



XMOS XTC Tools - Tools Guide

Release: 15.2

Publication Date: 2024/01/02

Document Number: XM-014363-PC

Table of Contents

1	Introduction	1
1.1	Supported devices	1
1.2	Upgrading from earlier xTIMEComposer releases	1
1.3	Document structure	2
1.4	References	2
1.5	Support	2
2	Installation and configuration	3
2.1	System requirements	3
2.2	Installation instructions	3
2.2.1	Download the tools	3
2.2.2	Install the tools	4
2.2.3	Check your installation	5
2.2.4	Configure the XTAG USB drivers	5
2.2.5	Check XTAG access	7
2.3	Configuring the command-line environment	7
2.3.1	Configure the environment	7
2.3.2	Quick check the environment	8
2.3.3	Query the XTC Tools version	8
2.4	Configuring an IDE	9
2.4.1	Configuration common to all IDEs	9
2.4.2	Visual Studio Code	10
2.4.3	Eclipse	13
2.4.4	xTIMEcomposer Studio	15
2.5	Transitioning from older tools releases	15
2.5.1	Programming language	16
2.5.2	Graphical IDE	17
2.5.3	Migrating existing projects	17
3	Quick start	19
3.1	A single-tile program	19
3.1.1	Build an executable	19
3.1.2	Running the program on the simulator	20
3.1.3	Run on real hardware	20
3.1.4	Summary	21
3.2	Targeting multiple tiles	22
3.2.1	Providing a multi-tile file	22
3.2.2	Summary	23
3.3	Communicating between tiles	23
3.3.1	Declaring a channel in the multi-tile file	23
3.3.2	Using the channel	24
3.3.3	Summary	25
3.4	Using XSCOPE for fast “printf debugging”	25
3.4.1	Configuring and using XSCOPE	25
3.4.2	Using XSCOPE “probes”	26
3.4.3	Summary	28
3.5	Debugging with XGDB	28
3.5.1	Create example	28
3.5.2	Interactive debugging	29
3.5.3	Scripted debugging	31
3.5.4	Summary	31



4	Advanced tutorials	32
4.1	Describe a target platform	32
4.1.1	Supported network topologies	32
4.1.2	A board with two packages	32
4.2	Working with targets	34
4.2.1	Terminology	34
4.2.2	Introduction	35
4.2.3	Key features	35
4.2.4	xTAG	36
4.2.5	Host tools	37
4.2.6	Target interaction	39
4.2.7	Host-IO	40
4.2.8	Xscope APIs	41
4.2.9	User-supplied host program	42
4.2.10	Host/target data throughput	46
4.2.11	Sample-based profiling of the target program	47
4.2.12	Target errors and warnings	48
4.2.13	Command examples	48
4.3	Design and manufacture systems with flash memory	51
4.3.1	Boot a program from flash memory	52
4.3.2	Generate a flash image for manufacture	52
4.3.3	Perform an in-field upgrade	52
4.3.4	Customize the flash loader	55
4.3.5	Reading the numerical and string identifiers in a flash image	56
4.4	Safeguard IP and device authenticity	57
4.4.1	The xCORE AES module	57
4.4.2	Develop with the AES module enabled	59
4.4.3	Production flash programming flow	59
4.4.4	Production OTP programming flow	60
4.4.5	Security implications of application design	60
4.5	Add support for a new flash device	61
4.5.1	Libflash Device ID	63
4.5.2	Page Size and Number of Pages	63
4.5.3	Address Size	63
4.5.4	Clock Rate	64
4.5.5	Read Device ID	65
4.5.6	Sector Erase	66
4.5.7	Write Enable/Disable	66
4.5.8	Memory Protection	67
4.5.9	Programming Command	68
4.5.10	Read Data	68
4.5.11	Sector Information	68
4.5.12	Status Register Bits	69
4.5.13	Quad Enable Bit	70
4.5.14	SFDP	70
4.5.15	Select a Flash Device	71
4.6	Using LPDDR	71
4.6.1	Accessing LPDDR in an application	71
4.6.2	Compiling for external memory (LPDDR) and software-defined memory	72
4.6.3	Hardware setup	73
4.6.4	Level 1 cache	75
4.7	Using software-defined memory	75
4.7.1	Level 1 cache	76
4.7.2	Application implementation	76
4.7.3	XTC Tools built-in support for flash storage	76
4.7.4	Compiling for software-defined memory	76
4.7.5	Examples	77
4.7.6	Using xrun and xgdb	80

4.8	How to use arguments and return codes	81
4.9	Using XSIM	81
4.9.1	Which tile is my code running on?	82
4.9.2	Using XScope during simulation	82
4.10	Understanding XE files and how they are loaded	82
4.10.1	Prepare an XE file	83
4.10.2	Examine the XE file	83
4.10.3	Load and execute the XE file	84
5	Reference	86
5.1	Command line tools	86
5.1.1	XRUN	86
5.1.2	XSIM	88
5.1.3	XCC	92
5.1.4	XOBJDUMP	101
5.1.5	XGDB	103
5.1.6	XFLASH	105
5.1.7	XBURN	110
5.1.8	XMAKE	113
5.2	File formats and data descriptions	114
5.2.1	XMOS executable (XE) file format	114
5.2.2	XN Specification	118
5.2.3	XSIM Trace output	129
5.2.4	XSCOPE config file	131
5.3	xSCOPE performance figures	132
5.3.1	Transfer rates between the xCORE Tile and XTAG-3 or XTAG-4	132
5.3.2	Transfer rates between the XTAG-3 or XTAG-4 and Host PC	132
5.4	Libraries	132
5.4.1	lib_xcore	133
5.4.2	lib_xs1	133
5.4.3	libflash API	133
5.4.4	libquadflash API	137
5.4.5	List of devices natively supported by libflash	141
5.4.6	List of devices natively supported by libquadflash	142
5.5	XCOMMON build system	142
5.5.1	Using the XCOMMON build system	143
5.5.2	Using XMOS Makefiles to create binary libraries	145

1 Introduction

Welcome to the XMOS XTC Tools User Guide. The XTC Tools provide:

- standards compliant C and C++ compilers and language libraries
- the xC language compiler
- an assembler
- a linker
- board support for building and running programs
- a simulator
- a symbolic debugger and program loader (which use the XTAG host-to-target adapters)
- runtime instrumentation and trace libraries
- flash firmware upgrade libraries
- flash device and security deployment tools

Multi-tile and multi-core support offers features for task based parallelism and communication, accurate timing and I/O, and safe memory management. All tools components are fully integrated to support the multicore functionality.

1.1 Supported devices

This release supports the following processor generations:

- xcore.ai
- XCORE-200

The xcore.ai processor can be evaluated using the XMOS XK-EVK-XU316 evaluation board. The XCORE-200 processor can be evaluated using the XMOS XCORE-200-EXPLORER evaluation board.

1.2 Upgrading from earlier xTIMEComposer releases

Applications built with earlier releases of xTIMEComposer may be built and deployed using this release of the XTC Tools. Modifications to the source or board definitions may be required, particularly when migrating to the xcore.ai generation of processors from an earlier generation. The XC, C and C++ language compilers, the linker, the target definition language, and the *xmake* & *xcommon* build tools are unchanged compared with earlier releases.

This release provides a new library *lib-xcore* which gives applications written in the C language a mechanism to use the XCORE resources (such as ports, timers and channel ends).

Existing applications which are based on the XC language and its constructs to use XCORE resources must NOT use the *lib-xcore* library, because the *lib-xcore* library functions do not co-operate.

1.3 Document structure

The *Tools Guide* provides instructions on installing and using the XTC Tools along with reference material on each of the individual tools, the libraries, board support, flash support and deploying secure systems.

The Programming Guide illustrates how to write multi-tasking programs for the XCORE-200 and xcore.ai processors, along with reference material related to assembly and high-level languages.

1.4 References

- [xcore.ai](#)
- [XCORE-200](#)
- [XTAG Adapters](#)
- [The XMOS XS3 Architecture](#)
- [xCORE-200: The XMOS XS2 Architecture](#)
- [xCONNECT architecture](#)
- [The XMOS Programming Guide](#)
- [XC Specification](#)
- Hoare, C. A. R. (2004) [1985]. *Communicating Sequential Processes*. Prentice Hall International. ISBN 978-0-13-153271-7.
- Programming with lib_xcore. [lib_xcore_prog_guide](#)

1.5 Support

For all support relating to the XTC Tools, [raise a ticket](#). Be sure to supply the *XTC Tools version* within your ticket.

2 Installation and configuration

2.1 System requirements

The tools are officially supported on the following platforms. Unofficial support is mentioned where appropriate.

Windows

Windows 10 Pro (all releases).

Linux

Official support is provided for CentOS 7.6 on which the tools are built and extensively regression tested.

The tools also work on many modern Linux distros including Fedora and Ubuntu. Support enquiries relating to these operating systems are highly likely to be serviced.

Recent Linux distros such as Fedora and Ubuntu have dropped the required version of `libtinfo`. If you see an error similar to:

```
error while loading shared libraries: libtinfo.so.5: cannot open shared object file: No such file or
→directory
```

...you will need to install the required library.

Listing 2.1: Fedora

```
$ yum install ncurses-compat-libs
```

Listing 2.2: Ubuntu

```
$ apt-get install libtinfo5
```

Mac

Operating systems macOS 10.15 (Catalina) and newer, on both Intel and Apple M1 processors.

2.2 Installation instructions

2.2.1 Download the tools

The tools and related drivers are provided in a single platform-specific downloadable file. By default, the file will be available from:

<https://www.xmos.com/software-tools>

In some circumstances you may be instructed to download specific versions of the tools from other locations.

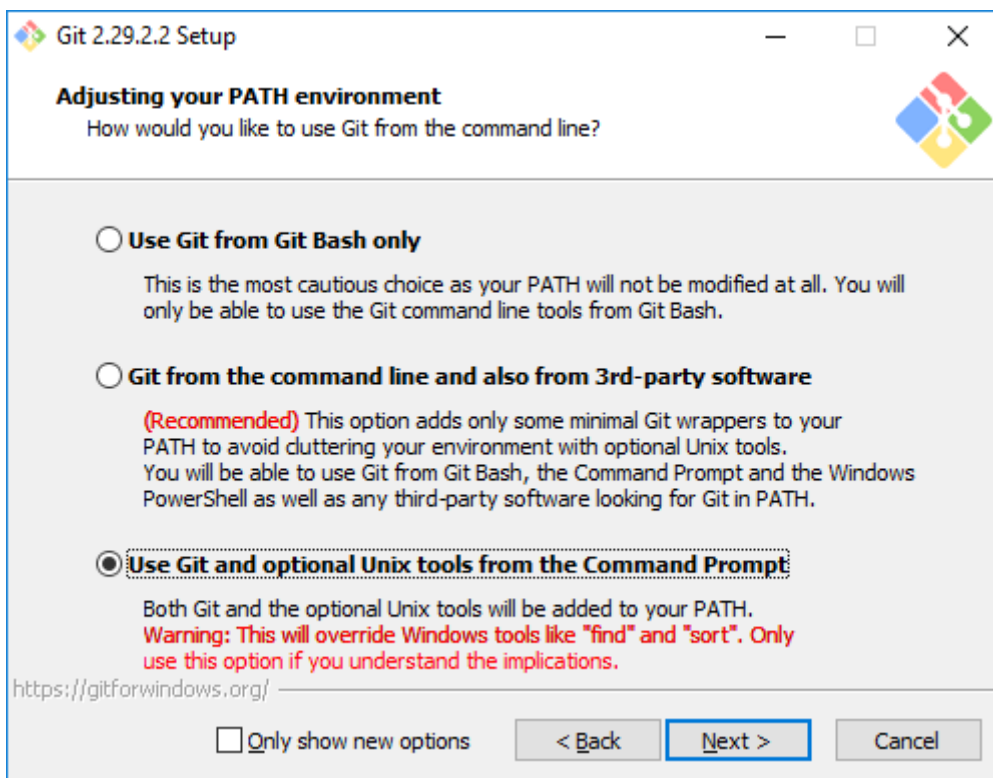
2.2.2 Install the tools

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

Windows

1. Run the installer. Click “Yes” when asked to allow the installer to make changes to your computer.
2. The installer GUI will guide you through the installation. For a default installation, click “Next” on every page, then “Install”.
3. *Recommendation:* Install a ‘bash’ command line environment and commonly used Linux tools. This is recommended because - for the sake of brevity - all the examples provided in this documentation assume a bash environment.

One way of installing a basic bash environment on a Windows machine is to install [Git for Windows](#). Follow the standard instructions for installation and, when asked, make the selection as below:



Linux

1. Uncompress the archive to your chosen installation directory. The example below will install to your home directory:

```
$ tar -xf archive.tgz -C ~
```

In this example, the tools will install to `~/XMO5/XTC/version`. This location will henceforth be known as your tools installation location.

Mac

1. Double-click the downloaded installer to open it, and then drag the 'X' icon into your Applications folder.
2. Unmount the installer.

2.2.3 Check your installation

After installation, check it by following the instructions at [Configuring the command-line environment](#).

2.2.4 Configure the XTAG USB drivers

The tools interface to development boards over USB via an XTAG adapter. Most development boards require an external XTAG adapter that connects to the board via an XSYS or XSYS2 connector. Some development boards include an integrated XTAG adaptor.

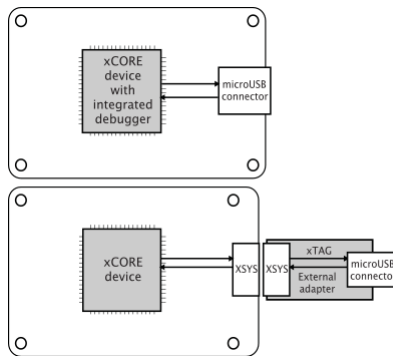


Fig. 2.1: XTAG adapter configurations used with xCORE development boards

Windows

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

Linux

The XTAG drivers for Linux are installed as part of the normal installation process. However, to make them accessible by all users, an additional step requiring root access is needed. This step is likely only required once on a given development machine, or when new XTAG devices are released by XMOS. Change directory to the `scripts` folder of your tools installation and run the setup script:

```
$ sudo ./setup_xmos_devices.sh
```

Reconnect any currently connected XTAG devices for the changes to be applied.

To check that the XTAG devices are available and accessible run the following script in the same directory:

```
$ ./check_xmos_devices.sh
```

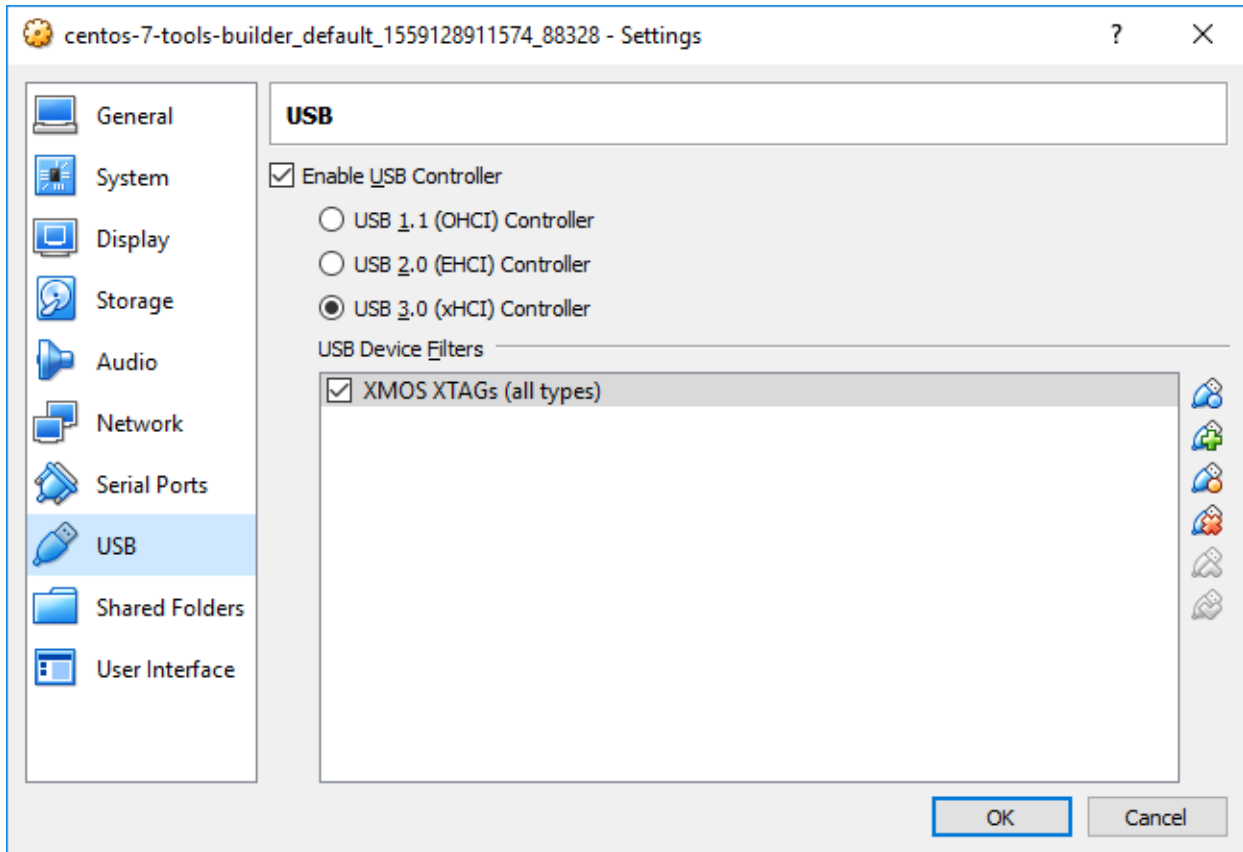
As shown, this script should be run as the normal user, not root.

Mac

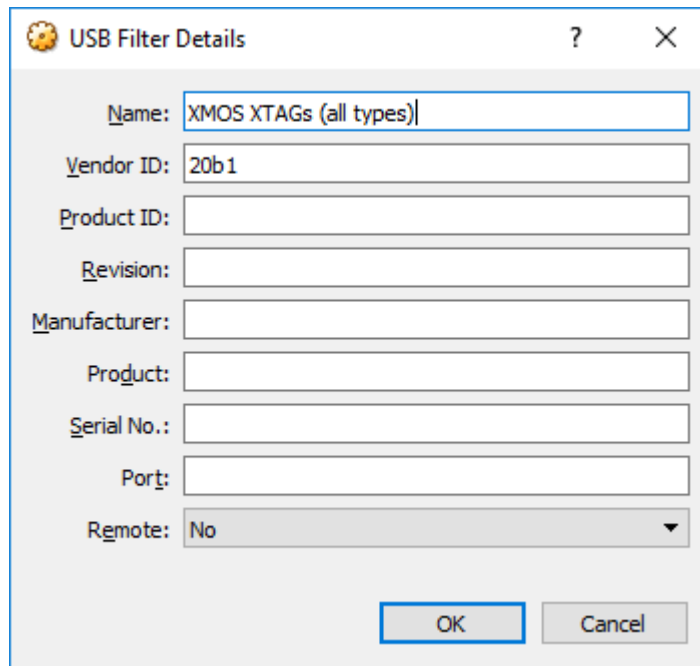
USB driver support is provided natively on OS X.

2.2.4.1 Using XTAG from within a VM

If you are using the tools from within a VM, you'll need to instruct that the XTAGs attached to the host be tunneled through to the guest machine. The USB settings required in Oracle VM VirtualBox Manager are shown below:



"Add" or "Edit" your USB filter details such that they appear as below. It is important that the "Product ID" field is empty.



2.2.5 Check XTAG access

Configure the environment. Connect a development board to your host machine via an XTAG adaptor as necessary. Ensure that the development board is powered.

Use `xrun -l` to list the devices available for debugging. The first time you issue this command, it may take several seconds for the XTAG firmware to be downloaded. An example of the expected result is:

```
$ xrun -l

Available XMOS Devices
-----

  ID   Name                Adapter ID      Devices
  --   ----                -
  0    XMOS XTAG-3          wff.G58J       P[0]
```

If the 'Devices' column lists None, it means the XTAG itself cannot gain access to the development board. Check that the development board itself is powered, and that the XSYS/XSYS2 connection is well made.

2.3 Configuring the command-line environment

2.3.1 Configure the environment

The tools use a set of environment variables when searching for header files, libraries and target devices (see [Environment](#)). To add the tools to the path and configure the default set of environment variables:

Windows

1. Choose *Start>XMOS>Command Prompt(15.x.x)*. This will open a DOS command prompt with the environment variables already set.
2. *Recommendation*: Start a bash environment.

For the sake of brevity, all the command line examples in this documentation assume that the user is using a *bash* environment. At the DOS command prompt, type:

```
> bash
```

To install *bash* and the basic Unix tools, see [Windows installation](#).

Linux

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

```
$ source SetEnv
```

Mac

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

```
$ SetEnv .command
```

2.3.2 Quick check the environment

You may wish to check that your tools environment has been correctly configured using, say, *xcc --help*. To do this, type:

```
$ xcc --help
```

If your environment is working, then this will print some quick help text.

2.3.3 Query the XTC Tools version

To find the XTC Tools version:

```
$ xcc --version
```

The reported version should be provided in any bug report.

The build version of individual program within the XTC Tools can be found using their *--version* option. For example, *xcc --version*.

2.4 Configuring an IDE

Some users may choose to use an Integrated Development Environment to augment their use of the tools. The tools can be integrated to varying degrees with a number of common user-supplied IDEs.

The instructions given below are simply suggestions; there are many ways of integrating the tools with even a single IDE, and the user is encouraged to experiment to suit their workflow.

2.4.1 Configuration common to all IDEs

2.4.1.1 Creation of project content

In order to reduce the amount of repetition later in this document, we define a set of example projects which will be used as a basis for each specific IDE.

Create two example projects:

```
$ mkdir -p projects/single-tile
$ mkdir -p projects/switch-setup
```

Within `projects/single-tile`:

- Create `main.c` as per [A single-tile program](#)
- Create the following three bash scripts:

Listing 2.3: `build.sh`

```
#!/bin/bash
CWD=`pwd`
xcc -target=XCORE-200-EXPLORER -g $CWD/main.c
```

Listing 2.4: `run.sh`

```
#!/bin/bash
xrun --io a.xe
```

Listing 2.5: `debug.sh`

```
#!/bin/bash
xgdb a.xe
```

Within `projects/switch-setup`:

- Create `multitile.xc` and `main.c` as per [Communicating between tiles](#)
- Create three bash scripts as above, but with a slight modification to `build.sh`:

Listing 2.6: `build.sh`

```
#!/bin/bash
CWD=`pwd`
xcc -target=XCORE-200-EXPLORER -g $CWD/mapfile.xc $CWD/main.c
```

You might later choose to customise these bash scripts; they will act as placeholders for now.

2.4.1.2 Choice of build System

Users of the tools may use any build system. The examples below demonstrate use of deliberately basic "build system" such that the examples are not dependent on installation of a build system.

2.4.1.3 Integration of "Run" and "Debug" functionality

The current architecture of the XRUN and XGDB tools is such that integration with 3rd-party IDEs is limited to terminal operation only. The guidance below therefore "integrates" the run and debug functionality into the terminal window of the respective IDE.

2.4.2 Visual Studio Code

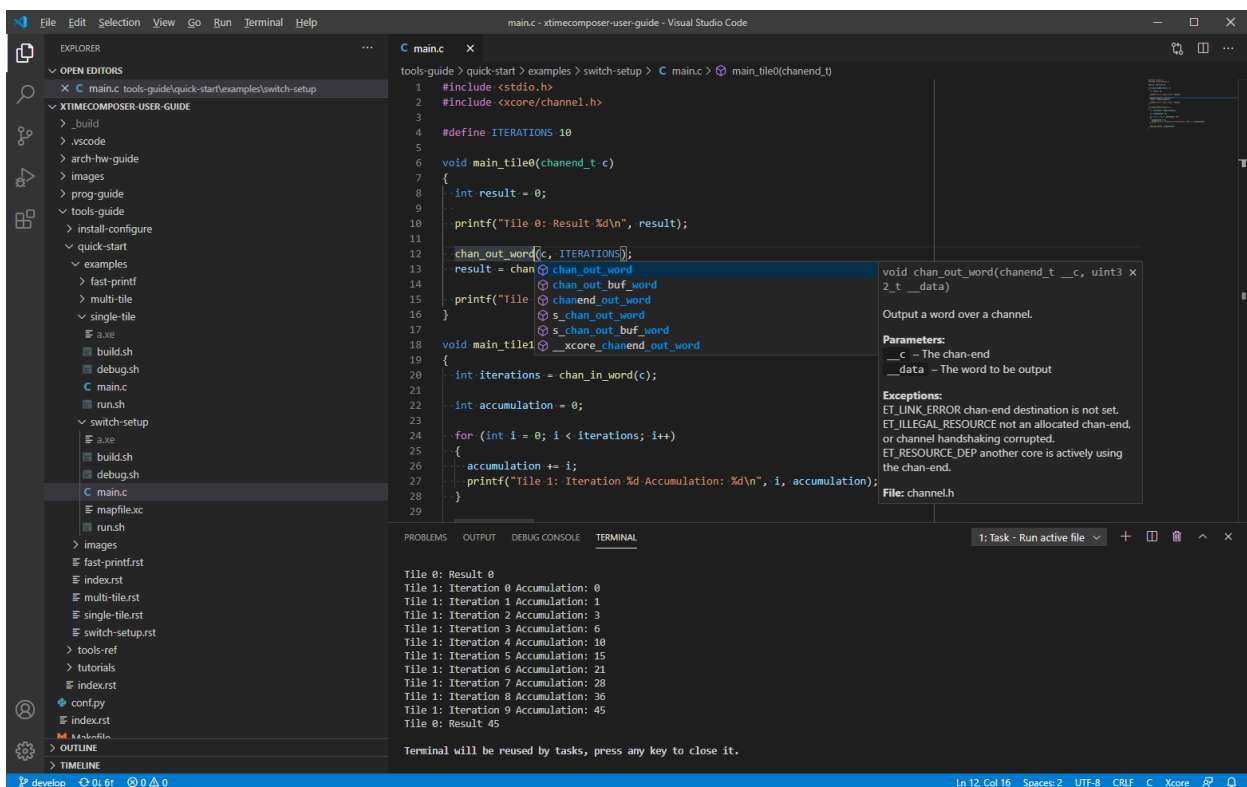


Fig. 2.2: Example of tools plugged into VS Code

2.4.2.1 Pre-requisites

1. Install [Visual Studio Code](#).
2. Install the [C/C++ extension](#) for VS Code.
3. Ensure the *command line environment is working*.
4. Create the *example project content*



2.4.2.2 Start VS Code and check the environment

From a bash terminal where the *command line environment is working*, start VS Code such that it can see both projects within its workspace:

```
$ cd projects
$ code .
```

Check that the tools are available within VS Code. Select *View>Terminal* and check that the following command gives the expected results:

```
$ xcc --version
```

2.4.2.3 Configure VS Code

If not already present, create a folder `.vscode` within the `projects` folder and, within it, create two files:

Listing 2.7: tasks.json

```
{
  "version": "2.0.0",
  "tasks": [
    {
      "label": "Build active file",
      "type": "shell",
      "command": "bash",
      "args": [
        "./build.sh"
      ],
      "options": {
        "cwd": "${fileDirname}"
      },
      "problemMatcher": [
        "$gcc"
      ],
      "group": {
        "kind": "build",
        "isDefault": true
      }
    },
    {
      "label": "Run active file",
      "type": "shell",
      "command": "bash",
      "args": [
        "./run.sh"
      ],
      "options": {
        "cwd": "${fileDirname}"
      },
      "problemMatcher": []
    },
    {
      "label": "Debug active file",
      "type": "shell",
      "command": "bash",
      "args": [
        "./debug.sh"
      ],
      "options": {
        "cwd": "${fileDirname}"
      }
    }
  ]
}
```

(continues on next page)

```

    },
    "problemMatcher": []
  }
]
}

```

Listing 2.8: c_cpp_properties.json

```

{
  "version": 4,
  "configurations": [
    {
      "name": "Xcore",
      "includePath": [
        "${workspaceFolder}/**"
      ],
      "defines": ["XSCOPE_IMPL"],
      "compilerPath": "${XMOS_TOOL_PATH}/bin/xcc",
      "cStandard": "gnu11",
      "cppStandard": "gnu++14",
      "intelliSenseMode": "clang-x64",
      "compilerArgs": [
        "-save-temps"
      ]
    }
  ]
}

```

2.4.2.4 Use VS Code

For the projects/single-tile application:

- Open projects/single-tile/main.c. With this file active, use the keyboard shortcut `Ctrl+Shift+B` to build the application.
- To run the application without debugging, hit `Ctrl+P`, then start typing task `Run`. When *Run active file* is highlighted, press enter. The application output will appear in the Terminal view.
- To debug the application, hit `Ctrl+P`, then start typing task `Debug`. When *Debug active file* is highlighted, press enter. An interactive debug session will begin in the Terminal view. When finished debugging, be sure to issue `quit` to end the session.

To build, run and debug the projects/switch-setup application, carry out the same actions, but first make sure that projects/switch-setup/main.c is the active file.

You may check that VS Code's 'Intellisense' is working correctly by hovering the mouse over `chan_out_word` in projects/switch-setup/main.c.

2.4.2.5 Further suggestions

With the tools now integrated into VS Code, you might try:

- Setting up some keyboard shortcuts for the Run and Debug tasks
- Adding the [XSCOPE example](#) to your projects folder and using the [WaveTrace](#) VS Code extension to view the generated .vcd file.

