XMOS XTC Tools - Tools Guide

Release: 15.3
Publication Date: 2024/07/24
Document Number: XM-014363-PC

Table of Contents

1 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 XTC Tools and third-party tools
- An XMOS target system which may require a separate xTAG

1.1 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.

1.1.1 Build an executable

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

Listing 1.1: main.c

```c
#include <stdio.h>

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

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.

1.1.1.1 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.

1.1.1.2 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.

1.1.2 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!
```

1.1.3 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.

1.1.3.1 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
```

Tip: The sequence of `load`, `break main`, `continue` can be simplified to just running the *`start`* command.

1.1.3.2 Debugging using XGDB on the simulator

It is possible to use XGDB to debug programs running 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
...
```

1.1.4 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*

1.2 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.

1.2.1 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 1.2: 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 1.3: 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:

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

1.2.2 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*

1.3 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.

1.3.1 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 1.4: 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` keyword 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.

1.3.2 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 1.5: main.c

```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
```

1.3.3 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 chanends of for each end of this channel to the C functions.

1.4 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 noticable 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.

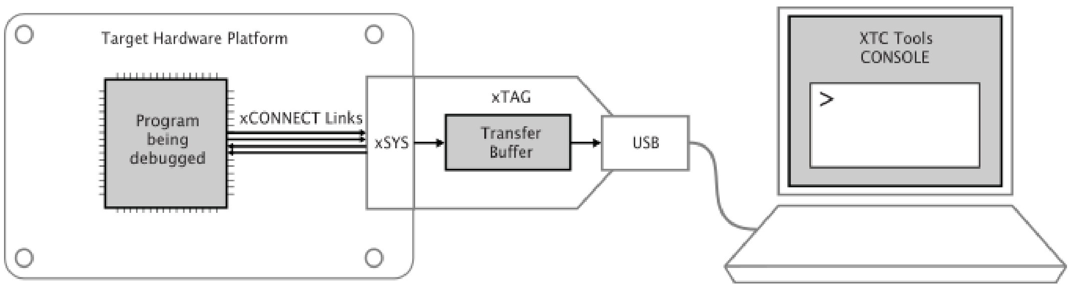


Fig. 1.1: xTAG debug setup showing JTAG and xLINK connections

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.

1.4.1 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 1.6: 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. If compiling and linking separately, the .xscope files must be provided to XCC at every stage, since they are used both to autogenerate definitions at compile time, and provide functionality at link time.

```
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 written data. Note that this also reduces the amount of data that can be buffered at any time.

1.4.2 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. For example, these could be streams of ADC samples.

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 file* as follows:

Listing 1.7: 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 1.8: 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(TILE0_RESULT, result);

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

  printf("Tile 0: Result %d\n", result);
  xscope_int(TILE0_RESULT, result);
```

(continues on next page)

(continued from previous page)

```
}

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(TILE1_I, i);
    xscope_int(TILE1_ACCUMULATION, accumulation);
  }

  chan_out_word(c, accumulation);
}
```

Build similarly to before:

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

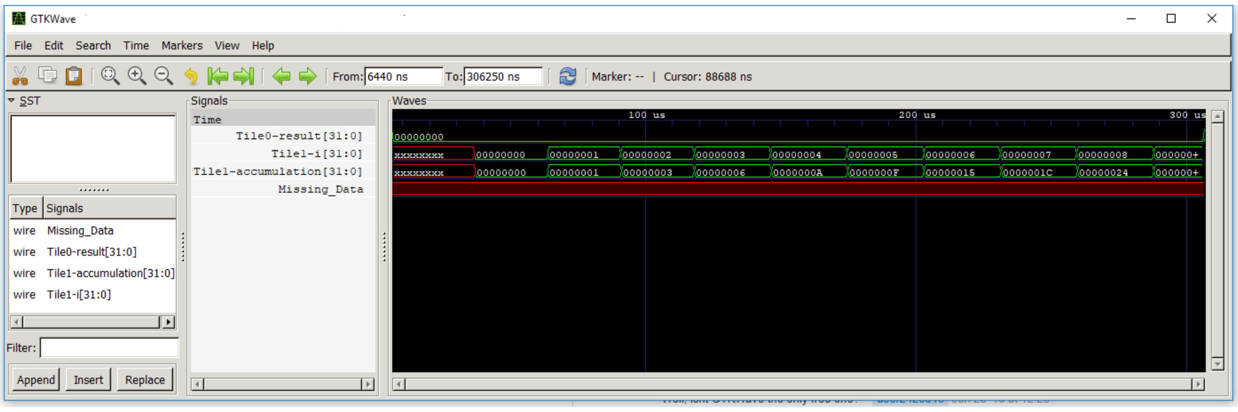
This time, when running, add *--xscope-file* to specify a file to write the probe output into:

```
$ 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 third-party VCD viewer. One option is GTKWave. To use **gtkwave** to view the VCD file:

```
$ gtkwave xscope.vcd
```

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



```
Tile0-result[31:0]        00000000
Tile1-i[31:0]             xxxxxxxx 00000000 00000001 00000002 00000003 00000004 00000005 00000006 00000007 00000008 000000+
Tile1-accumulation[31:0]  xxxxxxxx 00000000 00000001 00000003 00000006 0000000A 0000000F 00000015 0000001C 00000024 000000+
Missing_Data
```

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*.

1.4.3 Summary

xSCOPE can be used to perform fast "printf debugging". This method (using `xrun --xscope` instead of `xrun --io`) is faster than printf debugging over JTAG. It is much better for real-time performance, and should often be the preferred option.

1.5 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 third-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.

1.5.1 Create example

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

Listing 1.9: 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 1.10: main.c

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

DECLARE_JOB(process, (chanend_t, chanend_t, char *, int));
void process(chanend_t cIn, chanend_t cOut, char * name, int init)
{
  if (init)
  {
    chan_out_word(cOut, 1);
```

(continues on next page)

(continued from previous page)

```
  }

  while (1)
  {
    int token = chan_in_word(cIn);
    printf("%s\n", name);
    chan_out_word(cOut, token);
  }
}

void main_tile0(chanend_t cIn, chanend_t cOut)
{
  channel_t chan = chan_alloc();

  PAR_JOBS(
    PJOB(process, (cIn, chan.end_a, "tile0-core0", 0)),
    PJOB(process, (chan.end_b, cOut, "tile0-core1", 0)));

  chan_free(chan);
}

void main_tile1(chanend_t cIn, chanend_t cOut)
{
  channel_t chan = chan_alloc();

  PAR_JOBS(
    PJOB(process, (cIn, chan.end_a, "tile1-core0", 0)),
    PJOB(process, (chan.end_b, cOut, "tile1-core1", 1))); // Kick-start the ring

  chan_free(chan);
}
```

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
...
```

1.5.2 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 GDB prompt will be shown. 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 inferior 1 (tile[0])]
[Switching to thread 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 examples/debug/main.c, line 16. (2 locations)
```

Note that when setting a breakpoint on a line of source, that breakpoint will be set on every tile which that specific line of source can run on. In this case, the `process` function is called on both tiles, so a breakpoint gets set on each of the tiles.

