If the ready-in or ready-out ports are not 1-bit ports, an exception is raised. The ready-out port is asserted on the falling edge of the clock when the port's buffer is not full. The port samples its pins on its sampling edge when both the ready-in and ready-out ports are asserted. By default the port's sampling edge is the rising edge of clock. This can be changed by the function set_port_sample_delay().	
Type	`void configure_in_port_handshake(void port p, in port readyin, out port readyout, clock clk)`	
Parameters	`p`	The buffered port to configure.
	`readyin`	A 1-bit port to use for the ready-in signal.
	`readyout`	A 1-bit port to use for the ready-out signal.
	`clk`	The clock used to configure the port.

Function	**configure_out_port_handshake**
Description	Configures a buffered port to be a clocked output port in handshake mode. If the ready-in or ready-out ports are not 1-bit ports an exception is raised. The port drives the initial value on its pins until an output statement changes the value driven. The ready-in port is read on the sampling edge of the buffered port. Outputs are driven on the next falling edge of the clock where the previous value read from the ready-in port was high. On the falling edge of the clock the ready-out port is driven high if data is output on that edge, otherwise it is driven low. By default the port's sampling edge is the rising edge of clock. This can be changed by the function set_port_sample_delay().

Type	`void`
	`configure_out_port_handshake(void port p,`
	`in port readyin,`
	`out port readyout,`
	`clock clk,`
	`unsigned initial)`

Parameters	`p` The buffered port to configure. `readyin` A 1-bit port to use for the ready-in signal. `readyout` A 1-bit port to use for the ready-out signal. `clk` The clock used to configure the port. `initial` The initial value to output on the port.

Function	**configure_in_port_strobed_master**
Description	Configures a buffered port to be a clocked input port in strobed master mode. If the ready-out port is not a 1-bit port, an exception is raised. The ready-out port is asserted on the falling edge of the clock when the port's buffer is not full. The port samples its pins on its sampling edge after the ready-out port is asserted. By default the port's sampling edge is the rising edge of clock. This can be changed by the function set_port_sample_delay().

Type	`void`
	`configure_in_port_strobed_master(void port p,`
	`out port readyout,`
	`const clock clk)`

Parameters	`p` The buffered port to configure. `readyout` A 1-bit port to use for the ready-out signal. `clk` The clock used to configure the port.

Function	configure_out_port_strobed_master	
Description	Configures a buffered port to be a clocked output port in strobed master mode. If the ready-out port is not a 1-bit port, an exception is raised. The port drives the initial value on its pins until an output statement changes the value driven. Outputs are driven on the next falling edge of the clock. On the falling edge of the clock the ready-out port is driven high if data is output on that edge, otherwise it is driven low.	
Type	`void configure_out_port_strobed_master(void port p, out port readyout, const clock clk, unsigned initial)`	
Parameters	`p`	The buffered port to configure.
	`readyout`	A 1-bit port to use for the ready-out signal.
	`clk`	The clock used to configure the port.
	`initial`	The initial value to output on the port.

Function	**configure_in_port_strobed_slave**	
Description	Configures a buffered port to be a clocked input port in strobed slave mode. If the ready-in port is not a 1-bit port, an exception is raised. The port samples its pins on its sampling edge when the ready-in signal is high. By default the port's sampling edge is the rising edge of clock. This can be changed by the function set_port_sample_delay().	
Type	`void configure_in_port_strobed_slave(void port p, in port readyin, clock clk)`	
Parameters	`p`	The buffered port to configure.
	`readyin`	A 1-bit port to use for the ready-in signal.
	`clk`	The clock used to configure the port.

Function	configure_out_port_strobed_slave	
Description	Configures a buffered port to be a clocked output port in strobed slave mode. If the ready-in port is not a 1-bit port, an exception is raised. The port drives the initial value on its pins until an output statement changes the value driven. The ready-in port is read on the buffered port's sampling edge. Outputs are driven on the next falling edge of the clock where the previous value read from the ready-in port is high. By default the port's sampling edge is the rising edge of clock. This can be changed by the function set_port_sample_delay().	
Type	`void configure_out_port_strobed_slave(void port p, in port readyin, clock clk, unsigned initial)`	
Parameters	`p`	The buffered port to configure.
	`readyin`	A 1-bit port to use for the ready-in signal.
	`clk`	The clock used to configure the port.
	`initial`	The initial value to output on the port.

Function	configure_in_port
Description	Configures a port to be a clocked input port with no ready signals. This is the default mode of a port. The port samples its pins on its sampling edge. If the port is unbuffered, its direction can be changed by performing an output. This change occurs on the next falling edge of the clock. Afterwards, the port behaves as an output port with no ready signals. By default the port's sampling edge is the rising edge of the clock. This can be changed by the function set_port_sample_delay().
Type	`void configure_in_port(void port p, const clock clk)`
Parameters	`p` The port to configure, which may be buffered or unbuffered. `clk` The clock used to configure the port.

Function	configure_in_port_no_ready
Description	Alias for configure_in_port().
Type	`void configure_in_port_no_ready(void port p, const clock clk)`

Function	**configure_out_port**
Description	Configures a port to be a clocked output port with no ready signals. The port drives the initial value on its pins until an input or output statement changes the value driven. Outputs are driven on the next falling edge of the clock and every port-width bits of data are held for one clock cycle. If the port is unbuffered, the direction of the port can be changed by performing an input. This change occurs on the falling edge of the clock after any pending outputs have been held for one clock period. Afterwards, the port behaves as an input port with no ready signals.

Type	`void configure_out_port(void port p,`
	`const clock clk,`
	`unsigned initial)`

Parameters	`p` The port to configure, which may be buffered or unbuffered. `clk` The clock used to configure the port. `initial` The initial value to output on the port.

Function	**configure_out_port_no_ready**
Description	Alias for configure_out_port().

Type	`void`
	`configure_out_port_no_ready(void port p,`
	`const clock clk,`
	`unsigned initial)`

Function	**configure_port_clock_output**
Description	Configures a 1-bit port to output a clock signal. If the port is not a 1-bit port, an exception is raised. Performing inputs or outputs on the port once it has been configured in this mode results in undefined behaviour.
Type	`void` `configure_port_clock_output(void port p, const clock clk)`
Parameters	`p` The 1-bit port to configure. `clk` The clock to output.

Function	set_port_no_sample_delay
Description	Sets a port to no sample delay mode. This causes the port to sample input data on the rising edge of its clock. This is the default state of the port.
Type	`void` `set_port_no_sample_delay(void port p)`
Parameters	`p` The port to configure.

Function	set_port_sample_delay
Description	Sets a port to sample delay mode. This causes the port to sample input data on the falling edge of its clock.
Type	`void set_port_sample_delay(void port p)`
Parameters	`p` The port to configure.

Function	set_port_clock
Description	Attaches a clock to a port. This corresponds to using the SETCLK instruction on a port. The edges of the clock are used to sample and output data. Usually the use of the configure_*_port_* functions is preferred since they ensure that all the port configuration changes required for the desired mode are performed in the correct order.
Type	`void set_port_clock(void port p, const clock clk)`
Parameters	`p` The port to configure. `clk` The clock to attach.

Function	set_port_ready_src
Description	Sets a 1-bit port as the ready-out for another port. This corresponds with using the SETRDY instruction on a port. If the ready-out port is not a 1-bit port then an exception is raised. The ready-out port is used to indicate that the port is ready to transfer data. Usually the use of the configure_*_port_* functions is preferred since they ensure that all the port configuration changes required for the desired mode are performed in the correct order.
Type	`void set_port_ready_src(void port ready, void port p)`
Parameters	`p` The port to configure. `ready` The 1-bit port to use for the ready-out signal.

Function	set_port_use_on
Description	Turns on a port. The port state is initialised to the default state for a port of its type. If the port is already turned on its state is reset to its default state.
Type	`void set_port_use_on(void port p)`
Parameters	`p` The port to turn on.

Function	set_port_use_off
Description	Turns off a port. No action is performed if the port is already turned off. Any attempt to use the port while off will result in an exception being raised.
Type	`void set_port_use_off(void port p)`
Parameters	`p` The port to turn off.

Function	set_port_mode_data
Description	Configures a port to be a data port. This is the default state of a port. Output operations on the port are use to control its output signal.
Type	`void set_port_mode_data(void port p)`
Parameters	`p` The port to configure.

Function	set_port_mode_clock
Description	Configures a 1-bit port to be a clock output port. The port will output the clock connected to it. If the port is not a 1-bit port, an exception is raised. The function set_port_mode_data() can be used to set the port back to its default state.
Type	`void set_port_mode_clock(void port p)`
Parameters	`p` The port to configure.

Function	set_port_mode_ready
Description	Configures a 1-bit port to be a ready signal output port. The port will output the ready-out of a port connected with set_port_ready_src(). If the port is not a 1-bit port, an exception is raised. The function set_port_mode_data() can be used to set the port back to its default state. Usually the use of the configure_*_port_* functions is prefered since they ensure that all the port configuration changes required for the desired mode are performed in the correct order.
Type	`void set_port_mode_ready(void port p)`
Parameters	`p` The port to configure.

Function	set_port_drive
Description	Configures a port in drive mode. Values output to the port are driven on the pins. This is the default drive state of a port. Calling set_port_drive() has the side effect disabling the port's pull up or pull down resistor.
Type	`void set_port_drive(void port p)`
Parameters	`p` The port to configure.