2.4.3 Eclipse

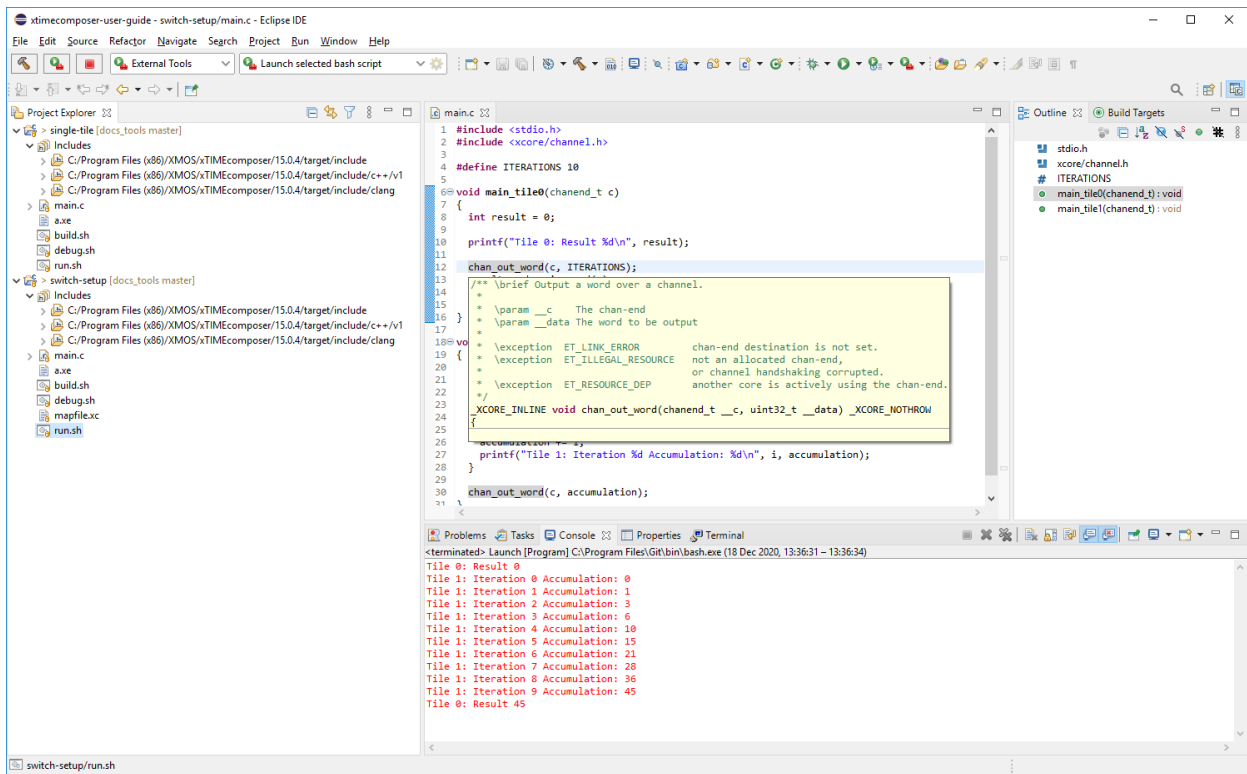


Fig. 2.3: Example of tools plugged into Eclipse

2.4.3.1 Pre-requisites

1. Install [Eclipse CDT](#).
2. Ensure the *command line environment is working*.
3. Create the *example project content*

2.4.3.2 Start Eclipse and check the environment

From a bash terminal where the *command line environment is working*, start Eclipse:

```
$ eclipse
```

Create a new Eclipse workspace in the `projects` folder you have created.

Next check that Eclipse can access the tools. Open a Terminal window using *Window>Show View>Terminal*. Within the window, click the “Open a Terminal” icon and launch a bash terminal. Confirm that the following command produces the expected results:

```
$ xcc --version
```

2.4.3.3 Configure Eclipse

Follow these steps to import the `single-tile` project content you’ve already created into your Eclipse workspace:

1. Navigate to *File>New>Makefile Project with Existing Code*
2. Project Name: `single-tile`
3. Existing Code Location: “Browse” to `projects/single-tile`
4. Toolchain for Indexer Settings: Select `Cross GCC` (Untick “Show only...” if not visible)
5. Click “Finish”

This will create the `single-tile` project in the “Project Explorer” window. We next make some adjustments (because we’re not actually using GCC as previously selected):

1. Right-click the `single-tile` project and select “Properties”
2. Navigate to *C/C++ General>Preprocessor Include Paths, Macros etc.*
3. Select the “Providers” tab
4. Adjust the “Command to get compiler specs” to read: `xcc $FLAGS -march=xs2a -E -P -vv -dD "$INPUTS"`
5. Click “Apply and Close”

These adjustments allow the Eclipse source code indexer to find the correct include files within the toolkit. Confirm this is working in two ways:

1. Within the “Project Explorer” window, navigate to `single-tile>Includes`. Confirm that the path to the contained include files is as expected.
2. Open the file `main.c`. Confirm that the “Problems” window is empty (which indicates that indexing has completed successfully).

We now make further modifications because, in this example, we’re not going to use a `Makefile` as previously selected:

1. Right-click the `single-tile` project and select “Properties”
2. Navigate to *C/C++ Build* and select the “Builder Settings” tab
3. Untick “Use default build command” and set “Build command” to `bash build.sh`
4. Click “Apply and Close”

Repeat these steps for the `switch-setup` project.

We now configure Eclipse for running and debugging:

1. Navigate to *Run>External Tools>External Tools Configurations...*

-
2. Click the icon “New launch configuration”.
 3. Select the “Main” tab and enter the following details:
 - Name: `Launch selected bash script`
 - Location: “Browse File System” to locate bash executable
 - Working Directory: `$container_loc`
 - Arguments: `$selected_resource_name`
 4. Select the “Build” tab and select “The project containing the selected resource”
 5. Click “Apply”
 6. Click “Close”

2.4.3.4 Use Eclipse

Within the `single-tile` project:

- To build:
 1. Select any file within the project
 2. Click on the “Build” icon
- To build and run:
 1. Select `run.sh` within the project
 2. Navigate to `Run>External Tools>Launch selected bash script`
- To build and debug:
 1. Select `debug.sh` within the project
 2. Navigate to `Run>External Tools>Launch selected bash script`

Now try the same within the `switch-setup` project.

2.4.4 xTIMEcomposer Studio

Integration of the XTC toolchain with the xTIMEcomposer Studio IDE packaged with previous tools releases is not recommended. Developers desiring a similar IDE experience should follow the instructions for [Eclipse](#).

2.5 Transitioning from older tools releases

Users of the XMOS xTIMEComposer Tools will notice some changes when using this XTC Tools release. This section aids users in migrating to the XTC Tools.

2.5.1 Programming language

A notable change within the XTC Tools is the move towards the use of the C language instead of XC as the preferred programming language for the xcore. The advantages of using C for xcore programming are:

- ANSI/ISO compliance
- Familiar to most embedded software developers
- 3rd-party algorithms and routines may be deployed
- There is an extensive support network

The xcore developer is encouraged to write communicating sequential processes (CSP) using tasks with their own private memory. This is largely enforced in XC and remains an excellent approach to parallel programming, and the same design pattern is recommended when programming in C. The move towards the use of C retains access to all of the unique benefits of the xcore architecture alongside benefits listed above.

For further information on communicating sequential processes see [References](#).

2.5.1.1 Existing applications using XC

Existing source code using the XC language, (with or without C or C++), can be built using this XTC Tools release. The XC compiler itself will be maintained for existing applications and libraries.

For guidance on projects developed within older tools releases, see [Migrating existing projects](#).

2.5.1.2 New applications

New applications code should be written in C or a combination of C and C++.

The underlying hardware features of the xcore can be accessed through the C language using a new system library [lib_xcore](#). Use of the new library is introduced in the guide: [lib_xcore_prog_guide](#).

Where new C code is required to interact with existing XC code, existing guidance should be followed. See [call_between_c_xc](#).

Writing multi-tile applications currently still requires the writing of a minimal declarative XC source file, often called `multitile.xc` as per the examples: [Targeting multiple tiles](#) and [Communicating between tiles](#). The file should not contain any procedural code, and should contain only the bare minimum to:

- Declare a C entry point on each tile
- Declare the channels over which the tiles communicate

The C language does not include equivalents for XC's `[[combine]]`, `[[combinable]]` or `[[distributable]]` keywords. Combining tasks onto a single logical core must be done manually, using functions in [lib_xcore](#) to wait for an event from one of many sources, or by using an RTOS.

Similarly, the C language does not have an equivalent for XC interfaces which are accessed via the `interface` keyword. Implementation of interfaces and their underlying transport and protocol can now be performed by the application developer, allowing greater scope for choosing an implementation appropriate to the real-time requirements and any resource availability constraints.

Users who rely on the XC language protecting them against unsafe concurrent access to shared memory should take appropriate care when programming in the C language. Similarly, care should be taken to ensure that resources are correctly allocated and deallocated before/after use, since such allocation is not handled by the language itself. These are all normal considerations for a C programmer.

Experienced XC users transitioning to C/[lib_xcore](#) may find the following resources useful:

The following bookmarks provide a convenient access into the video for later reference:

- [Ports](#)

-
- *Timers*
 - *Locks*
 - *Par*
 - *Channels*
 - *Select*
 - *Other features*
 - *Considerations*
 - *Automatic stack size calculation*
 - *Multi-tile applications*

The presentation used in the video may be downloaded: [C_for_xCore_2021_02_08.pdf](#)

In addition, an cheat is provided.

2.5.2 Graphical IDE

The XTC Tools no longer contain a graphical IDE within the installation package. Instead, the tools are intended to be integrated with the user's preferred IDE as per the examples in [Configuring an IDE](#).

Viewing of offline XSCOPE output is similarly delegated to the user's preferred VCD viewer. Use of GTKWave is demonstrated in [Using XSCOPE for fast "printf debugging"](#), though other viewers are available.

2.5.3 Migrating existing projects

Some very minor changes in configuration files and procedures are required when migrating to the XTC Tools from xTIMEComposer. The following changes may be required.

2.5.3.1 Flash-related changes

2.5.3.1.1 XFLASH option changes

The version of XFLASH supplied with tools 15.0.x no longer performs image compression thus its default and only behaviour is to not compress the image. The option `xflash --no-compression` is therefore no longer provided. Simply do not supply the option.

2.5.3.1.2 XFLASH upgrade image guidance

When using the XTC Tools to produce a flash upgrade image for a factory image created under earlier tools releases, be sure to supply the correct value for the `xflash --factory-version`.

2.5.3.2 XTAG adapters

The XTAG3 and XTAG4 adapters are supported by this release. The XTAG4 is a new-generation XTAG adapter. xcore.ai devices must use this adapter because it interfaces to them at lower voltages (1v8) which they require.

On Windows the XTAG drivers have been replaced, and a Windows Service is run when the host computer boots to manage connected XTAGs.

3 Quick start

This introduction to the tools aims to provide a quick tour of the available tools and how to use them to create, debug and deploy collaborative applications on an XCore platform.

Note: The reader is expected to already be familiar with:

- XCore architecture fundamentals
- XCore programming fundamentals

To try the examples described within this introduction, the reader will also need the following pre-requisites:

- [Installed and configured tools](#)
 - An XTAG and XCORE-200-EXPLORER hardware
-

3.1 A single-tile program

This section introduces some of the key tools required to build and run a simple program on a single tile.

To run on hardware one of the following XMOS evaluation boards is required:

- XK-EVK-XU316 (with an xcore.ai processor)
- XCORE-AI-EXPLORER (with an xcore.ai processor)
- XCORE-200-EXPLORER (with an XCORE-200 processor)

Replace the target definition string used in the following examples (*-target=XK-EVK-XU316*) with that of the board being used, for example: *-target=XCORE-AI-EXPLORER*.

The following board and XTAG type combinations are supported:

- The XK-EVK-XU316 board has an on-board XTAG4 adapter with a dedicated USB micro-B interface which must be plugged into the host computer
- The XCORE-AI-EXPLORER board requires an XTAG4 adapter to be plugged into its XSYS2 connector
- The XCORE-200-EXPLORER board requires an XTAG3 adapter to be plugged into its XSYS connector

See: [Configure the XTAG](#) and [Check the XTAG](#).

3.1.1 Build an executable

The following is a simple example contained in the single source file `main.c`:

Listing 3.1: `main.c`

```
1 #include <stdio.h>
2
3 int main(void) {
4     printf("Hello world!\n");
5     return 0;
6 }
```

This is built into an executable using the tool [XCC](#). The output is an xcore executable in the XE file `a.xe`:

```
$ xcc -target=XK-EVK-XU316 -g main.c
```

The option `-g` tells the XCC tool to add debug information. This will be used later.

3.1.1.1 The target definition

The option `-target=XK-EVK-XU316` provides the tools with a description of the target board on which the application will execute. This particular XMOS evaluation board description is provided with the XMOS XTC Tools (in the `target` subdirectory). A user will normally provide a description of their board, derived from an XMOS evaluation board description.

The board description includes information about the xcore devices present, their architecture, external flash devices and frequencies at which sub-systems are clocked.

3.1.1.2 The content of an XE file

An *XE file* is often referred to as an executable. However, it is actually a package of files which include an ELF file for each tile in the target definition.

3.1.2 Running the program on the simulator

The tools include a near cycle-accurate hardware simulator *XSIM*. XSIM simulates of entire XMOS device, and may also simulate certain external devices such as a QSPI flash memory.

```
$ xsim a.xe
Hello world!
```

3.1.3 Run on real hardware

The *XRUN* tool is used to launching an executable on real hardware.

Connect your XK-EVK-XU316 development board to your host PC via the XTAG3 or XTAG4 adaptor. Make sure you've also supplied power to the development board itself.

Run `a.xe` using `xrun --io`:

```
$ xrun --io a.xe
Hello world!
```

Note: The option `-io` is required to make *XRUN* show the application *printf* output on the host computer's console, which also makes it wait until the application has terminated.

Note: If you have problems with this step, you may need to *configure* and *check* your XTAG setup.

Congratulations! You've just built and executed your first xcore application.

3.1.3.1 Debugging using XGDB on real hardware

You have already used the *XGDB* debugger indirectly when you used the simplified *XRUN* tool. However, if you want greater insight into how the application is running, you need to use the *XGDB* debugger directly. Start the debugger:

```
$ xgdb a.xe
```

This starts a new debug session with a new (gdb) prompt. You must now *connect* to the attached hardware and *load* the application code:

```
(gdb) connect
...
(gdb) load
...
```

From now on, using *XGDB* on this single-threaded program is the same as using normal *GDB*. For instance, create a breakpoint, run up to that breakpoint, step to the next line and quit:

```
(gdb) break main
Breakpoint 1 at 0x400fc: file main.c, line 4.
(gdb) continue

Breakpoint 1, main () at main.c:4
4      printf("Hello world!\n");
Current language: auto; currently minimal
(gdb) step
Hello world!
5      return 0;
(gdb) quit
```

3.1.3.2 Debugging using XGDB

It is possible to use *XGDB* to debug on the *XSIM* simulator. The steps are identical to *Debugging using XGDB on real hardware*, except use *connect -s* to connect to the simulator instead of the hardware:

```
(gdb) connect -s
...
(gdb) load
...
```

3.1.4 Summary

In this brief overview of some command line tools, you have built an application to produce an *XE file*. You have run and debugged the *XE file* on real hardware, and done the same on the simulator.

Through the tour you have used the following tools:

- *XCC*
- *XRUN*
- *XGDB*
- *XSIM*

3.2 Targeting multiple tiles

In the [previous example](#), the code targeted only a single tile. Most real applications will use two or more tiles. It is not possible to build an application for multiple tiles using the C language (as the tiles are independent processors).

To build a multi-tile application, a top-level, multi-tile XC file must be provided to specify the entry-point for each tile. The code running on a tile communicates with another tile using XCORE channels.

Note: The syntax of the multi-tile XC file is C language like. However, the grammar is not a pure super-set of the C language.

3.2.1 Providing a multi-tile file

To place code onto both of the tiles on a XK-EVK-XU316, it is necessary to provide a file which we will call `multitile.xc`. An example is shown below:

Listing 3.2: multitile.xc

```
#include <platform.h>

extern "C" {

void main_tile0();
void main_tile1();

}

int main(void)
{
    par {
        on tile[0]: main_tile0();
        on tile[1]: main_tile1();
    }

    return 0;
}
```

This `multitile.xc` references the two functions in `main.c`:

Listing 3.3: main.c

```
#include <stdio.h>

void main_tile0()
{
    printf("Hello from tile 0\n");
}

void main_tile1()
{
    printf("Hello from tile 1\n");
}
```

Now build and execute this multi-tile application on real hardware to see the printed output:

```
$ gcc -target=XK-EVK-XU316 multitile.xc main.c
$ xrun --io a.xe
Hello from tile 0
Hello from tile 1
```

3.2.2 Summary

In this example, you have written a `multitile.xc` using the declarative components of XC language to deploy two C functions onto the two tiles of an XCORE.AI processor.

See also:

At this point, you might proceed to the next topic, or you might chose to explore this example further:

- [Which tile is my code running on?](#)
- [Understanding XE files and how they are loaded](#)

3.3 Communicating between tiles

Usually application code running on separate tiles will need to communicate and synchronise their activity. Such communication is passed via internal interconnect and between devices using XLINKS. See [xCONNECT architecture](#). A top-level multi-tile XC file can also declare channels between the tiles it places entry-points on.

3.3.1 Declaring a channel in the multi-tile file

The XK-EVK-XU316 has two tiles and interconnect for communication between the tiles within the package. The tools will automatically configure a channel via the interconnect using `multitile.xc` as below:

Listing 3.4: multitile.xc

```
#include <platform.h>

typedef chanend chanend_t;

extern "C" {
    void main_tile0(chanend_t);
    void main_tile1(chanend_t);
}

int main(void)
{
    chan c;

    par {
        on tile[0]: main_tile0(c);
        on tile[1]: main_tile1(c);
    }

    return 0;
}
```

In this example, the tile level entrypoint functions each accept a `chanend`. In the forward-declarations these functions take a `chanend` argument; this is compatible with a `lib_xcore chanend_t` at link-time. The `chan` key-

word is used in `main` to declare a channel. When a channel is passed to tile-level `main`, an end is automatically assigned to each entrypoint which uses it. In the task-level entrypoints, these `chanends` are used just like local ones.

3.3.2 Using the channel

In the tile-level entrypoints, the `chanends` are used as per `/prog-guide/quick-start/c-programming-guide/index`. This time in `main.c`, we have a more interesting pair of tile functions. Each is passed a `chanend_t c`, and they use it to communicate between tiles and synchronise their activities.

Listing 3.5: `main.c`

```
#include <stdio.h>
#include <xcore/channel.h>

#define ITERATIONS 10

void main_tile0(chanend_t c)
{
    int result = 0;

    printf("Tile 0: Result %d\n", result);

    chan_out_word(c, ITERATIONS);
    result = chan_in_word(c);

    printf("Tile 0: Result %d\n", result);
}

void main_tile1(chanend_t c)
{
    int iterations = chan_in_word(c);

    int accumulation = 0;

    for (int i = 0; i < iterations; i++)
    {
        accumulation += i;
        printf("Tile 1: Iteration %d Accumulation: %d\n", i, accumulation);
    }

    chan_out_word(c, accumulation);
}
```

Building and executing this multi-tile application produces the expected result:

```
$ xcc -target=XK-EVK-XU316 multitile.xc main.c
$ xrun --io a.xe
Tile 0: Result 0
Tile 1: Iteration 0 Accumulation: 0
Tile 1: Iteration 1 Accumulation: 1
Tile 1: Iteration 2 Accumulation: 3
Tile 1: Iteration 3 Accumulation: 6
Tile 1: Iteration 4 Accumulation: 10
Tile 1: Iteration 5 Accumulation: 15
Tile 1: Iteration 6 Accumulation: 21
Tile 1: Iteration 7 Accumulation: 28
Tile 1: Iteration 8 Accumulation: 36
Tile 1: Iteration 9 Accumulation: 45
Tile 0: Result 45
```

3.3.3 Summary

You have now written a multi-tile application which, through the declarations in `multitile.xc`, sets up a channel between the tiles and provides pre-allocated channels for each end of this channel to the C functions.

3.4 Using XSCOPE for fast “printf debugging”

The previous example in [Communicating between tiles](#) was very slow due to the time taken to print each line of text to the terminal. This is made particularly noticeable if the value of `ITERATIONS` is increased.

This default behaviour can be slow for a number of reasons:

- A JTAG interface is being used to control the transfer.
- There is no buffering for the transferred data.
- The data is transferred via the debug kernel. All running logical cores on a tile are halted whilst the transfer on a given logical core completes.

It is clear to see that the default behaviour can play havoc if the application being debugged has any real-time constraints. To mitigate this problem, the XSCOPE interface is provided.

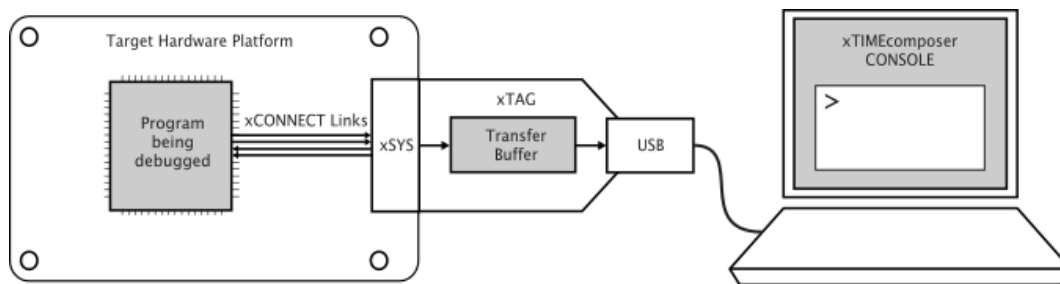


Fig. 3.1: XTAG debug setup showing JTAG and xlink connections

The XSCOPE interface makes use of a physical high bandwidth XLINK connection to the XTAG debugger. Buffering is provided on the XTAG debugger itself. Through this mechanism there is no need to halt all the logical cores whilst another conducts a transfer.

The net result is that XSCOPE can be used as a high performance debug interface with minimal impact on the real-time performance of the application under examination. Use of XSCOPE is thus recommended in almost all cases.

3.4.1 Configuring and using XSCOPE

This example presumes that we are using XSCOPE to accelerate debug of the previous example: [Communicating between tiles](#).

XSCOPE is configured by creating an XML *XSCOPE config file* with the suffix: `.xscope`. Here we create the most basic configuration file:

Listing 3.6: basic.xscope

```
<xSCOPEconfig ioMode="basic" enabled="true">
</xSCOPEconfig>
```

XCC recognises files with this suffix. The file is provided to XCC as an argument during building (specifically during the link step if compiling and linking as separate build steps):

```
xcc -target=XCORE-200-EXPLORER basic.xscope multitile.xc main.c
```

Now run the application using `xrun --xscope` (not `--io` as previously used):

```
xrun --xscope a.xe
```

The printed output is now produced seemingly instantaneously. Try increasing the value for `ITERATIONS`.

Important: Use of XSCOPE on one or more logical cores of a tile results in a single channel end being allocated for use by XSCOPE.

Tip: Try changing `ioMode` from `basic` to `timed`. This causes the output timestamp to be displayed with the printed data in the console. Note that this also reduces the amount of data that can be buffered at any time.

Warning: An alternative approach for configuring XSCOPE used the `xscope_user_init()` function. This approach has since been deprecated.

3.4.2 Using XSCOPE “probes”

As well as using XSCOPE to accelerate literal “printf debugging” as above, XSCOPE “probes” can be used to send named streams of data to the host for debugging purposes. These might be streams of ADC samples, say.

The underlying mechanism used for XSCOPE probes is the same as that used by the calls to `printf()` above. Probes however are even more efficient, as they avoid overheads both in CPU time and data transfer.

Probes are added into the XSCOPE config as follows:

Listing 3.7: probes.xscope

```
<xSCOPEconfig ioMode="basic" enabled="true">
  <Probe name="Tile0-result" type="CONTINUOUS" datatype="UINT" units="mV" enabled="true" />
  <Probe name="Tile1-i" type="CONTINUOUS" datatype="UINT" units="mV" enabled="true" />
  <Probe name="Tile1-accumulation" type="CONTINUOUS" datatype="UINT" units="mV" enabled="true" />
</xSCOPEconfig>
```

With XSCOPE probes now configured, they can be exploited by adding the highlighted modifications into `main.c`:

Listing 3.8: main.c

```
#include <stdio.h>
#include <xcore/channel.h>
#include <xscope.h>

#define ITERATIONS 10

void main_tile0(chanend_t c)
{
  int result = 0;

  printf("Tile 0: Result %d\n", result);
  xscope_int(0, result);
```

(continues on next page)

(continued from previous page)

```
chan_out_word(c, ITERATIONS);
result = chan_in_word(c);

printf("Tile 0: Result %d\n", result);
xscope_int(0, result);
}

void main_tile1(chanend_t c)
{
    int iterations = chan_in_word(c);

    int accumulation = 0;

    for (int i = 0; i < iterations; i++)
    {
        accumulation += i;
        printf("Tile 1: Iteration %d Accumulation: %d\n", i, accumulation);
        xscope_int(1, i);
        xscope_int(2, accumulation);
    }

    chan_out_word(c, accumulation);
}
```

Build similarly to before:

```
xcc -target=XCORE-200-EXPLORER probes.xscope multitile.xc main.c
```

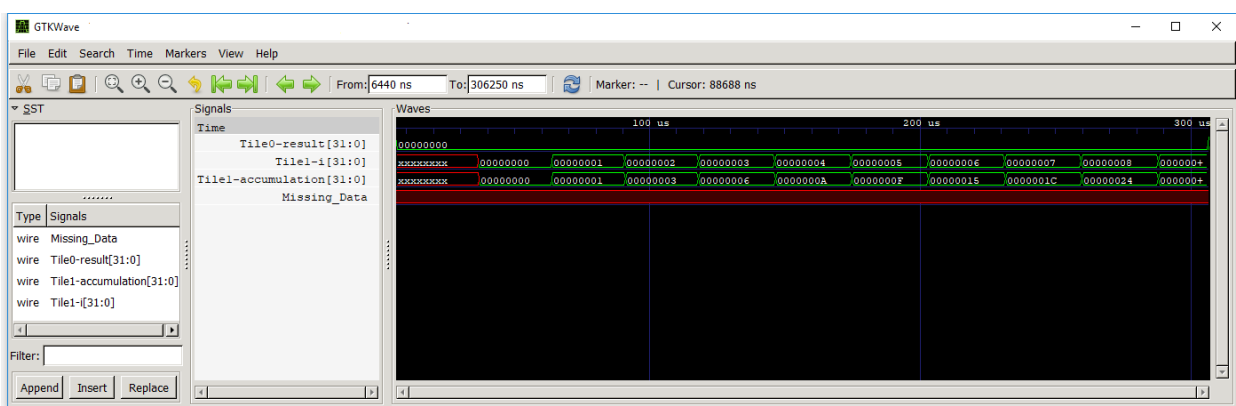
This time, when running, add `--xscope-file` to specify a file to write the probe output into:

```
$ xrun --xscope --xscope-file xscope.vcd a.xe
```

A standard VCD file `xscope.vcd` is produced in the current directory, which can be opened with any 3rd-party VCD viewer. One option is [GTKWave](#). To use `gtkwave` to view the VCD file:

```
$ gtwave xscope.vcd
```

After 'dragging' the signals into the viewing area, the display might look like this:



Installation and use of GTKWave or other VCD viewers is outside the scope of this document.

Note: XScope tracing as described in this example can be performed [using XSIM](#).

3.4.3 Summary

You can now use the XScope facility to perform “printf debugging”. In fact, this method (using `xrun --xscope`) should always be the preferred approach, instead of the default approach (using `xrun --io`) shown in previous tutorials.

3.5 Debugging with XGDB

XGDB is an extension of GDB which adds support for multi-tile debugging of Xcore applications in the form of *XE files*. Therefore, to the greatest possible extent, XGDB behaves like GDB and therefore most information can be found on 3rd-party websites and resources.

Its use for debugging multi-tile applications is more unusual; this example aims at demonstrating how XGDB can be used to follow an Xcore application as control passes from core to core and tile to tile.

3.5.1 Create example

To illustrate the use of XGDB for debugging, we create a token-passing ring operating across two tiles:

Listing 3.9: multitile.xc

```
#include <platform.h>

typedef chanend chanend_t;

extern "C" {
    void main_tile0(chanend_t, chanend_t);
    void main_tile1(chanend_t, chanend_t);
}

int main(void)
{
    chan tile0_to_tile1;
    chan tile1_to_tile0;

    par {
        on tile[0]: main_tile0(tile1_to_tile0, tile0_to_tile1);
        on tile[1]: main_tile1(tile0_to_tile1, tile1_to_tile0);
    }

    return 0;
}
```

On each tile, we pass the token via two cores. The ring is kick-started by tile[1]:

Listing 3.10: main.c

```
1 #include <stdio.h>
2 #include <xcore/channel.h>
3 #include <xcore/parallel.h>
4
5 DECLARE_JOB(process, (chanend_t, chanend_t, char *, int));
6 void process(chanend_t cIn, chanend_t cOut, char * name, int init)
7 {
8     if (init)
9     {
10         chan_out_word(cOut, 1);
```

(continues on next page)



(continued from previous page)

```
11 }
12
13 while (1)
14 {
15     int token = chan_in_word(cIn);
16     printf("%s\n", name);
17     chan_out_word(cOut, token);
18 }
19 }
20
21 void main_tile0(chanend_t cIn, chanend_t cOut)
22 {
23     channel_t chan = chan_alloc();
24
25     PAR_JOBS(
26         PJOB(process, (cIn, chan.end_a, "tile0-core0", 0)),
27         PJOB(process, (chan.end_b, cOut, "tile0-core1", 0)));
28
29     chan_free(chan);
30 }
31
32 void main_tile1(chanend_t cIn, chanend_t cOut)
33 {
34     channel_t chan = chan_alloc();
35
36     PAR_JOBS(
37         PJOB(process, (cIn, chan.end_a, "tile1-core0", 0)),
38         PJOB(process, (chan.end_b, cOut, "tile1-core1", 1))); // Kick-start the ring
39
40     chan_free(chan);
41 }
```

Build the example as normal, remembering to use `xcc -g`:

```
$ xcc -target=XCORE-200-EXPLORER -g multitile.xc main.c
```

Running the example produces the expected results:

```
$ xrun --io a.xe
tile0-core0
tile0-core1
tile1-core0
tile1-core1
tile0-core0
tile0-core1
...
```

3.5.2 Interactive debugging

We'll now use XGDB to observe the multi-tile example as the token is passed from core to core and tile to tile.