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  Address    What
1   breakpoint     keep y    <MULTIPLE>
1.1                     y    0x00040156 in process at examples/debug/main.c:16 inf 1
1.2                     y    0x0004015a in process at examples/debug/main.c:16 inf 2
```

Note: To restrict the breakpoint to only be set on a single tile, you can use the additional `inferior` argument to the breakpoint command, such as `break main.c:16 inferior 1`.

To find the relevant inferior numbers, the *info inferiors* command can be used:

```
(gdb) info inferiors
  Num  Description       Connection           Executable
  1    tile[0]           1 (remote :39211)    /tmp/.xgdb/593782/0/a.xe.i0_n0_t0.elf
* 2    tile[1]           1 (remote :39211)    /tmp/.xgdb/593782/0/a.xe.i1_n0_t1.elf
```

The * indicates the currently selected inferior.

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 1.1, process (cIn=2147614978, cOut=2147615234, name=0x4511c "tile0-core0", init=0) at
↪examples/debug/main.c:16
16          printf("%s\n", name);
Current language:  auto; currently minimal
(gdb) info threads
* 1.1  tile[0] core[0]  process (cIn=2147614978, cOut=2147615234, name=0x4511c "tile0-core0",
↪init=0) at examples/debug/main.c:16
  1.2  tile[0] core[1] 0x000403f4 in chan_in_word ()
  2.1  tile[1] core[0] 0x000403c4 in chan_in_word ()
  2.2  tile[1] core[1] 0x000403c4 in chan_in_word ()
```

The thread with the asterisk next to it shows which tile/core currently has focus for subsequent commands. You can observe the asterisk moving as the token moves around the ring by repeatedly issuing `continue` and *`info threads`*.

To end the interactive debug session:

```
(gdb) quit
```

1.5.3 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 scenario, and even share those scenarios with other developers.

To start a scripted debug session, use *`xgdb -x`* with the command filename:

```
$ xgdb --batch -x cmds.txt a.xe
```

This causes the following command file to be executed prior to any interactive debugging; the *`xgdb --batch`* option in the example command means that there there will be no interaction, gdb will terminate when the script file has been fully processed.