Function	set_port_drive_low
Description	Configures a port in drive low mode. When a 0 is output to a pin it is driven low and when 1 is output no value is driven. For XS1 devices, if the port is not a 1-bit port, the result of an output to the port is undefined. On XS2 and XS1-G devices calling set_port_drive_low() has the side effect of enabling the port's internal pull-up resistor. On XS1-L devices calling set_port_drive_low() has the side effect of enabling the port's internal pull-down resistor.
Type	`void set_port_drive_low(void port p)`
Parameters	`p` The port to configure.

Function	set_port_pull_up
Description	Enables a port's internal pull-up resistor. When nothing is driving a pin the pull-up resistor ensures that the value sampled by the port is 1. The pull-up is not strong enough to guarantee a defined external value. On XS2 and XS1-G devices calling set_port_pull_up() has the side effect of configuring the port in drive low mode. On XS1-L devices no pull-up resistors are available and an exception will be raised if set_port_pull_up() is called.
Type	`void set_port_pull_up(void port p)`
Parameters	`p` The port to configure.

Function	set_port_pull_down
Description	Enables a port's internal pull-down resistor. When nothing is driving a pin the pull-down resistor ensures that the value sampled by the port is 0. The pull-down is not strong enough to guarantee a defined external value. On XS2 devices calling set_port_pull_down() has the side effect of configuring the port in drive high mode. On XS1-G devices no pull-down resistors are available and an exception will be raised if set_port_pull_down() is called. On XS1-L devices calling set_port_pull_down() has the side effect of configuring the port in drive low mode.
Type	`void set_port_pull_down(void port p)`
Parameters	`p` The port to configure.

Function	set_port_pull_none
Description	Disables the port's pull-up or pull-down resistor. This has the side effect of configuring the port in drive mode.
Type	`void set_port_pull_none(void port p)`
Parameters	`p` The port to configure.

Function	set_port_master
Description	Sets a port to master mode. This corresponds to using the SETC instruction on the port with the value XS1_SETC_MS_MASTER. Usually the use of the functions configure_in_port_strobed_master() and configure_out_port_strobed_master() is preferred since they ensure that all the port configuration changes required for the desired mode are performed in the correct order.
Type	`void set_port_master(void port p)`
Parameters	`p` The port to configure.

Function	set_port_slave
Description	Sets a port to slave mode. This corresponds to using the SETC instruction on the port with the value XS1_SETC_MS_SLAVE. Usually the use of the functions configure_in_port_strobed_slave() and configure_out_port_strobed_slave() is preferred since they ensure that all the port configuration changes required for the desired mode are performed in the correct order.
Type	`void set_port_slave(void port p)`
Parameters	`p` The port to configure.

Function	set_port_no_ready
Description	Configures a port to not use ready signals. This corresponds to using the SETC instruction on the port with the value XS1_SETC_RDY_NOREADY. Usually the use of the functions configure_in_port() and configure_out_port() is preferred since they ensure that all the port configuration changes required for the desired mode are performed in the correct order.
Type	`void set_port_no_ready(void port p)`
Parameters	`p` The port to configure.

Function	set_port_strobed
Description	Sets a port to strobed mode. This corresponds to using the SETC instruction on the port with the value XS1_SETC_RDY_STROBED. Usually the use of the configure_*_port_strobed_* functions is preferred since they ensure that all the port configuration changes required for the desired mode are performed in the correct order.
Type	`void set_port_strobed(void port p)`
Parameters	`p` The port to configure.

Function	set_port_handshake
Description	Sets a port to handshake mode. This corresponds to using the SETC instruction on the port with the value XS1_SETC_RDY_HANDSHAKE. Usually the use of the configure_*_port_handshake functions is preferred since they ensure that all the port configuration changes required for the desired mode are performed in the correct order.
Type	`void set_port_handshake(void port p)`
Parameters	`p` The port to configure.

Function	set_port_no_inv
Description	Configures a port to not invert data that is sampled and driven on its pins. This is the default state of a port.
Type	`void set_port_no_inv(void port p)`
Parameters	`p` The port to configure.

Function	set_port_inv
Description	Configures a 1-bit port to invert data which is sampled and driven on its pin. If the port is not a 1-bit port, an exception is raised. If the port is used as the source for a clock then setting this mode has the effect of the swapping the rising and falling edges of the clock.
Type	`void set_port_inv(void port p)`
Parameters	`p` The 1-bit port to configure.

Function	set_port_shift_count
Description	Sets the shift count for a port. This corresponds with the SETPSC instruction. The new shift count must be less than the transfer width of the port, greater than zero and a multiple of the port width otherwise an exception is raised. For a port used for input this function will cause the next input to be ready when the specified amount of data has been shifted in. The next input will return transfer-width bits of data with the captured data in the most significant bits. For a port used for output this will cause the next output to shift out this number of bits. Usually the use of the functions partin() and partout() is preferred over `setpsc()` as they perform both the required configuration and the input or output together.
Type	`void set_port_shift_count(void port p, unsigned n)`
Parameters	`p` The buffered port to configure. `n` The new shift count.

Function	set_pad_delay
Description	Sets the delay on the pins connected to the port. The input signals sampled on the port's pins are delayed by this number of processor-clock cycles before they they are seen on the port. The default delay on the pins is 0. The delay must be set to values in the range 0 to 5 inclusive. If there are multiple enabled ports connected to the same pin then the delay on that pin is set by the highest priority port.
Type	`void set_pad_delay(void port p, unsigned n)`
Parameters	`p` The port to configure. `n` The number of processor-clock cycles by which to delay the input signal.

50.3 Clock Configuration Functions

Function	**configure_clock_src**
Description	Configures a clock to use a 1-bit port as its source. This allows I/O operations on ports to be synchronised to an external clock signal. If the port is not a 1-bit port, an exception is raised.
Type	`void configure_clock_src(clock clk, void port p)`
Parameters	`clk` The clock to configure. `p` The 1-bit port to use as the clock source.

Function	**configure_clock_ref**
Description	Configures a clock to use the reference clock as it source. If the divide is set to zero the reference clock frequency is used, otherwise the reference clock frequency divided by 2 * `divide` is used. By default the reference clock is configured to run at 100 MHz.
Type	`void configure_clock_ref(clock clk, unsigned char divide)`
Parameters	`clk` The clock to configure. `divide` The clock divide.

Function	**configure_clock_rate**	
Description	Configures a clock to run at a rate of (a/b) MHz. If the specified rate is not supported by the hardware, an exception is raised. The hardware supports rates of MHz and rates of the form (MHz or (MHz where is the reference clock frequency, is the xCORE tile frequency and is a number in the range 1 to 255 inclusive.	
Type	`void configure_clock_rate(clock clk, unsigned a, unsigned b)`	
Parameters	`clk`	The clock to configure.
	`a`	The dividend of the desired rate.
	`b`	The divisor of the desired rate.

Function	**configure_clock_rate_at_least**
Description	Configures a clock to run the slowest rate supported by the hardware that is equal to or exceeds (a/b) MHz. An exception is raised if no rate satisfies this criterion.

Type	`void`
	`configure_clock_rate_at_least(clock clk,`
	`unsigned a,`
	`unsigned b)`

Parameters	`clk` The clock to configure. `a` The dividend of the desired rate. `b` The divisor of the desired rate.

Function	**configure_clock_rate_at_most**
Description	Configures a clock to run at the fastest non-zero rate supported by the hardware that is less than or equal to (a/b) MHz. An exception is raised if no rate satisfies this criterion.
Type	`void` `configure_clock_rate_at_most(clock clk, unsigned a, unsigned b)`
Parameters	`clk` The clock to configure. `a` The dividend of the desired rate. `b` The divisor of the desired rate.

Function	set_clock_src	
Description	Sets the source for a clock to a 1-bit port. This corresponds with using the SETCLK instruction on a clock. If the port is not a 1-bit port, an exception is raised. In addition if the clock was previously configured with a non-zero divide then an exception is raised. Usually the use of configure_clock_src() which does not suffer from this problem is recommended.	
Type	`void set_clock_src(clock clk, void port p)`	
Parameters	`clk`	The clock to configure.
	`p`	The 1-bit port to use as the clock source.

Function	set_clock_ref	
Description	Sets the source for a clock to the reference clock. This corresponds with the using SETCLK instruction on a clock. The clock divide is left unchanged.	
Type	`void set_clock_ref(clock clk)`	
Parameters	`clk`	The clock to configure.

Function	set_clock_div	
Description	Sets the divide for a clock. This corresponds with the SETD instruction. On XS1 devices an exception is raised if the clock source is not the reference clock or the xCORE tile clock. If the divide is set to zero the source frequency is left unchanged, unless the clock source is the xCORE tile clock, in which case the behaviour is undefined. If the divide is non-zero, the source frequency is divided by 2 * `divide`.	
Type	`void set_clock_div(clock clk, unsigned char div)`	
Parameters	`clk`	The clock to configure.
	`div`	The divide to use.

Function	set_clock_rise_delay	
Description	Sets the delay for the rising edge of a clock. Each rising edge of the clock by `n` processor-clock cycles before it is seen by any port connected to the clock. The default rising edge delay is 0 and the delay must be set to values in the range 0 to 512 inclusive. If the clock edge is delayed by more than the clock period then no rising clock edges are seen by the ports connected to the clock.	
Type	`void set_clock_rise_delay(clock clk, unsigned n)`	
Parameters	`clk`	The clock to configure.
	`n`	The number of processor-clock cycles by which to delay the rising edge of the clock.