To start an interactive debug session:

```
$ xgdb a.xe
```

A familiar GDB prompt will be raised. Connect to a target and load the application using the `connect` and `load` commands:

```
(gdb) connect
0x00040000 in _start ()
(gdb) load
Loading setup image to XCore 0
...
```

Using the `tile` command, select a tile as the focus of subsequent commands:

```
(gdb) tile 0
[Switching to task 1 (tile[0] core[0])]#0 0x00040000 in _start ()
```

Add a breakpoint in the usual way:

```
(gdb) break main.c:16
Breakpoint 1 at 0x40156: file Z:/docs/docs_tools/tools-user-guide/tools-guide/quick-start/examples/
↪debug\main.c, line 16.
```

Switch to the `tile[1]` and add a breakpoint there too:

```
(gdb) tile 1
[Switching to task 2 (tile[1] core[0])]#0 0x00040000 in _start ()
(gdb) break main.c:16
Breakpoint 2 at 0x4015a: file Z:/docs/docs_tools/tools-user-guide/tools-guide/quick-start/examples/
↪debug\main.c, line 16.
```

Review all the breakpoints. Notice how the breakpoint appears at different addresses on each tile - this is not unexpected, as the two tiles have entirely independent address spaces with potentially different contents:

```
(gdb) info breakpoints
Num Type      Disp Enb  Core Address      What
1  breakpoint  keep y   0   0x00040156 in process at Z:/docs/docs_tools/tools-user-guide/tools-
↪guide/quick-start/examples/debug\main.c:16
2  breakpoint  keep y   1   0x0004015a in process at Z:/docs/docs_tools/tools-user-guide/tools-
↪guide/quick-start/examples/debug\main.c:16
```

We're now ready to run the multi-tile application. Use the `continue` and `info threads` commands to run the application to the next breakpoint and examine which tile/core it has halted at:

```
(gdb) continue
[Switching to tile[0] core[0]]

Breakpoint 1, process (cIn=2147614978, cOut=2147615234, name=0x4511c "tile0-core0", init=0) at Z:/
↪docs/docs_tools/tools-user-guide/tools-guide/quick-start/examples/debug\main.c:16
16      printf("%s\n", name);
Current language: auto; currently minimal
(gdb) info threads
  4  tile[1] core[1] (dual issue) 0x000403c4 in chan_in_word ()
  3  tile[1] core[0] (dual issue) 0x000403c4 in chan_in_word ()
  2  tile[0] core[1] (dual issue) 0x000403f4 in chan_in_word ()
* 1  tile[0] core[0] process (cIn=2147614978, cOut=2147615234, name=0x4511c "tile0-core0", init=0)
↪at Z:/docs/docs_tools/tools-user-guide/tools-guide/quick-start/examples/debug\main.c:16
```

The asterisk next to each thread shows which tile/core currently has focus for subsequent commands. Watch the asterisk move as the token moves round the ring by repeatedly issuing `continue` and `info threads`.

To end the interactive debug session:

```
(gdb) quit
```

3.5.3 Scripted debugging

Use of XGDB can be fully or partially scripted using Command Files (just like GDB). Scripted debugging using Command Files can be very powerful. It allows developers to quickly reproduce a particular scenarios, and even share those scenarios with other developers.

To start a scripted debug session, use `xgdb -ex` with the standard GDB **source** command:

```
$ xgdb -ex "source -v cmds.txt" a.xe
```

This causes the following command file to be executed prior to any interactive debugging; the trailing **quit** in the example below means that here there will be no interaction and thus the example is fully scripted.

Listing 3.11: cmds.txt

```
# Setup
connect
load

# Add breakpoints
tile 0
break main.c:16
tile 1
break main.c:16
info breakpoints

# Run
set $count=5
while $count > -1
  continue
  info threads
  set $count=$count-1
end

quit
```

3.5.4 Summary

This tutorial has demonstrated how XGDB can be used to debug a multi-tile Xcore application. To start exploring further XMOS-specific commands built on top of GDB, try:

```
(gdb) help xmos
```

4 Advanced tutorials

4.1 Describe a target platform

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.

4.1.1 Supported network topologies

To route messages across the xCONNECT Link network, the routing ID and routing table of each node on the network must be configured. The tools use the information in the XN file to setup the routing for the network before running the application.

If the routing configuration is explicitly specified in the XN file, the tools use this configuration. If the routing configuration is omitted from the XN file the tools choose a suitable set routing IDs and routing tables based on the network topology. The tools can automatically compute routing configurations for the the following network topologies.

Table 4.1: Topologies that can be automatically routed

Network Topology	Supported Configurations
Line	Not supported on XS1-G devices
Hypercube	Degree-2 (pair of nodes) Degree-3 (ring of 4 nodes) Degree-3 (cube of 8 nodes) Degree-4 (canonical cube of 16 nodes)
Hypercube with trees attached	Not supported on XS1-G devices

4.1.2 A board with two packages

Example hardware platform illustrates a board containing two XMOS L8-64 devices arranged in a line. A suitable XN description is described below.

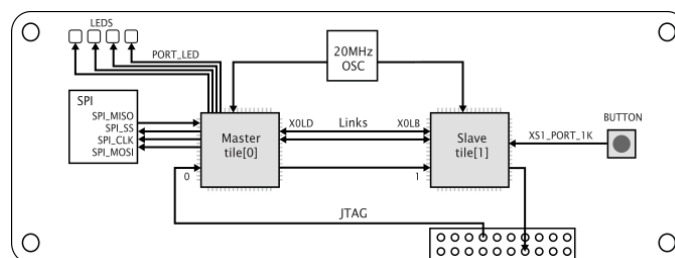


Fig. 4.1: Example hardware platform

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">
```

The following code declares two xCORE Tiles. The declaration "tileref tile[2];" is exported to the header file <platform.h>.

```
<Declarations>
  <Declaration>tileref tile[2]</Declaration>
</Declarations>
```

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

```
<Packages>
  <Package Id="P1" Type="XS1-LnA-64-TQ128">
    <Nodes>
      <Node Id="Master" Type="XS1-L8A-64" InPackageId="0"
        Oscillator="20MHz" SystemFrequency="400MHz">
        <Boot>
          <Source Location="SPI:bootFlash"/>
          <Bootee NodeId="Slave" Tile="0"/>
        </Boot>
        <Tile Number="0" Reference="tile[0]">
          <Port Location="XS1_PORT_1A" Name="PORT_SPI_MISO"/>
          <Port Location="XS1_PORT_1B" Name="PORT_SPI_SS"/>
          <Port Location="XS1_PORT_1C" Name="PORT_SPI_CLK"/>
          <Port Location="XS1_PORT_1D" Name="PORT_SPI_MOSI"/>
          <Port Location="XS1_PORT_4A" Name="PORT_LED"/>
        </Tile>
      </Node>
    </Nodes>
  </Package>
```

The node Master is a 400MHz XS1-L8A-64 device in a TQ128 package, clocked by a 20MHz oscillator. It is booted from an SPI device named "bootFlash" which has the class "SPIFlash".

The declaration of tile "0" is associated with tile[0] and the ports 1A, 1B, 1C, 1D and 4A are given symbolic names. These declarations are exported to the header file <platform.h>.

The following code declares a package named P2, which contains a single node named Slave.

```
<Package Id="P2" Type="XS1-LnA-64-TQ128">
  <Nodes>
    <Node Id="Slave" Type="XS1-L8A-64" InPackageId="0"
      Oscillator="20Mhz" SystemFrequency="400MHz">
      <Boot>
        <Source Location="LINK"/>
      </Boot>
      <Tile Number="0" Reference="tile[1]">
        <Port Location="XS1_PORT_1K" Name="PORT_BUTTON"/>
      </Tile>
    </Node>
  </Nodes>
</Package>
</Packages>
```

The node Slave is a 400MHz XS1-L8A-64 device in a TQ128 package, clocked by a 20MHz oscillator. It is booted from node Master over an xCONNECT Link.

The following code defines a 2-wire xCONNECT Link with, which connects the node Master on link X0LD to the node Slave on link X0LB.

```
<Links>
  <Link Encoding="2wire" Delays="4,4">
    <LinkEndpoint NodeId="Master" Link="X0LD" />
    <LinkEndpoint NodeId="Slave" Link="X0LB" />
  </Link>
</Links>
```

The links have intra-symbol and inter-symbol delays of 4 clock periods.

The following code specifies a list of components on the board that are connected to XMOS devices.

```
<ExternalDevices>
  <Device NodeId="Master" Tile="0" Name="bootFlash"
    Class="SPIFlash" Type="AT25FS010">
    <Attribute Name="PORT_SPI_MISO" Value="PORT_SPI_MISO" />
    <Attribute Name="PORT_SPI_SS" Value="PORT_SPI_SS" />
    <Attribute Name="PORT_SPI_CLK" Value="PORT_SPI_CLK" />
    <Attribute Name="PORT_SPI_MOSI" Value="PORT_SPI_MOSI" />
  </Device>
</ExternalDevices>
```

A device named bootFlash is connected to xCORE Tile 0 on Node Master, and is given attributes that associate the four SPI pins on the device with ports. (The class SPIFlash is recognized by XFLASH.)

The following code describes the JTAG scan chain.

```
<JTAGChain>
  <JTAGDevice NodeId="Master" Position="0" />
  <JTAGDevice NodeId="Slave" Position="1" />
</JTAGChain>

</Network>
```

4.2 Working with targets

This section shows in detail how to work with XMOS target systems.

4.2.1 Terminology

The following terms are defined:

Host computer (or host)

A desktop or laptop on which the XTC Tools are installed

Host tool (or tool)

A program supplied with the XTC Tools which may be launched from a shell or console

Target system (or target)

One or more printed circuit boards populated with XMOS devices

Target program

A program built with the XTC Tools which runs on a target system

4.2.2 Introduction

A set of host computer tools is provided for working with an XMOS target system. A target system is typically a single printed circuit board populated with one or more XMOS devices. A target program, (which is a .xe-suffixed file produced by the tool `xcc`), may be downloaded into the target RAM and executed using the tool `xrun`. Symbolic debugging of a target program may be carried out using the tool `xgdb`. A target program may be burned to non-volatile flash using the tool `xflash`. These tools require an XMOS xTAG to connect the target system to the host computer via USB 2.0.

4.2.3 Key features

The following features are provided by the XTC Tools:

1. Reset the target, download and launch a target program, and optionally support host-IO and xscope operations
2. Write a target program to a flash device in a target system
3. Dump the current state of the target (does not reset the target)
4. Provide symbolic debug of a target (where the program is either downloaded into RAM or booted from flash)
 1. The target may be debugged starting from a system-reset state or the debugger may be attached to a running program
 2. Host-IO and Xscope API calls may be left in programs which are deployed. The tools will optionally enable host-IO when a connection is made via an xTAG
5. Provide an API for a developer to use on the host computer to transfer application data to and from the target
6. Provide sample-based profiling of a target
7. Management of all XTAGs attached to the host
 1. List the xTAGs and detect the type of target system attached
 2. For all the tools, an xTAG may be specified by its list index or by unique identifier
 3. If only one xTAG is connected to the host an identifier does not need to be supplied to the tools

4.2.3.1 Targets

Three generations of XMOS target system are supported by the tools:

- xCORE (XS1 Instruction Set Architecture)
- xCORE-200 (XS2A Instruction Set Architecture)
- xcore.ai (XS3A Instruction Set Architecture)

Evaluation boards may be obtained from various sources, such as Farnell.

4.2.4 xTAG

The xTAG is the physical host-to-target interface. Two types are supported by the tools; xTAG v3.0 ([https://www.xmos.com/download/xTAG-3-Hardware-Manual\(1.0\).pdf](https://www.xmos.com/download/xTAG-3-Hardware-Manual(1.0).pdf)) and xTAG v4.0. Both xTAG types have a USB 2.0 High Speed (Micro-B) interface for connection to the host which also supplies the power to the xTAG. The xTAG3 has a proprietary xSYS interface to the target board and the xTAG4 has an xSYS2 interface to the target board.

Both xSYS and xSYS2 provide 4-wire IEEE1149.1 JTAG interface and a duplex serial xCONNECT link interface, plus some additional control signals. Note the optional JTAG signal nTRST is not pinned out.

When an xTAG is attached to a host it listens for a connection by certain host programs (`xrun`, `xgdb`, `xf1ash` or `xburn`). The first connection will download firmware to the xTAG which will support target-related operations. The firmware is not downloaded on subsequent connections until the xTAG is power-cycled.

The elapsed time required to connect to an xTAG is highly variable, due to the possibility of a firmware download occurring, and because the host operating system can introduce delays. This period may extend when multiple xTAGs are attached to the host computer. When a connection is initiated a lock is taken which may pause the execution of other independent host tools each using their own xTAG. The lock is released once the connection is established. Automated test systems should ensure any timeout for an xTAG response is sufficient for the worst case which may need to be determined empirically.

If the xTAG has been used previously with an earlier Tools 15 release the first connection will attempt to install the firmware required for use with this release. The xTAG will automatically reboot as part of this process which will be shown by the xTAG LEDs.

Note 1: If an xTAG has been used with a tools release earlier than Tools 15 it must be manually power cycled prior to making a connection with Tools 15.

Note 2: If the host is rebooted it may fail to enumerate the attached xTAGs. The xTAGs must be power-cycled. Some USB hubs provide a mechanism to power-cycle their downstream ports, and these are recommended for remote lab environments. Note Windows 10 Pro does not support this feature.

4.2.4.1 xTAG v3.0

xTAG v3.0 datasheet: [https://www.xmos.com/download/xTAG-3-Hardware-Manual\(1.0\).pdf](https://www.xmos.com/download/xTAG-3-Hardware-Manual(1.0).pdf)

The xTAG v3.0 provides the xSYS target interface (IDC 20-pin 0.1") and supports XS1 and XS2 target families. It supports only 3v3 target voltage levels.

4.2.4.2 xTAG v4.0

The xTAG v4.0 provides the xSYS2 target interface (Amphenol Minitek127 20-pin 0.05") and supports XS2 and XS3 (xcore.ai) target families. It supports 1v8 and 3v3 target voltages, automatically and independently for JTAG and xCONNECT Link interfaces.

The xCONNECT Link is optional – if it is not connected a reduced-footprint connector may be employed on the target board. To reduce the connector cost to a minimum, for production programming and testing, a Tag-Connect Plug-Of-Nails may be used. Only the IEEE1149.1 JTAG signals need to be connected.

4.2.4.3 Using xTAGs with Windows 10 Pro

On Windows 10 Pro host computers the click-through XTC installer sets up an XTAG service. This service starts following the installation procedure and each time the host boots, and requires no further administration.

On Windows 10 Pro it is not possible to switch between an XTC Tools 15 series release and an earlier XTC Tools release and use the host tools which require access to an xTAG.

On Windows 10 Pro the xTAG unique identifiers are shown in upper case but the tools are case-insensitive so scripts which drive the tools may be used interchangeably between all OS types supported by the tools.

4.2.5 Host tools

The host tools that interface with the target system are `xrun`, `xgdb`, `xflash` and `xburn`.

Unless stated otherwise the following examples in this section assume a single xTAG and target system are attached to the host computer.

4.2.5.1 `xrun`

The program `xrun` is used to:

1. List the xTAGs and target systems attached to the host computer
2. Download a target program to RAM and launch it
3. Download a target program to RAM and launch it, managing host-IO
4. Download a target program to RAM and launch it, and capture `xscope` data to a file
5. Dump the state of a target either with a `.xe` target program so it may be represented symbolically or without a `.xe` to show state as raw data

4.2.5.1.1 List the attached xTAGs

The command `xrun -l` lists the xTAGs attached to the host computer. When an xTAG is first plugged into the host the red LEDs should be illuminated in a dim state. These should change state when `xrun -l` is invoked. Sometimes the host fails to identify the xTAG. In this case it should be unplugged for several seconds and then re-inserted. If the xTAG is not listed check for its presence. On Linux hosts run the command:

```
$ lsusb
```

On Windows hosts open the Device Manager and expand Universal Serial Bus devices.

4.2.5.1.2 Launching a target program

A target program may be launched on the target system with:

```
$ xrun my_program.xe
```

`xrun` will exit immediately, leaving the target program running indefinitely. If the target program fails it may be debugged by attaching with `xgdb`.

A target program may be launched with host-IO enabled with:

```
$ xrun -io my_program.xe
```

or with xCONNECT Link host-IO enabled:

```
$ xrun -xscope my_program.xe
```

4.2.5.2 xgdb

The host tool `xgdb` is a variant of the open-source tool `gdb`. It supports the standard `gdb` command line options and commands, plus extra command options and builtin commands for use with XMOS targets.

It may be used to carry out all the functions supported by `xrun`, plus

1. Interactive, symbolic debug of the downloaded program
2. Scripted actions with the target
3. Attaching to a target which is running a program which either booted from flash or was downloaded and launched with `xrun` or `xgdb`

Symbolic debug is only available for a source compilation unit (that is, a `.c`, `.cpp` or `.xc` file) if it is compiled with the `-g` option passed to `xcc`. Reducing the optimisation level used by `xcc` with the `-O` option will improve debuggability at the expense of potential changes in program behaviour.

4.2.5.2.1 Debugging a target program

The following command will start a debug session of the program `my_program.xe` by connecting to the target, downloading the program and initialising the device based on the target XN used to build the program:

```
$ xgdb -ex "connect" -ex "load" my_program.xe
```

Commands supplied with the option `-ex` may be entered at the `(gdb)` prompt if preferred.

The `load` command must be supplied only once.

To start the program running the user must enter the command:

```
(gdb) continue
```

Before starting the program debugging commands may be supplied. For example, breakpoints may be set with the `break` command.

Once the target program is started the `(gdb)` prompt will not be available. Host-IO may appear on the console. Once started the target program may be stopped by pressing `Ctrl-C` which will report the stopped location and provide the `(gdb)` prompt.

The active logical cores may be listed with:

```
(gdb) info threads
```

Each active logical core is assigned a unique "gdb thread" number. Only active logical cores in all tiles in the system are listed. Note the "gdb thread" number assignments are not static and they may vary each time the target stops. A logical core may be selected for further inspection with the `thread` command. For example:

```
(gdb) thread 5
```

The back trace (call stack) for the logical core may be shown with the `where` command, for example:

```
(gdb) where
```

Local and global variables may be inspected with the standard `gdb` commands. See the `gdb` documentation for details.

4.2.5.2.2 Attaching to a target

In some cases it may be desirable to inspect the state of a target system without resetting it. It is possible to “attach” to a running target. For example:

```
$ xgdb -ex attach my_program.xe
```

xgdb will stop all the tiles and provide the (gdb) prompt. xgdb commands may be issued as illustrated in the previous section.

4.2.6 Target interaction

The JTAG interface is used by the tools to establish a connection to the target, to reset it and to download programs to RAM. It is used to interact with the target, for example, to extract register state and to set breakpoints. It is used to monitor the target to detect program termination, to detect the target taking a fatal exception and to interrupt the target. It may be used to handle host-I/O.

4.2.6.1 Debug mode

Each tile will be placed in its debug mode to: download a program, when a breakpoint is reached, to terminate a target program and when interrupted by a Ctrl-C operation in the xgdb console. In some cases it will enter debug mode to forward host-I/O data. When a tile enters debug mode all its logical cores will be paused.

4.2.6.2 Exceptions

The tools xrun and xgdb place a breakpoint on the exception handler entry point. When the target takes a fatal exception (for example, due to arithmetic divide-by-zero), the tools will report on the occurrence of the exception. xgdb may be used to debug the cause, symbolically, typically by showing the back-trace.

Note the message SIGTRAP is generated if a tile enters debug mode unexpectedly. This may occur when certain low-level operations are being carried out with the tiles. The target program has **not** taken an exception.

4.2.6.3 Launch with xrun

If the target program is launched by xrun without any optional arguments, xrun loads and runs the program, then exits immediately leaving the target program running. If the -io option or any of the -xscope options are supplied to xrun it will not exit until the target program terminates – see below.

4.2.6.4 Termination

If a tile’s control flow reaches the end of main() that tile will terminate and wait. When control flow for all tiles has reached the end of main(), and the target program has been launched with xrun supplying the option -io or any of the -xscope options, the xrun session will terminate. If the program was launched using xgdb it will report “Program exited normally” on termination.

If a tile calls exit() with a non-zero value, this will be returned by xrun as the process exit code. All tiles must terminate before xrun terminates. Only one tile should call exit(). If launched by xgdb, the exit code will be shown on the console.

4.2.7 Host-IO

If the target program calls standard library IO functions from `stdio.h` such as `printf()`, `fprintf()`, `fopen()`, `fread()` and `fwrite()`, and the program was launched with `xgdb`, or with `xrun` and the `-io` or `-xscope` options, these calls will be redirected to act on the host computer's filesystem (or its console, for the files `stdin`, `stdout` and `stderr`). Data will be sent from and to the target via the JTAG interface. To transfer data, the tile making the call will enter debug mode which pauses execution of all its logical cores.

These calls may be left in the program when it is deployed to boot from flash using `xflash`. If `xrun` or `xgdb` is later used with a such a system, the IO will be redirected to the host computer. However there is a deployment penalty in run-time and memory used; each call will process its arguments before inspecting internal state to determine whether `xrun` or `xgdb` is connected.

Note `stdio.h` and other headers mentioned in the following sections can be found under the tools installation in the sub-directory `target/include` on Linux hosts and `target\include` on Windows hosts.

An example `xrun` command to enable host-IO:

```
$ xrun -adapter-id DFW7DTYY -io test.xe
```

An example `xgdb` command (note IO is enabled by default when using `xgdb`):

```
$ xgdb -ex "connect -adapter-id DFW7DTYY" -ex load -ex continue test.xe
```

4.2.7.1 Reduced-overhead print functions

The header `print.h` provides a set of reduced overhead print functions such as `printstr()`. These do not perform any memory allocation or any runtime formatting, making them memory and run-time efficient compared with the standard library IO functions. For example, the following prints a set of characters without a new line:

```
printstr("Starting test 1");
```

4.2.7.2 Syscall host-IO

The header `syscall.h` provides the functions `_open()`, `_close()`, `_write()` and `_read()`. These may be used to perform host-IO with a reduced run-time and memory overhead compared with the standard library IO functions. The standard library IO functions call these underlying syscall functions. For example:

```
int fd = _open(path, O_WRONLY | O_CREAT | O_TRUNC, S_IREAD | S_IWRITE);
if (_write(fd, buf, len) != len) { // Handle error }
```

4.2.7.3 Host-IO via the xCONNECT link

The IO transfer rate via JTAG is relatively low. Furthermore, each IO call puts the tile in debug mode which is highly intrusive (as it pauses all logical cores). To achieve a significantly higher bandwidth and minimise intrusion into the target program, the `xCONNECT` link may be used as a transport. This may be configured in a lossy mode where data may be dropped if the host fails to keep pace with the target, or in lossless mode where the logical core emitting the data may be paused while data is being received by the host.

Note the write APIs (`printf()`, `fprintf()`, `fwrite()` and `_write()`) will use the `xCONNECT` Link exclusively. Other APIs such as `fopen()`, `fread()` or `_read()` always use the JTAG interface to transport data even if IO has been configured to use the `xCONNECT` link.

To use the `xCONNECT` Link as a transport the following should be provided in a source file from which the target program is built:

```
#include <xscope.h>

void xscope_user_init() {
    xscope_register(0);
    xscope_config_io(XSCOPE_IO_BASIC);
}
```

The function below may be called at any stage during target program execution to select lossless mode:

```
xscope_mode_lossless();
```

The function below may be called at any stage during target program execution to select lossy mode:

```
xscope_mode_lossy();
```

When lossy mode is selected and the data rate of the target cannot be met by the host, entire packets may be dropped.

Key points to note:

1. The program provides the function named `xscope_user_init()` (which is called automatically on program startup by the C-runtime initialisation code (CRT) as a static constructor)
2. The target program must be built with the option `-fxscope` supplied to `xcc`
3. One of the `--xscope` options must be supplied to `xrun` or the `xgdb connect` command

4.2.8 Xscope APIs

A set of target APIs is provided to send packets of data to the host to be visualised as waveforms with a 3rd party graphical tool. A set of virtual channels are defined in the target program and each appears as separate waveform trace.

The xCONNECT Link is always used to transport the data sent from the target. This may be configured in lossy mode where data may be dropped if the host fails to keep pace with the target, or in lossless mode where logical core emitting the data may be paused while data is transferred.

The APIs are defined in `xscope.h`. An example to emit float and integer data on two channels is:

```
void xscope_user_init(void) {
    xscope_register(2,
        XSCOPE_CONTINUOUS, "Float0", XSCOPE_FLOAT, "mV",
        XSCOPE_CONTINUOUS, "Iterations", XSCOPE_UINT, "Steps");
}

void emit(float f, unsigned int i) {
    xscope_float(0, f);
    xscope_int(1, i);
}
```

The `xcc` option `-fxscope` must be supplied when building the program.

The `xrun` option `--xscope-file <filename>` must be supplied to specify a file into which the data will be written in IEEE 1364-2001 Verilog VCD format.

Lossy or lossless data delivery may be selected as described above.

4.2.9 User-supplied host program

The user may supply a host program which handles target data while the target program is running. The target can send data to the host program using either the Xscope APIs, the host-IO APIs or both sets. The host program can send data back to the target program.

The host program can be written in either the C or C++ programming language. The header `xscope_endpoint.h` provides the API for the host program. It must be linked against the shared library `libxscope_endpoint.so` on Linux and Mac hosts and `xscope_endpoint.a` on Windows hosts, using C linkage. These libraries are provided by the XTC Tools in the `lib` sub-directory of the installation root.

In this example two consoles are used to launch the xgdb session and the host program.

<PORTNUM> must be substituted with a free system port number.

Console 1: launch the xgdb session:

```
$ xgdb -ex "connect --xscope-port --xscope-port localhost:<PORTNUM>"
```

Console 2: launch the host program:

```
$ host_example <PORTNUM>
```

4.2.9.1 Structure of the host program

The host program starts by registering callback functions to handle selected target activities, such as target calls to `printf()` or target calls to xscope APIs. It does this with calls to:

```
xscope_ep_set_print_cb();  
xscope_ep_set_register_cb();  
xscope_ep_set_record_cb();  
xscope_ep_set_exit_cb();
```

It then connects to the xgdb session using the <PORTNUM> argument passed to `main()`:

```
result = xscope_ep_connect("localhost", argv[1]);
```

If `result` is zero, the connection was successful. If non-zero the xgdb server was not ready. The `xscope_ep_connect()` API call may be re-tried until a connection is made.

Once connected the program should sleep in an efficient manner. The callback functions will be called when necessary as a direct result of API calls by the target program.

4.2.9.2 Example host program and target programs

In the following example the host program provides a set of call back functions which are called when the target-initiated events occur.

4.2.9.2.1 Compile

```
$ xcc -g -fxscope -target=XCORE-AI-EXPLORER -o target.xe main.xc target.c
$ gcc -g -I "${XMOS_TOOL_PATH}/include" "${XMOS_TOOL_PATH}/lib/xscope_endpoint.so -o host host.c
```

4.2.9.2.2 Run

Open two consoles.

In console 1:

```
$ xrun --xscope-port : target.xe
XScope Realtime Server Enabled (localhost:37316)
```

In console 2: Supply the port number printed by xrun following localhost: above:

```
$ ./host 37316
```

4.2.9.2.3 Host program

Listing 4.1: host.c

```
#include "xscope_endpoint.h"

/* Called when the target calls xscope_register() */
static void xscope_register(unsigned int id,
                           unsigned int type,
                           unsigned int r,
                           unsigned int g,
                           unsigned int b,
                           unsigned char *name,
                           unsigned char *unit,
                           unsigned int data_type,
                           unsigned char *data_name)
{
}

/* Called on each target call to xscope_int() and similar APIs */
static void xscope_record(unsigned int id,
                         unsigned long long timestamp,
                         unsigned int length,
                         unsigned long long dataval,
                         unsigned char *databytes)
{
    /* id is the virtual channel number used by the target in the xscope API call */
    switch (id) {
        /* handle each virtual channel id */
        case 1:
            /* do something for virtual channel 1 */
            break;
        case 2:
            /* do something for virtual channel 2 */
    }
}
```

(continues on next page)

```

    break;
}
/* Send a 2-byte reply to the host : max 256 bytes */
const char* s = "ok";
xscope_ep_request_upload(2, s);
}

static void xscope_print(unsigned long long timestamp,
                        unsigned int length,
                        unsigned char *data)
{
    for (int i = 0; i<length; i++) {
        printf("%c", data[i]);
    }
}

static int running;

/* Called when the target program terminates */
static void xscope_exit(void) {
    running = 0;
}

int main(int argc, char *argv[])
{
    if(argc != 2){
        fprintf(stderr, "ERROR missing xscope port number: Usage example %s localhost:12340\n",
        ↪argv[0]);
        exit(-1);
    }
    xscope_ep_set_print_cb(xscope_print);
    xscope_ep_set_register_cb(xscope_register);
    xscope_ep_set_record_cb(xscope_record);
    xscope_ep_set_exit_cb(xscope_exit);

    int error = 1;
    unsigned attempts = 0;
    const unsigned MAX_ATTEMPTS = 240;

    while (attempts<MAX_ATTEMPTS) {
        error = xscope_ep_connect("localhost", argv[1]);
        if (0 == error) {
            break;
        }
        sleep(1);
        attempts++;
    }

    if (error) {
        fprintf(stderr, "xscope_ep_connect: failed %d attempts\n", attempts);
        return 1;
    }
    running = 1;
    while (running) {
        sleep(1);
    }
    xscope_ep_disconnect();
    return 0;
}

```


4.2.9.2.4 Target program

Listing 4.2: main.xc

```
#include <platform.h>
#include <xscope.h>

extern "C" {
    void main_tile0(chanend);
}

int main (void)
{
    chan xscope_chan;
    par
    {
        xscope_host_data(xscope_chan);
        on tile[0]: main_tile0(xscope_chan);
    }
    return 0;
}
```

Listing 4.3: test.c

```
void xscope_user_init(void) {
    xscope_register(2,
        XSCOPE_CONTINUOUS, "V", XSCOPE_INT, "mV", /* Virtual channel 1 */
        XSCOPE_CONTINUOUS, "I", XSCOPE_INT, "mA", /* Virtual channel 2 */
    );
}

void main_tile0(chanend_t xscope_end)
{
    xscope_mode_lossless();
    xscope_connect_data_from_host(xscope_end);

    xscope_int(1, 45);          /* Send 45 to virtual channel 1 */

    // Read response from host
    int bytes_read = 0;
    SELECT_RES(CASE_THEN(xscope_end, read_host_data)) {
        read_host_data: {
            xscope_data_from_host(xscope_end, (char *)buffer_ptr, &bytes_read);
        }
    }
    xscope_int(2, 578);        /* Send 578 to virtual channel 2 */

