

Tools User Guide

Document Number: X1013E

Publication Date: 2012/8/2
XMOS © 2012, All Rights Reserved.



Table of Contents

A	Installation	9
1	System requirements for running the X MOS Tools	10
2	Installation Instructions	11
2.1	Install the tools	11
2.2	Install the USB drivers	12
B	Quick Start	13
3	Get started with the X MOS Tools	14
3.1	Start the XDE	14
3.1.1	Register Your Tools	14
3.2	Start the command-line tools	15
3.3	Quick Start Menu	15
4	Frequently Used Commands	16
4.1	XCC	16
4.2	XRUN	16
4.3	XGDB	17
4.4	XSIM	17
C	Developing in the XDE	18
5	Import or export a project in the XDE	19
5.1	Import a project	19
5.2	Export a project	19
D	Compilation	20
6	Use the XDE to build a project	21
7	XCC Pragma Directives	22
8	XCC command-line options	24
8.1	Overall Options	24
8.2	Warning Options	27
8.3	Debugging Options	30
8.4	Optimization Options	30
8.5	Preprocessor Options	31
8.6	Linker And Mapper Options	32
8.7	Directory Options	33
8.8	Environment Variables Affecting XCC	33
8.9	Board Support Provided by <platform.h>	34

9	Using XMOS Makefiles	35
9.1	Projects, Applications and Modules	35
9.1.1	Example Structure	37
9.2	The Application Makefile	37
9.3	The Project Makefile	39
9.4	The module_build_info file	40
10	Using XMOS Makefiles to create binary libraries	41
10.1	The module_build_info file	41
10.2	The module Makefile	42
10.3	Using the module	42
E	Timing	43
11	Use the XDE to time a program	44
11.1	Launch the timing analyzer	44
11.2	Time a section of code	45
11.2.1	Visualize a route	46
11.2.2	The Visualizations view	46
11.3	Specify timing requirements	47
11.4	Add program execution information	47
11.4.1	Refine the worst-case analysis	48
11.5	Validate timing requirements during compilation	48
F	Run on Hardware	50
12	Use the XDE to run a program	51
12.1	Create a Run Configuration	51
12.2	Re-run a program	52
13	XRUN Command-Line Manual	53
13.1	Overall Options	53
13.2	Target Options	53
13.3	Debugging Options	54
13.4	XScope Options	54
G	Application Instrumentation and Tuning	56
14	Use the XDE and XScope to trace data in real-time	57
14.1	Instrument a program	57
14.2	Configure and run a program with tracing enabled	59
14.3	Analyze data offline	59
14.4	Analyze data in real-time	60
14.4.1	Capture control	61
14.4.2	Signal Control	62
14.4.3	Trigger Control	62
14.4.4	Timebase Control	63
14.4.5	Screen Control	63
14.5	Trace using the UART interface	64

15 XScope performance figures	65
15.1 Transfer rates between the XCore and XTAG-2	65
15.2 Transfer rates between the XTAG-2 and Host PC	66
16 XScope Library API	67
16.1 Functions	67
16.2 Enumerations	68
 H Simulation	 70
17 Use the XDE to simulate a program	71
17.1 Configure the simulator	71
17.2 Trace a signal	72
17.2.1 Enable signal tracing	72
17.2.2 View a trace file	73
17.2.3 View a signal	73
17.3 Set up a loopback	74
17.4 Configure a simulator plugin	75
18 XSIM Command-Line Manual	76
18.1 Overall Options	76
18.2 Warning Options	76
18.3 Tracing Options	77
18.4 Loopback Plugin Options	79
 I Debugging	 81
19 Use the XDE to debug a program	82
19.1 Launch the debugger	83
19.2 Control program execution	83
19.3 Examine a suspended program	84
19.4 Set a breakpoint	86
19.5 View disassembled code	87
20 Debug with printf in real-time	88
20.1 Redirect stdout and stderr to the XTAG-2	89
20.2 Run a program with XTAG-2 output enabled	90
20.3 Output using the UART interface	90
 J Flash Programming	 92
21 Design and manufacture systems with flash memory	93
21.1 Boot a program from flash memory	93
21.2 Generate a flash image for manufacture	94
21.3 Perform an in-field upgrade	94
21.3.1 Write a program that upgrades itself	94
21.3.2 Build and deploy the upgrader	96
21.4 Customize the flash loader	96
21.4.1 Build the loader	97

21.4.2 Add additional images	97
22 libflash API	98
22.1 General Operations	98
22.2 Boot Partition Functions	99
22.3 Data Partition Functions	101
22.3.1 Page-Level Functions	101
22.3.2 Sector-Level Functions	102
23 List of devices natively supported by libflash	103
24 Add support for a new flash device	104
24.1 Libflash Device ID	105
24.2 Page Size and Number of Pages	105
24.3 Address Size	106
24.4 Clock Rate	106
24.5 Read Device ID	107
24.6 Sector Erase	108
24.7 Write Enable/Disable	108
24.8 Memory Protection	109
24.9 Programming Command	110
24.10 Read Data	111
24.11 Sector Information	111
24.12 Status Register Bits	112
24.13 Add Support to the XMOS Tools	113
24.14 Select a Flash Device	114
25 XFLASH Command-Line Manual	115
25.1 Overall Options	115
25.2 Target Options	116
25.3 Security Options	117
25.4 Programming Options	117
K Security and OTP Programming	119
26 Safeguard IP and device authenticity	120
26.1 The XMOS AES module	121
26.2 Develop with the AES module enabled	122
26.3 Production flash programming flow	123
26.4 Production OTP programming flow	124
27 XBURN Command-Line Manual	125
27.1 Overall Options	125
27.2 Security Register Options	126
27.3 Target Options	126
27.4 Programming Options	127
L Programming in C/XC	128
28 Calling between C/C++ and XC	129

28.1 Passing arguments from XC to C/C++	129
28.2 Passing arguments from C/C++ to XC	129
29 XC Implementation-Defined Behavior	130
30 C Implementation-Defined Behavior	132
30.1 Environment	132
30.2 Identifiers	133
30.3 Characters	133
30.4 Floating point	134
30.5 Hints	134
30.6 Preprocessing directives	134
30.7 Library functions	134
30.8 Locale-Specific Behavior	138
31 C and C++ Language Reference	141
31.1 Standards	141
31.2 Books	141
31.3 Online	141
M Programming in Assembly	142
32 Inline Assembly	143
33 Make assembly programs compatible with the XMOS XS1 ABI	145
33.1 Symbols	145
33.2 Alignment	145
33.3 Sections	146
33.3.1 Data	146
33.3.2 Arrays	147
33.4 Functions	147
33.4.1 Parameters and return values	147
33.4.2 Caller and callee save registers	148
33.4.3 Resource usage	148
33.4.4 Side effects	149
33.5 Elimination blocks	150
33.6 Typestrings	150
33.7 Example	151
34 Assembly Programming Manual	153
34.1 Lexical Conventions	153
34.1.1 Comments	153
34.1.2 Symbol Names	153
34.1.3 Directives	153
34.1.4 Constants	154
34.2 Sections and Relocations	154
34.3 Symbols	154
34.3.1 Attributes	154
34.4 Labels	155
34.5 Expressions	155
34.6 Directives	156

34.6.1 align	156
34.6.2 ascii, asciiz	156
34.6.3 byte, short, int, long, word	157
34.6.4 file	157
34.6.5 loc	157
34.6.6 weak	158
34.6.7 globl, global, extern, locl, local	158
34.6.8 typestring	159
34.6.9 ident, core, corerev	159
34.6.10 section, pushsection, popsection	159
34.6.11 text	160
34.6.12 set, linkset	160
34.6.13 cc_top, cc_bottom	161
34.6.14 scheduling	162
34.6.15 syntax	162
34.6.16 assert	162
34.6.17 language Directives	162
34.6.18 XMOS Timing Analyzer Directives	164
34.6.19 leb128, sleb128	164
34.6.20 pace, skip	165
34.6.21 type	165
34.6.22 size	165
34.6.23 jmptable, jmptable32	165
34.7 Instructions	166
34.7.1 Data Access	167
34.7.2 Branching, Jumping and Calling	168
34.7.3 Data Manipulation	168
34.7.4 Concurrency and Thread Synchronization	169
34.7.5 Communication	170
34.7.6 Resource Operations	170
34.7.7 Event Handling	171
34.7.8 Interrupts, Exceptions and Kernel Calls	171
34.7.9 Debugging	172
34.7.10 pseudo Instructions	172
34.8 Assembly Program	173

N Programming for XS1 Devices 174

35 XCC Target-Dependent Behavior for XS1 Devices	175
35.1 Support for Clock Blocks	175
35.2 Support for Ports	176
35.2.1 Serialization	176
35.2.2 Timestamping	176
35.2.3 Changing Direction of Buffered Ports	177
35.3 Channel Communication	177

36 XS1 Data Types 178

37 XS1 port-to-pin mapping 179

38 XS1 Library 181

38.1 Data types	181
38.2 Port Configuration Functions	181
38.3 Clock Configuration Functions	190
38.4 Port Manipulation Functions	194
38.5 Clock Manipulation Functions	197
38.6 Thread/Core Control Functions	197
38.7 Channel Functions	205
38.8 Predicate Functions	211
38.9 XS1-S Functions	213
38.10 Miscellaneous Functions	214
39 XMOS 32-Bit Application Binary Interface	217
O Platform Configuration	218
40 Describe a target platform	219
40.1 Supported network topologies	219
40.2 A board with two packages	219
41 XN Specification	224
41.1 Network Elements	224
41.2 Declaration	224
41.3 Package	225
41.4 Node	226
41.4.1 Core	227
41.4.2 Port	227
41.4.3 Boot	228
41.4.4 Source	228
41.4.5 Bootee	228
41.4.6 Service	229
41.4.7 Chanend	229
41.5 Link	230
41.5.1 LinkEndpoint	230
41.6 Device	231
41.6.1 Attribute	231
41.7 JTAGDevice	232

Part A

Installation

CONTENTS

- ▶ [System requirements for running the XMOS Tools](#)
- ▶ [Installation Instructions](#)

1 System requirements for running the XMOS Tools

The XMOS 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*



Mac OS X 10.5 +

- ▶ *Intel Processors*



Linux CentOS 5

- ▶ *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
-

All of the XMOS tools and 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 XMOS icon into your Applications folder.

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



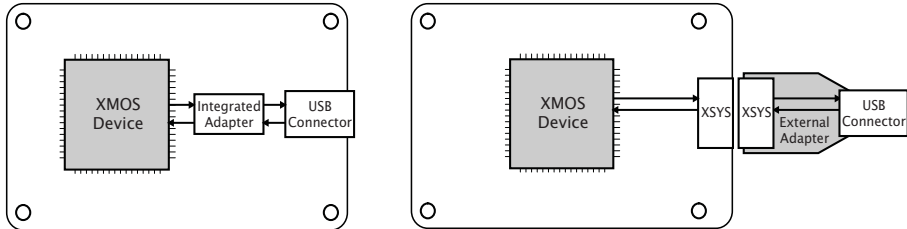
On Linux:

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

The XMOS tools interface to development boards over USB. Some boards provide an integrated debug adapter, and others require an external adapter that connects to the board via an XSYS connector, as shown in Figure 1.

Figure 1:
Adapter configurations
used with
X MOS
development
boards



The XMOS tools support adapters based on either FTDI or XMOS USB-to-JTAG chips. Consult your board manual to determine which driver to use.



On Windows:

The JTAG drivers are installed by the tools installer. Plug your XMOS 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 [X2612](#)).

Part B

Quick Start

CONTENTS

- ▶ [Get started with the XMOS Tools](#)
- ▶ [Frequently Used Commands](#)

3 Get started with the XMOS Tools

IN THIS CHAPTER

- ▶ Start the XDE
 - ▶ Start the command-line tools
 - ▶ Quick Start Menu
-

3.1 Start the XDE

To start the XDE:



In Windows:

Choose **Start ▶ Programs ▶ XMOS ▶ Development Tools 11.11 ▶ XDE**.



In OS X:

Open a new Finder window, navigate to the **Applications** folder, open the folder **XMOS_11.11** and double-click on the **XDE.app** icon.



In Linux:

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

- ▶ `source SetEnv`
- ▶ `xde`

3.1.1 Register Your Tools

The first time you start the XDE, you are asked whether you wish to register the tools with your XMOS account. Registration provides benefits such as automatic notifications of document and software updates directly within the XDE, and the option to manage account settings from within the tools.

To register later, choose **Help ▶ Registration Information**.

3.2 Start the command-line tools

The XMOS command-line tools use a set of environment variables when searching for header files, libraries and target devices (see §8.8). To add the XMOS tools to the path and configure the default set of environment variables:



In Windows:

Choose **Start ► Programs ► XMOS Development Tools 11.11 ► Command Prompt**.



In OS X:

Open a Terminal window, change to the installation directory and either open from the Finder, 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 Quick Start Menu

The **Quick Start Menu** in the XDE *Developer Column* provides a convenient starting point for all users, including developers who are new to XMOS and experienced users.

Developers with an XMOS development board, can use the page to create new projects for their board or find kit-specific documentation and tutorials. Those who do not have a board you can follow a tool tutorial using the simulator.

New developers can use the *Quick Start Menu* to link to key documents and information that will help them use the XMOS developer tools. For existing users there are links to documents that explain changes between the latest release and previous versions of the toolchain.

To make the most of the *Quick Start Menu* select your level of expertise or interest and then follow the instructions on screen.

In the XDE, choose **Help ► Quick Start** to view the Quick Start Menu in the *Developer Column*.

4 Frequently Used Commands

IN THIS CHAPTER

- ▶ XCC
 - ▶ XRUN
 - ▶ XGDB
 - ▶ XSIM
-

This document summarizes a number of frequently-used commands.

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 XMOS 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.

Part C

Developing in the XDE

CONTENTS

- [Import or export a project in the XDE](#)

5 Import or export a project in the XDE

IN THIS CHAPTER

- Import a project
 - Export a project
-

The XDE makes it easy to share projects with other developers.

5.1 Import a project

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**.

5.2 Export a project

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 the XDE to build a project](#)
- ▶ [XCC Pragma Directives](#)
- ▶ [XCC command-line options](#)
- ▶ [Using XMOS Makefiles](#)
- ▶ [Using XMOS Makefiles to create binary libraries](#)

6 Use the XDE to build a project



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

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

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

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

7 XCC Pragma Directives

The XMOS toolchain supports the following pragmas.

`#pragma unsafe arrays`

(XC Only) This pragma disables the generation of run-time safety checks that prevent indexing an invalid array element within the scope of the next `do`, `while` or `for` statement in the current function; outside of a function the pragma applies to 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 (`ints`) of stack space for the next function declaration in the current translation unit.

`#pragma stackcalls n`

(XC only) This pragma allocates `n` words (`ints`) 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.

8 XCC 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 XMOS C, C++ and XC compilers. Typical usage results in preprocessing, compilation, assembly, linking, and mapping code and data onto cores. 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.

8.1 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 core.

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

-xlanguage 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
none (turn off language specification)
```

Figure 2: File extensions recognized by XCC and their meaning	Extension	Type of File	Preprocessed by XCC
	.xc	XC source code	Y
	.c	C source code	Y
	.cpp	C++ source code (for compatability, the extensions cc, cp, c++, C and cxx are also recognized)	Y
	.S	Assembly code	Y
	.xta	XMOs Timing Analyzer script	N
	.xn	XMOs Network Description	N
	.xi	XC source code	N
	.i	C source code	N
	.ii	C++ source code	N
	.s	Assembly code	N
	<i>other</i>	Object file .o be given to the linker	N

-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 destination 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 §35.3.

-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 §8.8).

- `-fllvm` Use the experimental LLVM compiler backend. The LLVM backend adds support for floating point and long long data types. The following features are currently unsupported:
- ▶ Select functions
 - ▶ Boolean guards on default cases in select statements
 - ▶ XTA information
- `-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.

8.2 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.
- `-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 §8.7.

-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.

8.3 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 separated 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.

8.4 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.
- fconst-fold-addr Enable constant folding of expressions that are a stack location plus a constant offset or a label plus a constant offset. This is default at O1 and above, but in some cases disabling can improve performance.
If multiple -O options are used, the last one is effective.

8.5 Preprocessor Options

The following options control the preprocessor.

- E Preprocesses only, then exit.
- D*name* Predefines *name* as a macro with definition 1.
- D*name=definition* Tokenizes and preprocesses the contents of *definition* as if it appeared in a #define directive.
- U*name* 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 overridden with -MF.

-MMD	The same as -MD expect that dependencies on system headers are ignored.
-MF <i>file</i>	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 <code>make</code> gives if header files are removed without updating the Makefile to match.
-MT <i>file</i>	Specifies the target of the rule emitted by dependency generation.

8.6 Linker And Mapper Options

The following options control the linker/mapper.

-l <i>library</i>	Searches the library <i>library</i> when linking. The linker searches and processes libraries and object files in the order specified. The actual library name searched for is <code>lib<i>library</i>.a</code> . 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.

8.7 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.
- `-iquoteddir` 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`.

8.8 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 separated 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.

8.9 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 `core` (see §41.2).
- ▶ Macro definitions of port names (see §41.4.2).

9 Using XMOS Makefiles

IN THIS CHAPTER

- Projects, Applications and Modules
 - The Application Makefile
 - The Project Makefile
 - The `module_build_info` file
-

Projects created by the XMOS Development Environment have their build controlled by Makefiles. These Makefiles execute the build using the program `xmake` which is a port of Gnu Make¹. The build is executable either from within the XDE or from the command line by calling `xmake` directly.

You do not need to understand the Gnu Makefile language to develop applications using the XMOS tools. The **common XMOS Makefile** provides support for projects, applications and modules. You need only specify the required properties of the build in **Project Makefiles** and **Application Makefiles**.

9.1 Projects, 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*, *projects*, *applications* and *modules*.

Workspace

A *workspace* is a container for several projects.

Projects

A *project* is a directory possibly containing several applications and modules plus other files relating to a particular project. A project may contain the code for a particular board or reference design or be a software component containing modules for other projects to use.

Applications

An *application* is a directory 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 the XDE.

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

¹<http://www.gnu.org/software/make/>

convention module directories start with the prefix `module_`. These modules appear at the top level in the project explorer in the XDE.

9.1.1 Example Structure

An example workspace structure is shown below.