Function	set_clock_fall_delay	
Description	Sets the delay for the falling edge of a clock. Each falling edge of the clock is delayed by `n` processor-clock cycles before it is seen by any port connected to the clock. The default falling edge delay is 0. The delay can be set to values in the range 0 to 512 inclusive. If the clock edge is delayed by more than the clock period then no falling clock edges are seen by the ports connected to the clock.	
Type	`void set_clock_fall_delay(clock clk, unsigned n)`	
Parameters	`clk`	The clock to configure.
	`n`	The number of processor-clock cycles by which to delay the falling edge of the clock.

Function	**set_clock_ready_src**
Description	Sets a clock to use a 1-bit port for the ready-in signal. This corresponds with using the SETRDY instruction on a clock. If the port is not a 1-bit port then an exception is raised. The ready-in port controls when data is sampled from the pins. Usually the use of the configure_*_port_* functions is preferred since they ensure that all the port configuration changes required for the desired mode are performed in the correct order.
Type	`void set_clock_ready_src(clock clk, void port ready)`
Parameters	`clk` The clock to configure. `ready` The 1-bit port to use for the ready-in signal.

Function	set_clock_on
Description	Turns on a clock. The clock state is initialised to the default state for a clock. If the clock is already turned on then its state is reset to its default state.
Type	`void set_clock_on(clock clk)`
Parameters	`clk` The clock to turn on.

Function	set_clock_off
Description	Turns off a clock. No action is performed if the clock is already turned off. Any attempt to use the clock while it is turned off will result in an exception being raised.
Type	`void set_clock_off(clock clk)`
Parameters	`clk` The clock to turn off.

50.4 Port Manipulation Functions

Function	**start_port**
Description	Activates a port. The buffer used by the port is cleared.
Type	`void start_port(void port p)`
Parameters	`p` The port to activate.

Function	**stop_port**
Description	Deactivates a port. The port is reset to being a no ready port.
Type	`void stop_port(void port p)`
Parameters	`p` The port to deactivate.

Function	**peek**
Description	Instructs the port to sample the current value on its pins. The port provides the sampled port-width bits of data to the processor immediately, regardless of its transfer width, clock, ready signals and buffering. The input has no effect on subsequent I/O performed on the port.
Type	`unsigned peek(void port p)`
Parameters	`p` The port to peek at.
Returns	The value sampled on the pins.

Function	**clearbuf**
Description	Clears the buffer used by a port. Any data sampled by the port which has not been input by the processor is discarded. Any data output by the processor which has not been driven by the port is discarded. If the port is in the process of serialising output, it is interrupted immediately. If a pending output would have caused a change in direction of the port then that change of direction does not take place. If the port is driving a value on its pins when clearbuf() is called then it continues to drive the value until an output statement changes the value driven.
Type	`void clearbuf(void port p)`
Parameters	`p` The port whose buffer is to be cleared.

Function	**sync**
Description	Waits until a port has completed any pending outputs. Waits output all until a port has completed any pending outputs and the last port-width bits of data has been held on the pins for one clock period.
Type	`void sync(void port p)`
Parameters	`p` The port to wait on.

Function	**endin**
Description	Ends the current input on a buffered port. The number of bits sampled by the port but not yet input by the processor is returned. This count includes both data in the transfer register and data in the shift register used for deserialisation. Subsequent inputs on the port return transfer-width bits of data until there is less than one transfer-width bits of data remaining. Any remaining data can be read with one further input, which returns transfer-width bits of data with the remaining buffered data in the most significant bits of this value.
Type	`unsigned endin(void port p)`
Parameters	`p` The port to end the current input on.
Returns	The number of bits of data remaining.

Function	**partin**
Description	Performs an input of the specified width on a buffered port. The width must be less than the transfer width of the port, greater than zero and a multiple of the port width, otherwise an exception is raised. The value returned is undefined if the number of bits in the port's shift register is greater than or equal to the specified width.
Type	`unsigned partin(void port p, unsigned n)`
Parameters	`p` The buffered port to input on. `n` The number of bits to input.
Returns	The inputted value.

Function	partout
Description	Performs an output of the specified width on a buffered port. The width must be less than the transfer width of the port, greater than zero and a multiple of the port width, otherwise an exception is raised. The `n` least significant bits of `val` are output.
Type	`void partout(void port p, unsigned n, unsigned val)`
Parameters	`p` The buffered port to output on. `n` The number of bits to output. `val` The value to output.

Function	partout_timed
Description	Performs a output of the specified width on a buffered port when the port counter equals the specified time. The width must be less than the transfer width of the port, greater than zero and a multiple of the port width, otherwise an exception is raised. The `n` least significant bits of `val` are output.
Type	`unsigned partout_timed(void port p, unsigned n, unsigned val, unsigned t)`
Parameters	`p` The buffered port to output on. `n` The number of bits to output. `val` The value to output. `t` The port counter value to output at.

Function	partin_timestamped
Description	Performs an input of the specified width on a buffered port and timestamps the input. The width must be less than the transfer width of the port, greater than zero and a multiple of the port width, otherwise an exception is raised. The value returned is undefined if the number of bits in the port's shift register is greater than or equal to the specified width.
Type	`{unsigned , unsigned } partin_timestamped(void port p, unsigned n)`
Parameters	`p` The buffered port to input on. `n` The number of bits to input.
Returns	The inputted value and the timestamp.

Function	partout_timestamped
Description	Performs an output of the specified width on a buffered port and timestamps the output. The width must be less than the transfer width of the port, greater than zero and a multiple of the port width, otherwise an exception is raised. The `n` least significant bits of `val` are output.
Type	`unsigned partout_timestamped(void port p, unsigned n, unsigned val)`
Parameters	`p` The buffered port to output on. `n` The number of bits to output. `val` The value to output.
Returns	The timestamp of the output.

50.5 Clock Manipulation Functions

Function	**start_clock**
Description	Puts a clock into a running state. A clock generates edges only after it has been put into this state. The port counters of all ports attached to the clock are reset to 0.
Type	`void start_clock(clock clk)`
Parameters	`clk` The clock to put into a running state.

Function	**stop_clock**
Description	Waits until a clock is low and then puts the clock into a stopped state. In a stopped state a clock does not generate edges.
Type	`void stop_clock(clock clk)`
Parameters	`clk` The clock to put into a stopped state.

50.6 Logical Core/Tile Control Functions

Function	set_core_fast_mode_on
Description	Sets the current logical core to run in fast mode. The scheduler always reserves a slot for a logical core in fast mode regardless of whether core is waiting for an input or a select to complete. This reduces the worst case latency from a change in state happening to a paused input or select completing as a result of that change. However, putting a core in fast mode means that other logical cores are unable to use the extra slot which would otherwise be available while the core is waiting. In addition setting logical cores to run in fast mode may also increase the power consumption.
Type	`void set_core_fast_mode_on(void)`

Function	set_core_fast_mode_off
Description	Sets the current logical core to run in normal execution mode. If a core has previously been put into fast mode using set_core_fast_mode_on() this function resets the execution mode it to its default state.
Type	`void set_core_fast_mode_off(void)`

Function	**getps**
Description	Gets the value of a processor state register. This corresponds with the GETPS instruction. An exception is raised if the argument is not a legal processor state register.
Type	`unsigned getps(unsigned reg)`
Parameters	`reg` The processor state register to read.
Returns	The value of the processor state register.

Function	**setps**
Description	Sets the value of a processor state register. Corresponds with the SETPS instruction. An exception is raised if the argument is not a legal processor state register.
Type	`void setps(unsigned reg, unsigned value)`
Parameters	`reg` The processor state register to write. `value` The value to set the processor state register to.

Function	**read_pswitch_reg**
Description	Reads the value of a processor switch register. The read is of the processor switch which is local to the specified tile id. On success 1 is returned and the value of the register is assigned to `data`. If an error acknowledgement is received or if the register number or tile identifier is too large to fit in the read packet then 0 is returned.
Type	`int read_pswitch_reg(unsigned tileid, unsigned reg, unsigned &data)`
Parameters	`tileid` The tile identifier. `reg` The number of the register. `data` The value read from the register.
Returns	Whether the read was successful.

Function	write_pswitch_reg
Description	Writes a value to a processor switch register. The write is of the processor switch which is local to the specified tile id. If a successful acknowledgement is received then 1 is returned. If an error acknowledgement is received or if the register number or tile identifier is too large to fit in the write packet then 0 is returned.
Type	`int write_pswitch_reg(unsigned tileid, unsigned reg, unsigned data)`
Parameters	`tileid` The tile identifier. `reg` The number of the register. `data` The value to write to the register.
Returns	Whether the write was successful.

Function	write_pswitch_reg_no_ack
Description	Writes a value to a processor switch register without acknowledgement. The write is of the processor switch which is local to the specified tile id. Unlike write_pswitch_reg() this function does not wait until the write has been performed. If the register number or tile identifier is too large to fit in the write packet 0 is returned, otherwise 1 is returned. Because no acknowledgement is requested the return value does not reflect whether the write succeeded.
Type	`int write_pswitch_reg_no_ack(unsigned tileid, unsigned reg, unsigned data)`
Parameters	`tileid` The tile identifier. `reg` The number of the register. `data` The value to write to the register.
Returns	Whether the parameters are valid.

Function	**read_sswitch_reg**
Description	Reads the value of a system switch register. The read is of the system switch which is local to the specified tile id. On success 1 is returned and the value of the register is assigned to `data`. If an error acknowledgement is received or if the register number or tile identifier is too large to fit in the read packet then 0 is returned.
Type	`int read_sswitch_reg(unsigned tileid, unsigned reg, unsigned &data)`
Parameters	`tileid` The tile identifier. `reg` The number of the register. `data` The value read from the register.
Returns	Whether the read was successful.