    // Read response from host
    int bytes_read = 0;
    SELECT_RES(CASE_THEN(xscope_end, read_host_data)) {
        read_host_data: {
            xscope_data_from_host(xscope_end, (char *)buffer_ptr, &bytes_read);
        }
    }
}
```

4.2.9.3 Specifying a port number for xgdb and the user host program

An example of an explicitly chosen port number is:

```
xgdb -ex "connect --xscope-port-blocking --xscope-port localhost:45678" led-flash.xe
```

4.2.9.4 Allow xgdb to choose a port number and bind – avoiding a race condition

It is often difficult to choose a port number and avoid the race condition where another program on the host computer binds to that number. This race can be prevented by allowing xgdb to choose the port number, and bind to it (which is an *atomic* operation and prevents another program from using that number). xgdb prints the port number it has used, which is then passed to the host program. The following example shows the xgdb this:

```
$ xgdb -ex "connect --xscope-port-blocking --xscope-port :" led-flash.xe
```

4.2.10 Host/target data throughput

The throughput in each direction using both the xscope interface and the JTAG interface is shown below.

Table 4.2: xscope throughput

Host machine and OS	Host to target (xscope) [Kbytes/s]	Target to host (xscope) [Kbytes/s]	Host to target (JTAG) [Kbytes/s]	Target to host (JTAG) [Kbytes/s]
Intel I3 NUC PC Centos 7 running natively	1105	6926	2.7	2.8
Intel I3 NUC PC Ubuntu 20.04 running natively	1053	6929	2.8	2.9
Intel I3 NUC PC Windows 10 running natively	1031	6926	2.7	2.9
Intel I7 Laptop Centos 7 running in a VM	932	5183	4.2	15.9

It can be seen that in the direction from target to host, when using the xscope interface the maximum throughput of the XTAG device has been reached (for OSes running natively). This data is streamed from the target to the host without handshaking. In the direction from host to target the data is transferred in a series of blocks along with a handshake for each, which is why the throughput is significantly lower.

4.2.11 Sample-based profiling of the target program

The tools provide a mechanism to profile a target program and report the time spent executing its lines and functions, as a flat call graph. It does this non-intrusively by sampling at a low frequency, which might vary considerably, the program counter of each logical core. The samples are captured in a set of *gprof* data files, and a report generated with the tools *xgprof*.

This approach is useful for gaining insight into the behaviour of certain types of program, for example those which carry out a repetitive task. It may be useful for debugging a program which is failing to run as intended.

4.2.11.1 Capturing the sample data

Run and capture sample data using the `--gprof` option to the `connect` command to *xgdb*. When *xgdb* is quit a set of *gprof* files (with the extension `.gprof`) will be written in the current working directory. The filename of each *gprof* file indicates the tile and logical core which it covers. For example, start capture with:

```
$ xgdb -ex "connect --gprof" -ex load -ex continue CircularBuffer.xe
```

Leave *xgdb* running for a period to capture samples, interrupt it with `Ctrl-C` and exit *xgdb* with the `quit` command. On exit the *gprof* profile data files will be written to the current working directory.

4.2.11.2 Analysing the profile data

Spit the target program (`.xe`) file using the `xobjdump -s` to obtain the Elf files for each tile. For example:

```
$ xobjdump -s CircularBuffer.xe
```

For the next step, the Elf `image_n0c0_2.elf` is used for tile 0 analysis, the Elf `image_n0c1_2.elf` is used for tile 1 and so on. Ignore Elf images without the `_2` suffix (these are bootstrap images used to provide system initialisation).

Generate a flat profile report for a logical core with the tool *xgprof*, supplying the Elf for the tile and the *gprof* file for the logical core of the tile. It is typically necessary use a selection of report levels to understand the behaviour of the target program. For example, a summary report may be shown with:

```
$ xgprof -b image_n0c0_2.elf tile[0]_core0.gprof
```

A detailed report at the source line level may be shown with:

```
$ xgprof -l -b image_n0c0_2.elf tile[0]_core0.gprof
```

A very detailed report at the address level with hit-counts may be shown with:

```
$ xgprof -C -l -b image_n0c0_2.elf tile[0]_core0.gprof
```

4.2.11.3 Profiling a target program which boots from flash

The following command may be used to capture profile data for a system which booted from flash:

```
$ xgdb -ex "attach -gprof --nointerrupt" test.xe
```

where `test.xe` was used to build the flash image (or to build the upgrade image within the flash device) and is currently executing.

Leave *xgdb* running for a period to capture samples, interrupt with `Ctrl-C` and exit *xgdb* with the `quit` command. On exit the *gprof* profile data files will be written to the current working directory.

Follow the steps described above to analyse the data files.

4.2.12 Target errors and warnings

4.2.12.1 xSCOPE not supported

The following message may be shown when `xrun` or `xgdb` is launched with the `--xscope` option. This occurs because the application does not make the call `xscope_config_io(XSCOPE_IO_BASIC)` from a function named `xscope_user_init()`

```
WARNING: tile[0] - xSCOPE not supported by application, I/O will be via JTAG
```

This implies any host-IO will be transported over the JTAG interface which is highly intrusive to the target program.

4.2.12.2 Failure to parse XN file

When `xrun` or `xgdb` connects and loads a program to the target it configures the target using the target XN file (and associated configuration files) supplied to `xcc` with the `-target` option.

If the following message is shown the clocks and clock dividers in the target will not be set correctly and the target program may not function:

```
Warning: could not parse XN file, error /tmp/.xgdb21287-I8WMQBQG/platform.xn:11 Error: XN11101  
→ Configuration file for node type "XTAG4" not found.  
, PLL will not be set to expected value
```

This may be because a local XN file and configuration file was used to build the target program. In the example above the file `XTAG4.cfg` could not be found in the tools installation. The location of these local XN and configuration files may be specified by setting the search path environment variables `XCC_TARGET_PATH` and `XCC_DEVICE_PATH` respectively. For example, on a Linux host, the following command adds the current working directory to the paths that will be searched:

```
$ XCC_TARGET_PATH=. XCC_DEVICE_PATH=.:${XCC_DEVICE_PATH} xgdb ...
```

4.2.13 Command examples

4.2.13.1 xgdb examples

In the following examples `xgdb` commands are often supplied as a sequence of `-ex` options on the `xgdb` command line. But they may also be supplied in a command file or typed interactively at the `gdb` prompt. Commands may be abbreviated for convenience, but abbreviated forms must not be ambiguous.

4.2.13.1.1 Load and run a program

The following example connects to a target, loads the target program `test.xe` into RAM and starts it running. Once launched the program must be interrupted with a Ctrl-C action if it does not terminate, either to quit `xgdb` or to enter further `xgdb` commands. A Ctrl-C action is made by the holding down both the Ctrl key and the C key on the keyboard:

```
$ xgdb -ex connect -ex load -ex continue test.xe
```

Note the load command runs two phases. The first phase loads a setup image (which is automatically generated by the tools) to the target SRAM which is executed to initialise it. The second phase loads the target program which was built from the user's source code.

4.2.13.1.2 Breakpoints

A breakpoint may be set symbolically or with a numerical address to stop execution of all tiles when the control flow of a logical core reaches that address. The gdb prompt is shown along with a diagnostic message indicating the reason the program stopped.

The xgdb command “break” inserts a soft breakpoint in RAM. There is no limit to the number of soft breakpoints that may be set. A soft breakpoint is implemented by writing a dcall instruction to the breakpoint address in the RAM of the tile.

For example:

```
(gdb) break my_func
(gdb) break *0x80402
```

In some cases a soft breakpoint cannot be used, for example where a program will be loaded to RAM over an external link after the breakpoint is set. The xgdb command “hbreak” can be used. This will use the hardware emulation debug feature within the tile to set a breakpoint. A maximum of three hardware breakpoints may be set:

```
(gdb) hbreak my_func
```

4.2.13.1.3 Watchpoints

Xgdb may set watchpoints (also known as data breakpoints) to stop all tiles when an access is made to a watched address. Two types of watchpoint command are supported:

watch <location>

stops execution when a store occurs to the specified location

awatch <location>

stops execution when store occurs to or a load occurs from the specified location

xgdb infers the address range to watch based on the type of the argument <location>.

For example, if my_counter is declared as type int in the target program the following will watch for write accesses only to its 4-byte range:

```
(gdb) watch my_counter
```

The following will watch for write and read accesses only to my_counter:

```
(gdb) awatch my_counter
```

A watch may be set on an address with a defined range. The following sets a watch on address 0x80250 with a range of 8 bytes:

```
(gdb) watch *((char (*)[8]) 0x80250)
```

The debug hardware which provides watch support monitors the address issued by a load or store instruction, checking whether it is greater than or equal to the lower bound and less than or equal to the upper bound. If the lower bound is the address of the third byte of a word, and the upper bound is the address of the fourth byte of the word, and a 2-byte store is made to the address of the third byte the watch will trigger. But if a 4-byte store is made to the address of first byte the watch will not trigger (although this store will write to the third and fourth byte which is being watched).

If a watch is set on a global or static variable before the program is started it may be triggered when the startup code in the function _start() performs initialisation.

4.2.13.1.4 Attaching to a running target

A target may have booted from flash or launched with `xgdb`, or performed a flash firmware update. Symbolic debug may be carried out with:

```
xgdb -ex "attach --adapter-id pr1V0_15" led-flash.xe
```

If the target application was built to perform host-IO over JTAG (for example, by making a call to `printstrln()`) this will be enabled.

4.2.13.1.5 Attaching to a running target and capture data to generate a sample-based profiling report

The following command can be used to attach to a system which is running and collect profile data without intrusion:

```
$ xgdb -ex "attach --nointerrupt --gprof --adapter-id pr1V0_15" led-flash.xe
```

4.2.13.2 xrun examples

4.2.13.2.1 Listing available XTAGs

The `-l` option to the tool `xrun` will list all attached xTAGS, showing:

- The integer value that may be supplied to the `--id` option to `xrun` or to the `xgdb connect` command
- The xTAG type
- The unique adapter-id string which may be supplied to the `-adapter-id` option to `xrun`, to the `xgdb connect` or to the `xflash option --adapter-id <>`
- The type of target connected to the xTAG. The letter `O` indicates an xCORE-200 XS2A architecture and the letter `P` an XCORE-AI XS3A architecture. The digits in `[]` indicate the number of devices (XMOS nodes) (note there may be multiple in a package). A single `0` indicates one device.

```
% xrun -l
Available XMOS Devices
-----
ID      Name                Adapter ID          Devices
--      -
0       XMOS XTAG-3         .BT0u0X2           P[0]
1       XMOS XTAG-4         BEJMRJ7V           O[0]
2       XMOS XTAG-3         KAKqyceA           O[0]
3       XMOS XTAG-4         pr1v0_20           P[0]
```

If the device is described as "In Use" a host tool running is using it. In some cases, after interrupting a session, an `xgdb` process may be left without a parent (orphaned), and it continues to hold its xTAG. On Linux hosts the orphaned process should be killed with the command `kill` and if this fails, with `kill -9`.

If the device is described with "Invalid firmware" it was last used by a host tool from a different tools release (and may be used again by the same tool). If a connection is made to the device with `xrun` or `xgdb` the firmware will be replaced.

4.2.13.2.2 Dump the target state

The following will dump the state of the target:

```
$ xrun --dump-state test.xe
```

where `test.xe` is the target program that was launched on the target by an invocation of `xrun` or was burned to flash with the `xflash` tool.

The following will dump the state of the target without a `.xe` file:

```
$ xrun --dump-state-no-xe
```

Note in both cases all tiles will be put in their debug mode which will stop execution of all logical cores.

4.2.13.2.3 Reboot an XTAG which fails to respond

The `--xtagreboot` option to the `xgdb connect` command will force the xTAG reboot:

```
$ xgdb -ex "connect --xtagreboot"
```

If the xTAG is flagged **in use** by `xrun -1` the above command will release it. Note this reboot feature is not available on Windows 10 Pro due to limitations in the standard Windows USB driver.

4.3 Design and manufacture systems with flash memory

The tools can be used to target xCORE devices that use Quad SPI or SPI flash memory for booting and persistent storage.

By default XFLASH fully supports a range of *Quad SPI* and *SPI* devices implemented in `libquadflash` and `libflash` respectively. If the number of pages, page size and sector size are specified in the *XN file* - or the flash device supports *SFDP* - XFLASH can easily support significantly many more devices available on the market.

For some devices, particularly SPI flash devices, additional manual configuration in reference to the device's datasheet as described in the *Add support for a new flash device* section may be required for XFLASH to fully support the flash device. This configuration file can be specified to XFLASH using the `--spi-spec` option.

The same configuration file provided to XFLASH can also be used to develop target software which accesses persistent storage on the flash device using the `libquadflash` and `libflash` libraries.

The xCORE flash format is shown in *Flash format diagram*.

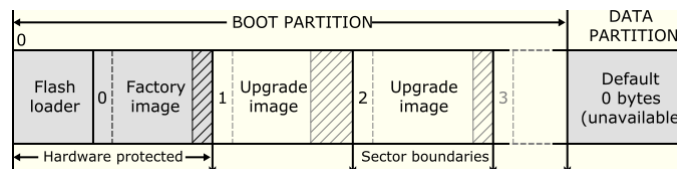


Fig. 4.2: Flash format diagram

The flash memory is logically split between a boot and data partition. The boot partition consists of a flash loader followed by a "factory image" and zero or more optional "upgrade images." Each image starts with a descriptor that contains a unique version number, a header that contains a table of code/data segments for each tile used by the program and a CRC. By default, the flash loader boots the image with the highest version with a valid CRC.

4.3.1 Boot a program from flash memory

To load a program into an SPI flash memory device on your development board, [start the tools](#) and enter the following commands:

1. **xflash -l**

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

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

2. **xflash --id ID *program*.xe**

XFLASH generates an image in the xCORE flash format that contains a first stage loader and factory image comprising the binary and data segments from your compiled program. It then writes this image to flash memory using the xCORE device.

Caution: The XN file used to compile your program must define an SPI flash device and specify the four ports of the xCORE device to which it is connected.

4.3.2 Generate a flash image for manufacture

In manufacturing environments, the same program is typically programmed into multiple flash devices.

To generate an image file in the xCORE flash format, which can be subsequently programmed into flash devices, [start the tools](#) 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.

4.3.3 Perform an in-field upgrade

The tools and the flash libraries libquadflash and libflash 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.

4.3.3.1 Write a program that upgrades itself using Quad SPI devices

The example program below uses the libquadflash library to upgrade itself.

```
#include <platform.h>
#include <quadflash.h>

#define MAX_PSIZE 256

/* initializers defined in XN file and available via platform.h */

f1_QSPIPorts QSPI = {
    PORT_SQI_CS,
```

(continues on next page)


```

PORT_SQI_CLK,
PORT_SQI_SIO,
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;
unsigned char page[MAX_PSIZE];
int psize;

fl_connect(QSPI);

psize = fl_getPageSize();
fl_getFactoryImage(b);
if(fl_getNextBootImage(b) == 0) {
    while(fl_startImageReplace(b, usize));
} else {
    while(fl_startImageAdd(b, usize, 0));
}

for (int i=0; i<usize/psize; i++) {
    for (int j=0; j<psize; j++) {
        c := page[j];
    }
    fl_writeImagePage(page);
}

fl_endWriteImage();

fl_disconnect();

return 0;
}

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

```

The call to `fl_connect` opens a connection between the xCORE and Quad SPI flash device, and the call to `fl_getPageSize` determines the flash 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 `fl_getNextBootImage`. Once located, `fl_startImageReplace` prepares this image for replacement with a new image with the specified (maximum) size. If there is no upgrade image already installed to the device, `fl_startImageAdd` prepares adding a new upgrade image, which can be replaced by future upgrades using `fl_startImageReplace`. `fl_startImageAdd` and `fl_startImageReplace` must be called until they return 0, signifying that the preparation is complete.

The function `fl_writeImagePage` writes the next page of data to the flash 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 flash 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 xCORE and flash device.

Note each of the `f1...()` functions typically returns a code which should be checked for success, prior to proceeding to the next step in the process of writing an upgrade image. These checks have been left out of the example above to aid clarity in following the process.

4.3.3.2 Build and deploy the upgrader

To build and deploy the first release of your program, *start the tools* and enter the following commands:

1. **`xcc *file*.xc -target=*boardname* -lquadflash -o first-release.xe`**

XCC compiles your program and links it against `libquadflash`. Alternatively add the option `-lflash` to your Makefile.

2. **`xflash first-release.xe -o manufacture-image`**

XFLASH generates an image in the xCORE flash format that contains a first stage loader and the first release of your program as the factory image.

To build and deploy an upgraded version of your program, enter the following commands:

1. **`xcc *file*.xc -target=*boardname* -lquadflash -o latest-release.xe`**

XCC compiles your program and links it against `libquadflash`.

2. **`xflash --upgrade *version* latest-release.xe --factory-version *tools-version* -o upgrade-image`**

XFLASH generates an upgrade image with the specified version number, which must be greater than 0. Your program should obtain this image to upgrade itself.

When the device is next reset, the loader boots the upgrade image, otherwise it boots the factory image.

4.3.3.3 Write a program that upgrades itself using SPI devices

The example above can be easily adjusted to use SPI devices with the `libflash` library. Note the Quad SPI devices and SPI devices use different xCORE ports and pins so `libquadflash` cannot be used with a SPI device (in serial mode).

Changes required to use a SPI devices are:

1. Include `flash.h` instead of `quadflash.h`
2. Replace the `f1_QSPIPorts` structure with an instance of `f1_SPIPorts` that references the correct ports in the XN file.
3. Link against `libflash` by supplying `-lflash`

The initial lines of the example above are shown below with the changes required:

```
#include <platform.h>
#include <flash.h>

#define MAX_PSIZE 256

/* initializers defined in XN file and available via platform.h */

f1_SPIPorts SPI = {
    PORT_SPI_MISO,
    PORT_SPI_SS,
    PORT_SPI_CLK,
    PORT_SPI_MOSI,
    XS1_CLKBLK_1
};
```

(continues on next page)

```

int upgrade(chanend c, int usize) {

    /* Obtain an upgrade image and write
     * it to flash memory.
     * Error checking omitted */

    fl_BootImageInfo b;
    unsigned char page[MAX_PSIZE];
    int psize;

    fl_connect(SPI);

    ...

```

Build for SPI linking against libflash:

1. **xcc *file*.xc -target=*boardname* -lflash -o first-release.xe**

4.3.4 Customize the flash loader

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

The XTC loader first calls the function `init`, and then iterates over each image in the boot partition. For each image, it calls `checkCandidateImageVersion` with the image version number and, if this function returns non-zero and its CRC is validated, it calls `recordCandidateImage` with the image version number and address. Finally, the loader calls `reportSelectedImage` to obtain the address of the selected image.

To produce a custom loader, you are required to define the functions `init`, `checkCandidateImageVersion`, `recordCandidateImage` and `reportSelectedImage`. The loader provides the function `readFlashDataPage`.

```

extern void * readFlashDataPage(unsigned addr);
int dpVersion;
void* imgAdr;

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

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

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

unsigned reportSelectedImage(void) {
    return imgAdr;
}

```

4.3.4.1 Build the loader and write to flash

To create a flash image that contains a custom flash loader and factory image, start the command-line tools 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.

4.3.4.2 Build with additional images and create a binary flash file

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  
-o image.bin
```

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.

4.3.5 Reading the numerical and string identifiers in a flash image

A 32-bit integer identifier may be stored in the flash device as shown by the following xflash invocation example:

```
xflash <options> --idnum 12345
```

A string identifier may be stored in the flash device as shown by the following xflash invocation example:

```
xflash <options> --idstr "My identifier string"
```

The following example application code shows how these identifiers may be read:

```
void get_ids() {  
    int success;  
    success = fl_connectToDevice(ports, deviceSpecs, 1);  
    // Check success  
  
    int identifier = fl_getFlashIdNum();  
  
    unsigned char idStr[MAX_LEN];  
    success = fl_getFlashIdStr(idStr, MAX_LEN);  
    // Check success  
  
    ...  
}
```

4.4 Safeguard IP and device authenticity

xCORE devices contain on-chip one-time programmable (OTP) memory that can be blown during or after device manufacture testing. You can program the xCORE AES Module into the OTP of a device, allowing programs to be stored encrypted on flash memory. This helps provide:

- **Secrecy**
Encrypted programs are hard to reverse engineer.
- **Program Authenticity**
The AES loader will not load programs that have been tampered with or other third-party programs.
- **Device Authenticity**
Programs encrypted with your secret keys cannot be cloned using xCORE devices provided by third parties.

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

- force boot from OTP to prevent bypassing,
- disable JTAG access to the tile to prevent the keys being read, and
- stop further writes to OTP to prevent updates.

Danger: 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.

4.4.1 The xCORE AES module

The xCORE AES Module authenticates and decrypts programs from SPI flash devices. When programmed into a device, it enables the following secure boot procedure, as illustrated in [Secure boot procedure used with the AES Module](#).

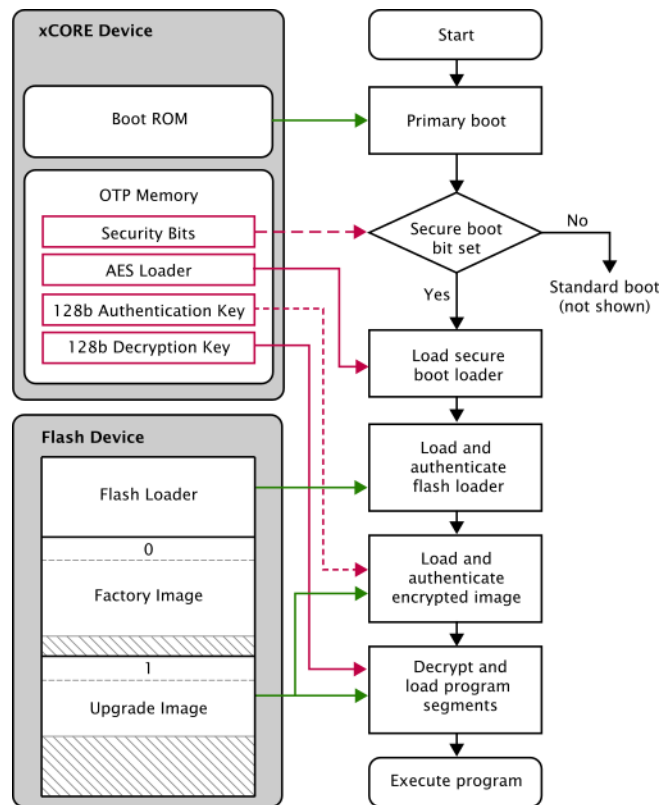


Fig. 4.3: Secure boot procedure used with the AES Module

1. The device loads the primary bootloader from its ROM, which detects that the secure boot bit is set in the OTP and then loads and executes the AES Module from OTP.
2. The AES Module loads the flash loader into RAM over SPI.
3. The AES Module authenticates the flash loader using the CMAC-AES-128 algorithm and the 128-bit authentication key. If authentication fails, boot is halted.
4. The AES Module places the authentication key and decryption key in registers and jumps to the flash loader.

The flash loader performs the following operations:

1. Selects the image with the highest number that validates against its CRC.
2. Authenticates the selected image header using its CMAC tag and authentication key. If the authentication fails, boot is halted.
3. Authenticates, decrypts and loads the table of program/data segments into memory. If any images fail authentication, the boot halts.
4. Starts executing the program.

For multi-node systems, the AES Module is written to the OTP of one tile, and a secure boot-from-xCONNECT Link protocol is programmed into all other tiles.

4.4.2 Develop with the AES module enabled

You can activate the AES Module at any time during development or device manufacture. In a development environment, you can activate the module but leave the security bits unset, enabling:

- XFLASH to use the device to load programs onto flash memory,
- XGDB to debug programs running on the device, and
- XBURN to later write additional OTP bits to protect the device.

In a production environment, you must protect the device to prevent the keys from being read out of OTP by the end user.

To program the AES Module into the xCORE device on your development board, [start the tools](#) 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.

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

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

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

`xburn --id *ID* --target-file *target*.xn --disable-jtag --enable-master-lock`

4.4.3 Production flash programming flow

In production manufacturing environments, the same program is typically programmed into multiple SPI devices.

To generate an encrypted image in the xCORE flash format, [start the tools](#) and enter the following command:

`xflash *prog*.xe -key *keyfile* -o *image-file*`

This image can be programmed directly into flash memory using a third-party flash programmer, or it can be programmed using XFLASH (via an xCORE device). To program using XFLASH, enter the following commands:

1. **`xflash -l`**

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

ID - NAME (ADAPTER-SERIAL-NUMBER)

2. **xflash --id *ID* --target-file *platform*.xn --write-all *image-file***

XFLASH generates an image in the xCORE flash format that contains a first stage loader and factory image comprising the binary and data segments from your compiled program. It then writes this image to flash memory using the xCORE device.

The XN file must define an SPI flash device and specify the four ports of the xCORE device to which it is connected.

4.4.4 Production OTP programming flow

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

To generate an image that contains the AES Module and security keys to be written to the OTP, [start the tools](#) 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.

Danger: 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.

4.4.5 Security implications of application design

The use of various software and hardware features by the program and their security implications must be carefully considered:

- Data partition

The data partition is not encrypted or signed, so must be considered untrusted by a secure application. A strong security scheme can be implemented for the data partition but this lives within the domain of the application, and is not provided by XFLASH.

- [Software-defined memory](#)

Use of software memory is supported for secure applications, but no signing or encryption of the data in “.SwMem” sections is performed. The application should consider the data placed in those sections and subsequently read from flash at runtime to be untrusted.

Secure schemes can be implemented within the application in a similar manner to the data partition but this may come at the cost of software memory performance.

- **LPDDR**

Data placed in “.ExtMem” sections is loaded from flash by the secure bootloader and is encrypted and signed using the secret keys provided to XFLASH.

This provides a basic level of security for the data placed in LPDDR memory, though it is important to note that LPDDR is external off-chip memory that can be sniffed and tampered with during the run time of your application using a hardware attack.

Use of LPDDR by a secure application should therefore be carefully implemented, following secure programming practices.

If the above features are required for your application and a strong level of security is desired, it is important to ensure the application does not trust any data residing in any external memory.

Practically, this will consist of:

1. Verifying authenticity of external data using modern cryptography standards

The data must be authenticated each time it is read to prevent attacks where the external data is changed following an initial authentication.

For large amounts of data, consider structures such as a hash tree to enable performant, authenticated accesses.

Encryption should be additionally considered if the external data must be kept secret.

It is safe to embed key material in your application code for these purposes as it will be protected by the xCORE AES module, though it is strongly recommended that you do not reuse the secure boot keys provided to XBURN and XFLASH within your application and instead use different keys. This will prevent leaking the secure boot keys in the event that the application is compromised.

2. Defensive programming of the application

Authentication is the first line of defence - the second is to design the application such that it cannot be compromised if external data was to be modified by an attacker.

This consists of thoroughly verifying your code to ensure it behaves predictably and safely upon receipt of unexpected external data.

Unpredictable behaviour may manifest as buffer overflows or similar bugs that can be exploited by an attacker to gain control over the device and reveal IP.

4.5 Add support for a new flash device

This section describes supporting a new flash device using libflash or libquadflash. libflash is a library implementing support for SPI flash devices and libquadflash implements Quad SPI devices using a similar API.

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 resulting configuration file can be used to support an unsupported flash device within XFLASH, or used to write target software using libflash or libquadflash directly.

The configuration file for the [Numonyx M25P10-A](#) is shown below. The device is described as an initializer for a C structure, the values of which are described in the following sections.

```

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 */
0x000000,   /* 24. RDID manufacturer ID bitmask to ignore */
0,          /* 25. libquadflash: QE bit location, libflash: Reserved */

```

The configuration structure implements a form of parameter inheritance, allowing for up to three sources of information. In order of priority, a device configuration file can override any value obtained as a result of SFDP query (for devices where this is available), which can override the default configuration file.

The device configuration file is the most authoritative parameter set used for establishing connection with the flash device. A device ID check is performed against the configuration to match the correct file with the attached flash device as described in [Read Device ID](#).

SFDP query is the second most authoritative source of information. Any parameter values set as -1 will be populated from the SFDP response according to the [SFDP](#) section, if available.

The final source of parameters is the default configuration file. This contains a fallback set of parameters that are used when either a specific device configuration cannot be matched, or when the device file inherits values on a per-value basis from the default file, and the parameter was not populated by SFDP.

By setting the libflash device ID to -1, the configuration file describes the default set of parameters.

By setting the remaining parameters to -1, the value is inherited from SFDP or the default configuration file, in that order.

The default configuration should not use the value of -1 for the remaining parameters, as there is nothing for it to inherit from. The default configuration describes fallback parameters used when information is missing from SFDP or the device configuration.

The values that can be currently populated from SFDP can be found in the [SFDP](#) section.

4.5.1 Libflash Device ID

```
10, /* 1. libflash device ID */
```

This value is returned by libflash on a call to the function `f1_getFlashType` so that the application can identify the connected flash device.

If this value is set to -1, the configuration file describes the default set of parameters.

4.5.2 Page Size and Number of Pages

```
256, /* 2. Page size */  
512, /* 3. Number of pages */
```

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 in section 1:

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.

4.5.3 Address Size

```
3, /* 4. Address size */
```

This value specifies the number of bytes used to represent an address. [Table 4 of M25P10-A datasheet](#) reproduces the part of the M25P10-A datasheet that provides this information. In the table, all instructions that require an address take three bytes.

Table 4.3: Table 4 of M25P10-A datasheet

Instruc- tion	Description	One-byte instruc- tion		Ad- dress byte	Dummy Data	
		code			bytes	bytes
WREN	Write Enable	0000 0110	06h	0	0	0
WRDI	Write Disable	0000 0100	04h	0	0	0
RDID	Read Identification	1001 1111	9Fh	0	0	1 to 3
RDSR	Read Status Register	0000 0101	05h	0	0	1 to in- finity
WRSR	Write Status Register	0000 0001	01h	0	0	1
READ	Read Data Bytes	0000 0011	03h	3	0	1 to in- finity
FAST_READ	Read Data Bytes at Higher Speed	0000 1011	0Bh	3	1	1 to in- finity
PP	Page Program	0000 0010	02h	3	0	1 to 256
SE	Sector Erase	1101 1000	D8h	3	0	0
BE	Bulk Erase	1100 0111	C7h	0	0	0
DP	Deep Power-down	1011 1001	B9h	0	0	0
RES	Release from Deep Power-down, and Read Electronic Signature	1010 1011	ABh	0	3	1 to in- finity
	Release from Deep Power-down			0	0	0

4.5.4 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.