Listing 1.11: 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
```

Further information for scripting with XGDB can be found *in the reference manual*.

1.5.4 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, see *XMOS XGDB commands*.

1.6 Build system

The previous examples in this quick-start guide are simple applications that can be built manually with XCC commands. For more complex applications, a CMake-based build and dependency management tool called XCommon CMake is provided.

An example application that can be built using XCommon CMake is distributed in the tools release under the `examples/ExampleXCommonCMake` directory. Instructions are provided in the `README` in that directory.

Normally a user cannot run this example in-place under the XTC Tools installation directory because they do not have permissions to create directories or files. The files in this example directory should be copied to a directory for which the user does have file write permissions, and run from that directory.

For full documentation, please refer to the XCommon CMake documentation.

2 Using VS Code

VS Code is a free Integrated Development Environment (IDE) provided by Microsoft. It may be used for editing code, Git repository management, building projects (source bases) to generate target binaries, launching target binaries and flashing target devices.

See https://code.visualstudio.com/docs for full documetation.

VS Code terminology is referred to in the remainder of this document. For details see: https://code.visualstudio.com/docs/getstarted/userinterface.

2.1 Building an example

The following section shows how to build a simple *XCommon CMake* based example program in VS Code. XCommon CMake is the build and dependency management system delivered with the XTC Tools and is required to build certain XMOS applications.

While following the steps below, some dialogs may pop up. For example the VS Code CMake Tools extension may ask whether an XCommon CMake project should be re-configured each time VS Code is started. Certain dialogs may be ignored and they will disappear after a few seconds. Depending on the responses provided these dialogs may pop up again later - for example when a workspace is opened in VS Code a second time.

2.1.1 Copy the example

1. Browse to the installation directory of the XTC Tools, for example `C:\Program Files\XMOS\XTC\15.3.0\`
2. Browse to `examples`
3. Copy the sub-directory `ExampleXCommonCMake` to a temporary working directory

2.1.2 Starting VS Code

Windows

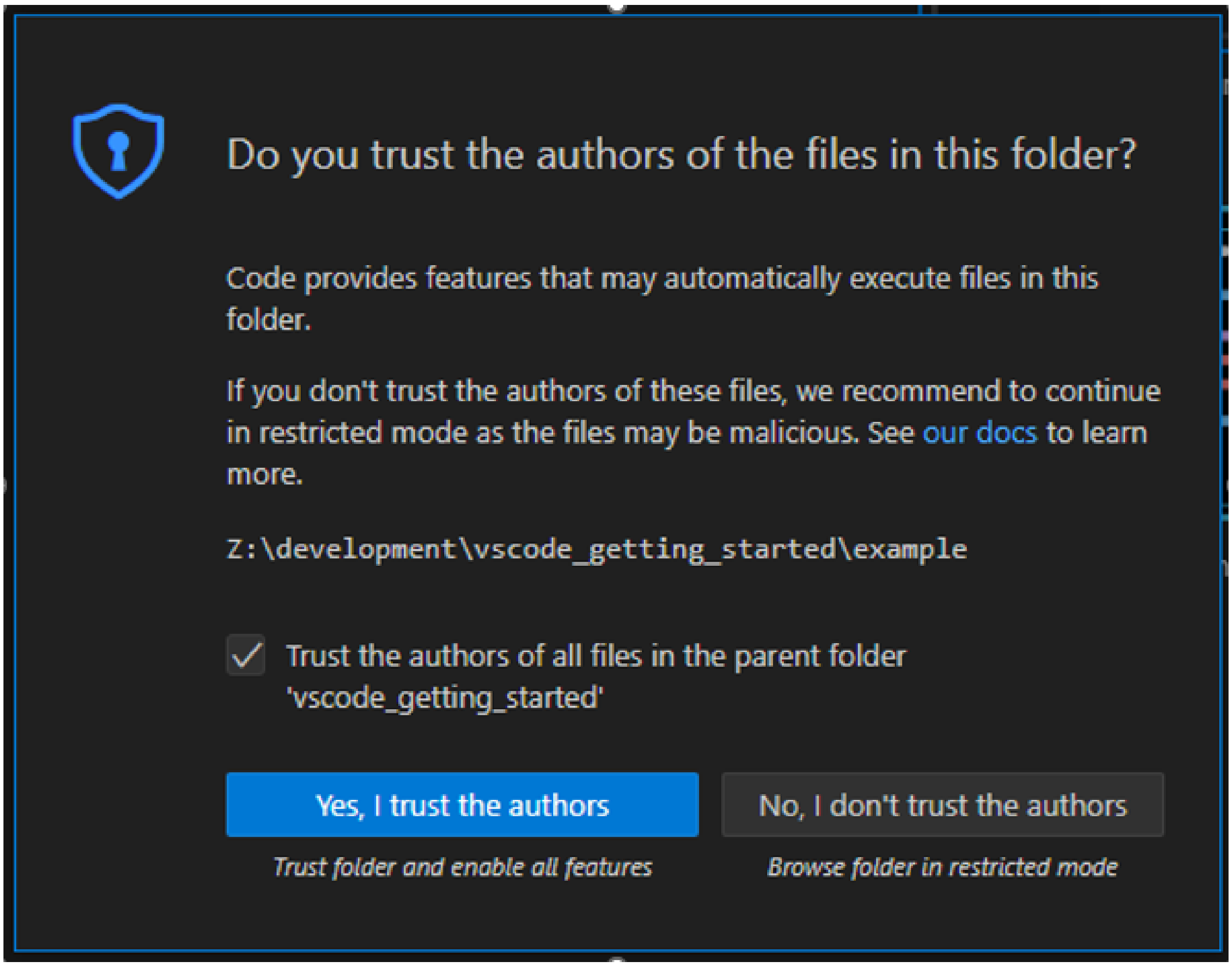
1. Open an XTC Tools Command Prompt
2. Change directory to the copied `ExampleXCommonCMake` directory
3. In the command prompt type

   ```
   $ code .
   ```

   Note the `.` character which tells VS Code to use the current directory as a workspace