Function	**write_sswitch_reg**
Description	Writes a value to a system switch register. The write is of the system switch which is local to the specified tile id. If a successful acknowledgement is received then 1 is returned. If an error acknowledgement is received or if the register number or tile identifier is too large to fit in the write packet then 0 is returned.
Type	`int write_sswitch_reg(unsigned tileid, unsigned reg, unsigned data)`
Parameters	`tileid` The tile identifier. `reg` The number of the register. `data` The value to write to the register.
Returns	Whether the write was successful.

Function	**write_sswitch_reg_no_ack**
Description	Writes a value to a system switch register without acknowledgement. The write is of the system switch which is local to the specified tile id. Unlike write_sswitch_reg() this function does not wait until the write has been performed. If the register number or tile identifier is too large to fit in the write packet 0 is returned, otherwise 1 is returned. Because no acknowledgement is requested the return value does not reflect whether the write succeeded.
Type	`int write_sswitch_reg_no_ack(unsigned tileid, unsigned reg, unsigned data)`
Parameters	`tileid` The tile identifier. `reg` The number of the register. `data` The value to write to the register.
Returns	Whether the parameters are valid.

Function	**read_node_config_reg**
Description	Reads the value of a node configuration register. The read is of the node containing the specified tile. On success 1 is returned and the value of the register is assigned to `data`. If an error acknowledgement is received or if the register number is too large to fit in the read packet then 0 is returned.
Type	`int read_node_config_reg(tileref tile, unsigned reg, unsigned &data)`
Parameters	`tile` The tile. `reg` The number of the register. `data` The value read from the register.
Returns	Whether the read was successful.

Function	**write_node_config_reg**
Description	Writes a value to a node configuration register. The write is of the node containing the specified tile. If a successful acknowledgement is received then 1 is returned. If an error acknowledgement is received or if the register number is too large to fit in the write packet then 0 is returned.
Type	`int write_node_config_reg(tileref tile, unsigned reg, unsigned data)`
Parameters	`tile` The tile. `reg` The number of the register. `data` The value to write to the register.
Returns	Whether the write was successful.

Function	**write_node_config_reg_no_ack**
Description	Writes a value to a node configuration register without acknowledgement. The write is of the node containing the specified tile. Unlike write_node_config_reg() this function does not wait until the write has been performed. If the register number is too large to fit in the write packet 0 is returned, otherwise 1 is returned. Because no acknowledgement is requested the return value does not reflect whether the write succeeded.
Type	`int write_node_config_reg_no_ack(tileref tile, unsigned reg, unsigned data)`
Parameters	`tile` The tile. `reg` The number of the register. `data` The value to write to the register.
Returns	Whether the parameters are valid.

Function	**read_periph_8**
Description	Reads `size` bytes from the specified peripheral starting at the specified base address. The peripheral must be a peripheral with a 8-bit interface. On success 1 is returned and `data` is filled with the values that were read. Returns 0 on failure.

Type	`int read_periph_8(tileref tile,`
	`unsigned peripheral,`
	`unsigned base_address,`
	`unsigned size,`
	`unsigned char data[size])`

Parameters	`tile` The tile. `peripheral` The peripheral number. `base_address` The base address. `size` The number of 8-bit values to read. `data` The values read from the peripheral.
Returns	Whether the read was successful.

Function	**write_periph_8**
Description	Writes `size` bytes to the specified peripheral starting at the specified base address. The peripheral must be a peripheral with a 8-bit interface. On success 1 is returned. Returns 0 on failure.

Type	`int write_periph_8(tileref tile,`
	`unsigned peripheral,`
	`unsigned base_address,`
	`unsigned size,`
	`const unsigned char data[size])`

Parameters	`tile` The tile. `peripheral` The peripheral number. `base_address` The base address. `size` The number of 8-bit values to write. `data` The values to write to the peripheral.
Returns	Whether the write was successful.

Function	**write_periph_8_no_ack**
Description	Writes `size` bytes to the specified peripheral starting at the specified base address without acknowledgement. The peripheral must be a peripheral with a 8-bit interface. Unlike write_periph_8() this function does not wait until the write has been performed. Because no acknowledgement is requested the return value does not reflect whether the write succeeded.

Type	`int write_periph_8_no_ack(tileref tile,`
	`unsigned peripheral,`
	`unsigned base_address,`
	`unsigned size,`
	`const unsigned char data[size])`

Parameters	`tile` The tile. `peripheral` The peripheral number. `base_address` The base address. `size` The number of 8-bit values to write. `data` The values to write to the peripheral.
Returns	Whether the parameters are valid.

Function	**read_periph_32**
Description	Reads `size` 32-bit words from the specified peripheral starting at the specified base address. On success 1 is returned and `data` is filled with the values that were read. Returns 0 on failure. When reading a peripheral with an 8-bit interface the most significant byte of each word returned is the byte at the lowest address (big endian byte ordering).

Type	`int read_periph_32(tileref tile,`
	`unsigned peripheral,`
	`unsigned base_address,`
	`unsigned size,`
	`unsigned data[size])`

Parameters	`tile` The tile. `peripheral` The peripheral number. `base_address` The base address. `size` The number of 32-bit words to read. `data` The values read from the peripheral.
Returns	Whether the read was successful.

Function	**write_periph_32**
Description	Writes `size` 32-bit words to the specified peripheral starting at the specified base address. On success 1 is returned. Returns 0 on failure. When writing to a peripheral with an 8-bit interface the most significant byte of each word passed to the function is written to the byte at the lowest address (big endian byte ordering).

Type	`int write_periph_32(tileref tile,`
	`unsigned peripheral,`
	`unsigned base_address,`
	`unsigned size,`
	`const unsigned data[size])`

Parameters	`tile` The tile. `peripheral` The peripheral number. `base_address` The base address. `size` The number of 32-bit words to write. `data` The values to write to the peripheral.
Returns	Whether the write was successful.

Function	**write_periph_32_no_ack**
Description	Writes `size` 32-bit words to the specified peripheral starting at the specified base address without acknowledgement. Unlike write_periph_32() this function does not wait until the write has been performed. Because no acknowledgement is requested the return value does not reflect whether the write succeeded. When writing to a peripheral with an 8-bit interface the most significant byte of each word passed to the function is written to the byte at the lowest address (big endian byte ordering).

Type	`int write_periph_32_no_ack(tileref tile,`
	`unsigned peripheral,`
	`unsigned base_address,`
	`unsigned size,`
	`const unsigned data[size])`

Parameters	`tile` The tile. `peripheral` The peripheral number. `base_address` The base address. `size` The number of 32-bit words to write. `data` The values to write to the peripheral.
Returns	Whether the parameters are valid.

Function	**get_local_tile_id**
Description	Returns the routing ID of the tile on which the caller is running. The routing ID uniquely identifies a tile on the network.
Type	`unsigned get_local_tile_id(void)`
Returns	The tile identifier.

Function	**get_tile_id**
Description	Returns the routing ID of the specified tile. The routing ID uniquely identifies a tile on the network.
Type	`unsigned get_tile_id(tileref t)`
Parameters	`tile` The tile.
Returns	The tile identifier.

Function	**get_logical_core_id**
Description	Returns the identifier of the logical core on which the caller is running. The identifier uniquely identifies a logical core on the current tile.
Type	`unsigned get_logical_core_id(void)`
Returns	The logical core identifier.

50.7 Channel Functions

Function	**outuchar**	
Description	Streams out a value as an unsigned char on a channel end. The protocol used is incompatible with the protocol used by the input (:>) and output (<:) operators.	
Type	`void outuchar(chanend c, unsigned char val)`	
Parameters	`c`	The channel end to stream data out on.
	`val`	The value to output.

Function	**outuint**	
Description	Streams out a value as an unsigned int on a channel end. The protocol used is incompatible with the protocol used by the input (:>) and output (<:) operators.	
Type	`void outuint(chanend c, unsigned val)`	
Parameters	`c`	The channel end to stream data out on.
	`val`	The value to output.

Function	inuchar
Description	Streams in a unsigned char from a channel end. If the next token in the channel is a control token then an exception is raised. The protocol used is incompatible with the protocol used by the input (:>) and output (<:) operators.
Type	`unsigned char inuchar(chanend c)`
Parameters	`c` The channel end to stream data in on.
Returns	The value received.

Function	inuint
Description	Streams in a unsigned int from a channel end. If the next word of data channel in the channel contains a control token then an exception is raised. The protocol used is incompatible with the protocol used by the input (:>) and output (<:) operators.
Type	`unsigned inuint(chanend c)`
Parameters	`c` The channel end to stream data in on.
Returns	The value received.

Function	inuchar_byref
Description	Streams in a unsigned char from a channel end. The inputted value is written to `val`. If the next token in channel is a control token then an exception is raised. The protocol used is incompatible with the protocol used by the input (:>) and output (<:) operators.
Type	`void inuchar_byref(chanend c, unsigned char &val)`
Parameters	`c` The channel end to stream data in on. `val` The variable to set to the received value.

Function	inuint_byref
Description	Streams in a unsigned int from a channel end. The inputted value is written to `val`. This function may be called in a case of a select, in which case it becomes ready as soon as there data available on the channel. The protocol used is incompatible with the protocol used by the input (:>) and output (<:) operators.
Type	`void inuint_byref(chanend c, unsigned &val)`
Parameters	`c` The channel end to stream data in on. `val` The variable to set to the received value.

Function	**outct**
Description	Streams out a control token on a channel end. Attempting to output a hardware control token causes an exception to be raised.
Type	`void outct(chanend c, unsigned char val)`
Parameters	`c` The channel end to stream data out on. `val` The value of the control token to output.