```
sw_avb/  
  app_avb_demo1/  
  app_avb_demo2/  
  module_avb1/  
  module_avb2/  
  doc/  
sc_tcp/  
  module_tcp/  
  module_zeroconf/  
sc_ethernet/  
  module_ethernet/
```

There are three projects within this workspace: `sw_avb`, `sc_tcp` and `sc_ethernet`. The `sw_avb` project contains two applications, each of which builds to a separate binary. These applications can use source from the modules within the projects and can use modules from their own project (`module_avb1` and `module_avb2`) and from other projects (`module_tcp`, `module_zeroconf` and `module_ethernet`).

Alternatively, a workspace may be structured in the following way:

```
app_avb_demo1/  
app_avb_demo2/  
module_avb1/  
module_avb2/  
doc/  
module_tcp/  
module_zeroconf/  
module_ethernet/
```

In this case, all applications and modules are at the top level of the workspace.

9.2 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 §10 for details.

9.3 The Project Makefile

As well as each application having its own Makefile, the project should have a Makefile at the top-level. This Makefile controls building the applications within the project. It has one variable assignment within it to do this:

`BUILD_SUBDIRS`

Specifies a space-separated list of application directories to build.

9.4 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.

10 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).

10.1 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
```

10.2 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
```

10.3 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 the XDE to time a program](#)

11 Use the XDE 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 XMOS Timing Analyzer lets you determine the time taken to execute code on your target platform. Due to the deterministic nature of the XMOS 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.

11.1 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**. The XDE creates a new configuration and displays the default settings in the right panel.
4. The XDE 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**.

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

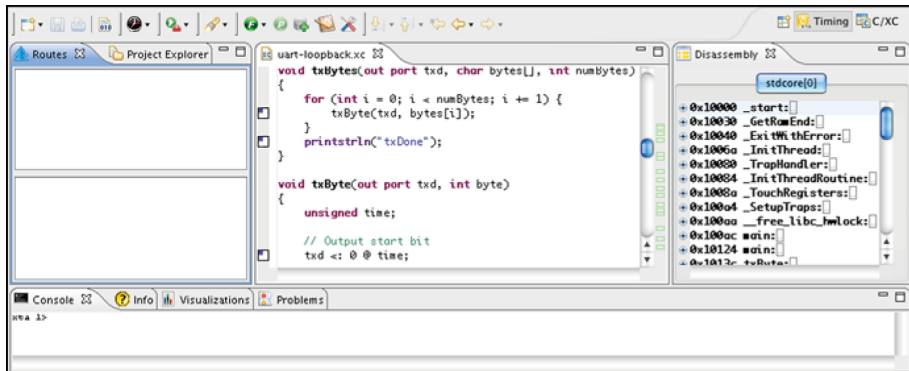


Figure 3:
Timing
perspective



The XDE 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.

11.2 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**. The XDE displays a green dot in the top-right quarter of the marker.



2. Right-click on an endpoint marker and choose **Set to endpoint**. The XDE 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. The XDE 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.

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

11.2.1 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.

11.2.2 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.

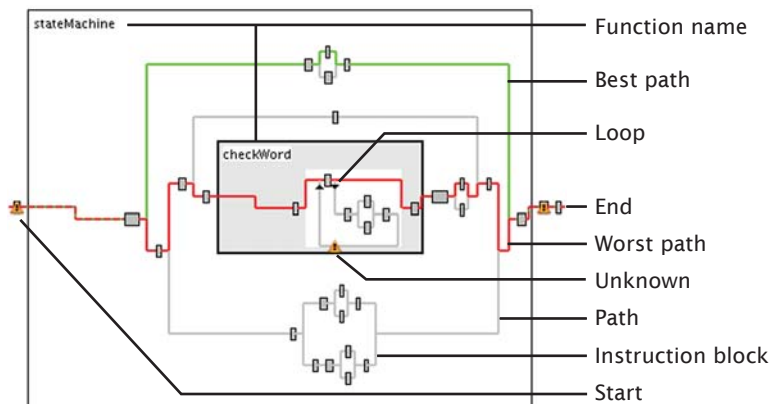






Figure 4:
Visualizations
view

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.

11.3 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**. The XDE updates the status of the route.

11.4 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**.

11.4.1 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**.

11.5 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. The XDE 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.

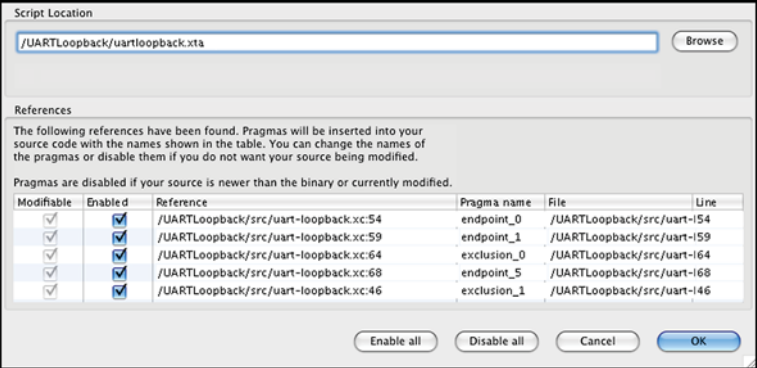


Figure 5:
Script
Options
dialog box

Part F

Run on Hardware

CONTENTS

- ▶ [Use the XDE to run a program](#)
- ▶ [XRUN Command-Line Manual](#)

12 Use the XDE to run a program

IN THIS CHAPTER

- Create a Run Configuration
 - Re-run a program
-

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

12.1 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**.
The XDE creates a new configuration and displays the default settings in the right panel, as shown in Figure 6.
4. In **Name**, enter a name for the configuration.
5. The XDE 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**.

The XDE loads your executable, displaying any output generated by your program in the **Console**.

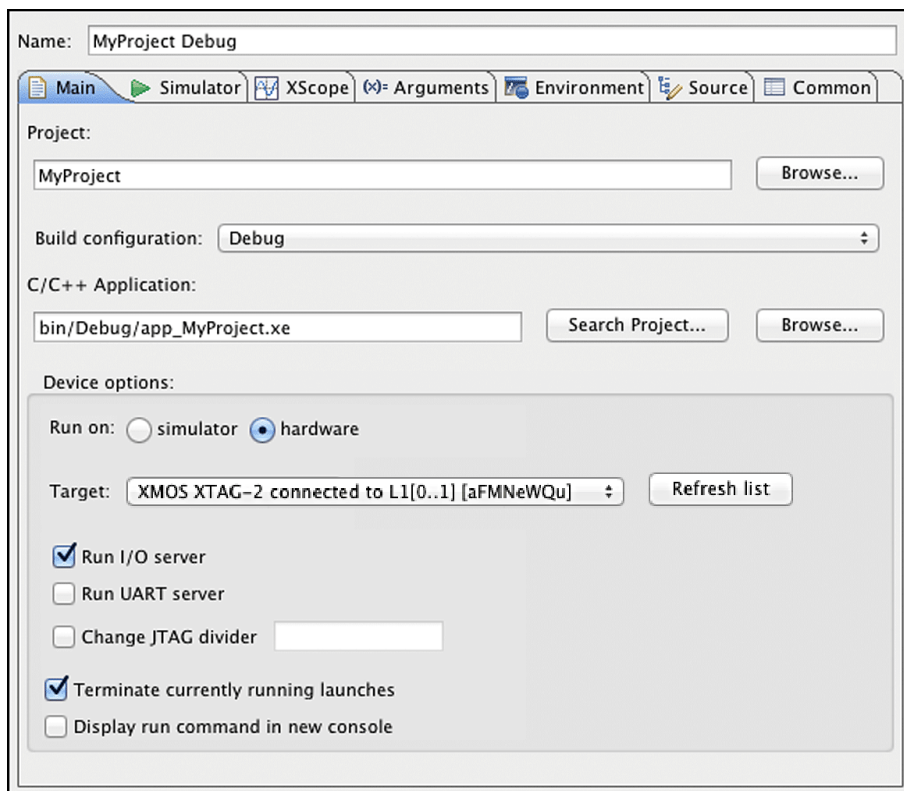


Figure 6:
Run Configu-
ration
window

12.2 Re-run a program



The XDE 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.

13XRUN 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).

13.1 Overall Options

The following options are used to specify an executable to run and, optionally, a core 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.

13.2 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.

13.3 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 thread, register and stack contents of all XCores in JTAG scan chain.

13.4 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 the XDE and XScope to trace data in real-time](#)
- ▶ [XScope performance figures](#)
- ▶ [XScope Library API](#)

14 Use the XDE and XScope to trace data in real-time

IN THIS CHAPTER

- ▶ Instrument a program
- ▶ Configure and run a program with tracing enabled
- ▶ Analyze data offline
- ▶ Analyze data in real-time
- ▶ Trace using the UART interface

The XDE and 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 the XDE for real-time display or written to a file for offline analysis.

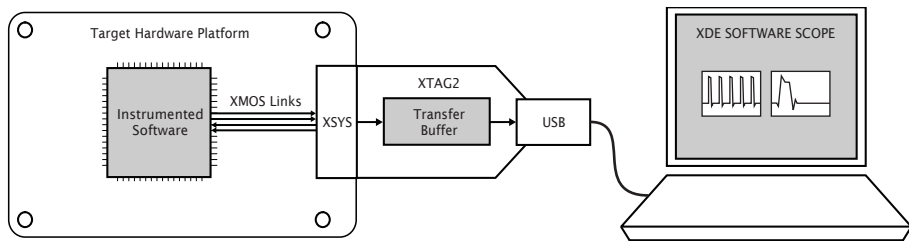


Figure 7:
Scope
connectivity



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 XMOS Link, you can configure the probes to output trace data over your adapter's UART interface instead (see [X9923](#)). Note that the UART interface is supported on a single core only and offers significantly reduced performance.

14.1 Instrument a program

The example program in [Figure 8](#) uses the XScope instrumentation functions to trace the 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 the XDE 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_probe_data_pred` each time it samples data from the microphone. This function creates a trace record and sends it to the PC.

```
#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_probe_data_pred(0, sample);
        micR :> sample;
        xscope_probe_data_pred(1, sample);
    }
}
```

Figure 8:
Program that
traces input
levels to a
microphone



Figure 9 summarizes the different types of probes that can be configured. Only **continuous** 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 9:
Supported
probe types

14.2 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_MAP_FLAGS_config` variable for your build configuration, for example `XCC_MAP_FLAGS_Release`.
3. Add the option `-lxscope`.
4. Create a Run Configuration for your target device (see §12.1).
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, the XDE 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, the XDE 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.

14.3 Analyze data offline

Double-click a trace file in *Project Explorer* to open it in the **Scope view**, as shown in Figure 10.

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.

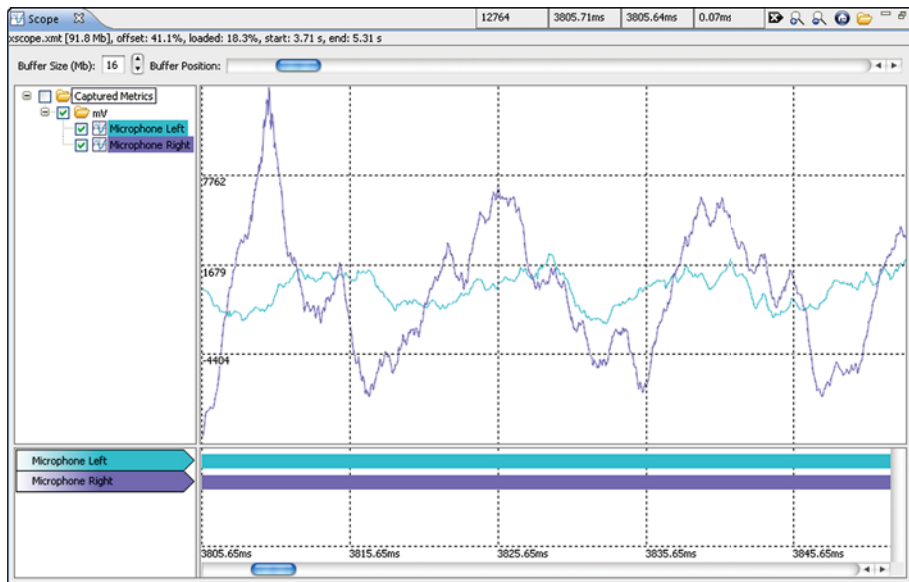


Figure 10:
Offline Scope
view

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.

14.4 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.

The left panel displays a list of the *continuous* probes registered by the application (see §14.1). 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

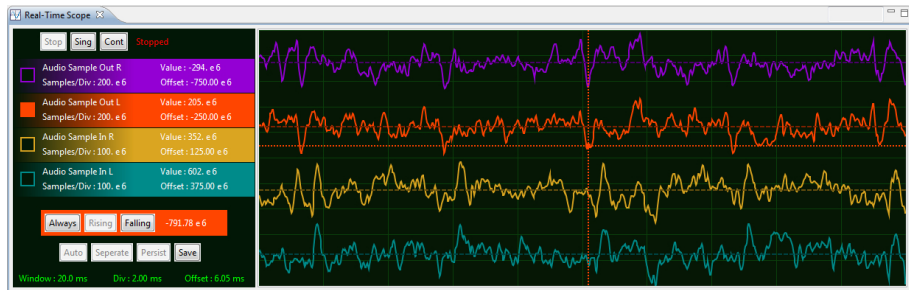


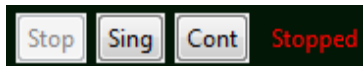
Figure 11:
Real-Time
Scope view

can be used if your platform supports it (Mac OS/X, Linux, and some but not all versions of Windows).

14.4.1 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 12:
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

Select single shot mode to capture one screen of data and return to the stopped state. If a trigger is enabled (see Figure 14) the system will wait for this trigger condition to be met before updating the screen and returning to the stopped state.

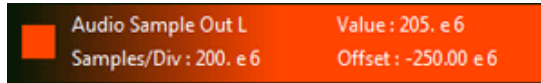
Continuous Capture

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.

14.4.2 Signal Control

The signal controls are available for each registered probe on the coloured label displayed in the left panel (see Figure 13)

Figure 13:
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 14) by clicking in trigger box to the left of the probe label. Only one signal can be used for triggering.

14.4.3 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 14:
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

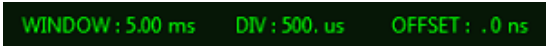
Trigger on a falling edge of the signal.

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.

14.4.4 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 15:
Timebase
Controls



WINDOW : 5.00 ms DIV : 500. us OFFSET : .0 ns

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.

14.4.5 Screen Control

Several commands are available that operate on all signals.

Figure 16:
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.

Persistent Display

Disabled.

Save Data

Saves the current scope view to a PNG file in a user-defined location.

14.5 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 XMOS 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 XMOS Link, the probe functions can be called on a single core only.

15 XScope performance figures

IN THIS CHAPTER

- Transfer rates between the XCore and XTAG-2
 - Transfer rates between the XTAG-2 and Host PC
-

Data transferred from the XMOS 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.

15.1 Transfer rates between the XCore and XTAG-2

The recommended XMOS 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 §41.5). The latencies and maximum call rates for the probe functions using an XMOS Link at this speed are given in Figure 17.

Figure 17: XScope performance figures for XMOs Link with 5-cycle interbit gap	Probe function	Latency (thread 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 §41.5). 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.

15.2 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.

16XScope Library API

IN THIS CHAPTER

- Functions
 - Enumerations
-

16.1 Functions

`void xscope_config_uart(port id)`

Configures XScope UART port.

Must be called before `xscope_register`.

This function has the following parameters:

`id` UART port id.

`void xscope_config_io(xscope_IORedirectionMode mode)`

Configures XScope I/O redirection.

This function has the following parameters:

`mode` I/O redirection mode.

`void xscope_probe(unsigned char id)`

Adds a record for a specific event.

This function has the following parameters:

`id` Probe id

`void xscope_probe_cpu(unsigned char id)`

Adds a record for a specific event, with cpu tracing.

This function has the following parameters:

`id` Probe id.

`void xscope_probe_cpu_data(unsigned char id, unsigned int data)`

Adds a record for a specific event with cpu tracing and user data.

This function has the following parameters:

`id` Probe id.

`data` User data value.