   Note: If VS Code restarts (for example after a machine reboot) it may not inherit the XTC Tools setup carried out above. If this occurs, close VS Code, and go back to *starting VS Code*

4. VS Code may present a dialog box *Do you trust the authors of the files in this folder*. Check the box *Trust the authors of all files in the parent folder …* and press the button *Yes, I trust the authors*


Do you trust the authors of the files in this folder?

Code provides features that may automatically execute files in this folder.

If you don't trust the authors of these files, we recommend to continue in restricted mode as the files may be malicious. See our docs to learn more.

Z:\development\vscode_getting_started\example

✓ Trust the authors of all files in the parent folder 'vscode_getting_started'

Yes, I trust the authors
Trust folder and enable all features
No, I don't trust the authors
Browse folder in restricted mode

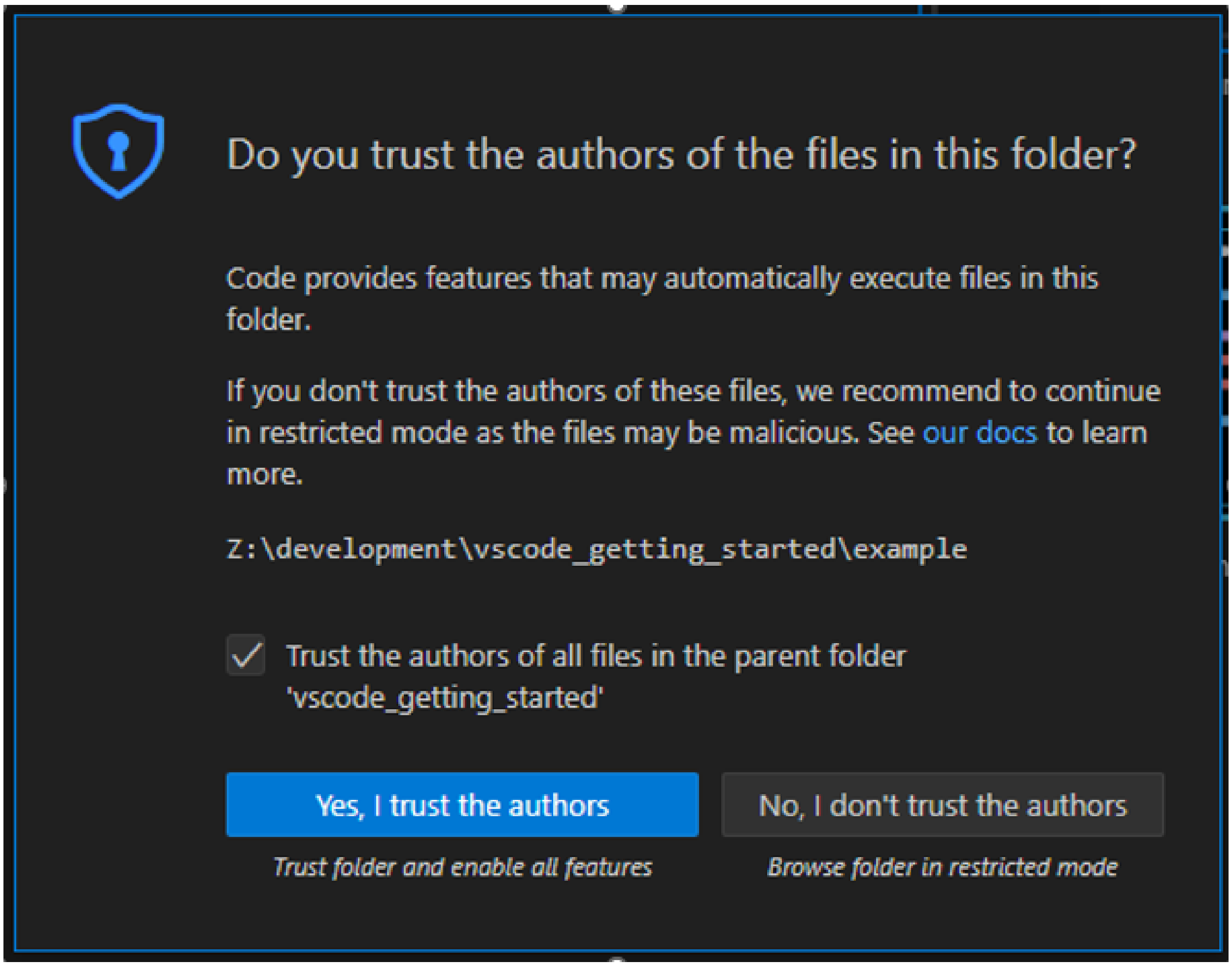
Mac

1. Ensure all instances of VSCode are closed in the *Dock* - this is required to ensure that when it is opened in the steps below it inherits the XTC Tools environment
2. Use a shell in which the XTC Tools have been setup for command line usage
3. Change directory to the copied `ExampleXCommonCMake` directory
4. Type

   ```
   $ open /Applications/Visual\ Studio\ Code.app
   ```

   Note 1: VS Code may open a workspace directory which had been used in a previous session. If this occurs, close the workspace directory with **File => Close Folder** and open the copied `ExampleXCommonCMake` directory with **File => Open Folder…**

   Note 2: If VS Code restarts (for example after a machine reboot) it may not inherit the XTC Tools setup carried out above. If this occurs, close VS Code, and go back to *starting VS Code*
5. VS Code may present a dialog box *Do you trust the authors of the files in this folder*. Check the box *Trust the authors of all files in the parent folder …* and press the button *Yes, I trust the authors*


Do you trust the authors of the files in this folder?

Code provides features that may automatically execute files in this folder.

If you don't trust the authors of these files, we recommend to continue in restricted mode as the files may be malicious. See our docs to learn more.

Z:\development\vscode_getting_started\example

✓ Trust the authors of all files in the parent folder 'vscode_getting_started'

Yes, I trust the authors
Trust folder and enable all features
No, I don't trust the authors
Browse folder in restricted mode

Linux

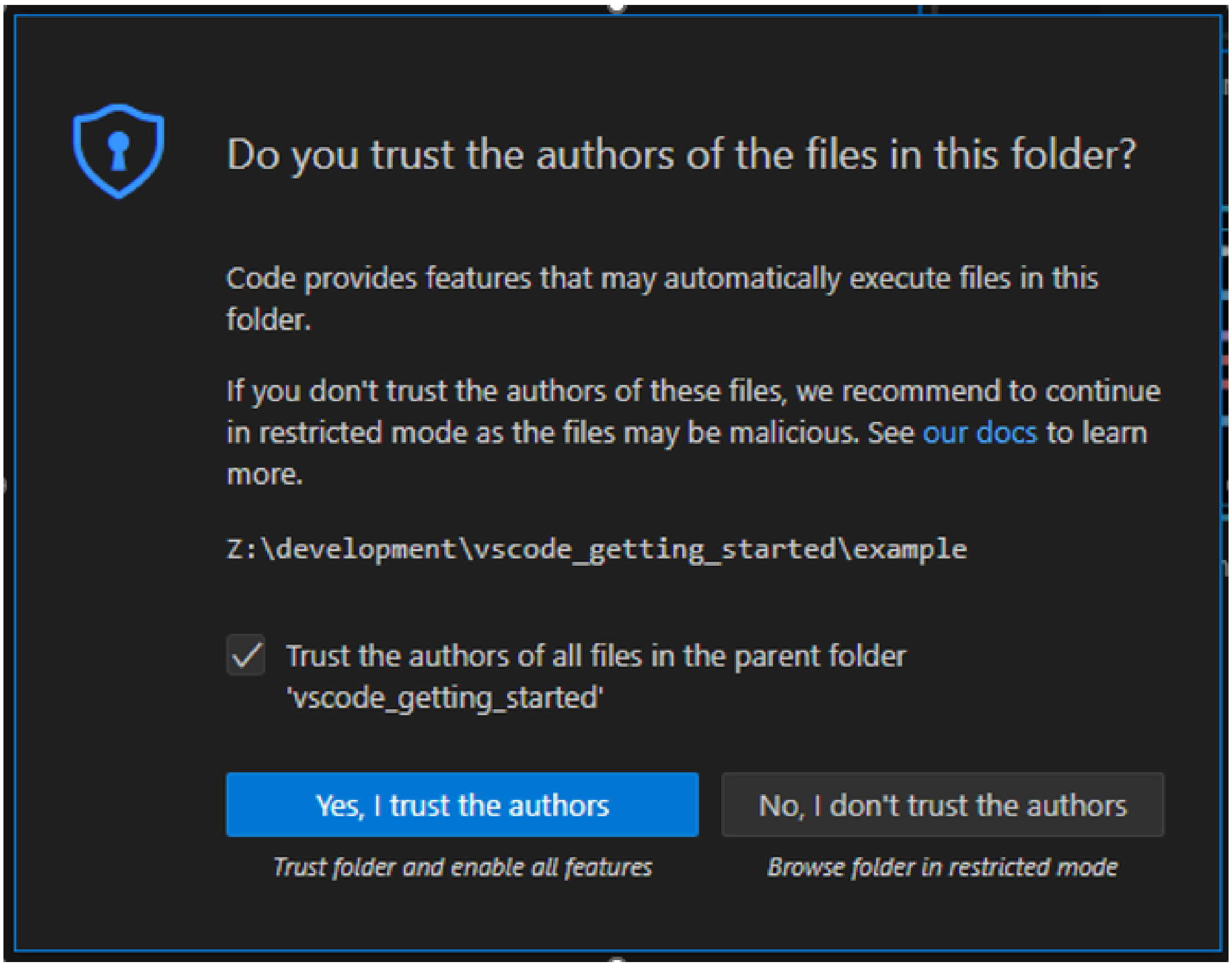
1. Use a shell in which the XTC Tools have been setup for command line usage
2. Change directory to the copied `ExampleXCommonCMake` directory
3. In the command prompt type