Function	chkct
Description	Checks for a control token of a given value. If the next byte in the channel is a control token which matches the expected value then it is input and discarded, otherwise an exception is raised.
Type	`void chkct(chanend c, unsigned char val)`
Parameters	`c` The channel end. `val` The expected control token value.

Function	inct
Description	Streams in a control token on a channel end. If the next byte in the channel is not a control token then an exception is raised, otherwise the value of the control token is returned.
Type	`unsigned char inct(chanend c)`
Parameters	`c` The channel end to stream data in on.
Returns	The received control token.

Function	inct_byref
Description	Streams in a control token on a channel end. The inputted value is written to `val`. If the next byte in the channel is not a control token then an exception is raised.
Type	`void inct_byref(chanend c, unsigned char &val)`
Parameters	`c` The channel end to stream data in on. `val` The variable to set to the received value.

Function	testct
Description	Tests whether the next byte on a channel end is a control token. The token is not discarded from the channel and is still available for input.
Type	`int testct(chanend c)`
Parameters	`c` The channel end to perform the test on.
Returns	1 if the next byte is a control token, 0 otherwise.

Function	testwct
Description	Tests whether the next word on a channel end contains a control token. If the word does contain a control token the position in the word is returned. No data is discarded from the channel.
Type	`int testwct(chanend c)`
Parameters	`c` The channel end to perform the test on.
Returns	The position of the first control token in the word (1-4) or 0 if the word contains no control tokens.

Function	**soutct**
Description	Outputs a control token on a streaming channel end. Attempting to output a hardware control token causes an exception to be raised. Attempting to output a `CT_END` or `CT_PAUSE` control token is invalid.
Type	`void soutct(streaming chanend c, unsigned char val)`
Parameters	`c` The channel end to stream data out on. `val` The value of the control token to output.

Function	**schkct**
Description	Checks for a control token of a given value on a streaming channel end. If the next byte in the channel is a control token which matches the expected value then it is input and discarded, otherwise an exception is raised.
Type	`void schkct(streaming chanend c, unsigned char val)`
Parameters	`c` The streaming channel end. `val` The expected control token value.

Function	**sinct**
Description	Inputs a control token on a streaming channel end. If the next byte in the channel is not a control token then an exception is raised, otherwise the value of the control token is returned.
Type	`unsigned char sinct(streaming chanend c)`
Parameters	`c` The streaming channel end to stream data in on.
Returns	The received control token.

Function	sinct_byref
Description	Inputs a control token on a streaming channel end. The inputted value is written to `val`. If the next byte in the channel is not a control token then an exception is raised.
Type	`void sinct_byref(streaming chanend c, unsigned char &val)`
Parameters	`c` The streaming channel end to stream data in on. `val` The variable to set to the received value.

Function	stestct
Description	Tests whether the next byte on a streaming channel end is a control token. The token is not discarded from the channel and is still available for input.
Type	`int stestct(streaming chanend c)`
Parameters	`c` The channel end to perform the test on.
Returns	1 if the next byte is a control token, 0 otherwise.

Function	stestwct
Description	Tests whether the next word on a streaming channel end contains a control token. If the word does contain a control token the position in the word is returned. No data is discarded from the channel.
Type	`int stestwct(streaming chanend c)`
Parameters	`c` The streaming channel end to perform the test on.
Returns	The position of the first control token in the word (1-4) or 0 if the word contains no control tokens.

Function	out_char_array	
Description	Output a block of data over a channel. A total of `size` bytes of data are output on the channel end. The call to out_char_array() must be matched with a call to in_char_array() on the other end of the channel. The number of bytes output must match the number of bytes input.	
Type	`transaction out_char_array(chanend c, const char src[size], unsigned size)`	
Parameters	`c`	The channel end to output on.
	`src`	The array of values to send.
	`size`	The number of bytes to output.

Function	in_char_array	
Description	Input a block of data from a channel. A total of `size` bytes of data are input on the channel end and stored in an array. The call to in_char_array() must be matched with a call to out_char_array() on the other end of the channel. The number of bytes input must match the number of bytes output.	
Type	`transaction in_char_array(chanend c, char dst[size], unsigned size)`	
Parameters	`c`	The channel end to input on.
	`dst`	The array to store the values input from on the channel.
	`size`	The number of bytes to input.

50.8 Predicate Functions

Function	**pinseq**
Description	Wait until the value on the port's pins equals the specified value. This function must be called as the expression of an input on a port. It causes the input to become ready when the value on the port's pins is equal to the least significant port-width bits of `val`.
Type	`void pinseq(unsigned val)`
Parameters	`val` The value to compare against.

Function	**pinsneq**
Description	Wait until the value on the port's pins does not equal the specified value. This function must be called as the expression of an input on a port. It causes the input to become ready when the value on the port's pins is not equal to the least significant port-width bits of `val`.
Type	`void pinsneq(unsigned val)`
Parameters	`val` The value to compare against.

Function	pinseq_at	
Description	Wait until the value on the port's pins equals the specified value and the port counter equals the specified time. This function must be called as the expression of an input on a unbuffered port. It causes the input to become ready when the value on the port's pins is equal to the least significant port-width bits of `val` and the port counter equals `time`.	
Type	`void pinseq_at(unsigned val, unsigned time)`	
Parameters	`val`	The value to compare against.
	`time`	The time at which to make the comparison.

Function	pinsneq_at	
Description	Wait until the value on the port's pins does not equal the specified value and the port counter equals the specified time. This function must be called as the expression of an input on a unbuffered port. It causes the input to become ready when the value on the port's pins is not equal to the least significant port-width bits of `val` and the port counter equals `time`.	
Type	`void pinsneq_at(unsigned val, unsigned time)`	
Parameters	`val`	The value to compare against.
	`time`	The time at which to make the comparison.

Function	**timerafter**
Description	Wait until the time of the timer equals the specified value. This function must be called as the expression of an input on a timer. It causes the input to become ready when timer's counter is interpreted as coming after the specified value timer is after the given value. A time A is considered to be after a time B if the expression is true.
Type	`void timerafter(unsigned val)`
Parameters	`val` The time to compare against.

50.9 XS1-S Functions

These functions to control the analogue-to-digital converter (ADC) on XS1-S devices.

Function	**enable_xs1_su_adc_input**	
Description	Enables the ADC input specified by `number`. Samples are sent to chanend `c`.	
Type	`void` `enable_xs1_su_adc_input(unsigned number, chanend c)`	
Parameters	`number`	The ADC input number.
	`c`	The channel connected to the XS1-SU ADC.

Function	**disable_xs1_su_adc_input**	
Description	Disables the ADC input specified by `number`.	
Type	`void` `disable_xs1_su_adc_input(unsigned number, chanend c)`	
Parameters	`number`	The ADC input number.
	`c`	The channel connected to the XS1-SU ADC.

Function	enable_xs1_su_adc_input_streaming
Description	Enables the ADC input specified by `number`. Samples are sent to chanend `c`.

Type	`void`
	`enable_xs1_su_adc_input_streaming(unsigned number,`
	`streaming chanend c)`

Parameters	`number` The ADC input number. `c` The channel connected to the XS1-SU ADC.

Function	disable_xs1_su_adc_input_streaming
Description	Disables the ADC input specified by `number`.

Type	`void`
	`disable_xs1_su_adc_input_streaming(unsigned number,`
	`streaming chanend c)`

Parameters	`number` The ADC input number. `c` The channel connected to the XS1-SU ADC.

50.10 Miscellaneous Functions

Function	**crc32**	

Description	Incorporate a word into a Cyclic Redundancy Checksum.
	The calculation performed is
	`for (int i = 0; i < 32; i++) {`
	`  int xorBit = (crc & 1);`
	`  checksum  = (checksum >> 1) | ((data & 1) << 31);`
	`  data = data >> 1;`
	`  if (xorBit)`
	`  checksum = checksum ^ poly;`
	`}`	

Type	`void crc32(unsigned &checksum, unsigned data, unsigned poly)`	
Parameters	`checksum`	The initial value of the checksum, which is updated with the new checksum.
	`data`	The data to compute the CRC over.
	`poly`	The polynomial to use when computing the CRC.

Function	**crc32_inc**	
Description	Incorporate a word into a Cyclic Redundancy Checksum (CRC), and simultaneously increment a register by a specified amount.	

Type	`void crc32_inc(unsigned int &checksum,`
	`  unsigned int data,`
	`  unsigned int poly,`
	`  unsigned int &value,`
	`  unsigned int increment)`	

Parameters	`checksum`	The inital value of the checksum, which is updated with the new checksum.
	`data`	The data to compute the CRC over.
	`poly`	The polynomial to use when computing the CRC.
	`value`	A value to be incremented.
	`increment`	The increment, one of bpw, 1, 2, 3, 4, 5, 6, 7, 8, 16, 24, 32.

Function	**crcn**
Description	Incorporate n-bits of a 32-bit word into a Cyclic Redundancy Checksum (CRC). Executing 32/N crcn calls sequentially has the same effect as executing a single crc call.
Type	`void crcn(unsigned int &checksum, unsigned int data, unsigned int poly, unsigned int n)`
Parameters	`checksum` The inital value of the checksum, which is updated with the new checksum. `data` The data to compute the CRC over. `poly` The polynomial to use when computing the CRC. `n` The number of lower bits of the data to incorporate.