```
void xscope_probe_data(unsigned char id, unsigned int data)
```

Adds a record for a specific event with user data.

This function has the following parameters:

`id` Probe id.

`data` User data value.

```
void xscope_probe_data_pred(unsigned char id, unsigned int data)
```

Adds a record for a specific event with user data and predictable timing.

Limited to calling from a single thread - use `xscope_probe_data` if calling from multiple threads.

This function has the following parameters:

`id` Probe id.

`data` User data value.

```
void xscope_register(int num_probes, ...)
```

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");
```

This function has the following parameters:

`num_probes` Number of probes that will be specified.

```
void xscope_set_register_location(int location)
```

16.2 Enumerations

```
xscope_IORedirectionMode
```

Enum of all I/O redirection modes.

This type has the following values:

`XSCOPE_IO_NONE`
I/O is not redirected.

XSCOPE_IO_BASIC
Basic I/O redirection.

XSCOPE_IO_TIMED
Timed I/O redirection.

xscope_UserDataType

Enum for all user data types.

This type has the following 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_EventType

Enum for all types of xscope events.

This type has the following 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.

Part H

Simulation

CONTENTS

- ▶ [Use the XDE to simulate a program](#)
- ▶ [XSIM Command-Line Manual](#)

17 Use the XDE to simulate a program

IN THIS CHAPTER

- ▶ Configure the simulator
 - ▶ Trace a signal
 - ▶ Set up a loopback
 - ▶ Configure a simulator plugin
-

The XMOS simulator provides a near cycle-accurate model of systems built from one or more XMOS 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.

17.1 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**. The XDE 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. The XDE 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 18.

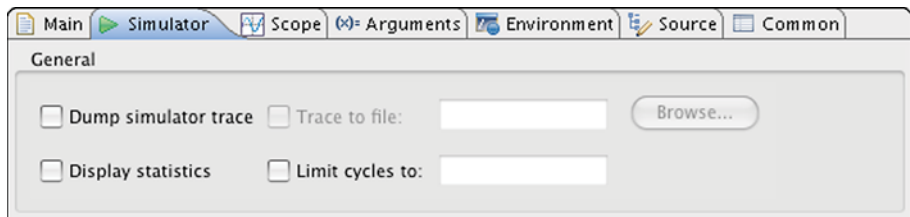


Figure 18:
Simulator
configuration
options

- ▶ 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 22.
- ▶ To view a summary of the program's execution once the program terminates, select **Display statistics**. The summary includes the instruction count for each hardware thread, 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**.

The XDE loads your executable, displaying any output generated by your program in the **Console**.



The XDE 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.

17.2 Trace a signal

The simulator can output signal tracing data to a VCD file which you can visualize with the XDE waveform viewer.

17.2.1 Enable signal tracing

To enable signal tracing during simulation, follow these steps:

1. Create a simulator **Run Configuration** (see §17.1).
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, threads, clock blocks, pads and processor instructions.
3. Click **Run**.

The XDE loads your program into the simulator and, on termination, adds the generated VCD file to your project.

17.2.2 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 19.

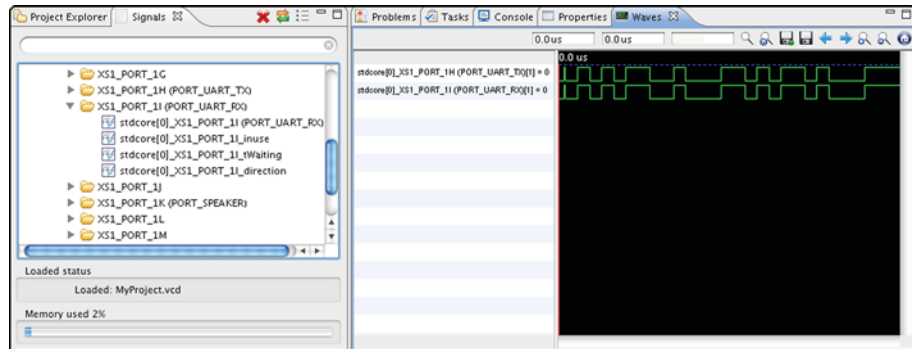


Figure 19:
Signals and
Waves views

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.

17.2.3 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.

17.3 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 §17.1).
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.

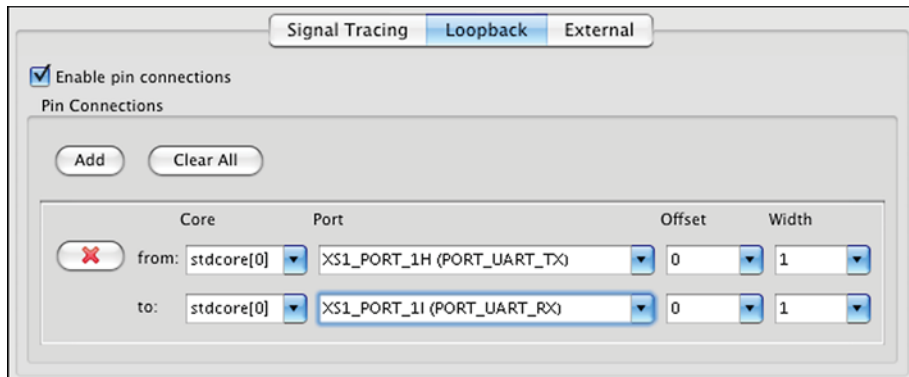


Figure 20:
Setting up a
loopback
connection

5. In the drop-down lists for each end of the connection, select a value for **Core** and **Port**. If you leave the core unspecified, the list of ports is taken from the project's XN file and the core is determined automatically. If you specify core, 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**.

17.4 Configure a simulator plugin

You can connect the simulator to any external plugin that has been compiled on your host PC using the X MOS simulator plugin interface. To configure an external plugin, follow these steps:

1. Create a simulator **Run Configuration** (see §17.1).
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.

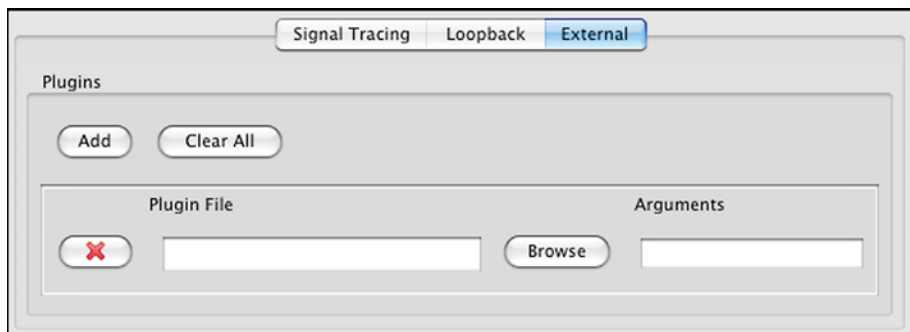


Figure 21:
Setting up an
external
plugin

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.

18XSIM Command-Line Manual

IN THIS CHAPTER

- ▶ Overall Options
 - ▶ Warning Options
 - ▶ Tracing Options
 - ▶ Loopback Plugin Options
-

XSIM performs a cycle-based simulation of an XMOS Executable (XE) file. The XE file contains a description of the target hardware.

18.1 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 hardware thread.
 - ▶ 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.

18.2 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 thread attempts to read or write to another thread's workspace. This can happen if the stack space for a thread is specified using either #pragma stackfunction (see §7) or #pragma stackcalls (see §7).

--no-warn-registers

Don't warn when a register is read before being written.

18.3 Tracing Options

--trace

-t Turns on instruction tracing for all cores (see Figure 22).

--trace-to *file*

Turns on instruction tracing for all cores. 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.

Figure 22: Trace output for XS1 processors

Core	Thread State										Address		Instruction		Mem	Cycle
Name from XN	l ₀	l ₁	l ₂	S ₀	S ₁ (T ₀) .. S ₀ S ₁ (T _n)	.	M	S	K	N	PC	(sym + offset) :	name	operands	address	eval
	-	*	-	-	n status pairs		-	-	-	-				val	L[adr]	
	D	P	d	a	b		m	s	k	n				r _n (val)	S[adr]	
				A	i									res[id]		
				I												
				p												
				m												
				s												
				w												
l ₀ : -	No debug interrupt												S ₁ : -	Interrupts and events disabled		
l ₀ : D	Instruction caused debug interrupt												S ₁ : b	Interrupts and events enabled		
l ₁ : *	Instruction excepted												S ₁ : i	Interrupts enabled and events disabled		
l ₁ : P	Instruction paused												S ₁ : e	Interrupts disabled and events enabled		
l ₂ : -	Not in debug mode												M : -	MSYNC not set		
l ₂ : d	XCore in debug mode												M : m	MSYNC set		
S ₀ : -	Thread not in use												S : -	SSYNC not set		
S ₀ : a	Thread active												S : s	SSYNC set		
S ₀ : A	Thread active (the instruction being traced belongs to this thread)												K : -	INK not set		
S ₀ : i	Thread active with ININT bit set												K : k	INK set		
S ₀ : I	Thread active with ININT bit set (belongs to this thread)												N : -	INENB not set		
S ₀ : p	Thread paused due to instruction fetch												N : n	INENB set		
S ₀ : m	Thread paused with MSYNC bit set												r _n (val)	Value of register <i>n</i>		
S ₀ : s	Thread paused with SSYNC bit set												res[id]	Resource identifier		
S ₀ : w	Thread paused with WAITING bit set												L/S[adr]	Load from/Store to address		

--vcd-tracing *args*

Enables signal tracing. The trace data is output in the standard VCD file format.

If *args* contains any spaces, it must be enclosed in quotes. Its format is:

global-options_{opt}⟨-core 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 core associated with the XN core declaration name, for example stdcore[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.

-threads Turns on thread tracing.

-instructions
 Turns on instruction tracing.

To output traces from different nodes, cores or threads to different files, this option can be specified multiple times.

For example, the following command configures the simulator to trace the ports on stdcore[0] to the file trace.vcd.

► `xsim a.xe --vcd-tracing "-o trace.vcd -core stdcore[0] -ports"`

Tracing by the VCD plugin can be enabled and disabled using the `_traceStart()` and `_traceStop()` syscalls, for example:

```
#include <xs1.h>
#include <syscall.h>

port p1 = XS1_PORT_1A;

int main() {
    _traceStop();

    p1 <: 1;
    p1 <: 0;

    _traceStart();
    p1 <: 1;
```

```
p1 <: 0;
_traceStop();

p1 <: 1;
p1 <: 0;

return 0;
}
```

18.4 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 §41.3) 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 §41.4.2) in the XN file used to compile the program.

Setting *offset* to a non-zero value specifies a subset of the available pins.

`-port core p n offset`

Specifies *n* pins that are connected to the port *p* on a *core*.

The value of *core* must match the `Reference` attribute of a `Core` node (see §41.4.1) 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 `stdcore[0]` to the pin defined by the port `UART_TX` in the program.

```
► xsim uart.xe --plugin LoopbackPort.dll '-port stdcore[0] XS1_PORT_1A 1 0 -port
  UART_TX 1 0'
```

Part I

Debugging

CONTENTS

- ▶ [Use the XDE to debug a program](#)
- ▶ [Debug with printf in real-time](#)

19 Use the XDE to debug a program

IN THIS CHAPTER

- ▶ Launch the debugger
- ▶ Control program execution
- ▶ Examine a suspended program
- ▶ Set a breakpoint
- ▶ View disassembled code

The XMOS 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.

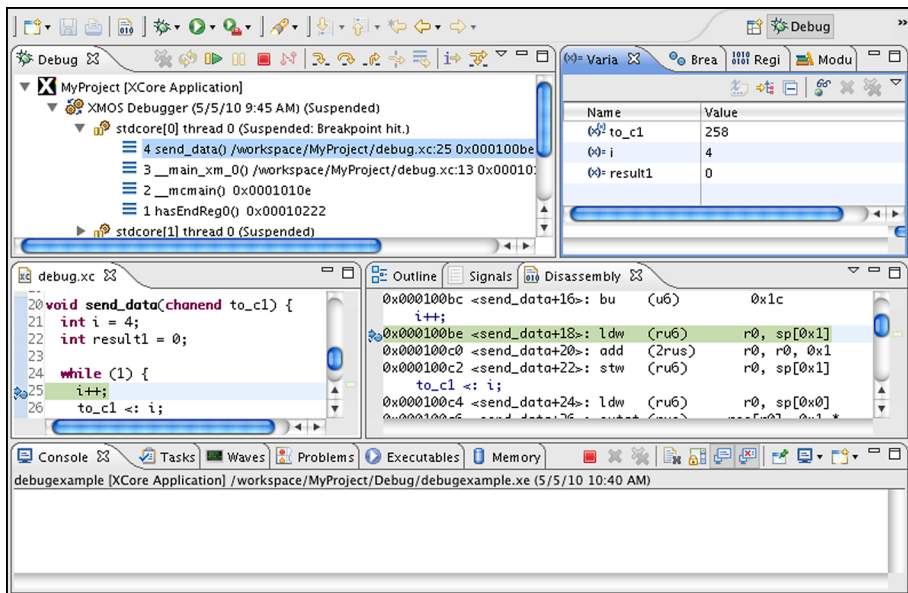


Figure 23:
Debug
perspective



For full visibility of your program, you must compile it with debugging enabled (see §8.3). 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 §8.4) can also make debugging more difficult.

19.1 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**. The XDE 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. The XDE 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 XMOS 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**.

The XDE loads your program in the debugger and opens it in the **Debug** perspective.



The XDE 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.

19.2 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 thread selected in the **Debug** view incrementally:



► **Step Into:** Executes a single line of source code on the thread 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 threads are resumed.



► **Step over:** Executes a single line of source code on the thread selected in the **Debug** view. All other threads are resumed.



► **Step return:** Steps the thread 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 threads are resumed.



► **Step through:** Switches the debugger context to the corresponding input thread of a channel output statement. This is useful for following the path of data as it flows between threads. No threads 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.

19.3 Examine a suspended program

Once a program is suspended, you can query the state of each thread and can inspect the values held in registers and memory.

► **Examine a thread's call stack:** The **Debug** view displays a list of software threads, each of which can be expanded to show its call stack, as shown in Figure 24.

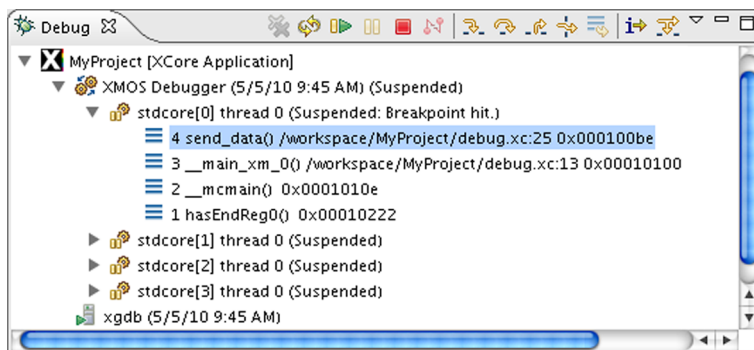


Figure 24:
Debug view

In the example above, processor `stdcore[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 thread's call stack to view its variables, as shown in Figure 19.3.

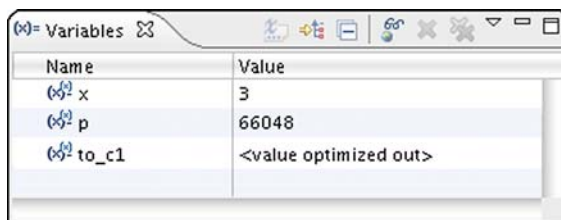


Figure 25:
Variables
view

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 thread to view the contents of its memory, as shown in Figure 19.3.