```
$ code .
```

   Note the `.` character which tells VS Code to use the current directory as a workspace

   Note: If VS Code restarts (for example after a machine reboot) it may not inherit the XTC Tools setup carried out above. If this occurs, close VS Code, and go back to *starting VS Code*

4. VS Code may present a dialog box *Do you trust the authors of the files in this folder*. Check the box *Trust the authors of all files in the parent folder …* and press the button *Yes, I trust the authors*


Do you trust the authors of the files in this folder?

Code provides features that may automatically execute files in this folder.

If you don't trust the authors of these files, we recommend to continue in restricted mode as the files may be malicious. See our docs to learn more.

```
Z:\development\vscode_getting_started\example
```

✓ Trust the authors of all files in the parent folder 'vscode_getting_started'

Yes, I trust the authors
Trust folder and enable all features
No, I don't trust the authors
Browse folder in restricted mode

2.1.3 Setup VS Code to use XCommon CMake

VS Code can be be extended by installing **Extensions** from the **Marketplace**. Extensions are provided by Microsoft and by third parties. Both Microsoft and third party extensions are used to use the XTC Tools with VS Code.

2.1.3.1 Install the CMake Tools extension

1. In the copied example directory ensure there is a subdirectory called `.vscode` and it has a file `settings.json`, which contains

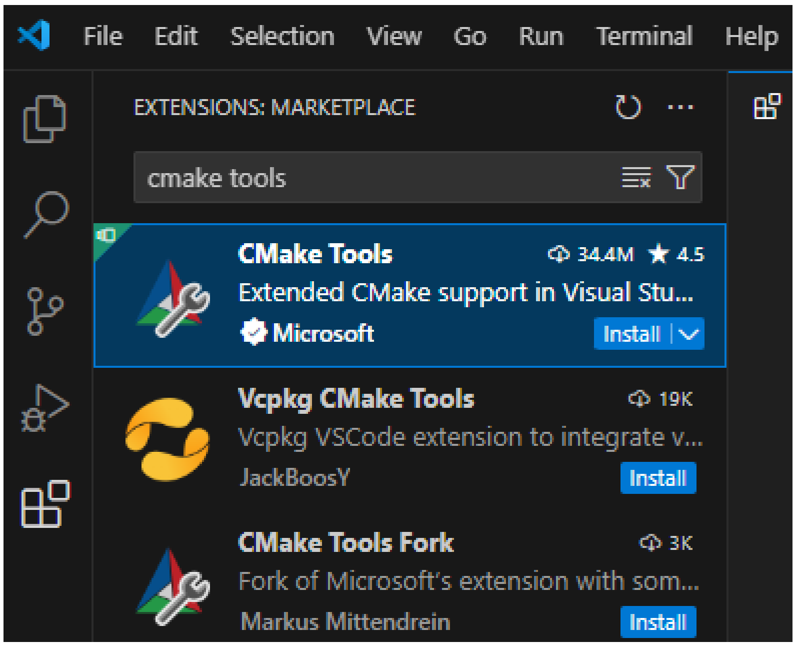
   ```
   {
     "cmake.generator": "Unix Makefiles"
   }
   ```

   Note 1: All XCommon CMake based projects require a `.vscode` subdirectory with a file `settings.json`, with the above content

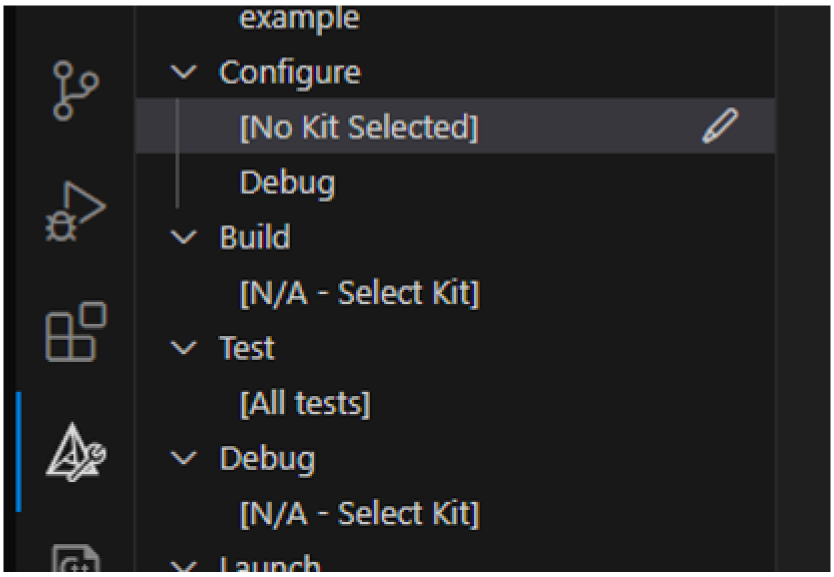
   Note 2: VS Code may write additional lines into this file. It may be necessary to delete these lines, particularly if the `.vscode` directory is copied into a new workspace

   Note 3: The file explorer or command line tools on some systems may default to hide directories which start with a `.` character. There is normally a way to make these directories visible

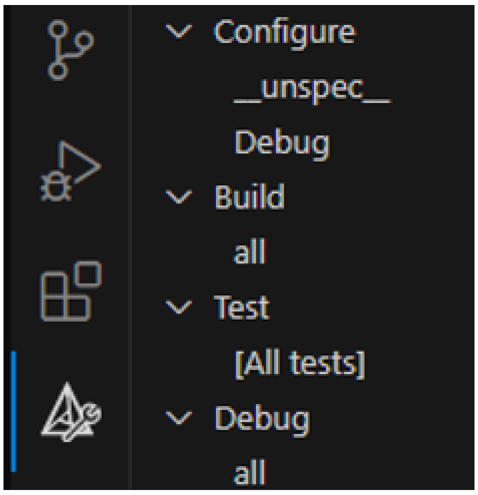
2. Select the **Extensions** icon on the left-hand **Activity bar** as shown below



3. Type `cmake tools` into the *Search Extensions in Marketplace* edit box
4. From the list of extensions presented, selected **CMake Tools** provided by Microsoft
5. Press the **Install** button in this extension
6. If a pop-up prompt appears over the VS Code top-centre search box with a list box titled: *Select a Kit for* ….
   1. Click on the *[unspecified]* option in this list box
   2. Go to *Install the Task Runner extension*
7. If this pop-up prompt does not appear or it disappears before a kit has been selected
   1. Go to the *CMAKE View* in the *Primary Sidebar*
   2. Select the *CMake icon* on the left hand side bar
   3. Expand the *Configure* item, and click on *[No Kit Selected]*
   4. Click on the pen icon. This is shown below



5. This will make a pull-down menu appear – select *[Unspecified]* and then the *Configure* pulldown will show *__unspec__* as shown below



2.1.3.2 Install the Task Runner extension

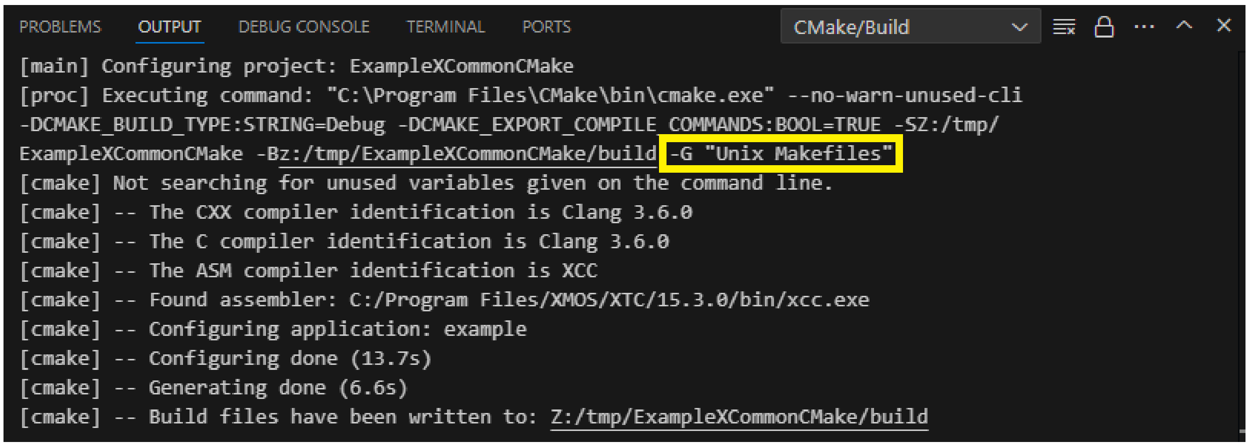
1. Select the **Extensions** icon on the left-hand **Activity bar**
2. Type `Task Runner` into the *Search Extensions in Marketplace* edit box
3. Select the extension *Task Runner* by *Sana Ajani* and press the *Install* button

VS Code is now set up for use with the XTC Tools. The *Setup* instructions above should not need to be repeated each time VS Code is started using the steps under *starting VS Code*

2.1.4 Configure the example

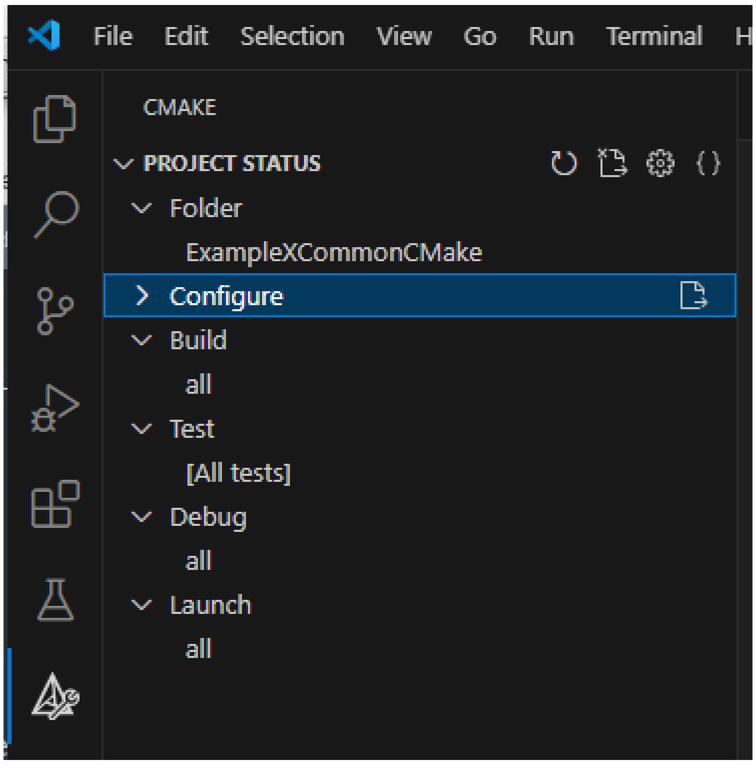
An XCommon CMake project must be configured before it can be built. This will generate subdirectories which are used in the subsequent build phase.

1. The *Configure* step may be performed automatically by VS Code at this point and the *OUTPUT* window in the *Panel* will show