Function	**crc8shr**
Description	Incorporate 8-bits of a word into a Cyclic Redundancy Checksum. The CRC is computed over the 8 least significant bits of the data and the data shifted right by 8 is returned. The calculation performed is `for (int i = 0; i < 8; i++) { int xorBit = (crc & 1); checksum = (checksum >> 1) | ((data & 1) << 31); data = data >> 1; if (xorBit) checksum = checksum ^ poly; }`
Type	`unsigned crc8shr(unsigned &checksum, unsigned data, unsigned poly)`
Parameters	`checksum` The initial value of the checksum which is updated with the new checksum. `data` The data. `poly` The polynomial to use when computing the CRC.
Returns	The data shifted right by 8.

Function	elate
Description	Raise an ET_ECALL exception if the specified time is less than the reference time.
Type	`void elate(unsigned int time)`
Parameters	`time` The time value.

Function	lmul
Description	Multiplies two words to produce a double-word and adds two single words. The high word and the low word of the result are returned. The multiplication is unsigned and cannot overflow. The calculation performed is `(uint64_t)a * (uint64_t)b + (uint64_t)c + (uint64_t)d`
Type	`{unsigned, unsigned} lmul(unsigned a, unsigned b, unsigned c, unsigned d)`
Returns	The high and low halves of the calculation respectively.

Function	lsats
Description	Perform saturation on a 64-bit value. If any arithmetic has overflowed beyond a given bit index, then the value is set to MININT or MAXINT, right shifted by the bit index.
Type	`signed long long lsats(signed long long value, unsigned int index)`
Parameters	`value` The 64-bit value to perform saturation on. `index` The bit index at which overflow is checked for.
Returns	The saturated 64-bit value.

Function	**lextract**
Description	Extract a bitfield from a 64-bit value.
Type	`unsigned int lextract(unsigned long long value, unsigned int position, unsigned int length)`
Parameters	`value` The value to extract the bitfield from. `position` The bit position of the field, which must be a value between 0 and bpw - 1, inclusive. `length` The length of the field, one of bpw, 1, 2, 3, 4, 5, 6, 7, 8, 16, 24, 32.
Returns	The value of the bitfield.

Function	**linsert**
Description	Insert a bitfield into a 64-bit value.
Type	`unsigned long long linsert(unsigned long long value, unsigned int bitfield, unsigned int position, unsigned int length)`
Parameters	`value` The 64-bit value to insert the bitfield in. `bitfield` The value of the bitfield. `position` The bit position of the field, which must be a value between 0 and bpw - 1, inclusive. `length` The length of the field, one of bpw, 1, 2, 3, 4, 5, 6, 7, 8, 16, 24, 32.
Returns	The 64-bit value with the inserted bitfield.

Function	mac
Description	Multiplies two unsigned words to produce a double-word and adds a double word. The high word and the low word of the result are returned. The calculation performed is: `(uint64_t)a * (uint64_t)b + (uint64_t)c<<32 + (uint64_t)d`
Type	`{unsigned, unsigned} mac(unsigned a, unsigned b, unsigned c, unsigned d)`
Returns	The high and low halves of the calculation respectively.

Function	macs
Description	Multiplies two signed words and adds the double word result to a double word. The high word and the low word of the result are returned. The calculation performed is: `(int64_t)a * (int64_t)b + (int64_t)c<<32 + (int64_t)d`
Type	`{signed, unsigned} macs(signed a, signed b, signed c, unsigned d)`
Returns	The high and low halves of the calculation respectively.

Function	**sext**
Description	Sign extends an input. The first argument is the value to sign extend. The second argument contains the bit position. All bits at a position higher or equal are set to the value of the bit one position lower. In effect, the lower b bits are interpreted as a signed integer. If b is less than 1 or greater than 32 then result is identical to argument a.
Type	`signed sext(unsigned a, unsigned b)`
Returns	The sign extended value.

Function	**zext**
Description	Zero extends an input. The first argument is the value to zero extend. The second argument contains the bit position. All bits at a position higher or equal are set to the zero. In effect, the lower b bits are interpreted as an unsigned integer. If b is less than 1 or greater than 32 then result is identical to argument a.
Type	`unsigned zext(unsigned a, unsigned b)`
Returns	The zero extended value.

Function	**unzip**
Description	Unzip a 64-bit value into two 32-bit values, with a granularity of bits, bit pairs, nibbles, bytes or byte pairs.
Type	`{unsigned int, unsigned int} unzip(unsigned long long value, unsigned int log_granularity)`
Parameters	`value` The 64-bit zipped value. `log_granularity` The logarithm of the granularity.
Returns	Two 32-bit unzipped values.

Function	**zip**
Description	Zip two 32-bit values into a single 64-bit value, with a granularity of bits, bit pairs, nibbles, bytes or byte pairs.
Type	`unsigned long long zip(unsigned int value1, unsigned int value2, unsigned int log_granularity)`
Parameters	`value1` The first 32-bit value. `value2` The second 32-bit value. `log_granularity` The logarithm of the granularity.
Returns	The 64-bit zipped value.

51 xCORE 32-Bit Application Binary Interface

Information on the XS1 32-ABI, the XE file format and System Call Interface is available in the *Tools Development Guide*[9].

[9] http://www.xmos.com/docnum/X9114

Part P

Platform Configuration

CONTENTS

- Describe a target platform
- xConnect link naming convention manual
- XN Specification

52 Describe a target platform

IN THIS CHAPTER

- Supported network topologies
- A board with two packages

Hardware platforms are described using XN. An XN file provides information to the XMOS compiler toolchain about the target hardware, including XMOS devices, ports, flash memories and oscillators.

The XMOS tools use the XN data to generate a platform-specific header file `<platform.h>`, and to compile, boot and debug multi-node programs.

52.1 Supported network topologies

To route messages across the xCONNECT Link network, the routing ID and routing table of each node on the network must configured. The tools use the information in the XN file to setup the routing for the network before running the application.

If the routing configuration is explicitly specified in the XN file, the tools use this configuration. If the routing configuration is omitted from the XN file the tools choose a suitable set routing IDs and routing tables based on the network topology. The tools can automatically compute routing configurations for the the following network topologies.

Figure 85: Topologies that can be automatically routed

Network Topology	Supported Configurations
Line	Not supported on XS1-G devices
Hypercube	Degree-2 (pair of nodes) Degree-3 (ring of 4 nodes) Degree-3 (cube of 8 nodes) Degree-4 (canonical cube of 16 nodes)
Hypercube with trees attached	Not supported on XS1-G devices

52.2 A board with two packages

Figure 86 illustrates a board containing two XMOS L8-64 devices arranged in a line. A suitable XN description is described below.

An XN file starts with an XML declaration.

```
<?xml version="1.0" encoding="UTF-8"?>
```

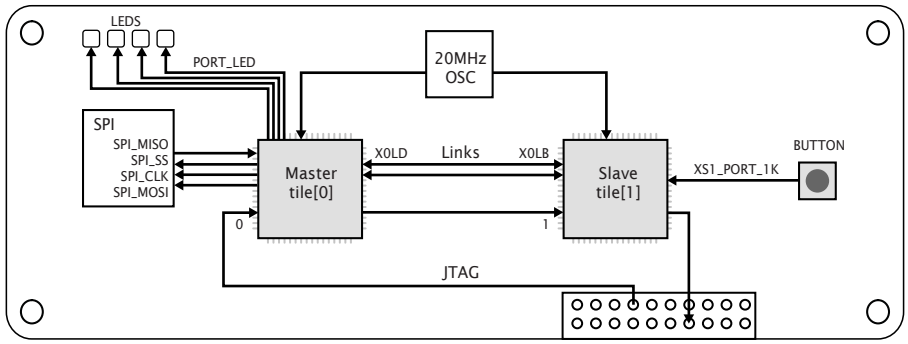


Figure 86: Example hardware platform

The following code provides the start of the network.

```
<Network xmlns="http://www.xmos.com"
         xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
         xsi:schemaLocation="http://www.xmos.com http://www.xmos.com">
```

The following code declares two xCORE Tiles. The declaration "`tileref tile[2];`" is exported to the header file `<platform.h>`.

```
<Declarations>
  <Declaration>tileref tile[2]</Declaration>
</Declarations>
```

The following code declares a package named `P1`, which contains a single node named `Master`.

```
<Packages>
  <Package Id="P1" Type="XS1-LnA-64-TQ128">
    <Nodes>
      <Node Id="Master" Type="XS1-L8A-64" InPackageId="0"
            Oscillator="20MHz" SystemFrequency="400MHz">
      <Boot>
        <Source Location="SPI:bootFlash"/>
        <Bootee NodeId="Slave" Tile="0"/>
      </Boot>
      <Tile Number="0" Reference="tile[0]">
        <Port Location="XS1_PORT_1A" Name="PORT_SPI_MISO"/>
        <Port Location="XS1_PORT_1B" Name="PORT_SPI_SS"/>
        <Port Location="XS1_PORT_1C" Name="PORT_SPI_CLK"/>
        <Port Location="XS1_PORT_1D" Name="PORT_SPI_MOSI"/>
        <Port Location="XS1_PORT_4A" Name="PORT_LED"/>
      </Tile>
    </Node>
  </Nodes>
</Package>
```

The node `Master` is a 400MHz XS1-L8A-64 device in a TQ128 package, clocked by a 20MHz oscillator. It is booted from an SPI device named "bootFlash" which has the class "SPIFlash".

The declaration of tile "0" is associated with `tile[0]` and the ports 1A, 1B, 1C, 1D and 4A are given symbolic names. These declarations are exported to the header file `<platform.h>`.