Figure 26:
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**. The XDE adds new tabs in the panel to the right of the **Memory** view, each showing a different interpretation of the values in memory.

19.4 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 thread 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**.

19.5 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**.

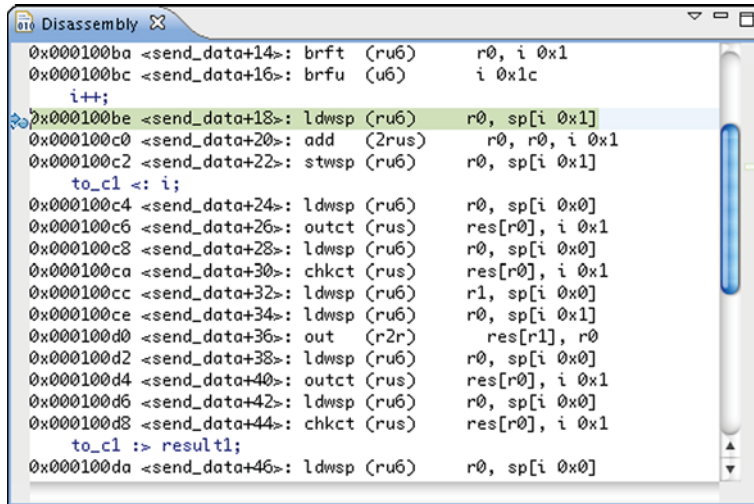


Figure 27:
Disassembly
view



The XDE 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.

20 Debug with printf in real-time

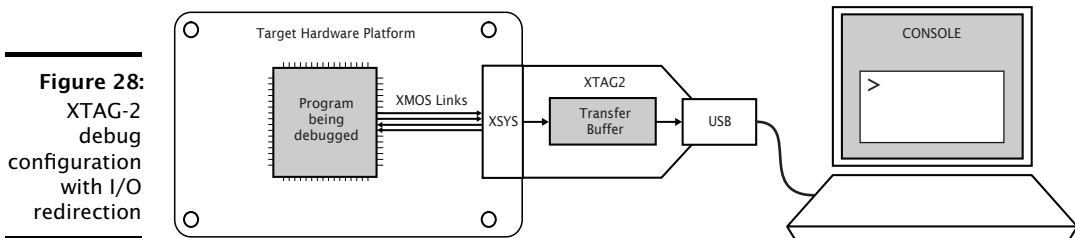
IN THIS CHAPTER

- Redirect `stdout` and `stderr` to the XTAG-2
- Run a program with XTAG-2 output enabled
- Output using the UART interface

The XMOS 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 thread execution, slowing your program down considerably. The XMOS tools let you redirect the standard streams `stdout` and `stderr` to an XTAG-2 debug adapter, where the data is buffered until it can be output to the host.



In this configuration, calls to output routines such as `printf` complete as soon as the data has been output on an XMOS 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-2 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 XMOS Link, you can output data over your adapter's UART interface instead (see §20.3). Note that the UART interface offers significantly reduced performance.

20.1 Redirect stdout and stderr to the XTAG-2

The program below redirects standard output to the XTAG-2.

```
#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-2 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 thread at the same time. This usage results in a single channel end being allocated on each core 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.

20.2 Run a program with XTAG-2 output enabled

To redirect standard output to the XTAG-2 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_MAP_FLAGS_config` variable for your build configuration, for example `XCC_MAP_FLAGS_Release`.
3. Add the option `-lxscope`.
4. If you are developing using the XDE, create a Run Configuration for your target device (see §12.1). In the **XScope** tab, select **Offline mode**. Click **Run** to save and run the configuration.

The XDE loads your program, displaying data received from the XTAG-2 in the console.

5. If you are developing using the command-line tools, pass the option `--xscope` to `XRUN`, for example:

► `xrun myprog.xe --xscope`

`XRUN` loads your program and remains attached to the XTAG-2 adapter, displaying data received from it in the terminal. `XRUN` terminates when the program performs a call to `exit`.

20.3 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 XMOS 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 the XDE, create a Run Configuration for your target device (see §12.1) 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 myprog.xe --uart --xscope`



Because the UART interface uses a port instead of an XMOS Link, you can use the C standard I/O functions on a single core only.

Part J

Flash Programming

CONTENTS

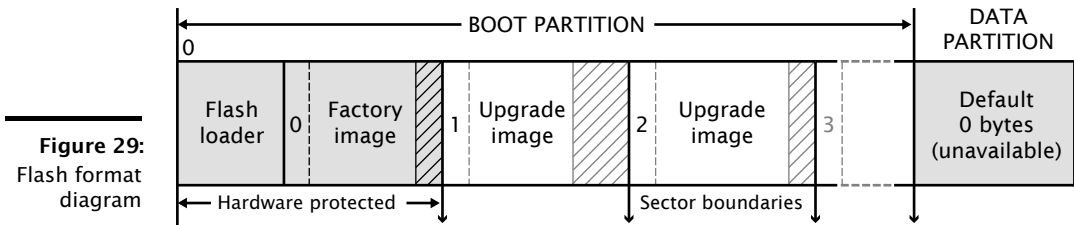
- ▶ [Design and manufacture systems with flash memory](#)
- ▶ [libflash API](#)
- ▶ [List of devices natively supported by libflash](#)
- ▶ [Add support for a new flash device](#)
- ▶ [XFLASH Command-Line Manual](#)

21 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

The XMOS tools can be used to target XMOS devices that use SPI flash memory for booting and persistent storage. The XMOS flash format is shown in Figure 29.



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 core used by the program and a CRC. By default, the flash loader boots the image with the highest version with a valid CRC.

21.1 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 XMOS 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 XMOS device.



The XN file used to compile your program must define an SPI flash device and specify the four ports of the XMOS device to which it is connected (see [X3944](#)).

21.2 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 XMOS 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.

21.3 Perform an in-field upgrade

The XMOS tools and 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.

21.3.1 Write a program that upgrades itself

The example program in [Figure 30](#) uses the libflash library to upgrade itself.

The call to `fl_connect` opens a connection between the XMOS and SPI device, 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.

The call `fl_disconnect` closes the connection between the XMOS and SPI device.

```

#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 < psize; i++)
        fl_writeImagePage(page, i);

    fl_endWriteImage();

    fl_disconnect();

    return 0;
}

int main() {
    /* main application - calls upgrade
     * to perform an in-field upgrade */
}

```

Figure 30:
C program
that uses
libflash to
upgrade itself

21.3.2 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 XMOS 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 -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.

21.4 Customize the flash loader

The XMOS tools let you customize the mechanism for choosing which image is loaded from flash. The example program in Figure 31 determines which image to load based on the value at the start of the data partition.

The XMOS 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`.

Figure 31:
C functions
that
customize
the flash
loader

```
extern void *readFlashDataPage(unsigned addr);

int    dpVersion;
void *imgAdr;

int init(void) {
    void *ptr = readFlashDataPage(0);
    dpVersion = *(int *)ptr;
}

int checkCandidateImageVersion(int v) {
    return v == dpVersion;
}

void recordCandidateImage(int v, unsigned adr) {
    imgAdr = adr;
    return 1;
}

unsigned reportSelectedImage(void) {
    return imgAdr;
}
```

21.4.1 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.

21.4.2 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.

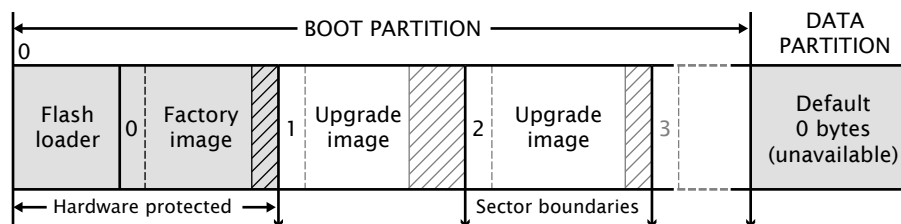
22 libflash API

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 XMOS format shown in the diagram below.

Figure 32: 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.

22.1 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;
```

```
int fl_connect(fl_SPIPorts *SPI)
```

`fl_connect` opens a connection to the specified SPI device.

```
int fl_connectToDevice(fl_SPIPorts *SPI, fl_DeviceSpec spec[], unsigned n)
```

`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.

```
int fl_getFlashType(void)
```

`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.

```
unsigned fl_getFlashSize(void)
```

`fl_getFlashSize` returns the capacity of the SPI device in bytes.

```
int fl_disconnect(void)
```

`fl_disconnect` closes the connection to the SPI device.

22.2 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.

```
int fl_getFactoryImage(fl_BootImageInfo *bootImageInfo)
```

`fl_getFactoryImage` provides information about the factory boot image.

```
int fl_getNextBootImage(fl_BootImageInfo *bootImageInfo)
```

`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.

```
unsigned fl_getImageVersion(fl_BootImageInfo *bootImageInfo)
```

`fl_getImageVersion` returns the version number of the specified image.

```
int fl_startImageReplace(fl_BootImageInfo *, unsigned maxsize)
```

`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.

```
int fl_startImageAdd(fl_BootImageInfo*, unsigned maxsize, unsigned padding)
```

`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.

```
int fl_startImageAddAt(unsigned offset, unsigned maxsize)
```

`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.

```
int fl_writeImagePage(const unsigned char page[])
```

`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.

```
int fl_writeImageEnd(void)
```

`fl_writeImageEnd` waits until the SPI device has written the last page of data to its memory.

```
int fl_startImageRead(fl_BootImageInfo *b)
```

`fl_startImageRead` prepares the SPI device for reading the contents of the specified upgrade image.

```
int fl_readImagePage(unsigned char page[])
```

`fl_readImagePage` inputs the next page of data from the SPI device and writes it to the array page.

```
int fl_deleteImage(fl_BootImageInfo* b)
```

`fl_deleteImage` erases the upgrade image with the specified image.

22.3 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.

```
unsigned fl_getDataPartitionSize(void)
```

`fl_getDataPartitionSize` returns the size of the data partition in bytes.

```
int fl_readData(unsigned offset, unsigned size, unsigned char dst[])
```

`fl_readData` reads a number of bytes from an offset into the data partition and writes them to the array `dst`.

```
unsigned fl_getWriteScratchSize(unsigned offset, unsigned size)
```

`fl_getWriteScratchSize` returns the buffer size needed by `fl_writeData` for the given parameters.

```
int fl_writeData(unsigned offset,  
                 unsigned size,  
                 const unsigned char src[],  
                 unsigned char buffer[])
```

`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.

22.3.1 Page-Level Functions

```
unsigned fl_getPageSize(void)
```

`fl_getPageSize` returns the page size in bytes.

```
unsigned fl_getNumDataPages(void)
```

`fl_getNumDataPages` returns the number of pages in the data partition.

```
unsigned fl_writeDataPage(unsigned n, const unsigned char data[])
```

`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.

```
unsigned fl_readDataPage(unsigned n, unsigned char data[])
```

`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.

22.3.2 Sector-Level Functions

unsigned fl_getNumDataSectors(void)

fl_getNumDataSectors returns the number of sectors in the data partition.

unsigned fl_getDataSectorSize(unsigned n)

fl_getDataSectorSize returns the size of the n -th sector in the data partition in bytes.

unsigned fl_eraseDataSector(unsigned n)

fl_eraseDataSector erases the n -th sector in the data partition.

unsigned fl_eraseAllDataSectors(void)

fl_eraseAllDataSectors erases all sectors in the data partition.

23 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 33 lists the flash devices for which SPI spec files are included with the development tools.

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
EMT	F25L004A	N
Macronix	MX25L1005C	N
NUMONYX	M25P10	N
	M25P16	N
	M25P40	N
	M45P10E	N
SST	SST25VF010	N
	SST25VF016	N
	SST25VF040	N
ST Microelectronics	M25PE10	Y
	M25PE20	Y
Winbond	W25X10	N
	W25X20	N
	W25X40	Y

Figure 33:
List of flash
devices
supported
natively by
libflash

24 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 the XMOS Tools
 - ▶ 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-A² is shown below. The device is described as an initializer for a C structure, the values of which are described in the following sections.

²<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 */

```

24.1 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.

24.2 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.

24.3 Address Size

3, /* 4. Address size */

This value specifies the number of bytes used to represent an address. Figure 34 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		3	1 to ∞
	Release from Deep Power-down			0	0	0

Figure 34:
Table 4 on
page 17 of
M25P10-A
datasheet

24.4 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 35 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 35:
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

24.5 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 34 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 36 and Figure 37, 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 36:
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.

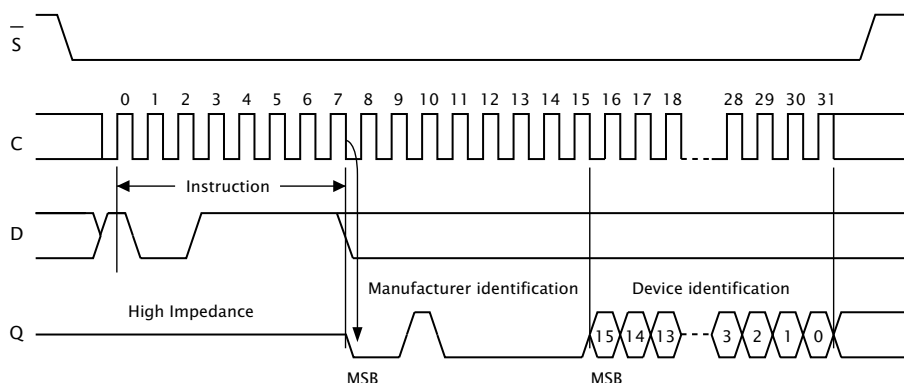


Figure 37:
Figure 9 on
page 19 of
M25P10-A
datasheet

24.6 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 34 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.

24.7 Write Enable/Disable

```
0x06,          /* 12. WREN cmd */
0x04,          /* 13. WRDI cmd */
```

Most flash devices provide instructions to enable and disable writes to memory. Figure 34 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.

24.8 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 38 and Figure 39 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.

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 38:
Table 2 on
page 13 of
M25P10-A
datasheet

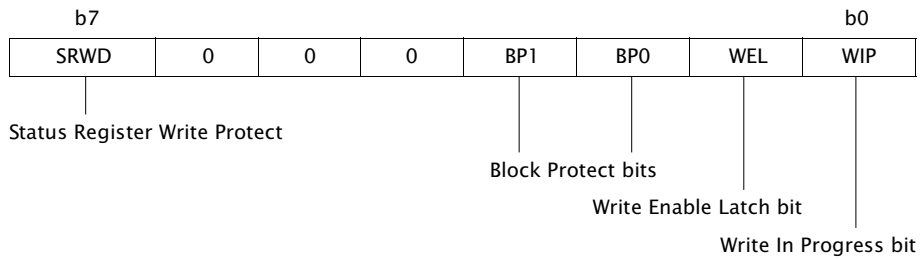


Figure 39:
Table 6 on
page 20 of
M25P10-A
datasheet

24.9 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 34 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 F25L004A³. Figure 40 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 40:
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 */
```

³<http://www.xmos.com/references/f25l004>

24.10 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 34 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.

24.11 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 A25L80P⁴. Figure 41 reproduces the part of this datasheet that provides the sector information.

⁴<http://www.xmos.com/references/a25l80p>

24.13 Add Support to the XMOS Tools

A configuration file can be used with libflash or xflash. The example program below uses libflash to connect to either a M25P10-A or A25L80P device, the configuration parameters which are specified in m25p10a and a25l80p.

```
#include <platform.h>
#include <flash.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"
},
{
#include "a25l80p"
}
};

int main() {
    if (fl_connectToDevice(SPI, myFlashDevices,
        sizeof(fl_DeviceSpecs)/sizeof(fl_DeviceSpec)) != 0) {
        fprintf(stderr, "No supported flash devices found.\n");
        exit(1);
    }
    return 0;
}
```

To generate an image file in the XMOS flash format, which can be subsequently programmed into the above two flash devices, start the command-line tools (see §3.2) and enter the following command:

► xflash prog.xe -o imgfile --spi-spec m25p10a --spi-spec a25l80p

XFLASH generates an image for the custom flash device, which it writes to the specified image file.

24.14 Select a Flash Device

When selecting a flash device for use with an XMOS 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.

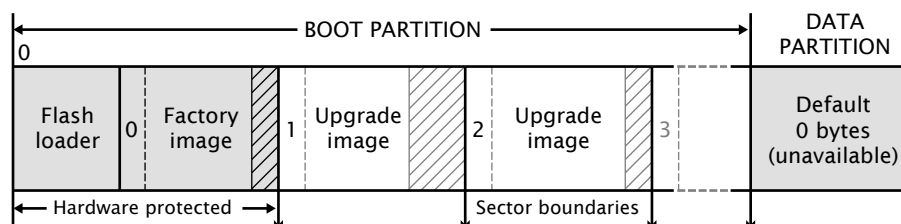
25XFLASH Command-Line Manual

IN THIS CHAPTER

- Overall Options
- Target Options
- Security Options
- Programming Options

XFLASH creates binary files in the XMOS flash format, as illustrated in the diagram below. It can also program these files onto flash devices used to boot XMOS systems.