[Table 18 of M25P10-A datasheet \(first four entries only\)](#). 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.

Table 4.4: Table 18 of M25P10-A datasheet (first four entries only).

Sym- bol	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	Clock High time	18			ns
t _{CL}	^{CLH} t _{CLL}	Clock Low time	18			ns

In general, if the SPI device supports different clock rates for different commands used by libflash, the lowest value must be specified.

4.5.5 Read Device ID

```
0x9f,          /* 6. RDID cmd */
0,            /* 7. RDID dummy bytes */
3,           /* 8. RDID data size in bytes */
0x202011,    /* 9. RDID manufacturer ID */
. . .
0x000000     /* 24. RDID manufacturer ID bitmask to ignore */
```

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.

[Table 4 of M25P10-A datasheet](#) 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 [Table 5 of M25P10-A datasheet](#) and [Figure 9 of M25P10-A datasheet](#), 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.

The bitmask of ID bits to ignore allows supporting a family of similar flash devices with a single configuration. libflash will attempt connection using the most specialised configurations first (those with fewest bits set).

The bitmask and device ID fields are special in that they cannot be set to -1 to trigger inheritance - the value is considered to be set to 0xffffffff, which in the case of the bitmask, ignores all bits of the device ID.

Table 4.5: Table 5 of M25P10-A datasheet

Manufacturer identification	Device identification	
	Memory type	Memory capacity
20h	20h	11h

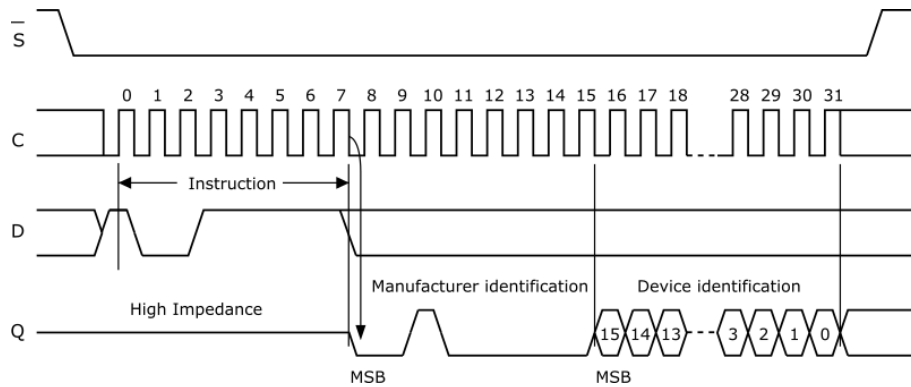


Fig. 4.4: Figure 9 of M25P10-A datasheet

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.

4.5.6 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.

[Table 4 of M25P10-A datasheet](#) 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 of section 6.9:

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.

4.5.7 Write Enable/Disable

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

Most flash devices provide instructions to enable and disable writes to memory. [Table 4 of M25P10-A datasheet](#) 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.

4.5.8 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.

If using SR protection, bit 30 of *a* and *b* can be set to enable a read-modify-write method of accessing the status register. This prevents clearing unrelated status bits. In this scenario, *b* is reinterpreted and must otherwise contain the bitmask of bits to clear - usually the same value set in *a*.

[Table 2 of M25P10-A datasheet](#) and [Table 6 of M25P10-A datasheet](#) 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.

Table 4.6: Table 2 of M25P10-A datasheet

Status Register		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

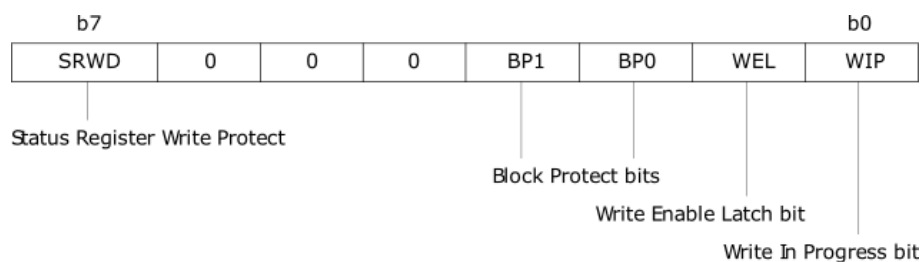


Fig. 4.5: Table 6 of M25P10-A datasheet

4.5.9 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.

[Table 4 of M25P10-A datasheet](#) reproduces the part of the M25P10-A datasheet that provides this information. In the table, a page program command is provided and has the value 0x02.

If page programming is not supported, this value is a concatenation of three separate values. Bits 0..7 must be set to 0. Bits 8..15 should contain the program command. Bits 16..23 should contain the number of bytes per command. The libflash library requires that the first program command accepts a three byte address but subsequent program command use auto address increment (AAI).

An example of a device without a PP command is the [ESMT F25L004A](#). [Table 7 of F25L004A datasheet](#) 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.

Table 4.7: Table 7 of F25L004A datasheet.

Symbol	Parameter	Minimum	Units
T _{PU-READ}	V _{DD} Min to Read Operation	10	us
T _{PU-WRITE}	V _{DD} Min to Write Operation	10	us

The corresponding entry in the specification file is:

```
0x00|(0xad<<8)|(2<<16), /* No PP, have AAI for 2 bytes */
```

4.5.10 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.

For libquadflash, command bits 0..7 denote the QUAD_READ command, typically 0xeb. Bits 8..15 can be used to set a READ or FAST_READ command but this defaults to 0x0b for FAST_READ.

[Table 4 of M25P10-A datasheet](#) 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.

4.5.11 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

4.5.13 Quad Enable Bit

```
0, /* 25. libquadflash: QE bit location, libflash: Reserved */
```

For Quad SPI devices, an additional field is provided for encoding the location of the Quad Enable bit. This field is reserved and has no effect when using libflash for a SPI device.

Bits 0..1 of this field describe the method used to set the bit:

Table 4.9: Quad Enable method

Value	Method	Description	Read-modify-write
0	Legacy flash behavior	libquadflash Set bit 6 in register 0 if manufacturer ID is ISSI or MACRONIX, otherwise set bit 1 in register 1	No
1	Bit position override	Set bit stored in bits 8..15 in register stored in bits 0..7, using the WRSR command	No
2	SFDP requirements	Set bit according to JEDEC Quad Enable Requirements (QER) value stored in bits 24..26	Yes
3	No QE bit	Does not attempt to set a QE bit	N/A

It is not recommended to override this field as the default behavior attempts SFDP query to derive a value using method 2. The value can be left as -1 to inherit from the SFDP query. This field is provided for devices that do not support SFDP query.

Note that XFLASH will populate this field with the QE bit position override, if any, stored in the XN file. This will couple the application XE file to a particular flash device, which may be undesirable. It is recommended to remove the QE bit location from the XN file for systems supporting SFDP.

If a method supports read-modify-write, other bits of the status register will be preserved when setting the QE bit.

4.5.14 SFDP

libquadflash implements a basic level of support for JEDEC JESD216 Serial Flash Discoverable Parameters (SFDP). This automatically populates the flash configuration structure with values obtained from the attached flash device at runtime, reducing the need to manually specify some properties of the device.

The currently supported SFDP parameters and mappings include:

- Quad Enable Requirements: 25 (quadEnable).
- Flash Memory Density: 3 (numPages).
- Page Size: 2 (pageSize).

To use this feature, the associated field in the device configuration file (if any) must be set to -1 to inherit the value from the SFDP query. Any other value will override the result of the query.



4.5.15 Select a Flash Device

When selecting a flash device for use with an xCORE device, the following guidelines are recommended:

- If access to the data partition is required, select a device with fine-grained erase granularity, as this will minimize the gaps between the factory and upgrade images, and will also minimize the amount of data that libflash needs to buffer when writing data.
- Select a device with sector protection if possible, to ensure that the bootloader and factory image are protected from accidental corruption post-deployment.
- Select a flash speed grade suitable for the application. Boot times are minimal even at low speeds.

4.6 Using LPDDR

This section shows how to use LPDDR1 devices in target systems which employ the xcore.ai (XU316) device.

An LPDDR1 memory device may be connected to certain xcore.ai devices. The memory device may be either 256Mbit, 512Mbit or 1024Mbit in size. See the relevant device datasheets for full connectivity details and electrical specifications.

The LPDDR memory contents may be accessed by application software executing on either tile 0 or on tile 1 of an xcore.ai device. It is not possible for both tiles to access LPDDR from a single application.

The application developer provides annotated C-language source to indicate which elements of the application reside in LPDDR. The bootloader performs the hardware setup required for the application to execute from, to read from and to write to this memory.

Note: In a system with two xcore.ai devices, the LPDDR device and the flash memory device (used to boot the system) must be connected to the same xcore.ai device.

4.6.1 Accessing LPDDR in an application

Executable code or data entities that must reside in LPDDR must be annotated with:

```
__attribute__((section(".ExtMem<qualifier>")))
```

<qualifier> must be the string `.bss` for data which must be initialised to 0 by the bootstrap (traditionally known as BSS) (and not occupy space in the flash image).

<qualifier> may be any other string for data which is initialised or for code.

The following examples illustrate how the `__attribute__` annotation should be applied:

```
// Place in BSS - zero-initialise in bootstrap - do not occupy space in the flash image
__attribute__((section(".ExtMem.bss")))
char working_area[10 * 1024 * 1024];

// Place in BSS - zero-initialise in bootstrap - do not occupy space in the flash image
__attribute__((section(".ExtMem.bss")))
extern char working_area[10 * 1024 * 1024];

// Annotation for executable code - occupies space in the flash image
__attribute__((section(".ExtMem.code")))
void procedure(int * p) {
    ...
}

// Annotation for initialised data - occupies space in the flash image
```

(continues on next page)

```

__attribute__((section(".ExtMem.data")))
unsigned ddr_data_word = 0xdeadbeef;

// Annotation for initialised data - occupies space in the flash image
__attribute__((section(".ExtMem.data")))
unsigned ddr_stuff[32768] = { 0x12345678, 0x234567ab, 0x4567abcd };

```

The annotation must be placed immediately before the definition and any declaration of the entity. Use `extern` in a declaration, to avoid multiple definitions of the same object.

4.6.2 Compiling for external memory (LPDDR) and software-defined memory

As described under memory models, accessing a data object in external or software-defined memory directly, by name, requires the `large` or `hybrid` memory model. But accessing it via a pointer can be done in any memory model. XMOS recommends choosing one of the following schemes:

4.6.2.1 Scheme 1: the general case: all objects may be accessed directly

Compile the whole application for the same model, either `large` or `hybrid`.

Model `large` is required only when the contiguous data area of any one tile (usually in internal RAM) exceeds 256KB, or if branches in code exceed the maximum range of the `small` model.

4.6.2.2 Scheme 2: access objects in LPDDR or software-defined memory via pointers

Can be used for external/software memory access when all of the following are true:

1. No code (functions) will be placed in external memory, only data.
2. External/software memory data is in a few, large objects which can be defined in dedicated source files. The scheme is not suitable if external/software memory objects are spread throughout an application.
3. Internal RAM data for a tile fits within 256KB.

Advantage: smaller and faster code is generated to access internal RAM data, than when the `large` model is used.

Define external/software memory objects in specific source files, with an accessor function, returning a pointer to the data. The source files defining external/software memory data and accessor functions must be compiled under the `hybrid` model. (The `large` model would also work, but is unnecessary if the conditions above are met.) All other source files can be compiled under the default (`small`) model.

This scheme is intended for the specific case of storing a small number of very large objects in a memory other than the internal RAM. Writing address-getter functions for many, smaller, objects is not the recommended model.

Example:

4.6.2.2.1 external.c

```
// xcc -mcmmodel=hybrid external.c ...

__attribute__((section(".ExtMem.data"))) int readings[MAX];
int * get_readings(void) { return readings; }
```

4.6.2.2.2 external.h

```
int * get_readings(void);
```

4.6.2.2.3 main.c

```
// xcc -mcmmodel=small main.c ...

#include "external.h"
...
int * r = get_readings();
r[3] = k;
int a = r[7];
```

4.6.3 Hardware setup

The xcore.ai device has a LPDDR controller which must be configured by the bootloader. The parameters required for configuration are provided in the target XN file. The following information is required:

1. The frequency at which the LPDDR interface is clocked
2. The size of the LPDDR device (256Mbit, 512Mbit or 1024Mbit)
3. xcore.ai output pad drive strengths (inputs to the LPDDR device)
4. LPDDR output drive strengths (inputs to the xcore.ai device)

4.6.3.1 LPDDR clock frequency specification

The LPDDR clock may be provided either by the system PLL or by the secondary PLL.

4.6.3.1.1 Using the primary (system) PLL

The LPDDR clock may be specified as a frequency which is derived from the system PLL. The system PLL operates at multiple of 100MHz. This is divided by a constant to give the LPDDR clock. The LPDDR clock is driven from the system PLL via a fixed divide-by-two followed by a programmable divider. The LPDDR clock frequency is:

```
f_lpddr = f_syspll / div
```

where `div` is an even integer in the inclusive range 0x2 to 0x20000. When using the system PLL, the value of `div` is computed based on the LPDDR frequency specified in the XN file. The following target XN file excerpt shows the parameters required to provide 100MHz a LPDDR clock:

```
<Extmem SizeMbit="1024" Frequency="100MHz">
```

This is shown in context along with the drive strength specification:

```

<Packages>
  <Package id="0" Type="XS3-UnA-1024-FB265">
    <Nodes>
      <Node Id="0" InPackageId="0" Type="XS3-L16A-1024" Oscillator="24MHz" SystemFrequency="600MHz"
      ↳ReferenceFrequency="100MHz">

        <Boot>
          <Source Location="bootFlash" />
        </Boot>

        <Extmem SizeMbit="1024" Frequency="100MHz">

          <!-- Attributes for Padctrl and Lpddr XML elements are as per equivalently named 'Node
          ↳Configuration' registers in datasheet -->
          <!--
            Padctrl attributes are applied to each named signal in the set below:
            [6] = Schmitt enable, [5] = Slew, [4:3] = drive strength, [2:1] = pull option, [0] =
            ↳read enable

            Therefore:
            0x30: 8mA-drive, fast-slew output
            0x31: 8mA-drive, fast-slew bidir
            -->
            <Padctrl clk="0x30" cke="0x30" cs_n="0x30" we_n="0x30" cas_n="0x30" ras_n="0x30" addr="0x30
            ↳" ba="0x30" dq="0x31" dqs="0x31" dm="0x30" />

          <!--
            LPDDR emr_opcode attribute:
            emr_opcode[7:5] = LPDDR drive strength to xcore.ai

            0x20: Half drive strength
            -->
            <Lpddr emr_opcode="0x20" />
          </Extmem>

```

The drive strength specifications should be provided based on board design parameters. The above values are for the XMOS XK-EVK-XU316 board which should be used as a reference design when creating a new board.

4.6.3.1.2 Using the secondary PLL

The secondary PLL may be used as a source for the LPDDR clock. This overcomes limits in the available LPDDR frequencies which exist when using the primary (system) PLL. There are two sets of parameters required:

1. The PLL configuration values required to obtain a specified PLL output frequency
2. A division value (which must be an even integer in the range 0x2 to 0x20000) used to divide the PLL output frequency to obtain the LPDDR clock frequency

The PLL output frequency is given by:

$$f_{\text{out}} = (\text{Oscillator} \times \text{SecondaryPllFeedbackDiv} / 2) / (\text{SecondaryPllInputDiv} \times \text{SecondaryPllOutputDiv})$$

The following XN excerpt illustrates the specification of these parameters to provide the PLL output frequency 322MHz and a 166MHz LPDDR clock:

```

<Node Id="0" InPackageId="0" Type="XS3-L16A-1024" Oscillator="24MHz"
SystemFrequency="600MHz" ReferenceFrequency="100MHz"
SecondaryPllInputDiv="1" SecondaryPllOutputDiv="3" SecondaryPllFeedbackDiv="83">

  <Extmem SizeMbit="1024" SourcePll="SecondaryPll" Divider="2">

```

The following shows this excerpt in context (note the detailed comments as shown in the setup using the primary PLL have been removed for brevity):

```
<Node Id="0" InPackageId="0" Type="XS3-L16A-1024" Oscillator="24MHz"
  SystemFrequency="600MHz" ReferenceFrequency="100MHz"
  SecondaryPllInputDiv="1" SecondaryPllOutputDiv="3" SecondaryPllFeedbackDiv="83">

  <Extmem SizeMbit="1024" SourcePll="SecondaryPll" Divider="2">

    <!-- Attributes for Padctrl and Lpddr XML elements are as per equivalently named 'Node
    ↳Configuration' registers in datasheet -->
    <Padctrl clk="0x30" cke="0x30" cs_n="0x30" we_n="0x30" cas_n="0x30" ras_n="0x30" addr="0x30" ba=
    ↳"0x30" dq="0x31" dqs="0x31" dm="0x30" />

    <Lpddr emr_opcode="0x20" />
  </Extmem>
```

4.6.4 Level 1 cache

A level 1 cache is situated between the xCORE tile and the LPDDR memory. This is a unified I and D cache, fully-associative, with write-back. It has 8 lines and the line size is 32 bytes. The replacement policy is pseudo-LRU (Least Recently Used).

xCORE instructions are provided to prefetch, invalidate and flush this cache.

It is not advisable to have more than 2 logical cores access the LPDDR because 'cache-thrashing' will occur (where data required by a logical core is repeatedly evicted by another logical core and must be re-loaded).

4.7 Using software-defined memory

This section shows how to use software-defined memory in xcore.ai target systems.

Software-defined memory is a region of memory with a base address of 0x40000000 and a size of 0x40000000 bytes. The content of this address range is provided by software and therefore software-defined.

When a program performs a read access in this range, the required content is looked up in a level 1 cache. If it is not present a software fill handler (running in another independent logical core) is triggered. This handler must provide an entire cache line of data to satisfy the read and place it in the cache.

When a program performs a write access in this range, the data is written to the cache memory.

When a line is written to it is flagged as "dirty". This indicates that the copy of the data in the cache is more recent than that in the software-defined backing store. When this dirty line has to be evicted from the cache to make way for another line, an "eviction" software handler is triggered.

A typical use of this feature is to provide a cache of data stored in flash where application accesses have properties of spatial and temporal locality.

4.7.1 Level 1 cache

A level 1 cache is present on each tile. This cache is the same cache that is used for LPDDR accesses. It is not recommended to use both LPDDR and software-defined memory together on a tile.

A level 1 cache is situated between the xCORE tile and the LPDDR memory. This is a unified I and D cache, fully-associative, with write-back. It has 8 lines and the line size is 32 bytes. The replacement policy is pseudo-LRU (Least Recently Used).

xCORE instructions are provided to prefetch, invalidate and flush this cache.

4.7.2 Application implementation

An application may manage the software-defined memory directly, by providing fill and evict software handlers that access the data in application-specific backing store. Alternatively the XTC Tools can be used to provide assistance in placing application objects in flash.

The software-defined memory region is not enabled on reset, and an access will cause a trap.

APIs to manage the cache content are provided by `xcore/swmem_fill.h` and `xcore/swmem_evict.h`.

4.7.3 XTC Tools built-in support for flash storage

Executable code or data may be stored in flash for subsequent access by the application via the software-defined memory region. The code or data is annotated to place it in a section and the section name must start with the string `.SwMem`, for example:

```
__attribute__((section(".SwMem_data")))
unsigned int mydata = 12345678;
```

The above will store `mydata` in the flash image built with `xflash` such that it may be accessed via the software-defined memory region.

Both executable code (functions) and data may be annotated to be stored in flash.

The code below will trigger the software fill handler to fetch the content of `mydata` from this region, because it is not yet resident in the level 1 cache:

```
unsigned int newdata = mydata;
```

Once the software fill handler has obtained the data and placed it in the cache, a subsequent read of `mydata` will not trigger the software fill handler, unless the line containing `mydata` has been evicted because eight other cache lines have been filled.

4.7.4 Compiling for software-defined memory

See [compiling for external memory \(LPDDR\) and software-defined memory](#).

4.7.5 Examples

Two separate examples are provided; one for the software fill handler and another for the software evict handler. These use the XTC Tools built-in support to place annotated objects in flash.

4.7.5.1 Fill handler example

The following example illustrates the use of this feature. Two logical cores are used; the first is the “application” which requires data stored in flash, and the second is the software fill handler, which is triggered to fetch data from flash and place it in the cache for the application.

The fill handler uses APIs provided by `xmos_flash.h` to read data from flash and APIs provided by `xcore/swmem_fill.h` to write data into the level 1 cache. The symbol `__swmem_address` must be defined and initialised to `0xFFFFFFFF`. The system bootstrap will overwrite this with a value which provides an offset into flash from which the annotated application data will be fetched.

In this example a read to the address `0x50000000` will cause the fill handler loop running on a logical core to terminate.

Build the example with:

```
$ gcc main.c main.xc -o main.xe -lquadspi -target=XCORE-AI-EXPLORER -mcmmodel=large
```

Listing 4.4: main.c

```
#include <stdio.h>
#include <xcore/parallel.h>
#include <xcore/swmem_fill.h>
#include <xmos_flash.h>

__attribute__((section(".SwMem_data")))
const unsigned int my_array[20] = {
    1,2,3,4,5,6,7,8,9,10,11,12,13,14,15,16,17,18,19,20
};

flash_ports_t flash_ports_0 =
{
    PORT_SQI_CS,
    PORT_SQI_SCLK,
    PORT_SQI_SIO,
    XS1_CLKBLK_5
};

flash_clock_config_t flash_clock_config =
{
    1,
    8,
    8,
    1,
    0,
};

flash_qe_config_t flash_qe_config_0 =
{
    flash_qe_location_status_reg_0,
    flash_qe_bit_6
};

flash_handle_t flash_handle;

// We must initialise this to a value such that it is not memset to zero during C runtime startup
```

(continues on next page)

(continued from previous page)

```
#define SWMEM_ADDRESS_UNINITIALISED 0xffffffff
volatile unsigned int __swmem_address = SWMEM_ADDRESS_UNINITIALISED;

static unsigned int nibble_swap_word(unsigned int x)
{
    return ((x & 0xf0f0f0f0) >> 4) | ((x & 0x0f0f0f0f) << 4);
}

void swmem_fill(swmem_fill_t handle, fill_slot_t address) {
    swmem_fill_buffer_t buf;
    unsigned int * buf_ptr = (unsigned int *) buf;

    flash_read_quad(&flash_handle, (address - (void *)XS1_SWMEM_BASE + __swmem_address) >> 2, buf_ptr,
    ↪SWMEM_FILL_SIZE_WORDS);
    for (unsigned int i=0; i < SWMEM_FILL_SIZE_WORDS; i++)
    {
        buf_ptr[i] = nibble_swap_word(buf_ptr[i]);
    }

    swmem_fill_populate_from_buffer(handle, address, buf);
}

swmem_fill_t swmem_setup() {
    flash_connect(&flash_handle, &flash_ports_0, flash_clock_config, flash_qe_config_0);

    if (__swmem_address == SWMEM_ADDRESS_UNINITIALISED)
    {
        __swmem_address = 0;
    }

    return swmem_fill_get();
}

void swmem_takedown(swmem_fill_t fill_handle) {
    swmem_fill_free(fill_handle);
    flash_disconnect(&flash_handle);
}

static const fill_slot_t swmem_terminate_address = (void *)0x50000000;

DECLARE_JOB(swmem_handler, (swmem_fill_t))
void swmem_handler(swmem_fill_t fill_handle)
{
    fill_slot_t address = 0;
    while (address != swmem_terminate_address)
    {
        address = swmem_fill_in_address(fill_handle);
        swmem_fill(fill_handle, address);
        swmem_fill_populate_word_done(fill_handle, address);
    }
}

DECLARE_JOB(use_swmem, (void))
void use_swmem(void)
{
    volatile unsigned long a = 0;

    for (int i = 0; i < 20; i++) {
        printf("Result: 0x%08x\n", my_array[i]);
        a = my_array[i];
    }
}
```

(continues on next page)



(continued from previous page)

```
a = *(const volatile unsigned long *)swmem_terminate_address;
}

void tile_main(void) {
    swmem_fill_t fill_handle = swmem_setup();

    PAR_JOBS(
        PJOB(swmem_handler, (fill_handle)),
        PJOB(use_swmem, ())
    );

    swmem_tear_down(fill_handle);
}
```

Listing 4.5: main.xc

```
#include <platform.h>
#include <stdio.h>

void tile_main(void);

int main(void) {
    par {
        on tile[0]: par {
            tile_main();
        }
        on tile[1]: par {
        }
    }
    return 0;
}
```

4.7.5.2 Evict handler example

The following example illustrates the main.c file for a software evict handler.

Listing 4.6: main.c

```
#include <stdio.h>
#include <xcore/parallel.h>
#include <xcore/swmem_evict.h>
#include <xcore/minicache.h>
#include <xmos_flash.h>

__attribute__((section(".SwMem_data")))
unsigned char my_array[512] = {};

DECLARE_JOB(swmem_handler, (swmem_evict_t))
void swmem_handler(swmem_evict_t evict_handle)
{
    for (unsigned evictions = 0; evictions < 16; evictions += 1)
    {
        evict_slot_t address = swmem_evict_in_address(evict_handle);
        unsigned long mask = swmem_evict_get_dirty_mask(evict_handle, address);
        unsigned long buf[SWMEM_EVICT_SIZE_WORDS];
        swmem_evict_to_buffer(evict_handle, address, buf);
        printf("Eviction of address %p with mask %lx; data:\n", address, mask);
    }
}
```

(continues on next page)



(continued from previous page)

```
    for (unsigned i = 0; i < SWMEM_EVICT_SIZE_WORDS; i += 1)
    {
        printf(i == SWMEM_EVICT_SIZE_WORDS - 1 ? "%lx\n" : "%lx ", buf[i]);
    }
}

DECLARE_JOB(use_swmem, (void))
void use_swmem(void)
{
    volatile unsigned char *a = my_array;

    for (unsigned i = 0; i < sizeof(my_array); i += 8)
    {
        *((volatile unsigned long *) (a + i)) = (unsigned long) &a[i];
        a[i + 4] = i/4;
        a[i + 5] = 0;
        a[i + 7] = 255;
        if (i % 16) { a[i + 6] = 10; }
    }
    // Performs asm volatile ("flush");
    minicache_flush();
}

void tile_main(void) {
    swmem_evict_t evict_handle = swmem_evict_get();

    PAR_JOBS(
        PJOB(swmem_handler, (evict_handle)),
        PJOB(use_swmem, ())
    );

    swmem_evict_free(evict_handle);
}
```

4.7.6 Using xrun and xgdb

When an application is written to flash using xflash the value of `__swmem_address` will be an offset into the flash storage from which the annotated application objects may be obtained.

But when using xrun or xgdb to run an application the flash bootloader does not execute so `__swmem_address` will retain the initialiser value of `0xFFFFFFFF`.

The data that would be placed at an offset by xflash needs to be extracted from the image and written to the the bottom of flash. The `__swmem_address` will be set to 0 when it contains the value `0xFFFFFFFF` as shown in the example above.

Extracting and writing the data to the bottom of flash is done as follows:

```
$ xobjdump --strip main.xe
$ xobjdump --split main.xb
$ xflash --reverse --write-all image_n0c0.swmem --target XCORE-AI-EXPLORER
```

The data nibbles must be swapped to match the format in which xflash stores a complete application in flash. In this example the the swap is done by the `--reverse` option to xflash.

4.8 How to use arguments and return codes

Arguments and return codes are not really useful in a true embedded application, since a user is not present to provide them or react to them. However, they are particularly useful during unit and regression testing, as they allow tests to be configured and pass/fail results returned simply.

Note: The facility to use arguments and return codes is only available for a single tile application; an application with a `main()` function written in the C language.

It makes no sense for a multi-tile application (with an XC `main()` function) to use arguments and return codes, because there is no mechanism to define which is the 'master' tile, nor define how it should distribute the arguments and collate the return code. Therefore this tutorial is suited only to single tile testing.

To add command line arguments, create a `main()` function with the usual prototype:

Listing 4.7: main.c

```
#include <stdio.h>

int main(int argc, char* argv[])
{
    for (int x = 0; x < argc; x++)
        printf("Arg %d %s\n", x, argv[x]);
    return argc;
}
```

Build with a non-zero value for `xcc -fcmdline-buffer-bytes`:

```
$ xcc main.c -target=XCORE-200-EXPLORER -fcmdline-buffer-bytes=1024
```

Note: If you forget to the `-fcmdline-buffer-bytes` parameter, then a buffer of size zero will be allocated, and `argc` will always hold a value of zero. No error or warning will be raised. So don't forget!

Run with using `xrun --args`, and examine the return code:

```
$ xrun --io --args a.xe giraffe elephant
Arg 0 a.xe
Arg 1 giraffe
Arg 2 elephant
$ echo $?
3
```

Similar behaviour can be found using `xsim --args` and `xgdb --args`.

4.9 Using XSIM

In this section, we have a series of examples which show some ways of using XSIM to gain a greater insight into how programs run on XCORE devices.

4.9.1 Which tile is my code running on?

Build and execute the example in *Targeting multiple tiles*:

```
$ xcc -target=XCORE-200-EXPLORER multitile.xc main.c
$ xrun --io a.xe
Hello from tile 0
Hello from tile 1
```

How can we convince ourselves that the output was generated by code running on the specified tiles? We can use `xsim -t` to determine which tile the system calls were generated by. Here we use `grep` to simplify and reduce the trace output by capturing 1 line before each of the relevant output lines:

```
$ xsim -t a.xe | grep -B 1 "Hello from"
tile[0]@0- -SI A-.----00040264 (_DoSyscall      + 0) : nop      @11057
Hello from tile 0
tile[1]@0- -SI A-.----00040234 (_DoSyscall      + 0) : nop      @11061
Hello from tile 1
```

This shows that the system calls were generated by the expected tiles.