The following code declares a package named `P2`, which contains a single node named `Slave`.

```
  <Package Id="P2" Type="XS1-LnA-64-TQ128">
    <Nodes>
      <Node Id="Slave" Type="XS1-L8A-64" InPackageId="0"
            Oscillator="20Mhz" SystemFrequency="400MHz">
        <Boot>
          <Source Location="LINK"/>
        </Boot>
        <Tile Number="0" Reference="tile[1]">
          <Port Location="XS1_PORT_1K" Name="PORT_BUTTON"/>
        </Tile>
      </Node>
    </Nodes>
  </Package>
</Packages>
```

The node `Slave` is a 400MHz XS1-L8A-64 device in a TQ128 package, clocked by a 20MHz oscillator. It is booted from node `Master` over an xCONNECT Link.

The following code defines a 2-wire xCONNECT Link with, which connects the node `Master` on link X0LD to the node `Slave` on link `X0LB`.

```
<Links>
  <Link Encoding="2wire" Delays="4,4">
    <LinkEndpoint NodeId="Master" Link="X0LD"/>
    <LinkEndpoint NodeId="Slave" Link="X0LB"/>
  </Link>
</Links>
```

The links have intra-symbol and inter-symbol delays of 4 clock periods.

The following code specifies a list of components on the board that are connected to XMOS devices.

```
<ExternalDevices>
  <Device NodeId="Master" Tile="0" Name="bootFlash"
          Class="SPIFlash" Type="AT25FS010">
     <Attribute Name="PORT_SPI_MISO" Value="PORT_SPI_MISO"/>
     <Attribute Name="PORT_SPI_SS"   Value="PORT_SPI_SS"/>
     <Attribute Name="PORT_SPI_CLK"  Value="PORT_SPI_CLK"/>
     <Attribute Name="PORT_SPI_MOSI" Value="PORT_SPI_MOSI"/>
   </Device>
 </ExternalDevices>
```

A device named `bootFlash` is connected to xCORE Tile 0 on Node `Master`, and is given attributes that associate the four SPI pins on the device with ports. (The class `SPIFlash` is recognized by XFLASH.)

The following code describes the JTAG scan chain.

```
  <JTAGChain>
    <JTAGDevice NodeId="Master" Position="0"/>
    <JTAGDevice NodeId="Slave" Position="1"/>
  </JTAGChain>

</Network>
```

53 xConnect link naming convention manual

IN THIS CHAPTER

- xCORE “L” series example
- xCORE-200 series example

This document describes the naming conventions used for xConnect LinkEndpoints within the XN specification using xTIMEcomposer 14. The naming convention used for the xCORE-200 device has been simplified within this version of the Tools. The naming convention for the xCORE “L” device has not changed.

A `LinkEndpoint` element describes one end of an xConnect Link. Each endpoint associates a node identifier to a physical xConnect Link. The table below highlights the correct link name to use for the `Link` attribute within the `LinkEndpoint` element.

xConnect link Number	xCORE “L” series link name	xCORE-200 link name
0	XLC	XL0
1	XLD	XL1
2	XLA	XL2
3	XLB	XL3
4	XLG	XL4
5	XLH	XL5
6	XLE	XL6
7	XLF	XL7
8	N/A	XL8

An xCORE “L” series link name cannot be used within an xCORE-200 XN specification.

An xCORE-200 link name cannot be used within an xCORE “L” series XN specification.

53.1 xCORE “L” series example

The following example demonstrates how Node 0 Link number 4 is connected to Node 1 Link number 7 in an xCORE “L” series XN specification.

```
<Links>
  <Link Encoding="5wire" Delays="0,1">
    <LinkEndpoint NodeId="0" Link="XLG"/>
    <LinkEndpoint NodeId="1" Link="XLF"/>
  </Link>
</Links>
```

53.2 xCORE-200 series example

The following example demonstrates how Node 0 Link number 4 is connected to Node 1 Link number 7 in an xCORE-200 series XN specification.

```
<Links>
  <Link Encoding="5wire" Delays="4,4">
    <LinkEndpoint NodeId="0" Link="XL4"/>
    <LinkEndpoint NodeId="1" Link="XL7"/>
  </Link>
</Links>
```

54XN Specification

IN THIS CHAPTER

- Network Elements
- Declaration
- Package
- Node
- Link
- Device
- JTAGDevice

54.1 Network Elements

xTIMEcomposer supports a single XN file that contains a single network definition. The network definition is specified as follows:

```
<?xml version="1.0" encoding="UTF-8"?>
<Network xmlns="http://www.xmos.com"
         xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
         xsi:schemaLocation="http://www.xmos.com http://www.xmos.com">
```

The XN hierarchy of elements is given in Figure 87

54.2 Declaration

A `Declaration` element provides a symbolic name for one or more xCORE Tiles. A single name or an array of names is supported with the form:

`tileref` *identifier*

`tileref` *identifier* [*constant-expression*]

An equivalent declaration is exported to the header file `<platform.h>` for use in XC programs. A `tileref` declaration is associated with physical xCORE tiles by the reference attribute of a `Tile` element (see §54.4.1).

Example

```
<Declaration>tileref master</Declaration>
<Declaration>tileref tile[8]</Declaration>
```

Node	Number	Description	Section
Network	1	An xCORE network	
Declarations	0+		
Declaration	1+	xCORE Tile declaration	§??
Packages	1+		
Package	1+	Device package	§??
Nodes	1		
Node	1+	Node declaration	§??
Tile	1+	An xCORE Tile	§??
Port	0+	An xCORE symbolic port name	§??
Boot	0 or 1	Boot method	§??
Source	1	Binary location	§??
Bootee	0+	Nodes booted	§??
RoutingTable	0 or 1		
Bits	1		
Bit	1+	Direction for bit	§??
Links	1		
Link	1+	Direction for link	§??
Service	0+	Service declaration	§??
Chanend	1+	Chanend parameter	§??
Links	0 or 1		
Link	1+	xCONNECT Link declaration	§??
LinkEndpoint	2	xCONNECT Link endpoint	§??
ExternalDevices	0 or 1		
Device	1+	External device	§??
Attribute	0+	A device attribute	§??
JTAGChain	0 or 1		
JTAGDevice	1+	A device in the JTAG chain	§??

Figure 87: XN hierarchy of elements

54.3 Package

A `Package` element refers to a package file that describes the mapping from xCORE ports and links to the pins on the package.

Attribute	Required	Type	Description
Id	Yes	String	A name for the package. All package names in the network must be unique.
Type	Yes	String	The name of the XML package. The tools search for the file *type*`.pkg` in the path specified by `XCC_DEVICE_PATH`.

Figure 88: XN `Package` element

Example

```
<Package id="L2" Type="XS1-L2A-QF124">
```

The package named `L2` is described in the file `XS1-L2A-QF124.xml`.

54.4 Node

A `Node` element defines a set of xCORE Tiles in a network, all of which are connected to a single switch. XMOS devices such as the G4 or L1 are both examples of nodes.

The PLL registers can be configured automatically using the attributes `SystemFrequency`, `PllFeedbackDivMin` and `ReferenceFrequency`, or can be configured manually using the attributes `PllDividerStageOneReg`, `PllMultiplierStageReg`, `PllDividerStageTwoReg` and `RefDiv`. If any of the first three attributes are provided, none of the last four attributes may be provided, and vice versa.

The PLL oscillator input frequency may be specifed using the `Oscillator` or `OscillatorSrc` attribute. If the `Oscillator` attribute is provided the `OscillatorSrc` attribute must not be provided, and vice versa.

If manual configuration is used, the attributes `PllDividerStageOneReg`, `PllMultiplierStageReg`, `PllDividerStageTwoReg` and `RefDiv` must be provided and the PLL oscillator input frequency must be specifed. The tools use these values to set the PLL registers and reference clock divider. Information on the PLL dividers can be found in xCORE frequency control documents for XS1-G processors (see X3221) and XS1-L processors (see X1433).

If the oscillator frequency is specifed and none of the manual PLL attributes are provided, automatic configuration is used. The tools attempt to program the PLL registers such that the target system frequency is achieved, the PLL feedback divider is greater than or equal to the minimum value and the target reference clock frequency is achieved. If any of these constraints cannot be met, the tools issue a warning and report the actual values used.

If the oscillator frequency is not specified, the tools do not attempt to configure the PLL. The PLL registers remain at their initial values as determined by the mode pins.

A network may contain either XS1-L devices or XS1-G devices, but not both.

Example

```
<Node Id="MyL1" Type="XS1-L1A" Oscillator="20Mhz"
      SystemFrequency="410MHz" ReferenceFrequency="98.5Mhz">
```

The node named `MyL1` is an L1 device, as described in the file `config_XS1-L1A.xml`.

Attribute	Required	Type	Description
Id	No	String	A name for the node. All node names in the network must be unique.
Type	Yes	String	If type is `periph:XS1-SU` the node is a XS1-SU peripheral node. Otherwise the type specifies the name of an XML file that describes the node. The tools search for the file `config_`*type*`.xml` in the path specified by `XCC_DEVICE_PATH`.
Reference	Yes	String	Associates the node with a xCORE Tile indentifer specified in a `Declaration`. This attribute is only valid on nodes with type `periph:XS1-SU`.
RoutingId	No	Integer	The routing identifier on the xCONNECT Link network.
InPackageId	Yes	String	Maps the node to an element in the package file.
Oscillator	No	String	The PLL oscillator input frequency, specified as a number followed by either `MHz`, `KHz` or `Hz`.
OscillatorSrc	No	String	The name of the node which supplies the PLL oscillator input.
SystemFrequency	No	String	The system frequency, specified as a number followed by either `MHz`, `KHz` or `Hz`. Defaults to 400MHz if not set.
PllFeedbackDivMin	No	Integer	The minimum allowable PLL feedback divider. Defaults to 1 if not set.
ReferenceFrequency	No	String	A reference clock frequency, specified as a number followed by either `MHz`, `KHz` or `Hz`. Defaults to 100MHz if not set.
PllDividerStageOneReg	No	Integer	The PLL divider stage 1 register value.
PllMultiplierStageReg	No	Integer	The PLL multiplier stage register value.
PllDividerStageTwoReg	No	Integer	The PLL divider stage 2 register value.
RefDiv	No	Integer	`SystemFrequency / RefDiv = ReferenceFrequency`

Figure 89: XN `Node` element

54.4.1 Tile

A `Tile` element describes the properties of a single xCORE Tile.