Figure 42: Flash format diagram



25.1 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 be 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`.

`--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 §21.4). The file may be either an object (.o) or archive (.a).

By default, the XMOS 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.

25.2 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 6 MHz.
- `--spi-spec file`
Enables support for the flash device specified in *file* (see §24).
- `--spi-div n` Sets the divider for the SPI clock to *n*, producing an SPI clock speed of $100/2n$ MHz. By default, if no target is specified, the divider value is set to 3 (16.7 MHz).
- `--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.
- `--disable-boot-link-warn`
Disables a warning emitted when the links between nodes do not allow for boot-from-link to work, for example only links lower than link 3 have been specified as connected on *bootee* nodes, whereas the boot rom enables links 4-7 (and link 3 if a *primary* core).

25.3 Security Options

The following options are used in conjunction with the AES Module (see §26.1).

- `--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.

25.4 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 §41.4) of the corresponding node.

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 §41.4) 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 §8.8).

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 §41.4) 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.

Part K

Security and OTP Programming

CONTENTS

- ▶ [Safeguard IP and device authenticity](#)
- ▶ [XBURN Command-Line Manual](#)

26Safeguard IP and device authenticity

IN THIS CHAPTER

- ▶ The XMOS AES module
 - ▶ Develop with the AES module enabled
 - ▶ Production flash programming flow
 - ▶ Production OTP programming flow
-

XMOS devices contain on-chip one-time programmable (OTP) memory that can be blown during or after device manufacture testing. You can program the XMOS 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 XMOS devices provided by third parties.

Once the AES Module is programmed, the OTP security bits are blown, transforming each core into a “secure island” in which all computation, memory access, I/O and communication are under exclusive control of the code running on the core. When set, these bits:

- ▶ force boot from OTP to prevent bypassing,
- ▶ disable JTAG access to the XCore 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.

26.1 The XMOS AES module

The XMOS 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 43.

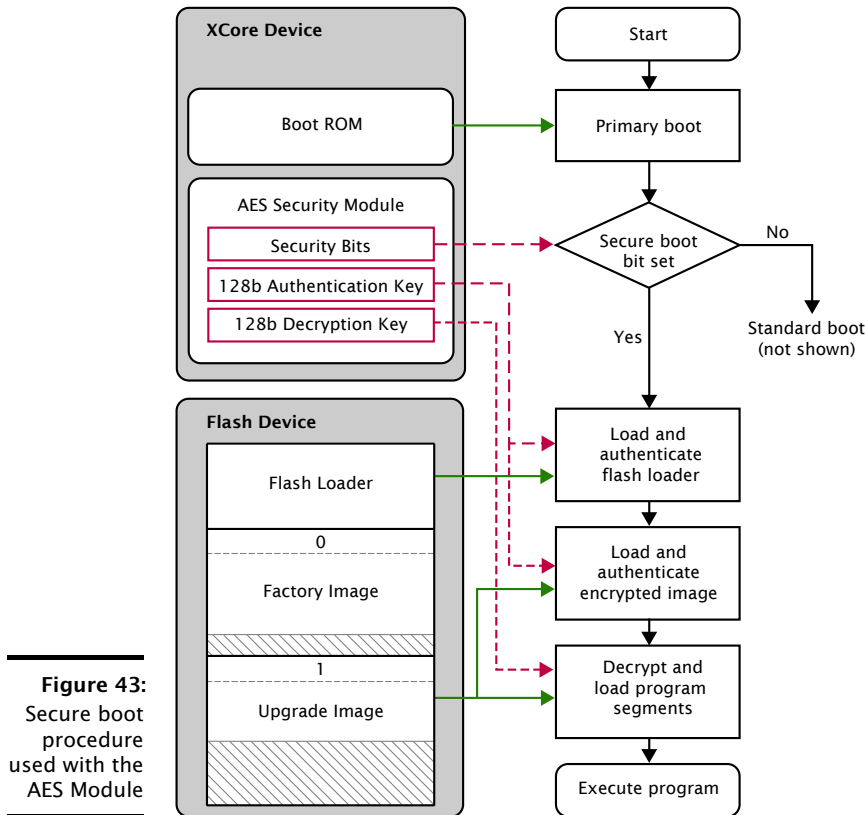


Figure 43:
Secure boot
procedure
used with the
AES Module

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 core, and a secure boot-from-XMOS Link protocol is programmed into all other cores.

26.2 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 XMOS 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 X3944).

To encrypt your program and write it to flash memory, enter the command:

► `xflash --id ID bin.xe --key keyfile`

To protect the XMOSE device, preventing any further development, enter the command:

► `xburn --id ID --target-file target.xn --disable-jtag --lock keyfile`

26.3 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 XMOSE 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 XMOSE 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 XMOSE 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 XMOSE device.

The XN file must define an SPI flash device and specify the four ports of the XMOSE device to which it is connected (see [X3944](#)).

26.4 Production OTP programming flow

In production manufacturing environments, the same keys are typically programmed into multiple XMOS 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.

27XBURN Command-Line Manual

IN THIS CHAPTER

- ▶ Overall Options
 - ▶ Security Register Options
 - ▶ Target Options
 - ▶ Programming Options
-

XBURN creates OTP images, and can program these images into the OTP memory of XMOS devices.

27.1 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 44).

otp-file Specifies the OTP segments from *otp-file* which includes the security register value.

`--lock keyfile` Specifies the XMOS AES boot module (see §26.1) and a default set of security bits (see Figure 44).

`--genkey keyfile` Outputs to *keyfile* two 128-bit keys used for authentication and decryption. The keys are generated using the open-source library `crypto++`.
This option is not valid with `--burn` or `--lock`.

`--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.

`--read` Prints the entire contents of the OTP.

`--help` Prints a description of the supported command-line options.

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

27.2 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 44.

Figure 44: 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

--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 XCores. Disabling plink access restricts all access of the registers of each plinks to the core local to that plink.

--disable-global-debug

Prevents the device from participating in global debug. Disabling global debug prevents the XCores from entering debug using the global debug pin.

--enable-master-lock

Enables the OTP master lock. No further modification of the OTP is permitted.

27.3 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 default value is 20 (2.5MHz).
This option is only valid with `--lock`.
- `--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 §8.8).

27.4 Programming Options

By default, XBURN writes the specified OTP images to the target platform.

- `-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 XMOS device performs the specified OTP burning operation; disables programming.
The XE file can be run later using XRUN.
- `--force`
`-f` Do not prompt before writing the OTP. This is not default.
- `--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.

Part L

Programming in C/XC

CONTENTS

- ▶ [Calling between C/C++ and XC](#)
- ▶ [XC Implementation-Defined Behavior](#)
- ▶ [C Implementation-Defined Behavior](#)
- ▶ [C and C++ Language Reference](#)

28 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.

28.1 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.”

28.2 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 XMOS linker inserts code to add an implicit array bound parameter equal to the maximum value of the `unsigned int` type.

29XC 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 §39).

► **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 §38.8.

► **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 §7) can be used to disable specific safety checks, resulting in undefined behavior for invalid operations.

30C 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*. The XMOS toolchain implementation-defined behavior matches that of GCC 4.2.1⁵ except for the choices listed below.

The following section headings refer to sections in the C99 specification (see §31.1) 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 §39).

Only the supported C99 features are documented.

30.1 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).**
No other signal values correspond to a computational exception.

⁵<http://www.xmos.com/references/gcc-4.2.1-c-implementation>

- ▶ **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.

30.2 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.

30.3 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.

30.4 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.

30.5 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.

30.6 Preprocessing directives

- ▶ **The behavior on each recognized non-STDC `#pragma` directive (6.10.6).**

This is documented in §7.

30.7 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, strtod, 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, strtod, 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`, `strtod` 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".

30.8 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`, `iswalph`, `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`, `iswalph`, `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 45.
- ▶ **Character classifications that are supported by the `iswctype` function (7.25.1).**
The character classifications supported by `iswctype` are given in Figure 46.

Figure 45: 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 46: Wide character mappings

Value	Description
WCT_TOLOWER	Convert to lower case
WCT_Toupper	Convert to upper case

31 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.

31.1 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.

31.2 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

31.3 Online

- ▶ comp.lang.c Frequently Asked Questions: <http://c-faq.com/>

Part M

Programming in Assembly

CONTENTS

- ▶ [Inline Assembly](#)
- ▶ [Make assembly programs compatible with the XMOS XS1 ABI](#)
- ▶ [Assembly Programming Manual](#)

32 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 ("getid r11\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 res[%1], %0" : "=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.

33 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.

33.1 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 once 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.

33.2 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:
```

33.3 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 47:
Sections
supported by
the XMOS
linker

Section	Used For	Directive
<code>.text</code>	Executable code	<code>.text</code>
<code>.dp.data</code>	Writable data	<code>.section .dp.data, "awd", @progbits</code>
<code>.cp.data</code>	Read only data	<code>.section .cp.rodata, "ac", @progbits</code>

33.3.1 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 48:
Directives for
emitting
different
types of data

Directive	Description
<code>.byte</code>	Emits a 1 byte value
<code>.short</code>	Emits a 2 byte value
<code>.word</code>	Emits a 4 byte value
<code>.space</code>	Emits an <i>n</i> -byte array of zero-initialized storage, where <i>n</i> is the argument to the directive
<code>.asciiz</code>	Emits a null terminated ASCII string
<code>.ascii</code>	Emits an ASCII string (no implicit terminating character)

33.3.2 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.

33.4 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.

33.4.1 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
```

33.4.2 Caller and callee save registers

The XS1 ABI specifies 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
```

33.4.3 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 49:
Resource
usage
symbols
defined by
the XS1 ABI

Symbol	Description
<i>f.nstackwords</i>	Stack size (in words)
<i>f.maxthreads</i>	Maximum number of threads allocated, including the current thread
<i>f.maxchanends</i>	Maximum number of channel ends allocated
<i>f.maxtimers</i>	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.

33.4.4 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 50:
Symbols for
denoting
side-effects

Symbol	Description
<code>f.locnoside</code>	Specifies whether the function is side effecting
<code>f.locnochandec</code>	Specifies whether the function allocates a channel end

33.5 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.

33.6 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 51:
Typestring
directives

Binding	Directive
Global	<code>.globl name, "typestring"</code>
External	<code>.extern name, "typestring"</code>
Local	<code>.loc1 name, "typestring"</code>

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.

33.7 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 :
    .ascii "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.locnohandec, 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 tpestrings, the linker can check type compatibility when different object files are linked. The linker can eliminate unused code and data since it is placed in elimination blocks.

34Assembly 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)⁶ with DWARF 3⁷ debugging information. Extensions to the ELF format are documented in the XMOS Application Binary Interface (see §39).

34.1 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.

34.1.1 Comments

The character # introduces a comment, which terminates with a newline. Comments do not occur within string literals.

34.1.2 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.

34.1.3 Directives

A directive begins with '.' followed by one or more letters. Directives instruct the assembler to perform some action (see §34.6).

⁶<http://www.xmos.com/references/elf>

⁷<http://www.xmos.com/references/dwarf3>

34.1.4 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.

34.2 Sections and Relocations

Named ELF sections are specified using directives (see §34.6.10). 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.

34.3 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.

34.3.1 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 §34.6.12), or through its use in a label (see §34.4). The default binding of symbols in the undefined section is *global*; for all other symbols the default binding is *local*.

34.4 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.

34.5 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 \pm a constant. An expression may be encoded in the ELF-extended expression section and its value evaluated by the linker (see §34.6.12); 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

unary-exp  ::= argument
            | prefix-op unary-exp

argument   ::= symbol
            | constant
            | ( expression )

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 52. For the \$D operator, the argument must be a symbol; the result is 1 if the symbol is defined and 0 otherwise.

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.

Figure 52: Valid operations for + and - operators	Op	Left Operand	Right Operand	Result
	+	{ <i>section</i> x}	{absolute y}	{ <i>section</i> x+y}
	+	{absolute x}	{ <i>section</i> y}	{ <i>section</i> x+y}
	+	{absolute x}	{absolute y}	{absolute x+y}
	-	{ <i>section</i> x}	{ <i>section</i> y}	{absolute x-y}
	-	{ <i>section</i> x}	{absolute y}	{ <i>section</i> x-y}
	-	{absolute x}	{absolute y}	{absolute x-y}

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.

34.6 Directives

Directives instruct the assembler to perform some action. The supported directives are given in this section.

34.6.1 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.

34.6.2 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
```


34.6.3 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

```

34.6.4 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.

34.6.5 loc

The `.loc` directive adds a row to the DWARF 3 `.debug_line` line number matrix.

```

loc-directive ::= constant constant constantopt
                  | 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.

34.6.6 weak

The `weak` directive sets the weak attribute on the specified symbol.

```
weak-directive ::= .weak symbol
```

34.6.7 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 ABL.

The `global` and `local` directives are synonyms for the `globl` and `locl` directives. They are provided for compatibility with other assemblers.

34.6.8 `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
```

34.6.9 `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.

34.6.10 `section, pushsection, popsection`

The section directives change the current ELF section (see §34.2).

```
section-directive ::= sec-or-push name
                      | sec-or-push name , flags sec-typeopt
                      | .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.

34.6.11 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
```

34.6.12 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.

34.6.13 `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.

34.6.14 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 §8.4).

```
scheduling-directive ::= .scheduling

scheduling-mode      ::= on
                       | off
                       | default
```

34.6.15 syntax

The `syntax` directive changes the current syntax mode. See §34.7 for details of how assembly instructions are specified in each mode.

```
syntax-directive    ::= .syntax syntax

syntax              ::= default
                       | architectural
```

34.6.16 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.

34.6.17 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 evaluation 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.

34.6.18 XMOS Timing Analyzer Directives

The XMOS Timing Analyzer directives add timing metadata to ELF sections.

```

xta-directive ::= .xtabbranch exp-listopt
               | .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.

- ▶ `xtabbranch` 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` specifies 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.

34.6.19 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

```


34.6.20 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.

34.6.21 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
```

34.6.22 size

The `size` directive specifies the size associated with a symbol.

```
size-directive ::= .size symbol , expression
```

34.6.23 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-listopt
                       | .jmptable32 jmptable-listopt

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.

34.7 Instructions

Assembly instructions are normally inserted into an ELF text section. The syntax of an instruction is:

```

instruction      ::= mnemonic instruction-argsopt

instruction-args ::= instruction-args , instruction-arg
                    | instruction-arg

instruction-arg  ::= symbol [ expression ]
                    | symbol [ expression ] : symbol
                    | expression

```

The assembly instructions are summarized below, with references to the XS1 architecture manual (see [X7879](#)). In this manual, an “architectural” assembly format is documented. The syntax directive is used to switch to this mode of encoding.

The following notation is used:

<i>bitp</i>	one of: 1, 2, 3, 4, 5, 6, 7, 8, 16, 24 and 32
<i>b</i>	register used as a base address
<i>c</i>	register used as a conditional operand
<i>d, e</i>	register used as a destination operand
<i>i</i>	register used as a index operand
<i>r</i>	register used as a resource identifier
<i>s</i>	register used as a source operand
<i>t</i>	register used as a thread identifier
<i>u_s</i>	small unsigned constant in the range 0...11
<i>u_x</i>	unsigned constant in the range 0...(2 ^{<i>x</i>} -1)
<i>v, w, x, y</i>	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.

The following tables refer to a page number of the instruction in the XS1 Architecture Manual (see [X7879](#)).