4.9.2 Using XScope during simulation

You can use XScope when running code on the simulator. It isn't faster than the non-Xscope approach, but it allows debug of any XScope-related issues.

Build the example in *Using XSCOPE for fast "printf debugging"*. Execute the application using the `xsim --xscope` option as follows:

```
$ xsim --xscope "--offline xscope.vcd" a.xe
Tile 0: Result 0
Tile 1: Iteration 0 Accumulation: 0
Tile 1: Iteration 1 Accumulation: 1
Tile 1: Iteration 2 Accumulation: 3
Tile 1: Iteration 3 Accumulation: 6
Tile 1: Iteration 4 Accumulation: 10
Tile 1: Iteration 5 Accumulation: 15
Tile 1: Iteration 6 Accumulation: 21
Tile 1: Iteration 7 Accumulation: 28
Tile 1: Iteration 8 Accumulation: 36
Tile 1: Iteration 9 Accumulation: 45
Tile 0: Result 45
```

XScope trace output will be placed in `xscope.vcd`.

4.10 Understanding XE files and how they are loaded

This short tutorial aims to help you understand:

- How a multi-tile application is stored inside an *XE file*
- How to use *XOBDUMP* to explore the contents of an XE file
- How *XRUN* (and thus XGDB) coordinates the loading and execution of an XE file

4.10.1 Prepare an XE file

First, build a .xe as per the example in [Targeting multiple tiles](#).

4.10.2 Examine the XE file

Before we run a .xe, let's understand what's inside it.

The XOBJDUMP tool is used to examine and manipulate the contents of an [XE file](#). Let's have a look at the innards of a .xe using `xobjdump --sector-info`. This lists the contents or 'sectors' of the .xe package:

```
$ xobjdump --sector-info a.xe
a.xe: file format: xcore-xe

Xmos binary sector information: file: a.xe

0: NODEDESC sector, part number: 0x5633
1: ELF sector for tile[0] (node 0, tile 0)
2: CALL sector. Address: 0x00000000
3: ELF sector for tile[1] (node 0, tile 1)
4: CALL sector. Address: 0x00000000
5: ELF sector for tile[0] (node 0, tile 0)
6: GOTO sector. Address: 0x00000000
7: ELF sector for tile[1] (node 0, tile 1)
8: GOTO sector. Address: 0x00000000
9: SYSCONFIG sector
10: XN sector
11: PROGINF0 sector
12: XSCOPE sector
13: LASTSEC sector
```

Why are there four ELF's within the package? We only wrote one application!

It's because the XCORE-200-EXPLORER target describes two cores or 'tiles' within one XMOS package. An application ELF is always generated by the tools for each tile. In this case our `Hello from tile 0` is generated by the ELF in sector 5 on executing on `tile[0]`; the ELF in sector 7 generates our `Hello from tile 1` by executing on `tile[1]`.

But what about the ELF's in sectors 1 and 3? These are automatically generated. They contain start-of-day SoC and tile setup code which is executed prior to loading of the application ELF(s). The setup code is added to the setup ELF's because:

- Some setup may be required before loading of the application ELF is possible and/or;
- Setup placed in the setup ELF's does not waste space in the application ELF.

Note: Single-tile applications on multi-tile targets

If you create the single-tile application:

Listing 4.8: single-tile.c

```
#include <stdio.h>

int main(void) {
    printf("Hello world!\n");
    return 0;
}
```

...and build it specifying a multi-tile target (say the two-tile XCORE-200-EXPLORER):

```
$ xcc -target=XCORE-200-EXPLORER single-tile.c
```

...then an XE file containing 4 ELF's will still be produced; two setup ELF's and two application ELF's. The single-tile application ELF will default to execute on tile[0]; another automatically generated application ELF will execute on tile[1]. The automatically generated application ELF simply halts tile[1].

4.10.3 Load and execute the XE file

We now aim to illustrate the general description of *how an XE file is booted* by showing how XRUN loads and executes a .xe. Run a .xe with XRUN using the `xrun --verbose` option:

```
$ xrun --io --verbose a.xe
```

A lot of output is produced. Here's a shortened summary of the interesting bits. The first part confirms that the XRUN tool is actually just a convenience wrapper of the debugger tool XGDB:

```
>>>> xgdb script (/tmp/.xrun11863-5NG8M6AT/xeload_auto.gdb)
...
<<<<< xgdb script

>>>> xgdb cmd:
...
<<<<< xgdb cmd
```

The next part shows XGDB loading and executing the two setup ELF's:

```
Loading setup image to XCore 0
Loading section .text, size 0x158 lma 0x40000
Loading section .cp.rodata, size 0x18 lma 0x40158
Loading section .dp.data, size 0x10 lma 0x40170
Start address 0x40000, load size 384
...

Loading setup image to XCore 1
Loading section .text, size 0x40 lma 0x40000
Loading section .cp.rodata, size 0x18 lma 0x40040
Start address 0x40000, load size 88
...
```

The next part shows XGDB loading and executing the two application ELF's:

```
Loading application image to XCore 0
Loading section .crt, size 0xac lma 0x40000
Loading section .init, size 0x1a lma 0x400ac
Loading section .fini, size 0x2e lma 0x400c6
Loading section .text, size 0x4b1c lma 0x400f4
Loading section .cp.rodata, size 0x108 lma 0x44c10
Loading section .cp.rodata.4, size 0x5c lma 0x44d18
Loading section .cp.const4, size 0x28 lma 0x44d74
Loading section .cp.rodata.string, size 0x84 lma 0x44d9c
Loading section .cp.rodata.cst4, size 0xcc lma 0x44e20
Loading section .dp.data, size 0x20 lma 0x44ef0
Loading section .dp.data.4, size 0x24 lma 0x44f10
Start address 0x40000, load size 20272
...

Loading application image to XCore 1
Loading section .crt, size 0xac lma 0x40000
```

(continues on next page)



(continued from previous page)

```
Loading section .init, size 0x1a lma 0x400ac
Loading section .fini, size 0x2e lma 0x400c6
Loading section .text, size 0x4aec lma 0x400f4
Loading section .cp.rodata, size 0x100 lma 0x44be0
Loading section .cp.rodata.4, size 0x5c lma 0x44ce0
Loading section .cp.const4, size 0x28 lma 0x44d3c
Loading section .cp.rodata.string, size 0x84 lma 0x44d64
Loading section .cp.rodata.cst4, size 0xcc lma 0x44de8
Loading section .dp.data, size 0x20 lma 0x44eb8
Loading section .dp.data.4, size 0x24 lma 0x44ed8
Start address 0x40000, load size 20216
...
```

The last part shows XGDB displaying the printf() output and waiting for the application to complete before returning control to the terminal:

```
Hello from tile 0
Hello from tile 1

Program exited normally.
```

5 Reference

The reference section is intended for users of the tools already familiar with basic tools usage who are looking for specific information.

5.1 Command line tools

This section describes the individual command line tools in a “man page” style.

5.1.1 XRUN

5.1.1.1 Synopsis

```
xrun [INQUIRY OPTIONS]
xrun [RUN OPTIONS] xe-file
xrun [RUN OPTIONS] --args xe-file arg1 arg2 .. argn
```

5.1.1.2 Description

The xrun tool has two key roles:

- To list the connected USB xtag devices
- To load and run *XMOS Executable (XE)* files on target hardware via a selected USB xtag debugger.

xrun is actually a wrapper around the xgdb tool, and is provided to simplify common usage patterns of xgdb. Therefore everything that can be done with xrun can also be done with xgdb (but not the other way round).

5.1.1.3 Options

--args <xe-file> <arg1> <arg2> ... <argn>

Provides an alternative way of supplying the XE file which also allows command line arguments to be passed to a program.

5.1.1.3.1 Inquiry options

The following options may be used without supplying an XE file. The most commonly used is *-l*.

--list-devices, -l

Prints an enumerated list of all JTAG adapters connected to the host and the devices on each JTAG chain. This example shows that there are two xtag devices present:

```
$ xrun -l
Available XMOS Devices
-----
ID      Name                               Adapter ID  Devices
```

(continues on next page)



--	----	-----	-----
0	XMOS XTAG-3	V0JhnXmh	0[0]
1	XMOS XTAG-3	wfF.G58J	P[0]

The adapters are ordered by their Adapter IDs.

--list-board-info, -lb

Displays information about the connected target board.

--help

Prints a description of the supported command line options.

--version

Displays the version number and copyrights.

5.1.1.3.2 Run options

If only one xtag device is present, it will be specified implicitly. If more than one device is present, the device must be specified using `--id` or `--adapter-id`.

--id <ID>

Specifies the adapter connected to the target hardware.

--adapter-id <ADAPTER-ID>

Specifies the serial number of the adapter connected to the target hardware.

--verbose

Prints information about the program loaded onto the target devices.

--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 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.

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.

Warning: The `--xscope` option should be used in preference to the `--io` option.

System calls delivered via JTAG are slow and cause all threads on a tile to be paused. Any real-time guarantees in a developer's application will likely be broken.

The `--io` option is only used for quick examples or for platforms where the XSCOPE XLINK interface to the XTAG debugger has not been wired.

--attach

Attaches to a JTAG adapter (of a running program), enabling system calls with the host. XRUN terminates when the program performs a call to `exit`.

An XE file must be specified with this option.

--dump-state

Prints the core, register and stack contents of all xCORE Tiles in JTAG scan chain.

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.

--xscope-io-only

Similar to `--xscope`

5.1.1.4 Examples

```
$ xrun a.xe
```

Asynchronously launch `a.xe` on the single connected target, and return control to the prompt immediately without waiting for the target to exit.

```
$ xrun -l
```

List the available USB xtag devices.

```
$ xrun --adapter-id V0JhnXmh --args a.xe giraffe elephant
```

Asynchronously launch `a.xe` on xtag with Adapter ID of 'V0JhnXmh' with command line arguments.

5.1.2 XSIM

5.1.2.1 Description

XSIM provides a near cycle-accurate model of systems built from one or more xCORE devices. Using the simulator, you can output data to VCD files that can be displayed in standard trace viewers such as GTKWave, including a processor's instruction trace and machine state. Loopbacks can also be configured to model the behavior of components connected to XMOS ports and links.

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

```
xsim <binary>
```

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

```
connect -s
```

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

5.1.2.2 Options

5.1.2.2.1 Overall Options

--args <xe-file> <arg1> <arg2> ... <argn>

Provides an alternative way of supplying the XE file which also allows command line arguments to be passed to a program.

<xe-file>

Specifies an XE file to simulate.

--max-cycles <n>

Exits when *n* system cycles is reached.

--plugin <name> <args>

Loads a plugin DLL. The format of *args* is determined by the plugin; if *args* contains any spaces, it must be enclosed in quotes.

--stats

On exit, prints the following:

- A breakdown of the instruction counts for each logical core.
- The number of data and control tokens sent through the switches.

--help

Prints a description of the supported command line options.

--version

Displays the version number and copyrights.

5.1.2.2.2 Warning Options

--warn-resources

Prints (on standard error) warning messages for the following:

- A timed input or output operation specifies a time in the past.
- The data in a buffered port's transfer register is overwritten before it is input by the processor.

--warn-stack

Turns on warnings about possible stack corruption.

xSIM prints a warning if one XC task attempts to read or write to another task's workspace. This can happen if the stack space for a task is specified using either `~~#pragma stackfunction~~` or `~~#pragma stackcalls~~`.

--no-warn-registers

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

5.1.2.2.3 Tracing Options

--trace, -t

Turns on instruction tracing for all tiles (see [XSIM Trace output](#)).

--trace-to <file>

Same as `--trace`, but redirects trace output to specified file.

--disable-rom-tracing

Turns off tracing for all instructions executed from ROM.

--enable-fnop-tracing

Turns on tracing of FNOP instructions.

--vcd-tracing <args>

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

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

```
[global-options] <-tile name <trace-options>>
```

The global options are:

-pads

Turns on pad tracing.

-o <file>

Places output in <file>.

The trace options are specific to the tile associated with the XN core declaration name, for example tile[0].

The trace options are:

-ports

Turns on port tracing.

-ports-detailed

Turns on more detailed port tracing.

-cycles

Turns on clock cycle tracing.

-clock-blocks

Turns on clock block tracing.

-cores

Turns on logical core tracing.

-instructions

Turns on instruction tracing.

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

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

```
xsim a.xe --vcd-tracing "-o trace.vcd -start-disabled -tile tile[0] -ports"
```

Tracing by the VCD plugin can be enabled and disabled using the `_traceStart()` and `_traceStop()` syscalls. The `-start-disabled` argument disables the vcd tracing from the start, allowing the user to enable/disable only those sections of code where tracing is desired. For example:

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

port p1 = XS1_PORT_1A;

int main() {
    p1 <: 1;
    p1 <: 0;

    _traceStart();
```

(continues on next page)

```

p1 <: 1;
p1 <: 0;
_traceStop();

p1 <: 1;
p1 <: 0;

return 0;
}

```

5.1.2.2.4 Profiling Options

--gprof

This option will profile the application at the function, statement and address access level. Profiling data will be written to a file for subsequent analysis. See section [Analysing the profile data](#).

5.1.2.2.5 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` 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` in the XN file used to compile the program.

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

-port <tile> <p> <n> <offset>

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

The value of *tile* must match the `Reference` attribute of a `~~Tile~~ node` in the XN file used to compile the program.

p can be any of the port identifiers defined in `<xs1.h>`. Setting *offset* to a non-zero value specifies a subset of the available pins.

The plugin options are specified in pairs, one for each end of the connection. For example, the following command configures the simulator to loopback the pin connected to port `XS1_PORT_1A` on `tile[0]` to the pin defined by the port `UART_TX` in the program.

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

5.1.2.2.6 xSCOPE Options

--xscope <args>

Enables xSCOPE. file format.

If <args> contains any spaces, it must be enclosed in quotes. One of the following 2 options is mandatory:

-offline <filename>

Runs with xSCOPE in offline mode, placing the xSCOPE output in the given file.

-realtime <URL:port>

Runs with xSCOPE in realtime mode, sending the xSCOPE output in the given URL:port.

The following argument is optional:

-limit <num records>

Limits the xSCOPE output records to the given number.

For example, the following will run xSIM with xSCOPE enabled in offline mode:

```
xsim app.xe --xscope "-offline xscope.xmt"
```

For example, the following will run xSIM with xSCOPE enabled in realtime mode:

```
xsim app.xe --xscope "-realtime localhost:12345"
```

5.1.3 XCC

5.1.3.1 Synopsis

```
xcc [-c|-S|-E] [-g] [-O<level>]
     [-W<warn>...]
     [-I<dir>...] [-L<dir>...]
     [-D<macro>[=<defn>]]
     [-f<option>...]
     [-target=<platform>]
     [-mcmmodel=<model>]
     [-o <outfile>] infile
```

Only the most useful options are listed here; see below for the remainder.

5.1.3.2 Description

XCC is deliberately analogous to GCC and imitates many of its behaviours. XCC is a wrapper around a collection of tools to make the use of those tools simpler. Those tools perform the steps of preprocessing, compilation (both C/C++ and XC), assembly and 'mapping'. Those tools may be used individually, but are more easily used via xcc.

XCC will normally run all of the steps. The first three steps are similar to GCC in that they are applied to an individual source file to produce an object file. The 'mapping' step, as the name suggests, is different. Rather than producing an executable for a single processor, the linker is invoked multiple times to produce an executable per processor. Furthermore, the process includes auto-generating source code, compiling it and invoking the linker further times. The resulting single output file is an *XE file*, which contains the multiple executables.

Options such as **-c**, **-S** and **-E** prevent all the steps running, and instead produce an intermediate output.

No matter which step is being performed, a platform conforming to the [XN Specification](#) must be specified. This is done via `XCC_DEFAULT_TARGET`, `-target` or, better, listed as one of the input files.

During compilation of a program, the compiler generates a temporary header file named `p1atform.h` that contains variable and macro definitions, as defined by the target XN file, which include:

- Declarations of variables of type `tileref` (see [Declaration](#)).
- Macro definitions of port names (see [Port](#)).

Many options are passed directly through to the tool performing the processing step. For instance, some options directly influence the behaviour of the compiler or the linker.

For the most part, the order you use for the options doesn't matter. Order does matter when you use several options of the same kind; for example, if you specify `-L` more than once, the directories are searched in the order specified. In other cases, such as with `-O` repeated usage will result in the rightmost usage to take precedence.

Many options have negative forms (for example, `-fno-<option>`).

A space between an option and its argument is permitted.

5.1.3.3 Options

5.1.3.3.1 Overall Options

-x <language>

Overrides default handling 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`
- `xscope`
- `none` (turn off language specification)

Default handling of the input files is determined by their suffix:

Table 5.1: File extensions recognized by XCC and their meaning

Extension	Type of File	Preprocessed by XCC
<code>.xc</code>	XC source code	Y
<code>.c</code>	C source code	Y
<code>.cpp</code>	CPP source code (for compatibility, the extensions <code>cc</code> , <code>cp</code> , <code>c++</code> , <code>C</code> and <code>cxx</code> are also recognized)	Y
<code>.S</code>	Assembly code	Y
<code>.xscope</code>	xSCOPE configuration file	N
<code>.xn</code>	xCORE Network Description	N
<code>.xi</code>	XC source code	N
<code>.i</code>	C source code	N
<code>.ii</code>	C++ source code	N
<code>.s</code>	Assembly code	N
<i>other</i>	Object file <code>.o</code> 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 `xs1_target_behavior_channel_communication`.

-target=<platform>

Convenience mechanism for specifying a pre-canned target platform. The platform configuration must be specified in the file `platform.xn`, which is searched for in the paths specified by the `XCC_TARGET_PATH` environment variable.

The use of this option is not recommended for long-lived projects due to the likelihood of the target platform changing in an uncontrolled fashion. It is best practice to specify a platform explicitly by defining and supplying a target platform as one of the `xcc` input files. For example:

```
$ xcc my_target.xn main.c
```

-mcmode1=<model>

Select memory model: `small`, `large`, or `hybrid`.

-foverlay

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

-foverlay=flash

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

-foverlay=syscall

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

-fxscope[=link|uart]

Enable support for tracing using xSCOPE (defaults to `link`). The XN file of the target must contain an xSCOPE link. The option should be passed when compiling and linking.

-fcmdline-buffer-bytes=<n>

Add a buffer of size <n> bytes to be used to hold command line arguments.

-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.

-o <file>

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 `-v` is also specified, `--help` is also passed to the subprocesses invoked by XCC.

--version

Displays the version number and copyrights.

-save-temps

Save intermediate files to current directory. Where possible, files are named based on the source file. The generated file `platform.h` is always written with the same name, and therefore parallel builds with this option enabled should be avoided.

This option is not forwarded to `xmap`; see `-Xmapper` if this is the desired behaviour.

5.1.3.3.2 Warning Options

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

-fsyntax-only

Checks the code for syntax errors only, then exits.

-w

Turns off all warning messages.

-Wbidirectional-buffered-port

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

-Wchar-subscripts

Warns if an array subscript has type `char`.

-Wcomment

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

-Wimplicit-int

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

-Wmain

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

-Wmissing-braces

Warns if an aggregate or union initializer is not fully bracketed.

-Wparentheses

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

-Wreturn-type

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

-Wswitch-default

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

-Wswitch-fallthrough

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

-Wtiming

Warns if timing constraints are not satisfied. This is default.

-Wtiming-syntax

Warns about invalid syntax in timing scripts. This is default.

-Wunused-function

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

-Wunused-parameter

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

-Wunused-variable

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

-Wunused

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

-Wall

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

The following `-W...` options are not implied by `-Wall`.

-Wextra, -W

Prints extra warning messages for the following:

- A function can return either with or without a value (C, C++ only).
- An expression statement or left-hand side of a comma expression contains no side effects. This warning can be suppressed by casting the unused expression to `void` (C, C++ only).

- An unsigned value is compared against zero with `<` or `<=`.
- Storage-class specifiers like `static` are not the first things in a declaration (C, C++ only).
- A comparison such as `x<=y<=z` appears (XC only).
- The return type of a function has a redundant qualifier such as `const`.
- Warns about unused arguments if `-Wall` or `-Wunused` is also specified.
- A comparison between signed and unsigned values could produce an incorrect result when the signed value is converted to unsigned. (Not warned if `-Wno-sign-compare` is also specified.)
- An aggregate has an initializer that does not initialize all members.
- An initialized field without side effects is overridden when using designated initializers (C, C++ only).
- A function parameter is declared without a type specifier in K&R-style functions (C, C++ only).
- An empty body occurs in an `if` or `else` statement (C, C++ only).
- A pointer is compared against integer zero with `<`, `<=`, `>`, or `>=`. (C, C++ only).
- An enumerator and a non-enumerator both appear in a conditional expression. (C++ only).
- A non-static reference or non-static `const` enumerator and a non-enumerator both appear in a conditional expression (C++ only).
- Ambiguous virtual bases (C++ only).
- Subscripting an array which has been declared `register` (C++ only).
- Taking the address of a variable which has been declared `register` (C++ only).
- A base class is not initialized in a derived class' copy constructor (C++ only).

-Wconversion

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

-Wdiv-by-zero

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

-Wfloat-equal

Warns if floating point values are used in equality comparisons.

-Wlarger-than-<len>

Warns if an object of larger than *len* bytes is defined.

-Wpadded

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

-Wreinterpret-alignment

Warns when a reinterpret cast moves to a larger alignment.

-Wshadow

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

-Wsign-compare

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

-Wsystem-headers

Prints warning messages for constructs found in system header files. This is not default. See [Directory Options](#).



-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.

5.1.3.3 Debugging Options

-g

Produces debugging information.

-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.

-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

Lists the target platforms (XN files) found via `XCC_TARGET_PATH`, excluding those in folders with name `.deprecated`.

-print-boards

Same as `-print-targets`, but only lists target platforms with the `< Type >` element set to "Board".

5.1.3.4 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.

-funroll-loops

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

-finline-functions

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

5.1.3.3.5 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* except that dependencies on system headers are ignored.

-MF <file>

Write dependency information to *file*.

-MP

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

-MT <file>

Specifies the target of the rule emitted by dependency generation.

5.1.3.3.6 Mapper Options

The following options control the mapper and its linker.

-l <library>

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

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

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

-nostartfiles

Do not link with the system startup files.

-nodefaultlibs

Do not link with the system libraries.

-nostdlib

Do not link with the system startup files or system libraries.

-s

Removes all symbol table and relocation information from the executable.

-default-clkblk <clk>

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

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

-Wm, <option>

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

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

-Xmapper <option>

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

-report

Prints a summary of resource usage.

5.1.3.3.7 Directory Options

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

-I <dir>

Adds *dir* to the list of directories to be searched for header files.

-isystem <dir>

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.

-iquote <dir>

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.

-L <dir>

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

5.1.3.4 Environment

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_TARGET_PATH

A list of directories to be searched for target configuration files. See `-target`, `-print-targets` and `-print-boards`.

XCC_EXEC_PREFIX

If set, subprograms executed by the compiler are prefixed with the value of this environment variable. No directory separator 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 provided as an input file.

5.1.4 XOBJDUMP

5.1.4.1 Synopsis

```
xobjdump [OPTIONS] xe-file
```

5.1.4.2 Description

The xobjdump tool is used to examine and manipulate the contents of *XMOS Executable (XE)* files.

5.1.4.3 Options

--help

Display a summary of the available options.

--version

Display the build version information.

--sector-info

Lists the contents or 'sectors' of the .xe file.

--strip

Creates a new XE file with suffix .xb in which the ELF sectors containing ELF files have been replaced with BINARY sectors containing flat binary images and a 'load address' at which to place them.

Also removes the SYSCONFIG sector.

XE files generated by this option retain the same format and can therefore be further manipulated by xobjdump.

--split, -s

Extracts ELF, BINARY, SYSCONFIG, XN, PROGINFO and XSCOPE sectors from the XE package and writes them as files in the current directory. Specifically:

Sector type	Default generated filename
ELF	image_n<node>c<tile>.elf
BINARY	image_n<node>c<tile>.bin
SYSCONFIG	config.xml
XN	platform_def.xn
PROGINFO	program_info.txt
XSCOPE	xscope.xscope

Where a sector exists with duplicate <node> and <core> value to a sector previously extracted (which will often be the case), the extracted filename will have an incrementing id added. For instance, the first ELF sector on node 0, tile 0 will be extracted as image_n0c0.elf. The second ELF sector on node 0, tile 0 will be extracted as image_n0c0_2.elf.

--split-dir

If **--split** is set, then extracts sectors into the directory <dir>. The directory must already exist.

-o <file>

When used with **--split**, causes ELF sectors to be extracted to <file>_n<node>c<tile>.elf.

When used with **--strip**, new XE file is given name <file>.

--disassemble, -d

Disassembles contents of all executable sectors in XE file

--source, -S

Same as **--disassemble**, but interleaves source code into disassembly output. Requires source to have been built with *xcc -g*.

--disassemble-all, -D

Same as **--disassemble**, but also provides binary contents of non-executable data sections.

5.1.4.3.1 Readelf-alike options

The following options all have more powerful alternatives that rely on the generally available [readelf](#) tool.

--syms, -t

Provides the symbol table for every ELF sector in the XE file.

More powerful example: Use `xobjdump --split` to obtain all the ELF files, then issue `readelf --syms *.elf`.

--size

Provides a summary of the code and data (both initialised and uninitialised) requirements for every ELF sector in the XE file.

More powerful example: Use `xobjdump --split` to obtain all the ELF files, then issue `readelf --sections *.elf`.

5.1.5 XGDB

5.1.5.1 Synopsis

```
xgdb [options] xe-file
xgdb [options] --args xe-file arg1 arg2 .. argn
```

5.1.5.2 Description

XGDB is an extended version of the more familiar [GDB debugger](#). The extensions allow XGDB to debug a multi-file Xcore application in the form of an *XE file*.

XGDB is able to attach to multiple targets including:

- Real hardware via an XTAG debug adapter
- Simulated hardware provided by XSIM

Most documentation about use of XGDB can be found from the [GDB documentation](#); this page largely only documents where XGDB extends or differs from GDB.

5.1.5.3 Options

The options for XGDB are largely the same as for GDB. Only options referenced by this documentation are described here. To find all the available commands, use the `--help` option.

--args <xe-file> <arg1> <arg2> ... <argn>

Provides an alternative way of supplying the XE file which also allows command line arguments to be passed to a program.

-ex <command>

Execute given command. If command contains spaces, it should be contained in quotes.

--help

List all available command line options

5.1.5.4 XMOS commands

The following commands are all provided as part of the XGDB extension to GDB. They can be listed within XGDB by issuing `help xmos` at the XGDB interactive prompt.

listdevices

List and enumerate all XTAG adapters connected to your PC.

connect

Connects to a target. If no options are supplied, it will connect to the single XTAG adapter connected to your PC.

Valid options are:

- `--id <id>`

Connect to a specific XTAG adapter as enumerated by [listdevices](#).

- `--adapter-id <adapter-id>`

Connect to a specific XTAG adapter as identified by [listdevices](#). The advantage of using an `adapter-id` (rather than an enumerated `id`) is that it is independent of the number of XTAGs that might be connected.

- `--sim, -s`

Connect to a simulated target.

- `--jtag-speed <n>`

Set JTAG clock speed divider, where the frequency generated is $25/(n + 2)$ (MHz). The default frequency is 12.5 MHz.

- `--xtagreboot`

Reboots the XTAG before connecting. Can be used to recover an XTAG which is marked 'in use' (by another process).

Changed in version 15.0.4: Does not work on Windows hosts 15.0.4 onwards due to move to latest `winusb.sys` driver.

xcommand

Send a command to the Xmos Target Debug Interface. Available commands:

- `jtagchain`

Print JTAG chain topology, including 'idcode' for each element in the chain. Also prints module ids within each element.

load

Load the binary (and run any setup ELF's as necessary).

tile

Change the focus of subsequent commands to tile `<n>`.

5.1.5.5 Environment

XDBG_LOG_LEVEL

Setting this to a value, where 9 is the maximum, will cause XGDB to output debug information. For example:

```
$ XDBG_LOG_LEVEL=9 xgdb a.xe
```

5.1.6 XFLASH

5.1.6.1 Synopsis

```
xflash xe-file  
xflash [options]
```

5.1.6.2 Description

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

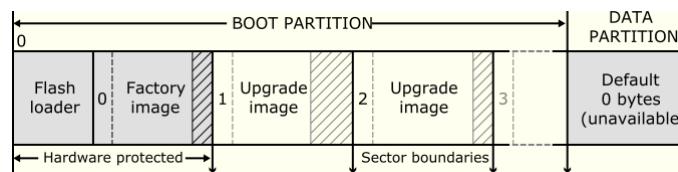


Fig. 5.1: Flash format diagram

5.1.6.3 Options

5.1.6.3.1 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.

--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.

--factory-version <version>

Specifies version as the tools release master version that was used to create the factory image. Accepted values are: 10, 11, 12, 13.0, 13.1, 13.2, 14.0, 14.1, 14.2, 14.3, 14.4, 15.0, 15.1 and 15.2. This option need only be specified when `--upgrade` is provided but `--factory` is not. This option will ensure that the produced flash upgrade image is of the correct format for the installed factory image.

--boot-partition-size <n>

Specifies the size of the boot partition to be n bytes. If left unspecified, the default size used is the total size of the flash device. n must be greater than or equal to the minimum size required to store the boot loader, factory image and any upgrade images. XFLASH will round up the actual boot partition size to the next sector boundary in flash memory.

--data [flash-name] <file>

Specifies the contents of file to be written to the data partition.

For a system with multiple flash boot sites, the data partition of the individual flash devices can be specified separately by repeatedly providing this option with flash-name set to the *Name attribute* of the flash device as defined in the XN file.

--loader <file>

Specifies custom flash loader functions in file, where file may be either an object (.o) or archive (.a).

By default, the XTC flash loader selects the image with the highest version number. A custom loader may choose to override the selected image. Only valid images passing CRC checks - and additional signature checks, under secure boot - are considered for selection. The custom loader is not expected to perform this validation and only informs the XTC flash loader which of the available valid images is preferred.

--idnum <32-bit-integer>

Specifies a numerical identifier which will be stored in the flash.

--idstr <a_string>

Specifies a string identifier which will be stored in the flash.

--analyze <file>

Prints the data structures and extracts the individual sections of the xCORE flash binary or flash dump in file.

If the factory image was built using an older tools release, the option `--factory-version` must be set accordingly. If the flash binary is encrypted, `--key` can be specified to enable decryption and signature verification of the sections.

The output code sections can be individually disassembled to aid debugging.

--force

Disables interactive prompts for user confirmation of any action. This can be used to bypass target warnings without user interaction.

--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.

5.1.6.3.2 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.

--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.

--noinq

Does not run the device inquirer program. The inquirer queries the device for flash memory density and sector size information. By default the inquirer runs when the user has not supplied the memory density in the XN file.

If `--noinq` is omitted XFLASH expects to be able to connect to the xCORE target via JTAG, in order to query the flash device.

--force-jtag

Will instruct the program to only communicate over JTAG, and not use the faster xSCOPE link. By default the xSCOPE link is used if declared in the XN file.

This option is recommended only when the xSCOPE link is physically disconnected.

--force-pll-reset

Will force an xCORE-200 (XS2) or `xcore.ai` (XS3) device to reset when the PLL register is written during the boot process. By default the device will not reset when the PLL register is written allowing for faster boot times.

This option is not recommended.

--jtag-speed <n>

Sets the divider for the JTAG clock to `n`. The corresponding JTAG clock speed is $6/(n+1)$ MHz. The default value of the divider for the JTAG clock is 0, representing 6MHz.

--spi-spec <file>

Enables support for the flash device specified in `file` (see [Add support for a new flash device](#)).

--spi-div <n>

Sets the divider for the SPI clock to `n`, producing an SPI clock speed of $100/2*n$ MHz. By default, if no target is specified, the divider value is set to 3 (16.7MHz).

--quad-spi-clock <arg>

Set the Quad SPI clock for the second stage loader. *arg* may be one of: 5MHz, 6.25MHz, 8.33MHz, 12.5MHz, 13.88MHz, 15.62MHz, 17.85MHz, 20.83MHz, 25MHz, 31.25MHz, 41.67MHz, 50MHz. By default, the clock is set to 15.62MHz.

--image-search-page

Will instruct the second stage bootloader to search flash memory for potential upgrade images at every page boundary within the boot partition. This was the default search mechanism in tools 14.0 and all previous tools versions.

--image-search-sector

Will instruct the second stage bootloader to search flash memory for potential upgrade images at every sector boundary within the boot partition. This is the new default search mechanism in tools 14.2.

--image-search-address <address>

Will instruct the second stage bootloader to search flash memory for potential upgrade images at a specified address within the boot partition. This option can be provided up to a maximum of three times, allowing for 3 separate upgrade images to be present within flash memory.

5.1.6.3.3 Security Options

The following options are used in conjunction with the [AES Module](#).

--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.

5.1.6.3.4 Programming Options

By default, XFLASH programs the generated binary file to the target flash device.

--outfile <file>, **-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](#) 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.

--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.

--spi-read-id <cmd>

Reads the SPI manufacturer's ID from the attached device. The `cmd` can be obtained from the SPI manufacturer's datasheet. If there is more than one device in a network then all IDs will be returned.

--spi-read-status <cmd>

Reads the flash status register from the attached device. The `cmd` can be obtained from the SPI manufacturer's datasheet. If there is more than one device in a network then all status registers will be returned.

--spi-command <cmd> [num-bytes-to-read] [bytes-to-write...]

Issues an arbitrary SPI command to the flash device at the lowest level. The command is sent to the device, followed by the `bytes-to-write`. `num-bytes-to-read` are then read from the device and printed to the host terminal.

If specifying `bytes-to-write`, one must specify `num-bytes-to-read`, which can be zero. This is to maintain the order of the parameters. This option can be provided repeatedly to issue multiple commands in a sequence.

Check the SPI manufacturer's datasheet for the commands supported by the device and see [SPI Command Option](#) for advanced usage and examples.

--no-reporting

Prevents the flash programmer from printing progress updates to the host terminal. This may result in a slight increase in programming performance, particularly over JTAG.

--no-verify-on-write

Prevents the flash programmer verifying pages after writing. By default, the flash programmer will read back each page and throw an error upon encountering a differing byte value. This assists diagnosing a faulty flash device or issues with the physical connection.

--no-reset-on-write

Prevents XFLASH from resetting the xCORE after programming the device.

5.1.6.3.5 SPI Command Option

The `--spi-command` option allows issuing raw command sequences to the flash device. Below are some examples of its usage and advanced features.

The examples in this section can be considered in relation to the [Adesto AT25FF321A](#) device though the commands are applicable to and supported on the vast majority of Quad SPI devices.