```
[main] Configuring project: ExampleXCommonCMake
[proc] Executing command: "C:\Program Files\CMake\bin\cmake.exe" --no-warn-unused-cli
-DCMAKE_BUILD_TYPE:STRING=Debug -DCMAKE_EXPORT_COMPILE_COMMANDS:BOOL=TRUE -SZ:/tmp/
ExampleXCommonCMake -Bz:/tmp/ExampleXCommonCMake/build -G "Unix Makefiles"
[cmake] Not searching for unused variables given on the command line.
[cmake] -- The CXX compiler identification is Clang 3.6.0
[cmake] -- The C compiler identification is Clang 3.6.0
[cmake] -- The ASM compiler identification is XCC
[cmake] -- Found assembler: C:/Program Files/XMOS/XTC/15.3.0/bin/xcc.exe
[cmake] -- Configuring application: example
[cmake] -- Configuring done (13.7s)
[cmake] -- Generating done (6.6s)
[cmake] -- Build files have been written to: Z:/tmp/ExampleXCommonCMake/build
```

2. If this configure step has not been done, it may be launched by selecting the **CMake** icon in the **Activity bar** and clicking on the *page* icon to the right of the *Configure* pull-down



3. Ensure the OUTPUT Window shows the `-G Unix Makefiles` option to `cmake` as shown in the figure above. And that the compiler identification reports: `The CXX compiler identification is Clang 3.6.0`. This is the underlying compiler tool used by the XTC Tools

   Note: If a Configure step is made a second time, reduced text is shown in the OUTPUT Window. The line with `-G Unix Makefiles` will be shown, but subsequent lines will not. To perform a full "Configure", delete the generated `build` directory from the top level of the project and repeat the Configure step

4. Wait for the *Configuring project:* … progress message in the bottom **Status bar** to disappear. A message of the form `[cmake] -- Build files have been written to:...` will be shown in the OUTPUT Window

5. A `build` directory will be created in the workspace directory containing the required files for the subsequent build step

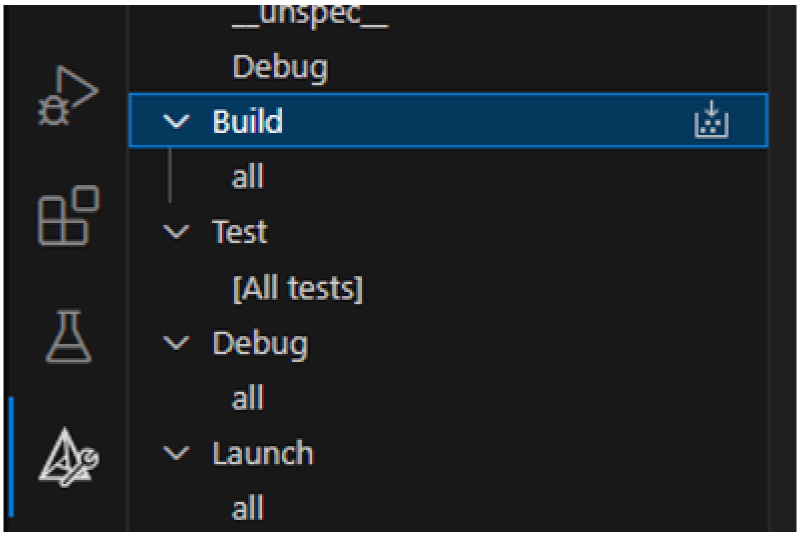
   Note: This build directory may need to be deleted manually if the workspace directory is copied or moved, and the *Configure* step repeated

2.1.5 Build the example

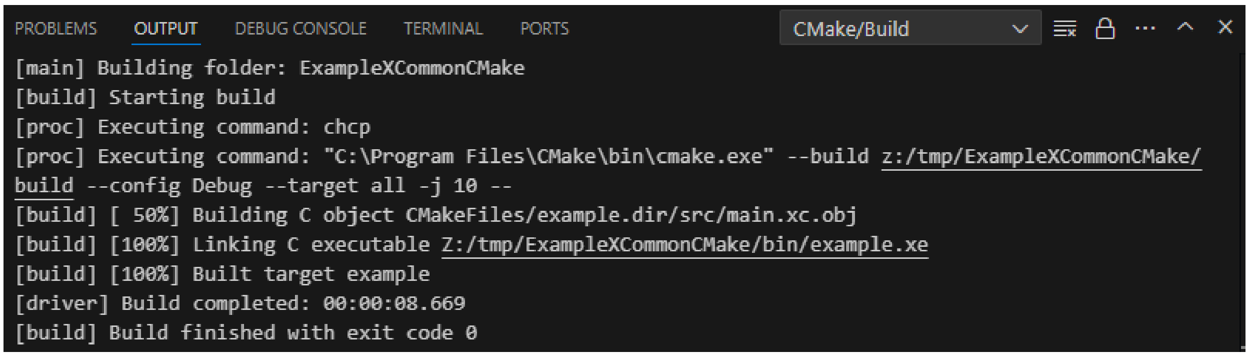
This will build the target binary files using the configuration generated in the above steps.

1. Select the **CMake** icon on the left-hand **Activity bar**

2. Build the configured example by pressing the *Build* icon to the right of the *Build* pulldown



3. The OUTPUT window will show



```
[main] Building folder: ExampleXCommonCMake
[build] Starting build
[proc] Executing command: chcp
[proc] Executing command: "C:\Program Files\CMake\bin\cmake.exe" --build z:/tmp/ExampleXCommonCMake/
build --config Debug --target all -j 10 --
[build] [ 50%] Building C object CMakeFiles/example.dir/src/main.xc.obj
[build] [100%] Linking C executable Z:/tmp/ExampleXCommonCMake/bin/example.xe
[build] [100%] Built target example
[driver] Build completed: 00:00:08.669
[build] Build finished with exit code 0
```

2.1.6 Running the example

The example will be run using a VS Code *Task*

Ensure the `.vscode` subdirectory contains the file `tasks.json` with the content

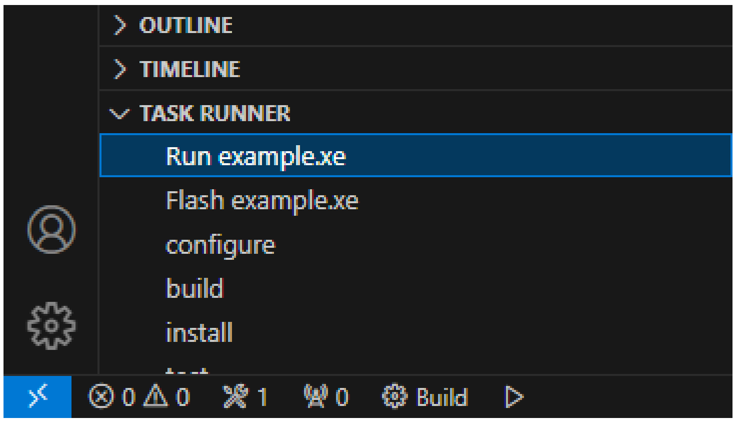
```
{
    "tasks": [
        {
            "label": "Run example.xe",
            "command": "xrun",
            "args": [
                "--io",
                "${workspaceFolder}/bin/example.xe"
            ],
            "options": {
                "cwd": "${workspaceFolder}"
            },
            "problemMatcher": []
        },
        {
            "label": "Flash example.xe",
            "command": "xflash",
            "args": [
                "${workspaceFolder}/bin/example.xe"
            ],
```

(continues on next page)

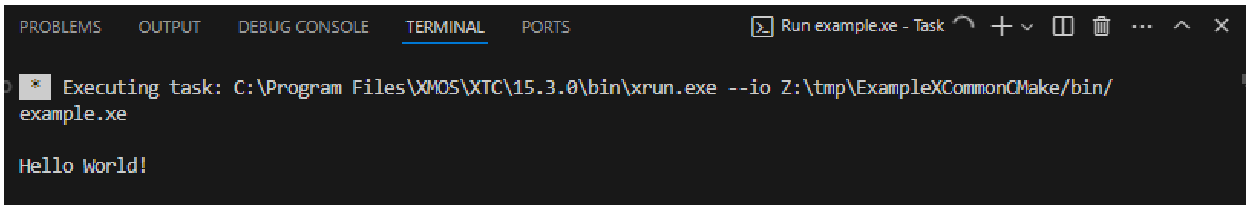
(continued from previous page)