34.7.1 Data Access

Mnemonic	Operands	Size	Meaning	Page
ld16s	d, b[i]	16	Load signed 16 bits	121
ld8u	d, b[i]	16	Load unsigned 8 bits	122
lda16	d, b[i]	32	Add to 16-bit address	123
lda16	d, b[-i]	32	Subtract from 16-bit address	124
ldap	r11, u ₂₀	16/32	Load pc-relative address	125
ldap	r11, -u ₂₀	16/32	Load pc-relative address	126
ldaw	d, b[i]	32	Add to a word address	131
ldaw	d, b[-i]	32	Subtract from a word address	127
ldaw	d, b[u ₈]	32	Add to a word address immediate	132
ldaw	d, b[-u ₈]	32	Subtract from a word address immediate	128
ldaw	r11, cp[u ₁₆]	16/32	Load address of word in constant pool	129
ldaw	d, dp[u ₁₆]	16/32	Load address of word in data pool	130
ldaw	d, sp[u ₁₆]	16/32	Load address of word on stack	133
ldw	et, sp[4]	16	Load ET from the stack	135
ldw	sed, sp[3]	16	Load SED from the stack	137
ldw	spc, sp[1]	16	Load SPC from the stack	138
ldw	ssr, sp[2]	16	Load SSR from the stack	139
ldw	d, b[i]	16	Load word	140
ldw	d, b[u ₈]	16	Load word immediate	141
ldw	d, cp[u ₁₆]	16/32	Load word from constant pool	142
ldw	r11, cp[u ₂₀]	16/32	Load word from constant pool	143
ldw	d, dp[u ₁₆]	16/32	Load word from data pool	144
ldw	d, sp[u ₁₆]	16/32	Load word from stack	145
set	cp, s	16	Set constant pool	175
set	dp, s	16	Set data pointer	177
set	sp, s	16	Set the stack pointer	185
st16	s, b[i]	32	16-bit store	196
st8	s, b[i]	32	8-bit store	197
stw	sed, sp[3]	16	Store SED on the stack	199
stw	et, sp[4]	16	Store ET on the stack	198
stw	spc, sp[1]	16	Store SPC on the stack	200
stw	ssr, sp[2]	16	Store SSR on the stack	201
stw	s, b[i]	32	Store word	202
stw	s, b[u ₈]	16	Store word immediate	203
stw	s, dp[u ₁₆]	16/32	Store word in data pool	204
stw	s, sp[u ₁₆]	16/32	Store word on stack	205

34.7.2 Branching, Jumping and Calling

Mnemonic	Operands	Size	Meaning	Page
bau	s	16	Branch absolute unconditional	53
bf	c, u ₁₆	16/32	Branch relative if false	63
bf	c, -u ₁₆	16/32	Branch relative if false	60
bl	u ₂₀	16/32	Branch and link relative	59
bl	-u ₂₀	16/32	Branch and link relative	58
bla	s	16	Branch and link absolute via register	55
bla	cp[u ₂₀]	16/32	Branch and link absolute via CP	56
blat	u ₁₆	16/32	Branch and link absolute via table	57
bru	s	16	Branch relative unconditional via register	66
bt	c, u ₁₆	16/32	Branch relative if true	64
bt	c, -u ₁₆	16/32	Branch relative if true	61
bu	u ₁₆	16/32	Branch relative unconditional	65
bu	-u ₁₆	16/32	Branch relative unconditional	62
entsp	u ₁₆	16/32	Adjust stack and save link register	90
extdp	u ₁₆	16/32	Extend data pointer	93
extsp	u ₁₆	16/32	Extend stack pointer	94
retsp	u ₁₆	16/32	Return	169

34.7.3 Data Manipulation

Mnemonic	Operands	Size	Meaning	Page
add	d, x, y	16	Add	47
add	d, x, u _s	16	Add immediate	48
and	d, x, y	16	Bitwise and	49
andnot	d, s	16	And not	50
ashr	d, x, y	32	Arithmetic shift right	51
ashr	d, x, bitp	32	Arithmetic shift right immediate	52
bitrev	d, s	32	Bit reverse	54
byterev	d, s	32	Byte reverse	67
clz	d, s	32	Count leading zeros	73
crc32	d, r, p	32	Word CRC	74
crc8	r, o, d, p	32	8-step CRC	75
divs	d, x, y	32	Signed division	79
divu	d, x, y	32	Unsigned division	80
eq	c, x, y	16	Equal	91
eq	c, x, u _s	16	Equal immediate	92
ladd	e, d, x, y, v	32	Long unsigned add with carry	120

(continued)

Mnemonic	Operands	Size	Meaning	Page
ldc	d, u ₁₆	16/32	Load constant	134
ldivu	d, e, v, x, y	32	Long unsigned divide	136
lmul	d, e, x, y, v, w	32	Long multiply	146
lss	c, x, y	16	Less than signed	147
lsu	c, x, y	16	Less than unsigned	148
lsub	e, d, x, y, v	32	Long unsigned subtract	149
maccs	d, e, x, y	32	Multiply and accumulate signed	150
maccu	d, e, x, y	32	Multiply and accumulate unsigned	151
mkmsk	d, s	16	Make mask	153
mkmsk	d, bitp	16	Make mask immediate	154
mul	d, x, y	32	Multiply	156
neg	d, s	16	Two's complement negate	157
not	d, s	16	Bitwise not	158
or	d, x, y	16	Bitwise or	159
rems	d, x, y	32	Signed remainder	167
remu	d, x, y	32	Unsigned remainder	168
sext	d, s	16	Sign extend	189
sext	d, bitp	16	Sign extend immediate	190
shl	d, x, y	16	Shift left	191
shl	d, x, bitp	16	Shift left immediate	192
shr	d, x, y	16	Shift right	193
shr	d, x, bitp	16	Shift right immediate	194
sub	d, x, y	16	Subtract	206
sub	d, x, u ₈	16	Subtract immediate	207
xor	d, x, y	32	Bitwise exclusive or	223
zext	d, s	16	Zero extend	224
zext	s, bitp	16	Zero extend immediate	225

34.7.4 Concurrency and Thread Synchronization

Mnemonic	Operands	Size	Meaning	Page
freet		16	Free unsynchronized thread	96
get	r11, id	16	Get thread ID	100
getst	d, res[r]	16	Get synchronized thread	108
mjoin	res[r]	16	Master synchronize and join	152
msync	res[r]	16	Master synchronize	155
ssync		16	Slave synchronize	195
init	t[r]:cp, s	16	Initialize thread's CP	212
init	t[r]:dp, s	16	Initialize thread's DP	213
init	t[r]:lr, s	32	Initialize thread's LR	214

(continued)

Mnemonic	Operands	Size	Meaning	Page
init	t[r]:pc, s	16	Initialize thread's PC	215
init	t[r]:sp, s	16	Initialize thread's SP	216
set	t[r]:d, s	16	Set register in thread	218
start	t[r]	16	Start thread	219
tsetmr	d, s	16	Set register in master thread	217

34.7.5 Communication

Mnemonic	Operands	Size	Meaning	Page
chkct	res[r], s	16	Test for control token	68
chkct	res[r], u _s	16	Test for control token immediate	69
getn	d, res[r]	32	Get network	103
in	d, res[r]	16	Input data	110
inct	d, res[r]	16	Input control token	111
int	d, res[r]	16	Input token of data	114
out	res[r], s	16	Output data	160
outct	res[r], s	16	Output control token	161
outct	res[r], u _s	16	Output control token immediate	162
outt	res[r], s	16	Output token of data	165
setn	res[r], s	32	Set network	180
testlcl	d, res[r]	32	Test local	209
testct	d, res[r]	16	Test for control token	210
testwct	d, res[r]	16	Test for position of control token	211

34.7.6 Resource Operations

Mnemonic	Operands	Size	Meaning	Page
clrpt	res[r]	16	Clear port time	71
endin	d, res[r]	16	End a current input	89
freer	res[r]	16	Free a resource	95
getd	d, res[r]	32	Get resource data	97
getr	d, u _s	16	Allocate resource	105
getts	d, res[r]	16	Get port timestamp	109
in	d, res[r]	16	Input data	110
inpw	d, res[r], bitp	32	Input a part word	112
inshr	d, res[r]	16	Input and shift right	113
out	res[r], s	16	Output data	160
outpw	res[r], s, bitp	32	Output a part word	163
outshr	res[r], s	16	Output data and shift	164

(continued)

Mnemonic	Operands	Size	Meaning	Page
peek	d, res[r]	16	Peek at port data	166
setc	res[r], s	32	Set resource control bits	172
setc	res[r], u ₁₆	16/32	Set resource control bits immediate	170
setclk	res[r], s	32	Set clock for a resource	174
setd	res[r], s	16	Set data	176
setev	res[r], r11	16	Set environment vector	178
setpsc	res[r], s	16	Set the port shift count	182
setpt	res[r], s	16	Set the port time	183
setrdy	res[r], s	32	Set ready input for a port	184
settw	res[r], s	32	Set transfer width for a port	187
setv	res[r], r11	16	Set event vector	188
syncr	res[r]	16	Synchronize a resource	208

34.7.7 Event Handling

Mnemonic	Operands	Size	Meaning	Page
clre		16	Clear all events	70
clrsr	u ₁₆	16/32	Clear bits in SR	72
edu	res[r]	16	Disable events	85
eef	d, res[r]	16	Enable events if false	86
eet	d, res[r]	16	Enable events if true	87
eeu	res[r]	16	Enable events	88
getsr	r11, u ₁₆	16/32	Get bits from SR	107
setsr	u ₁₆	16/32	Set bits in SR	186
waitef	c	16	Wait for event if false	220
waitet	c	16	Wait for event if true	221
waiteu		16	Wait for event	222

34.7.8 Interrupts, Exceptions and Kernel Calls

Mnemonic	Operands	Size	Meaning	Page
clrsr	u ₁₆	16/32	Clear bits in SR	72
ecallf	c	16	Raise exception if false	83
ecallt	c	16	Raise exception if true	84
get	r11, ed	16	Get ED into r11	98
get	r11, et	16	Get ET into r11	99
get	r11, kep	16	Get the kernel entry point	101
get	r11, ksp	16	Get the kernel stack pointer	102
getsr	r11, u ₁₆	16/32	Get bits from SR	107

(continued)

Mnemonic	Operands	Size	Meaning	Page
kcall	s	16	Kernel call	115
kcall	u ₁₆	16/32	Kernel call immediate	116
kentstp	u ₁₆	16/32	Switch to kernel stack	117
krestsp	u ₁₆	16/32	Restore stack pointer from kernel stack	118
kret		16	Kernel return	119
set	kep, r11	16	Set the kernel entry point	179
setsr	u ₁₆	16/32	Set bits in SR	186

34.7.9 Debugging

Mnemonic	Operands	Size	Meaning	Page
dcall		16	Cause a debug interrupt	76
dentstp		16	Save and modify stack pointer for debug	77
dgetreg	s	16	Debug read of another thread's register	78
drestsp		16	Restore non debug stack pointer	81
dret		16	Return from debug interrupt	82
get	d, ps[r]	32	Get processor state	104
set	ps[r], s	32	Set processor state	181

34.7.10 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 XS1 instructions.

Mnemonic	Operands	Definition
mov	d, s	add d, s, 0
nop		r0, r0, 0

34.8 Assembly Program

An assembly program consists of a sequence of statements.

```
program ::= <statement>*

statement ::= label-listopt dir-or-instopt separator

label-list ::= label
               | label-list label

dir-or-inst ::= directive
               | instruction

separator ::= newline
               | ;

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
               | jmp-table-directive
```

Part N

Programming for XS1 Devices

CONTENTS

- ▶ [XCC Target-Dependent Behavior for XS1 Devices](#)
- ▶ [XS1 Data Types](#)
- ▶ [XS1 port-to-pin mapping](#)
- ▶ [XS1 Library](#)
- ▶ [XMOS 32-Bit Application Binary Interface](#)

35XCC 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.

35.1 Support for Clock Blocks

An XS1 device provides a single reference clock that ticks at a frequency derived from an external oscillator. XC requires the system designer to ensure that the reference clock ticks at 100MHz for correct operation of timers.

Each XCore 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`.

35.2 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 §35.1) and use a single-entry buffer, even if their declarations are not qualified with the keyword `buffered`.

The table in Figure 53 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 53:
I/O
operations
supported on
XS1 ports

Mode	Operation		
	Serialization	Strobing	@ when
Unqualified	X	X	X
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`.

35.2.1 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.

35.2.2 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.

35.2.3 Changing Direction of Buffered Ports

An attempt to change the direction of a port qualified with `buffered` results in undefined behaviour.

35.3 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.

36XS1 Data Types

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 54 represents the size and alignment of the data types specified by the XMOS Application Binary Interface (see §39), which provides a standard interface for linking objects compiled from both C and XC.

Data Type	Size (bits)	Align (bits)	Supported		Meaning
			XC	C	
<code>char</code>	8	8	✓	✓	Character type
<code>short</code>	16	16	✓	✓	Short integer
<code>int</code>	32	32	✓	✓	Native integer
<code>long</code>	32	32	✓	✓	Long integer
<code>long long</code>	64	32	✗	✓	Long long integer
<code>float</code>	32	32	✗	✓	32-bit IEEE float
<code>double</code>	64	32	✗	✓	64-bit IEEE float
<code>long double</code>	64	32	✗	✓	64-bit IEEE float
<code>void *</code>	32	32	✗	✓	Data pointer
<code>port</code>	32	32	✓	✗	Port
<code>timer</code>	32	32	✓	✗	Timer
<code>chanend</code>	32	32	✓	✗	Channel end

Figure 54:
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).

37XS1 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 55 gives the XS1 port-to-pin mapping, which is interpreted as follows:

- ▶ The name of each pin is given in the format $XnDpq$ where n is a valid XCore 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 stdcore [0] : in port p = XS1_PORT_1A
```

uses the 1-bit port 1A on XCore 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 stdcore [2] : out port p1 = XS1_PORT_32A;  
on stdcore [2] : out port p2 = XS1_PORT_8B;  
on stdcore [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 55: Available ports and links for each pin

Pin	link	Precedence				
		≡ highest	lowest ⇒			
		1-bit ports	4-bit ports	8-bit ports	16-bit ports	32-bit port
XnD00		1A				
XnD01	A ⁴ in/out	1B				
XnD02	A ³ in/out		4A ⁰	8A ⁰	16A ⁰	32A ²⁰
XnD03	A ² in/out		4A ¹	8A ¹	16A ¹	32A ²¹
XnD04	A ¹ in/out		4B ⁰	8A ²	16A ²	32A ²²
XnD05	A ⁰ in/out		4B ¹	8A ³	16A ³	32A ²³
XnD06	A ⁰ out/in		4B ²	8A ⁴	16A ⁴	32A ²⁴
XnD07	A ¹ out/in		4B ³	8A ⁵	16A ⁵	32A ²⁵
XnD08	A ² out/in		4A ²	8A ⁶	16A ⁶	32A ²⁶
XnD09	A ³ out/in		4A ³	8A ⁷	16A ⁷	32A ²⁷
XnD10	A ⁴ out/in	1C				
XnD11		1D				
XnD12		1E				
XnD13	B ⁴ in/out	1F				
XnD14	B ³ in/out		4C ⁰	8B ⁰	16A ⁸	32A ²⁸
XnD15	B ² in/out		4C ¹	8B ¹	16A ⁹	32A ²⁹
XnD16	B ¹ in/out		4D ⁰	8B ²	16A ¹⁰	
XnD17	B ⁰ in/out		4D ¹	8B ³	16A ¹¹	
XnD18	B ⁰ out/in		4D ²	8B ⁴	16A ¹²	
XnD19	B ¹ out/in		4D ³	8B ⁵	16A ¹³	
XnD20	B ² out/in		4C ²	8B ⁶	16A ¹⁴	32A ³⁰
XnD21	B ³ out/in		4C ³	8B ⁷	16A ¹⁵	32A ³¹
XnD22	B ⁴ out/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 in/out					32A ⁰
XnD50	C ³ in/out					32A ¹
XnD51	C ² in/out					32A ²
XnD52	C ¹ in/out					32A ³
XnD53	C ⁰ in/out					32A ⁴
XnD54	C ⁰ out/in					32A ⁵
XnD55	C ¹ out/in					32A ⁶
XnD56	C ² out/in					32A ⁷
XnD57	C ³ out/in					32A ⁸
XnD58	C ⁴ out/in					32A ⁹
XnD61	D ⁴ in/out					32A ¹⁰
XnD62	D ³ in/out					32A ¹¹
XnD63	D ² in/out					32A ¹²
XnD64	D ¹ in/out					32A ¹³
XnD65	D ⁰ in/out					32A ¹⁴
XnD66	D ⁰ out/in					32A ¹⁵
XnD67	D ¹ out/in					32A ¹⁶
XnD68	D ² out/in					32A ¹⁷
XnD69	D ³ out/in					32A ¹⁸
XnD70	D ⁴ out/in					32A ¹⁹

38XS1 Library

IN THIS CHAPTER

- ▶ Data types
 - ▶ Port Configuration Functions
 - ▶ Clock Configuration Functions
 - ▶ Port Manipulation Functions
 - ▶ Clock Manipulation Functions
 - ▶ Thread/Core Control Functions
 - ▶ Channel Functions
 - ▶ Predicate Functions
 - ▶ XS1-S Functions
 - ▶ Miscellaneous Functions
-

38.1 Data types

`clock` 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.

38.2 Port Configuration Functions

```
void configure_in_port_handshake(void port p,  
                                in port readyin,  
                                out port readyout,  
                                clock clk)
```

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\(\)](#).

This function has the following 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.

```
void configure_out_port_handshake(void port p,  
                                in port readyin,  
                                out port readyout,  
                                clock clk,  
                                unsigned initial)
```

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\(\)](#).

This function has the following 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.