```
xflash --target-file *target*.xn --spi-command 0x06 --spi-command 0x05 1
```

This issues the Write Enable (WREN) command, followed by Read Status Register (RDSR).

The Write Enable Latch (WEL) bit will be set in the RDSR response, due to the use of WREN.

```
xflash --target-file *target*.xn --spi-command 0x44EB 258 0x00 0x01 0x00 0x00
```

For Quad SPI devices, data transfer in quad mode is supported.

This is enabled by using command ID bits 12..15 to specify the input data transfer mode (the address), and bits 8..11 for the output data transfer mode (the data).

This example reads the second page from the flash using the Fast Read Quad I/O command (0xEB). 258 bytes are read instead of 256 due to dummy cycles between the address and data transactions.

```
xflash --target-file *target*.xn --spi-command 0x06 --spi-command 0xC7 --spi-command 0x80000005 1 --spi-command 0x0B 257 0x00 0x00 0x00
```

When scripting a sequence of commands, it can be helpful to wait for a previous operation to complete.

To simplify scripting of sequences, the `--spi-command` option implements "special commands".

A special command requires that bit 31 is set. Bits 24..30 then encode the special command type:

0. Poll until bit clear

Repeatedly executes the command in bits 0..7, polling the response until the bit in position indicated in bits 8..15 is clear.

1. Poll until bit set

Repeatedly executes the command in bits 0..7, polling the response until the bit in position indicated in bits 8..15 is set.

2. Set QE bit (Quad SPI-only)

If command bit 0 is set, this command sets the Quad Enable bit.

If command bit 0 is clear, this command clears the Quad Enable bit.

This will make use of the SFDP Quad Enable Requirements (QER) field, unless overridden by the user in the XN file or SPI spec. Some flash devices do not have a QE bit, in which case this command will have no effect.

Note that a subsequent attempt to use XFLASH to write to the flash will set the QE bit.

3. Set factory image protection

If command bit 0 is set, this command enables factory image protection.

If command bit 0 is clear, this command disables factory image protection.

This requires a device that supports individual block locking, and a SPI spec file containing the protection commands to use. Some flash devices also require a status register bit to be set to enable the block locking - this can be configured using a standard `--spi-command` sequence. Refer to the SPI manufacturer's datasheet for more information.

Note that a subsequent attempt to use XFLASH to erase or write the flash will disable protection.

For special commands, bits 0..23 are arbitrary and their meaning depends on the special command type.

In the above example, type 0 is used with bit position 0. This waits for the busy bit to clear following the Chip Erase (0xC7) operation, guaranteeing that the following Fast Read (0x0B) returns an erased page.

```
xflash --target-file *target*.xn --spi-command 0x82000000 --spi-command 0x44EB 258 0x00 0x00 0x00 0x00 --spi-command 0x82000001 --spi-command 0x44EB 258 0x00 0x00 0x00 0x00
```

This example demonstrates the effect of the QE bit - a read in quad mode is attempted first with it cleared, and then with it set. The first read will fail to return the expected data from the flash, with the second providing the correct data. If the Quad SPI device has no QE bit, both reads will return the true flash data.

Note that XMOS Quad SPI boot requires the QE bit to be set.

Caution: The `--spi-command` option is powerful and it's possible to modify non-volatile flash state that may cause the device to be inoperable or incompatible with XMOS boot.

Be careful when using this option and refer to the SPI manufacturer's datasheet.

5.1.7 XBURN

5.1.7.1 Description

XBURN creates OTP images, and programs images into the OTP memory of xCORE devices.

5.1.7.2 Options

5.1.7.2.1 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*.

<otp-file>

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

--version

-v

Displays the version number and copyrights.

--help

-h

Prints a description of the supported command-line options.

5.1.7.2.2 Target Options

The following options are used to specify the target hardware platform.

--list-devices

-l

Prints an enumerated list of all JTAG adapters connected to the host and the devices on each JTAG chain, in the form:

ID - NAME (ADAPTER-SERIAL-NUMBER)

The adapters are ordered by their serial numbers.

--id <ID>

Specifies the adapter connected to the target hardware.

--adapter-id <ADAPTER-SERIAL-NUMBER>

Specifies the serial number of the adapter connected to the target hardware.

--jtag-speed <n>

Sets the divider for the JTAG clock to *n*. The corresponding JTAG clock speed is $6/(n+1)$ MHz. The default value is 0 (6MHz).

--spi-div <n>

Sets the divider used in the AES Module for the SPI clock to *n*. The corresponding SPI clock speed is set to $100/(2n)$ MHz. The default value is 20 (2.5MHz).

This option is only valid with --lock.

5.1.7.2.3 Security Options

`--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`.

`--lock *keyfile*`

Specifies the *XCORE AES boot module* and a *default set of security bits*.

5.1.7.2.4 Security Register Options

The following options are used to specify the contents of the OTP security register, overriding the default options for burning XE images, OTP images and the AES module, as given in *Default security bits written by XBURN*.

Table 5.2: Default security bits written by XBURN

Security Bit	XE Image	OTP Image	AES Module (<code>--lock</code>)
OTP Boot	Enabled	As per OTP image file	Enabled
JTAG Access	Enabled		Disabled
Plink Access	Enabled		Enabled
Global Debug	Enabled		Disabled
Master Lock	Disabled		Enabled
Secure Config Access	Enabled		Enabled

The following options support both the prefixes `--enable-...` and `--disable-...` to either enable or disable the feature.

`--enable-otp-boot / --disable-otp-boot`

Enables/disables boot from OTP.

`--enable-jtag / --disable-jtag`

Enables/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.

`--enable-plink-access / --disable-plink-access`

Applies to XS1 devices only. Enables/disables access to the plink registers from other tiles. Disabling plink access restricts all access of the registers of each plinks to the tile local to that plink.

`--enable-global-debug / --disable-global-debug`

Enables/disables the device from participating in global debug. Disabling global debug prevents the tiles from entering debug using the global debug pin.

`--enable-master-lock / --disable-master-lock`

Enables/disables the OTP master lock. No further modification of the OTP is permitted. Programming is disabled.

`--enable-secure-config-access / --disable-secure-config-access`

XCORE-200 (XS2A) and xcore.ai (XS3A) architectures only. Enables/disables access to the security config register.

5.1.7.2.5 Programming Options

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

--force
-f

Do not prompt before writing the OTP. This is not default.

--outfile <otp-file>
-o <otp-file>

Place output in <otp-file>, disabling programming.

--dump-otp-images

Dump the OTP images in separate binary files, disabling programming.

--make-exec <xe-file>

Place an executable in <xe-file> that when run on an xCORE device performs the specified OTP burning operation; disables programming.

The XE file can be run later using XRUN.

--target-file <xn-file>

Specifies <xn-file> as the target platform.

--target <platform>

Specifies a target platform. The platform configuration must be specified in the file *platform .xn*, which is searched for in the paths specified by the `XCC_DEVICE_PATH` environment variable.

--read

Prints the entire contents of the OTP.

--size-limit <n>

Limits the amounts of OTP memory written to the first *n* bytes of the OTP. If the image doesn't fit within the specified limit an error will be given.

5.1.8 XMAKE

5.1.8.1 Synopsis

```
xmake [options] [target] ...
```

5.1.8.2 Description

XMAKE is a special port of the [GNU Make tool](#). Its behavior and usage is therefore largely as per the [GNU Make Manual](#).

Warning: The XMAKE build tool is not recommended for new designs. This does not mean that the GNU Make tool is not recommended.

5.1.8.3 Options

`--help, -h`
Display all options.

5.2 File formats and data descriptions

This section collates and describes all the various data formats used or generated by the tools.

5.2.1 XMOS executable (XE) file format

The XMOS executable (XE) binary file format holds executable programs compiled to run on XMOS devices. The format supports distinct programs for each xCORE tile in a multi-tile or multi-chip design, and allows multiple loads and runs on each tile.

In addition to the program itself, an XE file contains a description of the system it is intended to run on. This description takes the form of either an XML system configuration description or a 64-bit per-node system identifier.

5.2.1.1 Binary format

The following sections explain the common elements of the binary format. All data is encoded as little endian.

5.2.1.1.1 XE header

An XE file must start with an XE header. It has the following format:

Table 5.3: XE header

Byte offset	Length (bytes)	Description
0x0	4	The string XMOS encoded in ASCII.
0x4	1	Major version number (2).
0x5	1	Minor version number (0).
0x6	2	Reserved. Must be set to zero.

5.2.1.1.2 Sectors

The XE header is followed by a list of sectors. The end of the sector list must be marked using a sector with a sector type of 0x5555. Each sector consists of a sector header, optionally followed by a variable-length sector contents block containing sector data. Padding is added after the sector data to make the sector contents block a whole number of 32-bit words.

Table 5.4: Sector header

Byte off-set	Length (bytes)	Description
0x0	2	Sector type.
0x2	2	Reserved. Must be set to zero.
0x4	8	Size in bytes of the sector contents block. Set to zero if this sector has no sector contents block.

Table 5.5: Sector contents block

Byte offset	Length (bytes)	Description
0x0	1	Size in bytes of the padding after the sector data.
0x1	3	Reserved. Must be set to zero.
0x4	n	Sector data.
0x4+n	p	Padding bytes to align to the next 32-bit word.
0x4+n+p	4	Sector CRC.

The sector CRC is calculated on the byte stream from the start of the sector header to the byte before the sector CRC. The polynomial used is 0x04C11DB7 (IEEE 802.3); the CRC register is initialized with 0xFFFFFFFF and residue is inverted to produce the CRC.

The following sector types are defined:

Table 5.6: Sector types

Value	Name	Description
0x1	Binary	Load binary image.
0x2	ELF	Load ELF image.
0x3	SysConfig	System description XML.
0x4	NodeDescriptor	Node description.
0x5	Goto	Start execution.
0x6	Call	Start execution and wait for return.
0x8	XN	XN description.
0x5555	Last sector	Marks the end of the file.
0xFFFF	Skip	Skip this sector.

The meaning of the sector data depends on the sector type. The following sections provide further details of the format of the sector data for each sector type.

5.2.1.1.2.1 SysConfig sector

The SysConfig sector contains a full XML description of the system, including number of nodes, xCORE tiles and link/interconnect configuration. This information is provided by XMOS to describe its chip products. The format of the SysConfig sector is currently undocumented.

5.2.1.1.2.2 Node descriptor sector

The NodeDescriptor sector describes an individual node, allowing the toolchain to validate an executable file matches the target device. There may be 0 or more NodeDescriptor sectors.

Table 5.7: NodeDescriptor sector

Data byte offset	Length (bytes)	Description
0x0	2	Index of the node in the JTAG scan chain.
0x2	2	Reserved.
0x4	4	Device JTAG ID.
0x8	4	Device JTAG user ID.

5.2.1.1.2.3 XN sector

The XN sector contains a *XN description* of the system.

5.2.1.1.2.4 Binary/ELF sectors

Binary or ELF sectors instruct the loader to load a program image on the specified xCORE tile. Binary/ELF sectors are formatted as shown in the following table:

Table 5.8: Binary/ELF sector

Data byte offset	Length (bytes)	Description
0x0	2	Index of the node in the JTAG scan chain.
0x2	2	xCORE tile number.
0x4	8	Load address of the binary image data. For ELF sectors this field should be set to 0.
0xC	n	Image data.

When a binary sector is loaded the data field is copied into memory starting at the specified load address. When a ELF sector is loaded the loadable segments of ELF image contained in the data field are loaded to the addresses specified in the ELF image.

5.2.1.1.2.5 Goto/call sectors

Goto and call sectors instruct the loader to execute code on the specified xCORE tile. If the last image loaded onto the tile was a ELF image execution starts at address of the `_start` symbol, otherwise execution starts at address specified as a field in the sector.

When processing a call sector the loader should wait for the code to indicate successful termination via a `done` or `exit` system call before processing the next sector.

Table 5.9: Goto/Call sector

Data byte offset	Length (bytes)	Description
0x0	2	Index of the node in the JTAG scan chain.
0x2	2	xCORE tile number.
0x4	8	Specifies the address to jump to if the last image loaded onto the tile was a binary image. This field should be set to 0 if the last image loaded was an ELF image.

5.2.1.1.2.6 Last sector

The last sector type is used to indicate the end of the sector list. A sector of this type should have no sector contents block.

5.2.1.1.2.7 Skip sector

A loader must ignore any skip sectors that appear in the sector list. Changing the type of an existing sector to the skip sector type allows removal of sectors without effecting the layout of the XE file.

5.2.1.2 Booting an XE File

To boot an XE file the sectors within the file must be processed in sequential order. This allows a loader to load and execute sectors to initialize the system in an order defined by the toolchain, using as many boot stages as required. If an image is loaded onto an xCORE tile there must be exactly one Goto sector. This sector must appear after all Call, Binary and ELF sectors for that tile.

A loader may choose to delay processing of Call sectors until a set of Call sectors have been accumulated for all xCORE tiles on the target device. This allows the loader to reduce boot time by executing as much code as possible in parallel.

The example in [Example XE file](#) shows a typical layout for an XE file containing a program compiled to run on a 4-tile XS1-G4 device.

Table 5.10: Example XE file

Sector type	Node	Tile	Description
SysConfig			XML System description, ignored by the loader.
XN			XN description, ignored by the loader.
ELF	0	3	Load ELF image onto node 0 tile 3.
Call	0	3	Execute program on node 0 tile 3 and wait for successful termination.
ELF	0	2	Load ELF image onto node 0 tile 2.
Call	0	2	Execute program on node 0 tile 2 and wait for successful termination.
ELF	0	1	Load ELF image onto node 0 tile 1.
Call	0	1	Execute program on node 0 tile 1 and wait for successful termination.
ELF	0	0	Load ELF image onto node 0 tile 0.
Call	0	0	Execute program on node 0 tile 0 and wait for successful termination.
ELF	0	3	Load ELF image onto node 0 tile 3.
Goto	0	3	Execute program on node 0 tile 3.
ELF	0	2	Load ELF image onto node 0 tile 2.
Goto	0	2	Execute program on node 0 tile 2.
ELF	0	1	Load ELF image onto node 0 tile 1.
Goto	0	1	Execute program on node 0 tile 1.
ELF	0	0	Load ELF image onto node 0 tile 0.
Goto	0	0	Execute program on node 0 tile 0.
Last sector			Last sector marker.

A further example is given in the tutorial [Understanding XE files and how they are loaded](#).

5.2.2 XN Specification

5.2.2.1 Network Elements

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 below:

Node	**Number**	**Description**
Network	1	An xCORE network
Declarations	0+	
Declaration	1+	xCORE Tile declaration
Packages	1+	
Package	1+	Device package
Nodes	1	
Node	1+	Node declaration
Tile	1+	An xCORE Tile
Port	0+	An xCORE symbolic port name
Boot	0 or 1	Boot method
Source	1	Binary location
Bootee	0+	Nodes booted
Extmem	0 or 1	External memory configuration
Lpddr	1	Pad control for LPDDR device outputs
Padctrl	1	Pad control for xcore.ai outputs
RoutingTable	0 or 1	

(continues on next page)

(continued from previous page)

Bits	1	
Bit	1+	Direction for bit
Links	1	
Link	1+	Direction for link
Service	0+	Service declaration
Chanend	1+	Chanend parameter
Links	0 or 1	
Link	1+	xCONNECT Link declaration
LinkEndpoint	2	xCONNECT Link endpoint
ExternalDevices	0 or 1	
Device	1+	External device
Attribute	0+	A device attribute
JTAGChain	0 or 1	
JTAGDevice	1+	A device in the JTAG chain

5.2.2.2 Declaration

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

```
tileref identifier
```

```
tileref identifier [ constant-expression ]
```

An equivalent declaration is exported to the header file `<platform.h>` for use in XC programs. A `tileref` declaration is associated with physical xCORE tiles by the reference attribute of a `Tile` element.

Example

```
<Declaration>tileref master</Declaration>  
<Declaration>tileref tile[8]</Declaration>
```

5.2.2.3 Package

A Package element refers to a package file that describes the mapping from xCORE ports and links to the pins on the package.

Table 5.11: XN Package element

At-tribute	Re-quired	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> .

Example

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

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

5.2.2.4 Node

A Node element defines a set of xCORE Tiles in a network, all of which are connected to a single switch. The XMOS xcore.ai device XU316-1024-FB265 is an example of a node.

Table 5.12: XN Node element

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> .
ReferenceRoutingId	Yes	String	Associates the node with a xCORE Tile identifier specified in a Declaration. This attribute is only valid on nodes with type <code>periph:XS1-SU</code> .
InPackageId	No	Integer	The routing identifier on the xCONNECT Link network.
Oscillator	Yes	String	Maps the node to an element in the package file.
OscillatorSrc	No	String	The PLL oscillator input frequency, specified as a number followed by either MHz, KHz or Hz.
SystemFrequency	No	String	The name of the node which supplies the PLL oscillator input.
PIIFeedbackDivMin	No	String	The system frequency, specified as a number followed by either MHz, KHz or Hz. Defaults to 400MHz if not set.
ReferenceFrequency	No	Integer	The minimum allowable PLL feedback divider. Defaults to 1 if not set.
PII-Divider-StageOneReg	No	String	A reference clock frequency, specified as a number followed by either MHz, KHz or Hz. Defaults to 100MHz if not set.
PII-Multiplier-StageReg	No	Integer	The PLL divider stage 1 register value.
PII-Divider-StageTwoReg	No	Integer	The PLL multiplier stage register value.
Secondary-PII-Input-Div	No	Integer	The PLL divider stage 2 register value.
Secondary-PII-Output-Div	No	Integer	The secondary PLL input divider register value
Secondary-PII-Feedback-Div	No	Integer	The secondary PLL output divider register value
RefDiv	No	Integer	The secondary PLL feedback divider register value
	No	Integer	$\text{SystemFrequency} / \text{RefDiv} = \text{ReferenceFrequency}$

The PLL registers can be configured automatically using the attributes `SystemFrequency`, `PIIFeedbackDivMin`

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 xCORE frequency control documents [XS1 L Clock Frequency Control](#).

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 PLLs. The PLL registers remain at their initial values as determined by the mode pins.

The secondary PLL can only be configured manually by supplying SecondaryPllInputDiv, SecondaryPllOutputDiv and SecondaryPllFeedbackDiv.

Example

```
<Node Id="0" InPackageId="0" Type="XS3-L16A-1024" Oscillator="24MHz" SystemFrequency="600MHz" ReferenceFrequency="100MHz">
```

5.2.2.4.1 Tile

A Tile element describes the properties of a single xCORE Tile.

Table 5.13: XN Tile element

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

Example

```
<Tile Number="0" Reference="tile[0]">
```

5.2.2.4.2 Port

A Port element provides a symbolic name for a port.

Table 5.14: XN Port element

Attribute	Required	Type	Description
Location	Yes	String	A port identifier defined in the standard header file <code><xs1.h></code> . The ports are described in the <i>XC Programming Guide</i> https://www.xmos.com/developer/doc/XM-000223-UG .
Name	Yes	String	A valid C preprocessor identifier. All port names declared in the network must be unique.

Example

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

5.2.2.4.3 Boot

A Boot element defines the how the node is booted. It contains one *Source* element and zero or more *Bootee* elements that are booted over xCONNECT Links. If the source specifies an xCONNECT Link, no Bootee elements may be specified. In a line of XS1-L devices, bootees must be contiguous to the device booting from SPI.

Tip: The XMOS tools require a Boot element to be able to *boot programs from flash memory*.

5.2.2.4.4 Source

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

Table 5.15: XN Source element

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

Important: XMOS XS1-G devices cannot be configured to boot over xCONNECT Links.

Example

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

5.2.2.4.5 Bootee

A Bootee element specifies another node in the system that this node boots via an xCONNECT Link. If more than one xCONNECT Link is configured between this node and one of its bootees (see [Link](#) and [LinkEndpoint](#)), the tools pick one to use for booting.

Table 5.16: XN Bootee element

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

Example

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

5.2.2.4.6 Bit

A Bit element specifies the direction for messages whose first mismatching bit matches the specified bit number.

Table 5.17: XN Bit element

Attribute	Required	Type	Description
number	Yes	Integer	The bit number, numbered from the least significant bit.
direction	Yes	Integer	The direction to route messages.

Example

```
<Bit number="1" direction="0"/>
```

5.2.2.4.7 Link

When it appears within a RoutingTable element, a Link element specifies the direction of an xCONNECT Link.

Table 5.18: XN Link element

Attribute	Required	Type	Description
name	Yes	String	A link identifier in the form X<n>L<m> where <n> denotes a tile number and <m> the link letter. See the corresponding package datasheet for available link pinouts.
direction	Yes	Integer	The direction of the link.

Example

```
<Link number="XLA" direction="2"/>
```


5.2.2.4.8 Service

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

Table 5.19: XN Service element

At-tribute	Re-quired	Type	Description
Proto	Yes	String	The prototype for the service function, excluding the <code>service</code> keyword. This prototype is exported to the header file <code><platform.h></code> for use in XC programs.

Example

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

XSCOPE Example

The text below is required to support XSCOPE and must reside under a `<Network>` element (at the same level as `<Links>`):

```
<Nodes>
  <Node Id="2" Type="device:" RoutingId="0x8000">
    <Service Id="0" Proto="xscope_host_data(chanend c);">
      <Chanend Identifier="c" end="3"/>
    </Service>
  </Node>
</Nodes>
```

5.2.2.4.9 Chanend

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

Table 5.20: XN Service element

At-tribute	Re-quired	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"/>
```

5.2.2.5 Link

xCONNECT Links are described in the system specification documents ([XS1 L System Specification](#)) and link performance documents ([XS1 L Link Performance/Design Guidelines](#)).

A Link element describes the characteristics of an xCONNECT Link. It must contain exactly two `LinkEndpoint` children.

Table 5.21: XN Link element

Attribute	Required	Type	Description
Encoding	Yes	String	Must be either <code>2wire</code> or <code>5wire</code> .
Delays	Yes	String	Of the form <code><x>clk, <y>clk</code> where <code><x></code> specifies the inter-token delay value for the endpoint, and <code><y></code> specifies the intra-token delay value for the endpoint. <code><x></code> and <code><y></code> are specified as a number of <code>SystemFrequency</code> cycles. If <code><y>clk</code> is omitted, the <code><x>clk</code> value is used for the intra-token delay value.
Flags	No	String	Specifies additional properties of the link. Use the value <code>xSCOPE</code> to specify a link used to send xSCOPE trace information.

Example

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

5.2.2.5.1 LinkEndpoint

A `LinkEndpoint` describes one end of an xCONNECT Link, the details of which can be found in the system specification documents ([XS1 L System Specification](#)). Each endpoint associates a node identifier to a physical xCONNECT Link.

Table 5.22: XN LinkEndpoint element

Attribute	Required	Type	Description
NodeID	No	String	A valid node identifier.
Link	No	String	A link identifier in the form <code>X<n>L<m></code> where <code><n></code> denotes a tile number and <code><m></code> the link letter. See the corresponding package datasheet for available link pinouts.
RoutingId	No	Integer	The routing identifier on the xCONNECT Link network.
ChannelEnd	No	Integer	A channel end.
BootRomEnabled	No	Integer	Link is enabled at boot and can therefore be considered for use within the spanning network.

An endpoint is usually described as a combination of a node identifier and link identifier. For a streaming debug link, one of the endpoints must be described as a combination of a routing identifier and a channel end. For example:

```
<LinkEndpoint NodeId="0" Link="X0LD" />
<LinkEndpoint RoutingId="0x8000" Chanend="1">
```

The table below highlights the correct link name to use for the Link attribute within the LinkEndpoint element.

xConnect link Number	xCORE "L" series link name	xCORE-200 link name
0	XLC	XL0
1	XLD	XL1
2	XLA	XL2
3	XLB	XL3
4	XLG	XL4
5	XLH	XL5
6	XLE	XL6
7	XLF	XL7
8	N/A	XL8

An xCORE "L" series link name cannot be used within an xCORE-200 XN specification.

An xCORE-200 link name cannot be used within an xCORE "L" series XN specification.

The following example demonstrates how Node 0 Link number 4 is connected to Node 1 Link number 7 in an xCORE "L" series XN specification:

```
<Links>
  <Link Encoding="5wire" Delays="0,1">
    <LinkEndpoint NodeId="0" Link="XLG" />
    <LinkEndpoint NodeId="1" Link="XLF" />
  </Link>
</Links>
```

The following example demonstrates how Node 0 Link number 4 is connected to Node 1 Link number 7 in an xCORE-200 series XN specification:

```
<Links>
  <Link Encoding="5wire" Delays="4,4">
    <LinkEndpoint NodeId="0" Link="XL4" />
    <LinkEndpoint NodeId="1" Link="XL7" />
  </Link>
</Links>
```

5.2.2.6 Device

A Device element describes a device attached to an xCORE Tile that is not connected directly to an xCONNECT Link.

Table 5.23: XN Device element

Attribute	Required	Type	Description
Name	Yes	String	An identifier that names the device
NodeId	Yes	String	The identifier for the node that the device is connected to
Tile	Yes	Integer	The tile in the node that the device is connected to
Class	Yes	String	The class of the device
Type	No	String	The type of the device (class dependent)
PageSize	Yes	Integer	The program page size of the device
SectorSize	Yes	Integer	The erase sector size of the device
NumPages	Yes	Integer	The number of pages in the device



The following attribute values for the attribute name are recognised: class:

SPIFlash

Device is SPI flash memory

SQIFlash

Device is QuadSPI flash memory

Use the Type attribute to identify the model of the flash device.

5.2.2.6.1 Attribute

An Attribute element describes one aspect of a *Device*.