```
            "options": {
                "cwd": "${workspaceFolder}"
            },
            "problemMatcher": []
        }
    ]
}
```

1. Select the **Explorer** icon on the left-hand **Activity bar**
2. Expand the *TASK RUNNER* drop-down at the bottom of the *Primary Side Bar*
3. Select *Run example.xe*



4. The output from the run is shown in the *TERMINAL* window in the *Panel*. The text `Hello World!` will be shown and the LEDs will flash. This program will run until it is terminated by the user.



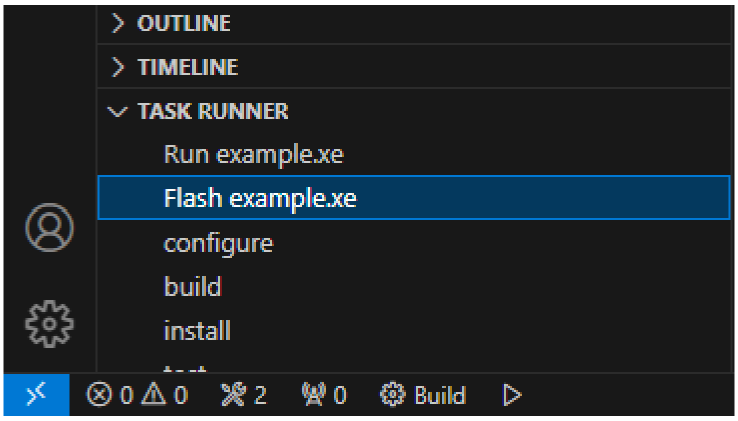
```
 * Executing task: C:\Program Files\XMOS\XTC\15.3.0\bin\xrun.exe --io Z:\tmp\ExampleXCommonCMake/bin/
example.xe

Hello World!
```

5. If the task runs forever (typically when xrun is launched with –io, and the target program never terminates), a spinning symbol will be shown next to the running task info button.
6. It may be terminated by pressing `Ctrl-C` in the *TERMINAL* window or by terminating the *TERMINAL* in which it is running by clicking on the trash-can icon to the right of the task info button.

   Note: On some systems, when using the trash-can icon, the LEDs may continue to flash due to the way VS Code terminates the xrun process

2.1.7 Flashing the example

1. Select the **Explorer** icon on the left-hand **Activity bar**
2. Expand the *TASK RUNNER* drop-down at the bottom of the *Primary Side Bar*
3. Select *Flash example.xe*



4. Flash progress is shown in the *TERMINAL* window in the *Panel*

2.1.8 Using the TERMINAL

The *TERMINAL* in the *Panel* may be used to run commands interactively. For example `xflash` or `xgdb` may be launched from the *TERMINAL*.

2.2 Using VS Code with a real project

The previous example shows how to build a target binary using XCommon CMake, how to run it on the target and how to flash the target with the binary.

If a new project is created, it should be based on the XTC Tools example `examples\ExampleXCommonCMake`, by copying this example into the top level of the new project. This ensures it can be built with XCommon CMake and it can be run and flashed using VS Code.

If an existing project is to be used with VS Code the `.vscode` directory should be copied from the example into the top-level of this project.

Note: if this `.vscode` directory is copied from an existing workspace, ensure the `settings.json` does not contain additional lines (written by VS Code). Some lines written by VS Code may contain paths which are invalid for the new project.

The `tasks.json` file will need to be amended to pick up the target binary built by the project for running on the target and for flashing the target.

2.2.1 Specifying extra arguments to xrun and xflash

Additional command line arguments are typically required. These are supplied in the `"args"` list which is associated with a command,

For example, if multiple targets and corresponding XTAG adapters are connected to the host machine, an adapter id must be specified to run and flashing steps. This can be done by modifying the `tasks.json` file to add an argument, as shown below:

```
{
    "tasks": [
        {
            "label": "Run example.xe",
            "command": "xrun",
            "args": [
                // each argument is supplied on the command line is provided in the list
↪of strings below
                "--adapter-id",
                "4NV62AKC"
                "--io",
                "${workspaceFolder}/bin/example.xe"
            ],
            "options": {
                "cwd": "${workspaceFolder}"
            },
            "problemMatcher": []
        },
        {
            "label": "Flash example.xe",
            "command": "xflash",
            "args": [
                // each argument is supplied on the command line is provided in the list
↪of strings below
                "--adapter-id",
                "4NV62AKC"
                "${workspaceFolder}/bin/example.xe"
            ],
            "options": {
                "cwd": "${workspaceFolder}"
            },
            "problemMatcher": []
        }
    ]
}
```

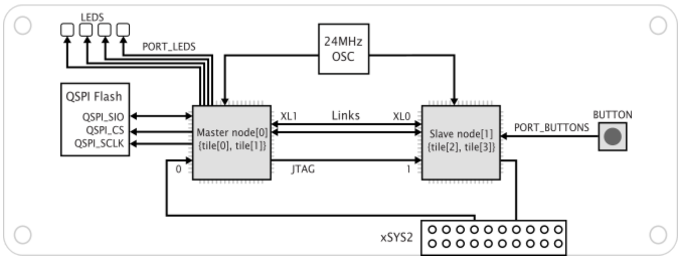
3 Tools and target features

3.1 Describe a target platform

Hardware target platforms are described using an XN file. This file provides information to the XMOS XTC tools 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 programs.

An XMOS target platform comprises one or more nodes, where each node has one or two tiles. A package (or device) typically contains one node, but some packages contain two nodes.



3.1.1 Supported network topologies

To route messages across the xCONNECT Link network, the routing ID and routing table of each node on the network must 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 of routing IDs and routing tables based on the network topology in the XN file. The tools support the following network topologies.

Table 3.1: Topologies supported by the tools

Network Topology	Supported Configurations
Single node	Single XCORE-200 or xcore.ai node
Line of nodes	Two XCORE-200 nodes or two xcore.ai nodes

The xTAG provides an optional xCONNECT link to the target board. When it is connected to a board with a single device containing one node, a line of two nodes is formed. A board with two nodes and an xTAG is a line of three nodes.

3.1.2 A board with a single package

The following example shows how to provide the XN description for a board containing a single xcore.ai package with one node. The XN file fragments below have been copied from the file in this XTC release `targets\XK-EVK-XU316\XK-EVK-XU316.xn`.

3.1.2.1 Initial declarations

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 provides the board description information (shown by `xcc -print-boards` and when `xcc` does recognise the target name supplied with `-target`).