```
void configure_in_port_strobed_master(void port p,  
                                     out port readyout,  
                                     const clock clk)
```

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\(\)](#).

This function has the following 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.

```
void configure_out_port_strobed_master(void port p,  
                                       out port readyout,  
                                       const clock clk,  
                                       unsigned initial)
```

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.

This function has the following 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.

```
void configure_in_port_strobed_slave(void port p,  
                                     in port readyin,  
                                     clock clk)
```

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\(\)](#).

This function has the following 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.

```
void configure_out_port_strobed_slave(void port p,  
                                       in port readyin,  
                                       clock clk,  
                                       unsigned initial)
```

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\(\)](#).

This function has the following parameters:

<code>p</code>	The buffered port to configure.
<code>readyin</code>	A 1-bit port to use for the ready-in signal.
<code>clk</code>	The clock used to configure the port.
<code>initial</code>	The initial value to output on the port.

```
void configure_in_port(void port p, const clock clk)
```

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\(\)](#).

This function has the following parameters:

<code>p</code>	The port to configure, which may be buffered or unbuffered.
<code>clk</code>	The clock used to configure the port.

```
void configure_in_port_no_ready(void port p, const clock clk)
```

Alias for [configure_in_port\(\)](#).

```
void configure_out_port(void port p, const clock clk, unsigned initial)
```

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.

This function has the following parameters:

<code>p</code>	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.

```
void configure_out_port_no_ready(void port p,  
                                const clock clk,  
                                unsigned initial)
```

Alias for `configure_out_port()`.

```
void configure_port_clock_output(void port p, const clock clk)
```

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.

This function has the following parameters:

`p` The 1-bit port to configure.

`clk` The clock to output.

```
void set_port_no_sample_delay(void port p)
```

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.

This function has the following parameters:

`p` The port to configure.

```
void set_port_sample_delay(void port p)
```

Sets a port to sample delay mode.

This causes the port to sample input data on the falling edge of its clock.

This function has the following parameters:

`p` The port to configure.

```
void set_port_clock(void port p, const clock clk)
```

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.

This function has the following parameters:

`p` The port to configure.

clk The clock to attach.

```
void set_port_ready_src(void port p, void port ready)
```

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.

This function has the following parameters:

p The port to configure.

ready The 1-bit port to use for the ready-out signal.

```
void set_port_use_on(void port p)
```

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.

This function has the following parameters:

p The port to turn on.

```
void set_port_use_off(void port p)
```

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.

This function has the following parameters:

p The port to turn off.

```
void set_port_mode_data(void port p)
```

Configures a port to be a data port.

This is the default state of a port. Output operations on the port are used to control its output signal.

This function has the following parameters:

p The port to configure.

```
void set_port_mode_clock(void port p)
```

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.

This function has the following parameters:

p The port to configure.

```
void set_port_mode_ready(void port p)
```

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 preferred since they ensure that all the port configuration changes required for the desired mode are performed in the correct order.

This function has the following parameters:

p The port to configure.

```
void set_port_drive(void port p)
```

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.

This function has the following parameters:

p The port to configure.

```
void set_port_drive_low(void port p)
```

Configures a port in drive low mode.

For 1-bit ports when 0 is output its pin is driven low and when 1 is output no value is driven. If the port is not a 1-bit port, the result of an output to the port is undefined. On 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.

This function has the following parameters:

p The port to configure.

```
void set_port_pull_up(void port p)
```

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 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.

This function has the following parameters:

p The port to configure.

```
void set_port_pull_down(void port p)
```

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 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.

This function has the following parameters:

p The port to configure.

```
void set_port_pull_none(void port p)
```

Disables the port's pull-up or pull-down resistor.

This has the side effect of configuring the port in drive mode.

This function has the following parameters:

p The port to configure.

```
void set_port_master(void port p)
```

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.

This function has the following parameters:

p The port to configure.

```
void set_port_slave(void port p)
```

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.

This function has the following parameters:

p The port to configure.

```
void set_port_no_ready(void port p)
```

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.

This function has the following parameters:

p The port to configure.

```
void set_port_strobed(void port p)
```

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.

This function has the following parameters:

p The port to configure.

```
void set_port_handshake(void port p)
```

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.

This function has the following parameters:

p The port to configure.

```
void set_port_no_inv(void port p)
```

Configures a port to not invert data that is sampled and driven on its pins.

This is the default state of a port.

This function has the following parameters:

p The port to configure.

```
void set_port_inv(void port p)
```

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.

This function has the following parameters:

p The 1-bit port to configure.

```
void set_port_shift_count(void port p, unsigned n)
```

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.

This function has the following parameters:

p The buffered port to configure.

n The new shift count.

```
void set_pad_delay(void port p, unsigned n)
```

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 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.

This function has the following parameters:

p The port to configure.

n The number of processor-clock cycles by which to delay the input signal.

38.3 Clock Configuration Functions

```
void configure_clock_src(clock clk, void port p)
```

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.

This function has the following parameters:

clk The clock to configure.

p The 1-bit port to use as the clock source.

```
void configure_clock_ref(clock clk, unsigned char divide)
```

Configures a clock to use the reference clock as its source.

If the divide is set to zero the reference clock frequency is used, otherwise the reference clock frequency divided by $2 * \text{divide}$ is used. By default the reference clock is configured to run at 100 MHz.

This function has the following parameters:

clk The clock to configure.

divide The clock divide.

```
void configure_clock_rate(clock clk, unsigned a, unsigned b)
```

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 where is the reference clock frequency and is a number in the range 1 to 255 inclusive.

This function has the following parameters:

clk The clock to configure.

a The dividend of the desired rate.

b The divisor of the desired rate.

```
void configure_clock_rate_at_least(clock clk, unsigned a, unsigned b)
```

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.

This function has the following parameters:

clk The clock to configure.

a The dividend of the desired rate.

b The divisor of the desired rate.

```
void configure_clock_rate_at_most(clock clk, unsigned a, unsigned b)
```

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.

This function has the following parameters:

clk The clock to configure.

a The dividend of the desired rate.

b The divisor of the desired rate.

```
void set_clock_src(clock clk, void port p)
```

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.

This function has the following parameters:

clk The clock to configure.

p The 1-bit port to use as the clock source.

```
void set_clock_ref(clock clk)
```

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.

This function has the following parameters:

clk The clock to configure.

```
void set_clock_div(clock clk, unsigned char div)
```

Sets the divide for a clock.

This corresponds with the SETD instruction. The clock source must be set to the reference clock, otherwise an exception is raised. If the divide is set to zero the source frequency is left unchanged, otherwise the source frequency is divided by $2 * \text{divide}$.

This function has the following parameters:

clk The clock to configure.

div The divide to use.

```
void set_clock_rise_delay(clock clk, unsigned n)
```

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.

This function has the following parameters:

clk The clock to configure.

n The number of processor-clock cycles by which to delay the rising edge of the clock.

```
void set_clock_fall_delay(clock clk, unsigned n)
```

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.

This function has the following parameters:

clk The clock to configure.

n The number of processor-clock cycles by which to delay the falling edge of the clock.

```
void set_clock_ready_src(clock clk, void port ready)
```

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.

This function has the following parameters:

clk The clock to configure.

ready The 1-bit port to use for the ready-in signal.

```
void set_clock_on(clock clk)
```

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.

This function has the following parameters:

clk The clock to turn on.

```
void set_clock_off(clock clk)
```

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.

This function has the following parameters:

clk The clock to turn off.

38.4 Port Manipulation Functions

`void start_port(void port p)`

Activates a port.

The buffer used by the port is cleared.

This function has the following parameters:

p The port to activate.

`void stop_port(void port p)`

Deactivates a port.

The port is reset to being a no ready port.

This function has the following parameters:

p The port to deactivate.

`unsigned peek(void port p)`

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.

This function has the following parameters:

p The port to peek at.

This function returns:

The value sampled on the pins.

`void clearbuf(void port p)`

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.

This function has the following parameters:

p The port whose buffer is to be cleared.

`void sync(void port p)`

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.

This function has the following parameters:

p The port to wait on.

`unsigned endin(void port p)`

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.

This function has the following parameters:

p The port to end the current input on.

This function returns:

The number of bits of data remaining.

`unsigned partin(void port p, unsigned n)`

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.

This function has the following parameters:

p The buffered port to input on.

n The number of bits to input.

This function returns:

The inputted value.

`void partout(void port p, unsigned n, unsigned val)`

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.

This function has the following parameters:

`p` The buffered port to output on.

`n` The number of bits to output.

`val` The value to output.

`unsigned partout_timed(void port p, unsigned n, unsigned val, unsigned t)`

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.

This function has the following 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.

`{unsigned /* value */, unsigned /* timestamp */} p partin_timestamped(void port p, unsigned n)`

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.

This function has the following parameters:

`p` The buffered port to input on.

`n` The number of bits to input.

This function returns:

The inputted value and the timestamp.

`unsigned partout_timestamped(void port p, unsigned n, unsigned val)`

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.

This function has the following parameters:

p The buffered port to output on.
n The number of bits to output.
val The value to output.

This function returns:

The timestamp of the output.

38.5 Clock Manipulation Functions

```
void start_clock(clock clk)
```

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.

This function has the following parameters:

clk The clock to put into a running state.

```
void stop_clock(clock clk)
```

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.

This function has the following parameters:

clk The clock to put into a stopped state.

38.6 Thread/Core Control Functions

```
void set_thread_fast_mode_on(void)
```

Sets the current thread to run in fast mode.

The thread scheduler always reserves a slot for a thread in fast mode regardless of whether thread 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 thread in fast mode means that other threads are unable to use the extra slot which would otherwise be available while the thread is waiting. In addition setting threads to run in fast mode may also increase the power consumption.

```
void set_thread_fast_mode_off(void)
```

Sets the current thread to run in normal execution mode.

If a thread has previously been put into fast mode using [set_thread_fast_mode_on\(\)](#) this function resets the execution mode it to its default state.

```
unsigned getps(unsigned reg)
```

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.

This function has the following parameters:

reg The processor state register to read.

This function returns:

The value of the processor state register.

```
void setps(unsigned reg, unsigned value)
```

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.

This function has the following parameters:

reg The processor state register to write.

value The value to set the processor state register to.

```
int read_pswitch_reg(unsigned coreid, unsigned reg, unsigned &data)
```

Reads the value of a processor switch register.

The read is of the processor switch which is local to the specified core 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 core identifier is too large to fit in the read packet then 0 is returned.

This function has the following parameters:

coreid The core identifier

reg The number of the register.

data The value read from the register.

This function returns:

Whether the read was successful.

```
int write_pswitch_reg(unsigned coreid, unsigned reg, unsigned data)
```

Writes a value to a processor switch register.

The write is of the processor switch which is local to the specified core id. If a successful acknowledgement is received then 1 is returned. If an error acknowledgement is received or if the register number or core identifier is too large to fit in the write packet then 0 is returned.

This function has the following parameters:

`coreid` The core identifier.
`reg` The number of the register.
`data` The value to write to the register.

This function returns:

Whether the write was successful.

```
int write_pswitch_reg_no_ack(unsigned coreid, unsigned reg, unsigned data)
```

Writes a value to a processor switch register without acknowledgement.

The write is of the processor switch which is local to the specified core id. Unlike [write_pswitch_reg\(\)](#) this function does not wait until the write has been performed. If the register number or core 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.

This function has the following parameters:

`coreid` The core identifier.
`reg` The number of the register.
`data` The value to write to the register.

This function returns:

Whether the parameters are valid.

```
int read_sswitch_reg(unsigned coreid, unsigned reg, unsigned &data)
```

Reads the value of a system switch register.

The read is of the system switch which is local to the specified core 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 core identifier is too large to fit in the read packet then 0 is returned.

This function has the following parameters:

`coreid` The core identifier.
`reg` The number of the register.
`data` The value read from the register.

This function returns:

Whether the read was successful.

```
int write_sswitch_reg(unsigned coreid, unsigned reg, unsigned data)
```

Writes a value to a system switch register.

The write is of the system switch which is local to the specified core id. If a successful acknowledgement is received then 1 is returned. If an error acknowledgement is received or if the register number or core identifier is too large to fit in the write packet then 0 is returned.

This function has the following parameters:

coreid	The core identifier.
reg	The number of the register.
data	The value to write to the register.

This function returns:

Whether the write was successful.

```
int write_sswitch_reg_no_ack(unsigned coreid, unsigned reg, unsigned data)
```

Writes a value to a system switch register without acknowledgement.

The write is of the system switch which is local to the specified core id. Unlike [write_sswitch_reg\(\)](#) this function does not wait until the write has been performed. If the register number or core 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.

This function has the following parameters:

coreid	The core identifier.
reg	The number of the register.
data	The value to write to the register.

This function returns:

Whether the parameters are valid.

```
int read_node_config_reg(core c, unsigned reg, unsigned &data)
```

Reads the value of a node configuration register.

The read is of the node containing the specified core. 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.

This function has the following parameters:

c	The core.
---	-----------

reg The number of the register.

data The value read from the register.

This function returns:

Whether the read was successful.

```
int write_node_config_reg(core c, unsigned reg, unsigned data)
```

Writes a value to a node configuration register.

The write is of the node containing the specified core. 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.

This function has the following parameters:

c The core.

reg The number of the register.

data The value to write to the register.

This function returns:

Whether the write was successful.

```
int write_node_config_reg_no_ack(core c, unsigned reg, unsigned data)
```

Writes a value to a node configuration register without acknowledgement.

The write is of the node containing the specified core. 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.

This function has the following parameters:

c The core.

reg The number of the register.

data The value to write to the register.

This function returns:

Whether the parameters are valid.

```
int read_periph_8(core c,
                    unsigned peripheral,
                    unsigned base_address,
                    unsigned size,
                    unsigned char data[])
```

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.

This function has the following parameters:

<code>c</code>	The core.
<code>peripheral</code>	The peripheral number.
<code>base_address</code>	The base address.
<code>size</code>	The number of 8-bit values to read.
<code>data</code>	The values read from the peripheral.

This function returns:

Whether the read was successful.

```
int write_periph_8(core c,
                  unsigned peripheral,
                  unsigned base_address,
                  unsigned size,
                  const unsigned char data[])
```

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.

This function has the following parameters:

<code>c</code>	The core.
<code>peripheral</code>	The peripheral number.
<code>base_address</code>	The base address.
<code>size</code>	The number of 8-bit values to write.
<code>data</code>	The values to write to the peripheral.

This function returns:

Whether the write was successful.

```
int write_periph_8_no_ack(core c,
```

```
    unsigned peripheral,  
    unsigned base_address,  
    unsigned size,  
    const unsigned char data[])
```

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.

This function has the following parameters:

<code>c</code>	The core.
<code>peripheral</code>	The peripheral number.
<code>base_address</code>	The base address.
<code>size</code>	The number of 8-bit values to write.
<code>data</code>	The values to write to the peripheral.

This function returns:

Whether the parameters are valid.

```
int read_periph_32(core c,  
    unsigned peripheral,  
    unsigned base_address,  
    unsigned size,  
    unsigned data[])
```

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).

This function has the following parameters:

<code>c</code>	The core.
<code>peripheral</code>	The peripheral number.
<code>base_address</code>	The base address.

`size` The number of 32-bit words to read.

`data` The values read from the peripheral.

This function returns:

Whether the read was successful.

```
int write_periph_32(core c,
                    unsigned peripheral,
                    unsigned base_address,
                    unsigned size,
                    const unsigned data[])
```

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).

This function has the following parameters:

`c` The core.

`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.

This function returns:

Whether the write was successful.

```
int write_periph_32_no_ack(core c,
                           unsigned peripheral,
                           unsigned base_address,
                           unsigned size,
                           const unsigned data[])
```

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).

This function has the following parameters:

`c` The core.

`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.

This function returns:

Whether the parameters are valid.

```
unsigned get_core_id(void)
```

Returns the core identifier of the core on which the caller is running.

The core identifier uniquely identifies a core on the network.

This function returns:

The core identifier.