Table 5.24: 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 following attribute names for the device are supported: 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" />
```

The following attribute names for the device are supported: class SQIFlash:

PORT_SQI_CS

QuadSPI Chip Select signal.

PORT_SQI_SCLK

QuadSPI Clock signal.

PORT_SQI_SIO

QuadSPI In/Out signal.

QE_REGISTER

This is optional and only required for devices which do not support JEDEC SFDP. Valid values are flash_qe_location_status_reg_0 and flash_qe_location_status_reg_1.

QE_BIT

This is optional and only required for devices which do not support JEDEC SFDP. Valid values are flash_qe_bit_0 through to flash_qe_bit_7.

Example

```
<Attribute Name="PORT_SQI_SIO" Value="PORT_SQI_SIO" />
```

Flash device example

The following example shows the complete description of a flash device.



```

<ExternalDevices>
  <Device NodeId="0" Tile="0" Class="SQIFlash" Name="bootFlash" Type="S25FL116K" PageSize="256"
  ↳SectorSize="4096" NumPages="16384">
    <Attribute Name="PORT_SQI_CS" Value="PORT_SQI_CS" />
    <Attribute Name="PORT_SQI_SCLK" Value="PORT_SQI_SCLK" />
    <Attribute Name="PORT_SQI_SIO" Value="PORT_SQI_SIO" />
    <Attribute Name="QE_REGISTER" Value="flash_qe_location_status_reg_0" />
    <Attribute Name="QE_BIT" Value="flash_qe_bit_6" />
  </Device>
</ExternalDevices>

```

5.2.2.7 JTAGChain

The JTAGChain element describes a device in the JTAG chain. The order of these elements defines their order in the JTAG chain.

Table 5.25: XN JTAGChain element

Attribute	Required	Type	Description
JTAGSpeed	No	String	Sets the JTAG clock speed.

5.2.2.7.1 JTAGDevice

Table 5.26: 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">

```

5.2.3 XSIM Trace output

XSIM trace output is produced by using `xsim -t`, `xsim --trace` or `xsim --trace-to`. The format provides an insight into the internal operations of the xcore which is not possible to see by using the hardware itself.

Here's a random example of the output, taken from a two-tile processor:

```

tile[0]@0- -SI A-.----0004012c (write_switch_reg_send+ 8) : out    res[r4(0x80020102)], r2(0x18)
↳@3261
tile[1]@0- -SI A-.----0004003a (_done                + 6) : bu    -0x1 @3261
tile[0]@0- -SI A-.----0004012e (write_switch_reg_send+ a) : outct  res[r4(0x80020102)], 0x1 @3266
tile[1]@0- -SI A-.----0004003a (_done                + 6) : bu    -0x1 @3266
tile[0]@0- -SI A-.----00040130 (write_switch_reg_send+ c) : bf    r11(0x1), 0x9 @3271
tile[1]@0- -SI A-.----0004003a (_done                + 6) : bu    -0x1 @3271
tile[0]@0-P-SI A-.----00040132 (write_switch_reg_send+ e) : inct  r0(0x1), res[r4(0x80020102)]
↳@3276
tile[1]@0- -SI A-.----0004003a (_done                + 6) : bu    -0x1 @3276

```

(continues on next page)

(continued from previous page)

```

tile[1]@0- -SI A-.----0004003a (_done          + 6) : bu      -0x1 @3281
tile[1]@0- -SI A-.----0004003a (_done          + 6) : bu      -0x1 @3286
tile[1]@0- -SI A-.----0004003a (_done          + 6) : bu      -0x1 @3291
tile[0]@0- -SI A-.----00040132 (write_switch_reg_send+ e) : inct  r0(0x3), res[r4(0x80020102)]
→@3293
tile[1]@0- -SI A-.----0004003a (_done          + 6) : bu      -0x1 @3296
tile[0]@0- -SI A-.----00040134 (write_switch_reg_send+ 10) : eq     r0(0x1), r0(0x3), 0x3 @3298
tile[1]@0- -SI A-.----0004003a (_done          + 6) : bu      -0x1 @3301
tile[0]@0- -SI A-.----00040136 (write_switch_reg_send+ 12) : chkct  res[r4(0x80020102)], 0x1 @3303
tile[1]@0- -SI A-.----0004003a (_done          + 6) : bu      -0x1 @3306
tile[0]@0- -SI A-.----00040138 (write_switch_reg_send+ 14) : freer  res[r4(0x80020102)] @3308
tile[1]@0- -SI A-.----0004003a (_done          + 6) : bu      -0x1 @3311
tile[0]@0- -SI A-.----0004013a (write_switch_reg_send+ 16) : ldw    r4(0x0), sp[0x0] L[0x42000]
→@3313
tile[1]@0- -SI A-.----0004003a (_done          + 6) : bu      -0x1 @3316
tile[0]@0- -SI A-.----0004013c (write_switch_reg_send+ 18) : retsp  0x0 L[0x42000] @3318

```

Each row represents the execution of a single instruction. Here's a quick overview of what it means. From left to right:

- tile[0]: this is the tile executing the instruction
- 0004012c: program counter address
- (write_switch_reg_send+ 8): the symbol and offset from it
- out res[r4(0x80020102)], r2(0x18): instruction
- @3261: processor cycle (time, basically)

If you look closer, you can also see the contents of each register and, towards the bottom of the sample, memory access too (L[0x42000]).

Further detail is contained in the table below:

Table 5.27: Trace output for XS1 processors

Tile	Core State			Address							Instruction	Mem	Cycle	
Name from XN	I0	I1	I2	S0,S1(T0) .. S0,S1(Tn)	.	M	S	K	N	PC	(sym+offset)	operands	addr	@val
	- D	* P	- d	- a A	n	- m	- s	- k	- n			val	L[adr]	
				i I p	sta-							rn(val)	S[adr]	
				m s w	tus							res[id]		
					pairs									

- I0: - No debug interrupt
- I0: D Instruction caused debug interrupt
- I1: * Instruction excepted
- I1: P Instruction paused
- I2: - Not in debug mode
- I2: d Tile in debug mode
- S0: - Core not in use
- S0: a Core active
- S0: A Core active (the instruction being traced belongs to this core)
- S0: i Core active with ININT bit set
- S0: I Core active with ININT bit set (belongs to this core)
- S0: p Core paused due to instruction fetch



S0: m Core paused with MSYNC bit set
S0: s Core paused withSSYNC bit set
S0: w Core paused with WAITING bit set
S1: - Interrupts and events disabled
S1: b Interrupts and events enabled
S1: i Interrupts enabled and events disabled
S1: e Interrupts disabled and events enabled
M: - MSYNC not set
M: m MSYNC set
S: - SSYNC not set
S: s SSYNC set
K: - INK not set
K: k INK set
N: - INENB not set
N: n INENB set
rn(va1) Value of register n
res[id] Resource identifier
L/S[adr] Load from/Store to address

5.2.4 XSCOPE config file

An XSCOPE config file is an XML file, usually with a suffix of .xscope. Here is an example:

```
<xSCOPEconfig ioMode="basic" enabled="true">
  <Probe name="Tile0-result" type="CONTINUOUS" datatype="UINT" units="mV" enabled="true" />
  <Probe name="Tile1-i" type="CONTINUOUS" datatype="UINT" units="mV" enabled="true" />
  <Probe name="Tile1-accumulation" type="CONTINUOUS" datatype="UINT" units="mV" enabled="true" />
</xSCOPEconfig>
```

The xscopeconfig element is required. Attributes are:

iomode

May be set to none, basic or timed

enabled

May be set to true or false

The probe elements are optional, depending on the desired number of named XSCOPE "probes". Attributes are:

name

Set to a string representing the name of the probe

type

May be set to STARTSTOP, CONTINUOUS, DISCRETE or STATEMACHINE

datatype

May be set to NONE, UINT, INT or FLOAT

units

Set to a string representing the units of measurement for the probe

enabled

May be set to true or false

5.3 xSCOPE performance figures

Data transferred from the xCORE device to the XTAG can be set to lossless, where the target application may be stalled, or to lossy where the target application is not stalled but data packets may be dropped.

Data packets transferred from the XTAG to the host computer may be dropped if the host is unable to keep pace with the transmit rate of the target application.

5.3.1 Transfer rates between the xCORE Tile and XTAG-3 or XTAG-4

The recommended xCONNECT target-to-XTAG link inter- and intra-token delay for most target hardware is approximately 10ns between transitions.

For a tile frequency of 500MHz this can be achieved by setting the link inter- and intra-token delay to 5 cycles (see [Link](#)). The latencies and maximum call rates for the probe functions using an xCONNECT Link at this speed are given in [xSCOPE performance figures for xCONNECT Link with 5-cycle intra-token delay](#).

Table 5.28: xSCOPE performance figures for xCONNECT Link with 5-cycle intra-token delay

Probe function	Latency (core cycles)	Max calls/sec
xscope_probe_data_pred	15 (always)	666,000
xscope_probe	20 (with no contention)	999,000
xscope_probe_cpu	27 (with no contention)	666,000
xscope_probe_data	22 (with no contention)	666,000
xscope_probe_cpu_data	28 (with no contention)	555,000

If two subsequent calls are made, the second call may be delayed in line with the maximum frequency. For example, if `xscope_probe_data_pred` is called twice, the second call is delayed by approximately 1.5 us.

The maximum call rates can be increased by reducing the inter- and intra-token delay (see [Link](#)). A small delay requires careful layout of the link and choice of cabling, since it increases link frequency.

5.3.2 Transfer rates between the XTAG-3 or XTAG-4 and Host PC

The host PC has a limit at which it can receive trace data from the XTAG. If the PC is unable to keep up with the rate at which the target is transmitting, it will drop trace data records.

5.4 Libraries

The XTC tools ship with a comprehensive suite of libraries that provide access to the underlying hardware at a tile and processor register level. Additional libraries to access attached flash memory devices are also included to provide a consistent interface to a range of different devices.

The library contents are summarised below and details for the individual API calls can be found in the HTML version of the XTC documentation.

This searchable list can be accessed online at <https://www.xmos.com/view/Tools-15-Documentation> and this information is also contained in the XTC release package and can be found at:

```
$XMOS_DOC_PATH/doc/html/index.html
```


5.4.1 lib_xcore

lib_xcore is a system library that provides a C API for the underlying hardware features of an xcore tile. A header file is provided for each functional area, and can be included with a line such as:

```
#include <xcore/port.h>
```

By default, lib_xcore is automatically added to the list of libraries for linking, so there is no need to use `xcc -l`.

5.4.2 lib_xs1

lib_xs1 is a system library that provides low level tools for accessing the XCORE hardware. Primarily, this is useful for accessing processor and system registers.

For other types of hardware access, such as channels and locks, see [lib_xcore](#).

lib_xs1 is automatically linked, so to use it, just include the header:

```
//Note: <xs1.h> is still the correct header, even if using an xs2 or xs3 architecture  
#include <xs1.h>
```

5.4.3 libflash API

The libflash library provides functions for reading and writing data to SPI flash devices that use the xCORE format shown in the diagram below.

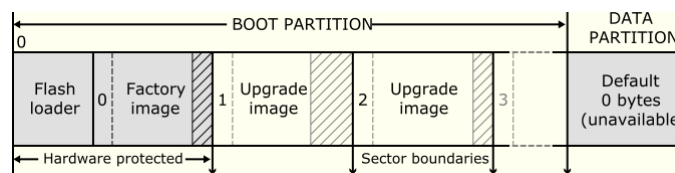


Fig. 5.2: Flash format diagram

All functions are prototyped in the header file `<flash.h>`. Except where otherwise stated, functions return 0 on success and non-zero on failure.

5.4.3.1 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 `f1_connect` and `f1_connectToDevice` require an argument of type `f1_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;  
} f1_SPIPorts;
```

int **f1_connect**(fl_SPIPorts *SPI)

f1_connect opens a connection to the specified SPI device.

int **f1_connectToDevice**(fl_SPIPorts *SPI, fl_DeviceSpec spec[], unsigned n)

f1_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.

unsigned **f1_getJedecId**(void)

f1_getJedecId returns the response to the 'Read Identification' command used during connection, typically the JEDEC ID. This uses the command and size defined by the SPI specification in use.

For the majority of flash devices, this will be the response to command 0x9f.

int **f1_getFlashType**(void)

f1_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 f1_connectToDevice(*p*, *spec*, *n*) is used to connect to a flash device, f1_getFlashType returns the parameter value *spec*[*i*].flashId where *i* is the index of the connected device.

unsigned **f1_getFlashSize**(void)

f1_getFlashSize returns the capacity of the SPI device in bytes.

void **f1_copySpec**(fl_DeviceSpec *dest)

f1_copySpec exports the completed SPI specification in use for the current connection.

unsigned **f1_getLibraryStatus**(void)

f1_getLibraryStatus returns a bitmask of errors and warnings from connection. The bits are defined as follows:

```
typedef enum {
    LIBRARY_ERROR_PAGESIZE_MISSING = 1 << 8,
    LIBRARY_ERROR_NUMPAGES_MISSING = 1 << 9,
    LIBRARY_ERROR_SECTORSIZE_MISSING = 1 << 10,
} fl_LibraryStatus;
```

These generally indicate the reason for connection failure, typically due to incomplete or incorrect SPI specification.

int **f1_command**(unsigned int cmd, unsigned char input[], unsigned int num_in, unsigned char output[], unsigned int num_out)

f1_command issues a command to the SPI device at the lowest level. The *cmd* is sent first, followed by *num_in* bytes from *input*. *num_out* bytes are then read to *output*.

int **f1_disconnect**(void)

f1_disconnect closes the connection to the SPI device.

5.4.3.2 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_getImageInfo**(fl_BootImageInfo *bootImageInfo, const unsigned char page[])

fl_getImageInfo provides information about a boot image header stored in memory. A single page of the image data must be provided.

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_getImageTag**(fl_BootImageInfo *bootImageInfo)

fl_getImageTag returns the magic number of the specified image.

unsigned **fl_getImageVersion**(fl_BootImageInfo *bootImageInfo)

fl_getImageVersion returns the version number of the specified image.

unsigned **fl_getImageAddress**(fl_BootImageInfo *bootImageInfo)

fl_getImageAddress returns the start address in flash of the specified image.

unsigned **fl_getImageSize**(fl_BootImageInfo *bootImageInfo)

fl_getImageSize returns the size of the specified image.

int **fl_getToolsMajor**(fl_BootImageInfo *bootImageInfo)

fl_getToolsMajor returns the tools major version used to build the specified image, or -1 on failure.

int **fl_getToolsMinor**(fl_BootImageInfo *bootImageInfo)

fl_getToolsMinor returns the tools minor version used to build the specified image, or -1 on failure.

int **fl_getToolsPatch**(fl_BootImageInfo *bootImageInfo)

fl_getToolsPatch returns the tools patch version used to build the specified image, or -1 on failure.

int **fl_getImageFormat**(fl_BootImageInfo *bootImageInfo)

fl_getImageFormat returns the compatibility version of the specified image, or -1 on failure.

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 **f1_writeImagePage**(const unsigned char page[])

f1_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 f1_startImageReplace, f1_startImageAdd or f1_startImageAddAt is invalid.

int **f1_writeImageEnd**(void)

f1_writeImageEnd waits until the SPI device has written the last page of data to its memory.

int **f1_startImageRead**(fl_BootImageInfo *b)

f1_startImageRead prepares the SPI device for reading the contents of the specified upgrade image.

int **f1_readImagePage**(unsigned char page[])

f1_readImagePage inputs the next page of data from the SPI device and writes it to the array page.

int **f1_deleteImage**(fl_BootImageInfo *b)

f1_deleteImage erases the upgrade image with the specified image.

5.4.3.3 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 **f1_getDataPartitionSize**(void)

f1_getDataPartitionSize returns the size of the data partition in bytes.

int **f1_readData**(unsigned offset, unsigned size, unsigned char dst[])

f1_readData reads a number of bytes from an offset into the data partition and writes them to the array dst.

unsigned **f1_getWriteScratchSize**(unsigned offset, unsigned size)

f1_getWriteScratchSize returns the buffer size needed by f1_writeData for the given parameters.

int **f1_writeData**(unsigned offset, unsigned size, const unsigned char src[], unsigned char buffer[])

f1_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.

5.4.3.3.1 Page-Level Functions

unsigned **f1_getPageSize**(void)

f1_getPageSize returns the page size in bytes.

unsigned **f1_getNumDataPages**(void)

f1_getNumDataPages returns the number of pages in the data partition.

unsigned **f1_writeDataPage**(unsigned n, const unsigned char data[])

f1_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 **f1_readDataPage**(unsigned n, unsigned char data[])

f1_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.

5.4.3.3.2 Sector-Level Functions

unsigned **f1_getNumDataSectors**(void)

f1_getNumDataSectors returns the number of sectors in the data partition.

unsigned **f1_getDataSectorSize**(unsigned n)

f1_getDataSectorSize returns the size of the *n*-th sector in the data partition in bytes.

unsigned **f1_eraseDataSector**(unsigned n)

f1_eraseDataSector erases the *n*-th sector in the data partition.

unsigned **f1_eraseAllDataSectors**(void)

f1_eraseAllDataSectors erases all sectors in the data partition.

5.4.4 libquadflash API

The libquadflash library provides functions for reading and writing data to Quad-SPI flash devices that use the xCORE format shown in the diagram below.

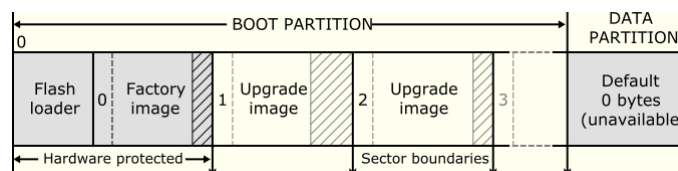


Fig. 5.3: Flash format diagram

All functions are prototyped in the header file `<quadflash.h>`. Except where otherwise stated, functions return 0 on success and non-zero on failure.

5.4.4.1 General Operations

The program must explicitly open a connection to the Quad-SPI device before attempting to use it, and must disconnect once finished accessing the device.

The functions `f1_connect` and `f1_connectToDevice` require an argument of type `f1_QSPIPorts`, which defines the four ports and clock block used to connect to the device.

```
typedef struct {
    out port qspiCS;
    out port qspiSCLK;
    out buffered port:32 qspiSI0;
    clock qspiClkblk;
} f1_QSPIPorts;
```

int **f1_connect**(f1_QSPIPorts *SPI)

f1_connect opens a connection to the specified Quad-SPI device.

int **f1_connectToDevice**(f1_QSPIPorts *SPI, f1_QuadDeviceSpec spec[], unsigned n)

f1_connectToDevice opens a connection to an Quad-SPI device. It iterates through an array of *n* Quad-SPI device specifications, attempting to connect using each specification until one succeeds.

unsigned **f1_getJedecId**(void)

f1_getJedecId returns the response to the 'Read Identification' command used during connection, typically the JEDEC ID. This uses the command and size defined by the SPI specification in use.

For the majority of flash devices, this will be the response to command `0x9f`.

int **f1_getFlashType**(void)

f1_getFlashType returns an enum value for the flash device. The enumeration of devices known to libquadflash is given below.

```
typedef enum {
    UNKNOWN = 0,
    ISSI_IS25LQ016B,
    ISSI_IS25LQ032B,
    ISSI_IS25LQ080B,
    SPANSION_S25FL116K,
    SPANSION_S25FL132K,
    SPANSION_S25FL164K,
} f1_QuadFlashId;
```

If the function call f1_connectToDevice(p, spec, n) is used to connect to a flash device, f1_getFlashType returns the parameter value spec[i].flashId where i is the index of the connected device.

unsigned **f1_getFlashSize**(void)

f1_getFlashSize returns the capacity of the Quad-SPI device in bytes.

void **f1_copySpec**(f1_DeviceSpec *dest)

f1_copySpec exports the completed SPI specification in use for the current connection. Where applicable, this may have been amended at runtime using the SFDP response.

unsigned **f1_getLibraryStatus**(void)

f1_getLibraryStatus returns a bitmask of errors and warnings from connection. The bits are defined as follows:

```
typedef enum {
    LIBRARY_ERROR_PAGESIZE_MISSING = 1 << 8,
    LIBRARY_ERROR_NUMPAGES_MISSING = 1 << 9,
    LIBRARY_ERROR_SECTORSIZE_MISSING = 1 << 10,
    LIBRARY_WARNING_PAGESIZE_MISMATCH = 1 << 16,
    LIBRARY_WARNING_NUMPAGES_MISMATCH = 1 << 17,
    LIBRARY_WARNING_QUADENABLE_MISMATCH = 1 << 18,
} f1_LibraryStatus;
```

These generally indicate the reason for connection failure, typically due to incomplete or incorrect SPI specification. Mismatch warnings indicate that a configuration parameter has been overridden in the device specification but does not match the SFDP response.

int **f1_command**(unsigned int cmd, unsigned char input[], unsigned int num_in, unsigned char output[], unsigned int num_out)

f1_command issues a command to the SPI device at the lowest level. The cmd is sent first, followed by num_in bytes from input. num_out bytes are then read to output.

Bits 12..15 of cmd set the number of data lines used for input. Bits 8..11 of cmd set the number of data lines used for output.

The supported values for the above fields are 1 and 4. If unspecified, they default to 1.

int **f1_disconnect**(void)

f1_disconnect closes the connection to the Quad-SPI device.

5.4.4.2 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_getImageInfo**(fl_BootImageInfo *bootImageInfo, const unsigned char page[])

fl_getImageInfo provides information about a boot image header stored in memory. A single page of the image data must be provided.

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_getImageTag**(fl_BootImageInfo *bootImageInfo)

fl_getImageTag returns the magic number of the specified image.

unsigned **fl_getImageVersion**(fl_BootImageInfo *bootImageInfo)

fl_getImageVersion returns the version number of the specified image.

unsigned **fl_getImageAddress**(fl_BootImageInfo *bootImageInfo)

fl_getImageAddress returns the start address in flash of the specified image.

unsigned **fl_getImageSize**(fl_BootImageInfo *bootImageInfo)

fl_getImageSize returns the size of the specified image.

int **fl_getToolsMajor**(fl_BootImageInfo *bootImageInfo)

fl_getToolsMajor returns the tools major version used to build the specified image, or -1 on failure.

int **fl_getToolsMinor**(fl_BootImageInfo *bootImageInfo)

fl_getToolsMinor returns the tools minor version used to build the specified image, or -1 on failure.

int **fl_getToolsPatch**(fl_BootImageInfo *bootImageInfo)

fl_getToolsPatch returns the tools patch version used to build the specified image, or -1 on failure.

int **fl_getImageFormat**(fl_BootImageInfo *bootImageInfo)

fl_getImageFormat returns the compatibility version of the specified image, or -1 on failure.

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

fl_startImageReplace prepares the Quad-SPI device for replacing an image. The old image can no longer be assumed to exist after this call.

Attempting to write into the data partition or the space of another upgrade image is invalid. A non-zero return value signifies that the preparation is not yet complete and that the function should be called again. This behavior allows the latency of a sector erase to be masked by the program.

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

fl_startImageAdd prepares the Quad-SPI device for adding an image after the specified image. The start of the new image is at least padding bytes after the previous image.

Attempting to write into the data partition or the space of another upgrade image is invalid. A non-zero return value signifies that the preparation is not yet complete and that the function must be called again. This behavior allows the latency of a sector erase to be masked by the program.

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

fl_startImageAddAt prepares the Quad-SPI device for adding an image at the specified address offset from the base of the first sector after the factory image.

Attempting to write into the data partition or the space of another upgrade image is invalid. A non-zero return value signifies that the preparation is not yet complete and that the function must be called again.

int **f1_writeImagePage**(const unsigned char page[])

f1_writeImagePage waits until the Quad-SPI device is able to accept a request and then outputs the next page of data to the device. Attempting to write past the maximum size passed to f1_startImageReplace, f1_startImageAdd or f1_startImageAddAt is invalid.

int **f1_writeImageEnd**(void)

f1_writeImageEnd waits until the Quad-SPI device has written the last page of data to its memory.

int **f1_startImageRead**(fl_BootImageInfo *b)

f1_startImageRead prepares the Quad-SPI device for reading the contents of the specified upgrade image.

int **f1_readImagePage**(unsigned char page[])

f1_readImagePage inputs the next page of data from the Quad-SPI device and writes it to the array page.

int **f1_deleteImage**(fl_BootImageInfo *b)

f1_deleteImage erases the upgrade image with the specified image.

5.4.4.3 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 **f1_getDataPartitionSize**(void)

f1_getDataPartitionSize returns the size of the data partition in bytes.

int **f1_readData**(unsigned offset, unsigned size, unsigned char dst[])

f1_readData reads a number of bytes from an offset into the data partition and writes them to the array dst.

unsigned **f1_getWriteScratchSize**(unsigned offset, unsigned size)

f1_getWriteScratchSize returns the buffer size needed by f1_writeData for the given parameters.

int **f1_writeData**(unsigned offset, unsigned size, const unsigned char src[], unsigned char buffer[])

f1_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.

5.4.4.3.1 Page-Level Functions

unsigned **f1_getPageSize**(void)

f1_getPageSize returns the page size in bytes.

unsigned **f1_getNumDataPages**(void)

f1_getNumDataPages returns the number of pages in the data partition.

unsigned **f1_writeDataPage**(unsigned n, const unsigned char data[])

f1_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 **f1_readDataPage**(unsigned n, unsigned char data[])

f1_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.

5.4.4.3.2 Sector-Level Functions

unsigned **f1_getNumDataSectors**(void)

f1_getNumDataSectors returns the number of sectors in the data partition.

unsigned **f1_getDataSectorSize**(unsigned n)

f1_getDataSectorSize returns the size of the *n*-th sector in the data partition in bytes.

unsigned **f1_eraseDataSector**(unsigned n)

f1_eraseDataSector erases the *n*-th sector in the data partition.

unsigned **f1_eraseAllDataSectors**(void)

f1_eraseAllDataSectors erases all sectors in the data partition.

5.4.5 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 [List of flash devices supported natively by libflash](#) lists the flash devices for which SPI spec files are included with the tools.

Table 5.29: List of flash devices supported natively by libflash

Manufacturer	Part Number	Enabled in libflash by default
Altera	EPCS1	Y
AMIC	A25L016	N
	A25L40P	N
	A25L40PT	N
	A25L40PUM	N
	A25L80P	N
Atmel	AT25DF021	N
	AT25DF041A	Y
	AT25F512	N
	AT25FS010	Y
ESMT	F25L004A	N
Macronix	MX25L1005C	N
Micron	M25P40	N
NUMONYX	M25P10	N
	M25P16	N
	M45P10E	N
SPANSION	S25FL204K	N
SST	SST25VF010	N
	SST25VF016	N
	SST25VF040	N
ST Microelectronics	M25PE10	Y
	M25PE20	Y
Winbond	W25X10	N
	W25X20	N
	W25X40	Y

Further details can be found by examining `$XMOS_TOOL_PATH/target/include/SpecEnum.h` and `$XMOS_TOOL_PATH/target/include/SpecMacros.h`.

Refer to [Add support for a new flash device](#) for information on the specification file format and advice on supporting other flash devices.

5.4.6 List of devices natively supported by libquadflash

libquadflash supports a limited range of flash devices available in the market. Each flash device is described using a Quad-SPI specification file. The table in [List of flash devices supported natively by libquadflash](#) lists the flash devices for which Quad-SPI spec files are included with the tools.

Table 5.30: List of flash devices supported natively by libquadflash

Manufacturer	Part Number	Enabled in libquadflash by default
ISSI	IS25LP016D	Y
	IS25LP032	Y
	IS25LP064	Y
	IS25LP080D	Y
	IS25LP128	Y
	IS25LQ016B	Y
	IS25LQ032B	Y
	IS25LQ080B	Y
SPANSION	S25FL116K	Y
	S25FL132K	Y
	S25FL164K	Y
WINBOND	W25Q128JV	Y
	W25Q16JV	Y
	W25Q32JV	Y
	W25Q64JV	Y

Further details can be found by examining `$XMS_T00L_PATH/target/include/QuadSpecEnum.h` and `$XMS_T00L_PATH/target/include/QuadSpecMacros.h`.

Refer to [Add support for a new flash device](#) for information on the specification file format and advice on supporting other flash devices.

5.5 XCOMMON build system

The XCOMMON build system is built on top of the GNU Makefile build system. The aspiration of the XCOMMON build system is to accelerate the development of Xcore applications. Instead of having to express dependencies explicitly in Makefiles, users are intended to follow particular folder structures and naming conventions, from which dependencies are inferred automatically.

The XCOMMON build system depends on use of `XMAKE` specifically. It cannot be used with a generic port of GNU Make.

5.5.1 Using the XCOMMON build system

The **common XMOS Makefile** provides support for building applications and source code modules. You need only specify the required properties of the build in **Application Makefiles** and `module_build_info` files.

5.5.1.1 Applications and Modules

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

Workspace

A *workspace* is a container for several projects.

Applications

An *application* is a project containing source files and a Makefile that builds into a single executable (`.xe`) file. By convention application directories start with the prefix `app_`.

Modules

A *module* is a directory containing source files and/or binary libraries. The source does not build to anything by itself but can be used by applications. By convention module directories start with the prefix `module_`.

5.5.1.1.1 Workspace structure and automatic module detection

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

The simplest structure is shown below:

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

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

Sometimes applications and modules are organized in separate repositories:

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

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

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

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

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

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

In addition to the automatic searching for modules, the environment variable `XMOS_MODULE_PATH` can be set to a list of paths that the Makefiles should search. If you just want to solely use the user specified search path then the automatic searching for modules can be disabled by setting the environment variable `XCOMMON_DISABLE_AUTO_MODULE_SEARCH` to 1.

5.5.1.2 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 [Using XMOS Makefiles to create binary libraries](#) for details.

5.5.1.3 The module_build_info file

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

DEPENDENT_MODULES

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

MODULE_XCC_FLAGS

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

```
MODULE_XCC_FLAGS = $(XCC_FLAGS) -O3
```

MODULE_XCC_XC_FLAGS

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

MODULE_XCC_C_FLAGS

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

MODULE_XCC_ASM_FLAGS

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

OPTIONAL_HEADERS

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

5.5.2 Using XMOS Makefiles to create binary libraries

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/
```

(continues on next page)

(continued from previous page)

```
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).

5.5.2.1 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:

```
LIBRARIES = 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
```

5.5.2.2 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/
```

(continues on next page)



(continued from previous page)

```
support_fns.xc
include/
my_library.h
```

5.5.2.3 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.



Copyright © 2024, XMOS Ltd

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, XCORE, VocalFusion 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. Company and product names mentioned in this document are the trademarks or registered trademarks of their respective owners.