Example

```
<Tile Number="0" Reference="tile[0]">
```

Attribute	Required	Type	Description
Number	Yes	Integer	The unique number for the tile in the node. A value between 0 and *n*-1 where *n* is the number of tiles as defined in the node's XML file.
Reference	No	String	Associates the tile with an identifier with the form `tile[n]` in a `Declaration`. A tile may be associated with at most one identifier.

Figure 90: XN `Tileref` element

54.4.2 Port

A `Port` element provides a symbolic name for a port.

Attribute	Required	Type	Description
Location	Yes	String	A port identifier defined in the standard header file `<xs1.h>`. The ports are described in the XC manual (see X1009).
Name	Yes	String	A valid C preprocessor identifier. All port names declared in the network must be unique.

Figure 91: XN `Port` element

Example

```
<Port Location="XS1_PORT_1I" Name="PORT_UART_TX"/>
<Port Location="XS1_PORT_1J" Name="PORT_UART_RX"/>
```

54.4.3 Boot

A `Boot` element defines the how the node is booted. It contains one `Source` element (see §54.4.4) and zero or more `Bootee` elements (see §54.4.5) that are booted over xCONNECT Links. If the source specifies an xCONNECT Link, no `Bootee` elements may be specified. In a line of XS1-L devices, bootees must be contiguous to the device booting from SPI.

The XMOS tools require a `Boot` element to be able to boot programs from flash memory (see §26.1).

54.4.4 Source

A `Source` element specifies the location from which the node boots. It has the following attributes.

Attribute	Required	Type	Description
Location	Yes	String	Has the form `SPI:` or `LINK`. The `device-name` must be declared in the set of `Device` elements.

Figure 92: XN `Source` element

Only XMOS XS1-L devices can be configured to boot over xCONNECT Links.

Example

```
<Source Location="SPI:bootFlash"/>
```

54.4.5 Bootee

A `Bootee` element specifies another node in the system that this node boots via an xCONNECT Link. If more than one xCONNECT Link is configured between this node and one of its bootees (see §54.5 and §54.5.1), the tools pick one to use for booting.

Figure 93: XN `Bootee` element

Attribute	Required	Type	Description
NodeId	Yes	String	A valid identifier for another node.

Example

```
<Bootee NodeId="Slave">
```

54.4.6 Bit

A `Bit` element specifies the direction for messages whose first mismatching bit matches the specified bit number.

Figure 94: XN `Bit` element

Attribute	Required	Type	Description
number	Yes	Integer	The bit number, numbered from the least significant bit.
direction	Yes	Integer	The direction to route messages.

Example

```
<Bit number="1" direction="0"/>
```

54.4.7 Link

When it appears within a `RoutingTable` element, a `Link` element specifies the direction of an xCONNECT Link.

Example

```
<Link number="XLA" direction="2"/>
```

Attribute	Required	Type	Description
name	Yes	String	A link identifier in the form X*n*L*m* where *n* denotes a tile number and *m* the link letter. See the corresponding package datasheet for available link pinouts.
direction	Yes	Integer	The direction of the link.

Figure 95: XN `Link` element

54.4.8 Service

A `Service` element specifies an XC service function provided by a node.

Attribute	Required	Type	Description
Proto	Yes	String	The prototype for the service function, excluding the `service` keyword. This prototype is exported to the header file `<platform.h>` for use in XC programs.

Figure 96: XN `Service` element

Example

```
<Service Proto="service_function(chanend c1, chanend c2)">
```

54.4.9 Chanend

A `Chanend` element describes a channel end parameter to an XC service function.

Attribute	Required	Type	Description
Indentifier	Yes	String	The identifier for the chanend argument in the service function prototype.
end	Yes	Integer	The number of the channel end on the current node.
remote	Yes	Integer	The number of the remote channel end that is connected to the channel end on the current node.

Figure 97: XN `Service` element

Example

```
<Chanend Identifier="c" end="23" remote="5"/>
```

54.5 Link

xCONNECT Links are described in the system specification documents (XS1-G: X7507, XS1-L: X1151) and link performance documents (XS1-G: X7561, XS1-L: X2999).

A Link element describes the characteristics of an xCONNECT Link. It must contain exactly two `LinkEndpoint` children (see §54.5.1).

Figure 98: XN `Link` element

Attribute	Required	Type	Description
Encoding	Yes	String	Must be either `2wire` or `5wire`.
Delays	Yes	String	Of the form *x*,*y* where *x* specifies the inter delay value for the endpoint, and *y* specifies the intra delay value for the endpoint. If a value for *y* is omitted, `x,1` is used. If both values are omitted, `1,1` is used.
Flags	No	String	Specifies additional properties of the link. Use the value `XSCOPE` to specify a link used to send xSCOPE trace information.

Example

```
<Link Encoding="2wire" Delays="4,4">
```

54.5.1 LinkEndpoint

A `LinkEndpoint` describes one end of an xCONNECT Link, the details of which can be found in the system specification documents (XS1-G: X7507, XS1-L: X1151). Each endpoint associates a node identifier to a physical xCONNECT Link.

Figure 99: XN `LinkEndpoint` element

Attribute	Required	Type	Description
NodeID	No	String	A valid node identifier.
Link	No	String	A link identifier in the form X*n*L*m* where *n* denotes a tile number and *m* the link letter. See the corresponding package datasheet for available link pinouts.
RoutingId	No	Integer	The routing identifier on the xCONNECT Link network.
Chanend	No	Integer	A channel end.
BootRomEnabled	No	Integer	Link is enabled at boot and can therefore be considered for use within the spanning network.

An endpoint is usually described as a combination of a node identifier and link identifier. For a streaming debug link, one of the endpoints must be described as a combination of a routing identifier and a channel end.

Example

```
<LinkEndpoint NodeId="0" Link="X0LD"/>
<LinkEndpoint RoutingId="0x8000" Chanend="1">
```

54.6 Device

A `Device` element describes a device attached to an xCORE Tile that is not connected directly to an xCONNECT Link.

Figure 100: XN `Device` element

Attribute	Required	Type	Description
Name	Yes	String	An identifier that names the device.
NodeId	Yes	String	The identifier for the node that the device is connected to.
Tile	Yes	Integer	The tile in the node that the device is connected to.
Class	Yes	String	The class of the device.
Type	No	String	The type of the device (class dependent).

xTIMEcomposer recognizes the following attribute values for the attribute name `Class`:

`SPIFlash`
: Device is SPI flash memory

`SQIFlash`
: Device is QuadSPI flash memory

`ARMBridge`
: Device is ARM flash memory

Use the Type attribute to identify the model of the flash device.

54.6.1 Attribute

An `Attribute` element describes one aspect of a `Device` (see §54.6).

Figure 101: XN `Attribute` element

Attribute	Required	Type	Description
Name	Yes	String	Specifies an attribute of the device.
Value	Yes	String	Specifies a value associated with the attribute.

xTIMEcomposer supports the following attribute names for the device class `SPIFlash`:

`PORT_SPI_MISO`
: SPI Master In Slave Out signal.

`PORT_SPI_SS`
: SPI Slave Select signal.

`PORT_SPI_CLK`
: SPI Clock signal.

`PORT_SPI_MOSI`
: SPI Master Out Slave In signal.

Example

```
<Attribute Name="PORT_SPI_MISO" Value="PORT_SPI_MISO"/>
```

xTIMEcomposer supports the following attribute names for the device class `SQIFlash`:

`PORT_SQI_CS`
: QuadSPI Chip Select signal.

`PORT_SQI_SCLK`
: QuadSPI Clock signal.

`PORT_SQI_SIO`
: QuadSPI In/Out signal.

Example

```
<Attribute Name="PORT_SQI_SIO" Value="PORT_SQI_SIO"/>
```

54.7 JTAGDevice

xTIMEcomposer loads and debugs programs on target hardware using JTAG. The JTAGChain element describes a device in the JTAG chain. The order of these elements defines their order in the JTAG chain.

Figure 102: XN `JTAGDevice` element

Attribute	Required	Type	Description
NodeID	Yes	String	A valid node identifier.

Example

```
<!-- N1 comes before N2 in the JTAG chain -->
<JTAGDevice NodeId="N1">
<JTAGDevice NodeId="N2">
```

XMOS®

Copyright © 2015, All Rights Reserved.

Xmos Ltd. is the owner or licensee of this design, code, or Information (collectively, the "Information") and is providing it to you "AS IS" with no warranty of any kind, express or implied and shall have no liability in relation to its use. Xmos Ltd. makes no representation that the Information, or any particular implementation thereof, is or will be free from any claims of infringement and again, shall have no liability in relation to any such claims.

XMOS and the XMOS logo are registered trademarks of Xmos Ltd. in the United Kingdom and other countries, and may not be used without written permission. All other trademarks are property of their respective owners. Where those designations appear in this book, and XMOS was aware of a trademark claim, the designations have been printed with initial capital letters or in all capitals.