```
unsigned get_thread_id(void)
```

Returns the thread identifier of the thread on which the caller is running.

The thread identifier uniquely identifies a thread on the current core.

This function returns:

The core identifier.

38.7 Channel Functions

```
void start_streaming_master(chanend c)
```

Start streaming communication on the channel.

A call to this function must be matched with a call to [start_streaming_slave\(\)](#) on the other end of the channel. A path between the two channel ends is opened which can be used to perform unsynchronized communication using the streaming input and output functions. This path is held open until it is closed using the function [stop_streaming_master\(\)](#). Note that if the number of channels held open between two points on the network is equal to the number of possible paths these two points then no other channel communication can take place between these two points, which may cause the program deadlock.

This function has the following parameters:

`c` The channel end to start streaming on

```
void stop_streaming_master(chanend c)
```

Stop streaming communication on the channel.

A call to this function must be matched with a call to a [stop_streaming_slave\(\)](#) on the other end of the channel. The the path previously opened using [start_streaming_master\(\)](#) is closed down, making it available for other channel communications.

This function has the following parameters:

c The channel end to stop streaming on

```
void start_streaming_slave(chanend c)
```

Start streaming communication on the channel.

A call to this function must be matched with a call to [start_streaming_master\(\)](#) on the other end of the channel. A path between the two channel ends is opened which can be used to perform unsynchronized communication using the streaming input and output functions. This path is held open until it is closed using the function [stop_streaming_slave\(\)](#). Note that if the number of channels held open between two points on the network is equal to the number of possible paths these two points then no other channel communication can take place between these two points, which may cause the program deadlock. The function [start_streaming_slave\(\)](#) may be called in a case of a select, in which case it becomes ready when [start_streaming_master\(\)](#) is called on the other end of the channel.

This function has the following parameters:

c The channel end to start streaming on

```
void stop_streaming_slave(chanend c)
```

Stop streaming communication on the channel.

A call to this function must be matched with a call to [stop_streaming_master\(\)](#) on the other end of the channel. The the path previously opened using [start_streaming_slave\(\)](#) is closed down, making it available for other channel communications.

This function has the following parameters:

c The channel end to stop streaming on

```
void outuchar(chanend c, unsigned char val)
```

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.

This function has the following parameters:

c The channel end to stream data out on.

val The value to output.

void outuint(chanend c, unsigned val)

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.

This function has the following parameters:

c The channel end to stream data out on.

val The value to output.

unsigned char inuchar(chanend c)

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.

This function has the following parameters:

c The channel end to stream data in on.

This function returns:

The value received.

unsigned inuint(chanend c)

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.

This function has the following parameters:

c The channel end to stream data in on.

This function returns:

The value received.

void inuchar_byref(chanend c, unsigned char &val)

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.

This function has the following parameters:

c The channel end to stream data in on.

`val` The variable to set to the received value.

```
void inuint_byref(chanend c, unsigned &val)
```

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.

This function has the following parameters:

`c` The channel end to stream data in on.

`val` The variable to set to the received value.

```
void outct(chanend c, unsigned char val)
```

Streams out a control token on a channel end.

Attempting to output a hardware control token causes an exception to be raised.

This function has the following parameters:

`c` The channel end to stream data out on.

`val` The value of the control token to output.

```
void chkct(chanend c, unsigned char val)
```

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.

This function has the following parameters:

`c` The channel end.

`val` The expected control token value.

```
unsigned char inct(chanend c)
```

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.

This function has the following parameters:

`c` The channel end to stream data in on.

This function returns:

The received control token.

```
void inct_byref(chanend c, unsigned char &val)
```

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.

This function has the following parameters:

`c` The channel end to stream data in on.

`val` The variable to set to the received value.

```
int testct(chanend c)
```

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.

This function has the following parameters:

`c` The channel end to perform the test on.

This function returns:

1 if the next byte is a control token, 0 otherwise.

```
int testwct(chanend c)
```

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.

This function has the following parameters:

`c` The channel end to perform the test on.

This function returns:

The position of the first control token in the word (1-4) or 0 if the word contains no control tokens.

```
void soutct(streaming chanend c, unsigned char val)
```

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.

This function has the following parameters:

`c` The channel end to stream data out on.

`val` The value of the control token to output.

```
void schkct(streaming chanend c, unsigned char val)
```

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.

This function has the following parameters:

`c` The streaming channel end.

`val` The expected control token value.

```
unsigned char sinct(streaming chanend c)
```

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.

This function has the following parameters:

`c` The streaming channel end to stream data in on.

This function returns:

The received control token.

```
void sinct_byref(streaming chanend c, unsigned char &val)
```

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.

This function has the following parameters:

`c` The streaming channel end to stream data in on.

`val` The variable to set to the received value.

```
int stestct(streaming chanend c)
```

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.

This function has the following parameters:

`c` The channel end to perform the test on.

This function returns:

1 if the next byte is a control token, 0 otherwise.

```
int stestwct(streaming chanend c)
```

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.

This function has the following parameters:

`c` The streaming channel end to perform the test on.

This function returns:

The position of the first control token in the word (1-4) or 0 if the word contains no control tokens.

`transaction out_char_array(chanend c, const char src[], unsigned size)`

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.

This function has the following parameters:

`c` The channel end to output on.

`src` The array of values to send.

`size` The number of bytes to output.

`transaction in_char_array(chanend c, char src[], unsigned size)`

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.

This function has the following parameters:

`c` The channel end to input on.

`src` The array to store the values input from on the channel.

`size` The number of bytes to input.

38.8 Predicate Functions

`void pinseq(unsigned val)`

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`.

This function has the following parameters:

`val` The value to compare against.

`void pinsneq(unsigned val)`

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`.

This function has the following parameters:

`val` The value to compare against.

`void pinseq_at(unsigned val, unsigned time)`

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`.

This function has the following parameters:

`val` The value to compare against.

`time` The time at which to make the comparison.

`void pinsneq_at(unsigned val, unsigned time)`

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`.

This function has the following parameters:

`val` The value to compare against.

`time` The time at which to make the comparison.

`void timerafter(unsigned val)`

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.

This function has the following parameters:

val The time to compare against.

38.9 XS1-S Functions

These functions to control the analogue-to-digital converter (ADC) on XS1-S devices.

```
void enable_xs1_su_adc_input(unsigned number, chanend c)
```

Enables the ADC input specified by *number*.

Samples are sent to chanend *c*.

This function has the following parameters:

number The ADC input number.

c The channel connected to the XS1-SU ADC.

```
void enable_xs1_su_adc_input_streaming(unsigned number,  
                                         streaming chanend c)
```

Enables the ADC input specified by *number*.

Samples are sent to chanend *c*.

This function has the following parameters:

number The ADC input number.

c The channel connected to the XS1-SU ADC.

```
void disable_xs1_su_adc_input(unsigned number, chanend c)
```

Disables the ADC input specified by *number*.

This function has the following parameters:

number The ADC input number.

c The channel connected to the XS1-SU ADC.

```
void disable_xs1_su_adc_input_streaming(unsigned number,  
                                         streaming chanend c)
```

Disables the ADC input specified by *number*.

This function has the following parameters:

number The ADC input number.

c The channel connected to the XS1-SU ADC.

38.10 Miscellaneous Functions

`void crc32(unsigned &checksum, unsigned data, unsigned poly)`

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;
}
```

This function has the following 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.

`unsigned crc8shr(unsigned &checksum, unsigned data, unsigned poly)`

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;
}
```

This function has the following 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.

This function returns:

The data shifted right by 8.

```
{unsigned, unsigned} l lmul(unsigned a, unsigned b, unsigned c, unsigned d)
```

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
```

This function returns:

The high and low halves of the calculation respectively.

```
{unsigned, unsigned} m mac(unsigned a, unsigned b, unsigned c, unsigned d)
```

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
```

This function returns:

The high and low halves of the calculation respectively.

```
{signed, unsigned} m macs(signed a, signed b, signed c, unsigned d)
```

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
```

This function returns:

The high and low halves of the calculation respectively.

```
signed sext(unsigned a, unsigned b)
```

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.

This function returns:

The sign extended value.

`unsigned zext(unsigned a, unsigned b)`

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.

This function returns:

The zero extended value.

39X MOS 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*⁸.

⁸<http://www.xmos.com/docnum/X9114>

Part O

Platform Configuration

CONTENTS

- ▶ [Describe a target platform](#)
- ▶ [XN Specification](#)

40 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.

40.1 Supported network topologies

The tools support the following network topologies.

	Supported network topologies	
	Network Topology	Supported Configurations
Figure 56: Supported network topologies	Line	XS1-L devices only, up to a maximum of 16 nodes
	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)

40.2 A board with two packages

Figure 57 illustrates a board containing two XMOS L1 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"?>
```

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">
```

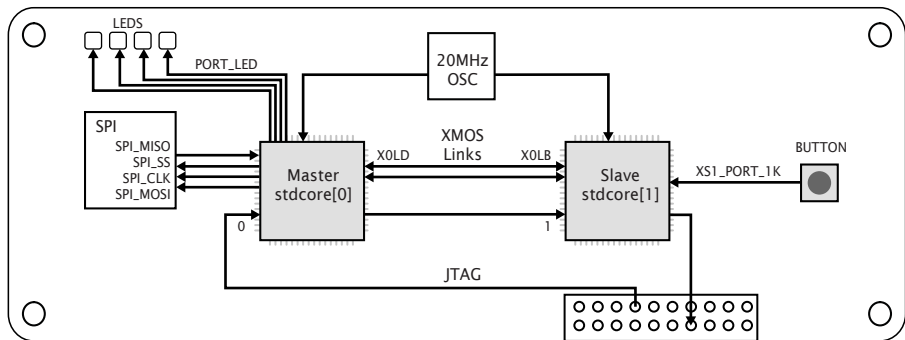


Figure 57:
Example
hardware
platform

The following code declares two XCores. The declaration “core stdcore[2];” is exported to the header file <platform.h>.

```
<Declarations>
  <Declaration>core stdcore[2]</Declaration>
</Declarations>
```

The following code declares a package named P1, which contains a single node named Master.

```
<Packages>
  <Package Id="P1" Type="XS1-L1A-TQ128">
    <Nodes>
      <Node Id="Master" Type="XS1-L1A" InPackageId="0"
        Oscillator="20MHz" SystemFrequency="400MHz">
        <Boot>
          <Source Location="SPI:bootFlash"/>
          <Bootee NodeId="Slave" Core="0"/>
        </Boot>
        <Core Number="0" Reference="stdcore[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"/>
        </Core>
      </Node>
    </Nodes>
  </Package>
```

The node Master is a 400MHz XS1-L1A 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 XCore "0" is associated with `stdcore[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-L1A-TQ128">
  <Nodes>
    <Node Id="Slave" Type="XS1-L1A" InPackageId="0"
      Oscillator="20Mhz" SystemFrequency="400MHz">
      <Boot>
        <Source Location="XMOSLINK"/>
      </Boot>
      <Core Number="0" Reference="stdcore[1]">
        <Port Location="XS1_PORT_1K" Name="PORT_BUTTON"/>
      </Core>
    </Node>
  </Nodes>
</Package>
</Packages>
```

The node Slave is a 400MHz XS1-L1A device in a TQ128 package, clocked by a 20MHz oscillator. It is booted from node Master over an XMOS Link.

The following code defines a 2-wire XMOS 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" Core="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 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>
```

41 XN Specification

IN THIS CHAPTER

- ▶ Network Elements
 - ▶ Declaration
 - ▶ Package
 - ▶ Node
 - ▶ Link
 - ▶ Device
 - ▶ JTAGDevice
-

41.1 Network Elements

The XMOS tools support 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 58](#)

41.2 Declaration

A **Declaration** element provides a symbolic name for one or more XCores. A single name or an array of names is supported with the form:

core identifier

core identifier [*constant-expression*]

An equivalent declaration is exported to the header file `<platform.h>` for use in XC programs. A core variable is assigned to a physical XCore by a **Core** element (see [§41.4.1](#)).

Example

```
<Declaration>core master</Declaration>
<Declaration>core stdcore[8]</Declaration>
```

Node	Number	Description	Section
Network	1	An XMOS network	
Declarations	0+		
Declaration	1+	XCore declaration	§??
Packages	1+		
Package	1+	Device package	§??
Nodes	1		
Node	1+	Node declaration	§??
Core	1+	An XCore	§??
Port	0+	An XCore symbolic port name	§??
Boot	0 or 1	Boot method	§??
Source	1	Binary location	§??
Bootee	0+	Nodes booted	§??
Service	0+	Service declaration	§??
Chanend	1+	Chanend parameter	§??
Links	0 or 1		
Link	1+	XMOS Link declaration	§??
LinkEndpoint	2	XMOS 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 58:
XN hierarchy
of elements

41.3 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 <code>type.pkg</code> in the path specified by <code>XCC_DEVICE_PATH</code> .

Figure 59:
XN Package
element

Example

```
<Package id="L2" Type="XS1-L2A-QF124">
```

The package named L2 is described in the file `XS1-L2A-QF124.xml`.

41.4 Node

A Node element defines a set of XCores in a network, all of which are connected to a single switch. XN devices such as the G4 or L1 are both examples of nodes.

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 <code>periph:XS1-SU</code> 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 <code>config_type.xml</code> in the path specified by <code>XCC_DEVICE_PATH</code> .
Reference	Yes	String	Associates the node with a core identifier specified in a Declaration. This attribute is only valid on nodes with type <code>periph:XS1-SU</code> .
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 unset.
PllFeedbackDivMin	No	Integer	The minimum allowable PLL feedback divider. Defaults to 1 if unset.
ReferenceFrequency	No	String	A reference clock frequency, specified as a number followed by either MHz, KHz or Hz. Defaults to 100MHz if unset.
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	$\text{SystemFrequency} / \text{RefDiv} = \text{ReferenceFrequency}$

Figure 60:
XN Node
element

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 specified 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 specified. The tools use these values to set the PLL registers and reference clock divider. Information on the PLL dividers can be found in XNOS frequency control documents for XS1-G processors (see [X3221](#)) and XS1-L processors (see [X1433](#)).

If the oscillator frequency is specified 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`.

41.4.1 Core

A Core element describes the properties of a single XCore.

Attribute	Required	Type	Description
Number	Yes	Integer	A unique number for the core in the node. Must be a value between 0 and <i>n</i> -1 where <i>n</i> is the number of cores as defined in the node's XML file.
Reference	No	String	Associates the core with an identifier with the form <code>stdcore[n]</code> in a Declaration. A core may be associated with at most one identifier.

Figure 61:
XN Core
element

Example

```
<Core Number="0" Reference="stdcore[0]">
```

41.4.2 Port

A Port element provides a symbolic name for a port.

Example

Figure 62:
XN Port
element

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.

```
<Port Location="XS1_PORT_1I" Name="PORT_UART_TX"/>
<Port Location="XS1_PORT_1J" Name="PORT_UART_RX"/>
```

41.4.3 Boot

A `Boot` element defines the how the node is booted. It contains one `Source` element (see §41.4.4) and zero or more `Bootee` elements (see §41.4.5) that are booted over XMOS Links. If the source specifies an XMOS 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 §21.1).

41.4.4 Source

A `Source` element specifies the location from which the node boots. It has the following attributes.

Figure 63:
XN Source
element

Attribute	Required	Type	Description
Location	Yes	String	Has the form SPI: or XMOSLINK. The device-name must be declared in the set of Device elements.



Only XMOS XS1-L devices can be configured to boot over XMOS Links.

Example

```
<Source Location="SPI:bootFlash"/>
```

41.4.5 Bootee

A `Bootee` element specifies another node in the system that this node boots via an XMOS Link. If more than one XMOS Link is configured between this node and one of its bootees (see §41.5 and §41.5.1), the tools pick one to use for booting.

Example

Figure 64:
XN Bootee
element

Attribute	Required	Type	Description
NodeId	Yes	String	A valid identifier for another node.

```
<Bootee NodeId="Slave">
```

41.4.6 Service

A Service element specifies an XC service function provided by a node.

Figure 65:
XN Service
element

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.

Example

```
<Service Proto="service_function(chanend c1, chanend c2)">
```

41.4.7 Chanend

A Chanend element describes a channel end parameter to an XC service function.

Figure 66:
XN Service
element

Attribute	Required	Type	Description
Identifier	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.

Example

```
<Chanend Identifier="c" end="23" remote="5"/>
```

41.5 Link

XMOS 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 XMOS Link. It must contain exactly two LinkEndpoint children (see §41.5.1).

Figure 67:
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 SOD to specify a "streaming output for debug link."

Example

```
<Link Encoding="2wire" Delays="4,4">
```

41.5.1 LinkEndpoint

A LinkEndpoint describes one end of an XMOS 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 XMOS Link.

Figure 68:
XN
LinkEndpoint
element

Attribute	Required	Type	Description
NodeID	No	String	A valid node identifier.
Link	No	String	A link identifier in the form XnLm where n denotes a core number and m the link letter. See the corresponding package datasheet for available link pinouts.
RoutingId	No	Integer	The routing identifier on the XMOS Link network.
Chanend	No	Integer	A channel end.

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">
```

41.6 Device

A `Device` element describes a device attached to an XCore that is not connected directly to an XMOS Link.

Figure 69:
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.
Core	Yes	Integer	The core 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).

The XMOS tools recognize the Class `SPIFlash` and use the `Type` attribute to identify the model of the flash device.

41.6.1 Attribute

An `Attribute` element describes one aspect of a `Device` (see §41.6).

Figure 70:
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.

The XMOS tools support 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"/>
```

41.7 JTAGDevice

The XMOS tools load and debug 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 71:

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">
```



Copyright © 2012, 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.