```
<Type>Board</Type>
<Name>xcore.ai Explorer Kit (XK-EVK-XU316)</Name>
```

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>
```

3.1.2.2 The package with clocks and tiles

The next code fragment declares a single package (a BGA FB265 package containing the XS3 processor architecure and 1024 kilo bytes of internal RAM). A crystal oscillator provides an input clock to this package at 24 MHz. Each tile clock oscillates at a frequency of 600 MHz, and the reference clock used for timers oscillates at 100 MHz. The bootloader will setup these frequencies using the system PLL and clock dividers in the device.

It is typical to clock the reference clock at 100 MHz and the tiles at a multiple of 100 MHz.

The code descibes the boot source. In this case it is a flash device referred to as `bootFlash`. This will be described in more detail later.

The code then describes the LPDDR on this board in the section `ExtMem`. See the section *Hardware setup* for full details on the contents of this section.

The code then describes each tile in the package referring to the tiles declared in the `tileref` section above.

The physical ports connected to a tile may be provided with convenient board or application names as shown in this example. These names are provided in the generated file `<platform.h>`, and may be used by application software written in the C language, xC language or assembly language.

```
<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>
```

(continues on next page)

(continued from previous page)

```
          <Source Location="bootFlash"/>
        </Boot>
        <Extmem sizeMbit="1024" Frequency="100MHz">
        </Extmem>
        <Tile Number="0" Reference="tile[0]">
          <Port Location="XS1_PORT_1B" Name="PORT_SQI_CS"/>
          <Port Location="XS1_PORT_1C" Name="PORT_SQI_SCLK"/>
          <Port Location="XS1_PORT_4B" Name="PORT_SQI_SIO"/>
          <Port Location="XS1_PORT_4C" Name="PORT_LEDS"/>
          <Port Location="XS1_PORT_4D" Name="PORT_BUTTONS"/>
        </Tile>
        <Tile Number="1" Reference="tile[1]">
          <Port Location="XS1_PORT_1G" Name="PORT_PDM_CLK"/>
          <Port Location="XS1_PORT_1F" Name="PORT_PDM_DATA"/>
        </Tile>
      </Node>
    </Nodes>
  </Package>
</Packages>
```

3.1.2.3 Routing to the xTAG

The code then describes network routing required to send data from the xTAG to the device over the xCONNECT link. This code should be provided exactly as shown below.

```
<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>
```

The code then provides the xCONNECT physical routing between the target device and the xTAG. This sets the inter-symbol delay between edge transitions at 5 clock cycles where the clock is running at the tile frequency (600 MHz in the above example). Usually, This code should be provided exactly as shown below.

```
<Links>
  <Link Encoding="2wire" Delays="5clk" Flags="XSCOPE">
    <LinkEndpoint NodeId="0" Link="XL0"/>
    <LinkEndpoint NodeId="2" Chanend="1"/>
  </Link>
</Links>
```

3.1.2.4 Boot device

The code then provides information about the `bootFlash` device referred to above. This is a QSPI device (`Class="SQIFlash"`) with a size of 16384 pages (`NumPages="16384"`), a page size of 256 bytes (`PageSize="256"`) and a sector size of 4096 bytes (`SectorSize="4096"`).

Note all modern flash devices have a page size of 256 bytes.

This device is connected to tile 0, via the ports with convenience names `PORT_SQI_CS`, `PORT_SQI_SCLK` and `PORT_SQI_SIO`. The bootrom is “hard-wired” to use a QSPI device connected to these ports when instructed to boot from this source. (See the device datasheet for further information on how to specify which source the bootrom uses.)

```
<ExternalDevices>
  <Device NodeId="0" Tile="0" Class="SQIFlash" Name="bootFlash" 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"/>
  </Device>
</ExternalDevices>
```

3.1.2.5 JTAG connectivity

Finally the devices connected in the JTAG chain to the xTAG are described. In this case there is only one.

```
<JTAGChain>
  <JTAGDevice NodeId="0"/>
</JTAGChain>
```

3.1.3 A board with a package with two nodes

The following example describes a board with the XUF232-1024-FB167 package, which contains two nodes, each comprising two xCORE tiles.

Note additional usb_tile declarations are provided to allow access to the USB PHYs. Only one of these PHYs is connected to external package pins.

```
<?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">
  <Type>Board</Type>
  <Name>XUF232-1024-FB167-C40 Bringup</Name>

  <Declarations>
    <Declaration>tileref tile[4]</Declaration>
    <Declaration>tileref usb_tile[2]</Declaration>
  </Declarations>

  <Packages>
    <Package id="0" Type="XS2-UFnA-1024-FB167">
      <Nodes>
        <Node Id="0" InPackageId="0" Type="XS2-L16A-512" SystemFrequency="500MHz" OscillatorSrc="1">
          <Boot>
            <Source Location="bootFlash0"/>
            <Bootee NodeId="2"/>
          </Boot>
          <Tile Number="0" Reference="tile[0]">
            <Port Location="XS1_PORT_1B" Name="PORT_SQI_CS_0"/>
            <Port Location="XS1_PORT_1C" Name="PORT_SQI_SCLK_0"/>
            <Port Location="XS1_PORT_4B" Name="PORT_SQI_SIO_0"/>
          </Tile>
          <Tile Number="1" Reference="tile[1]"/>
        </Node>
        <Node Id="2" InPackageId="2" Type="XS2-L16A-512" SystemFrequency="500MHz" OscillatorSrc="3">
          <Boot>
            <Source Location="LINK" BootMode="4"/>
          </Boot>
          <Tile Number="0" Reference="tile[2]"/>
```

(continues on next page)

(continued from previous page)

```
        <Tile Number="1" Reference="tile[3]"/>
      </Node>
      <Node Id="1" InPackageId="1" Type="periph:XS1-SU" Reference="usb_tile[0]">
      </Node>
      <Node Id="3" InPackageId="3" Type="periph:XS1-SU" Reference="usb_tile[1]">
      </Node>
    </Nodes>
    <Links>
      <Link Encoding="5wire" Delays="3clk">
        <LinkEndpoint NodeId="0" Link="7"/>
        <LinkEndpoint NodeId="2" Link="0"/>
      </Link>
      <Link Encoding="5wire" Delays="3clk">
        <LinkEndpoint NodeId="0" Link="4"/>
        <LinkEndpoint NodeId="2" Link="3"/>
      </Link>
      <Link Encoding="5wire" Delays="3clk">
        <LinkEndpoint NodeId="0" Link="6"/>
        <LinkEndpoint NodeId="2" Link="1"/>
      </Link>
      <Link Encoding="5wire" Delays="3clk">
        <LinkEndpoint NodeId="0" Link="5"/>
        <LinkEndpoint NodeId="2" Link="2"/>
      </Link>
      <Link Encoding="5wire">
        <LinkEndpoint NodeId="0" Link="8" Delays="52clk,52clk"/>
        <LinkEndpoint NodeId="1" Link="XL0" Delays="1clk,1clk"/>
      </Link>
      <Link Encoding="5wire">
        <LinkEndpoint NodeId="2" Link="8" Delays="52clk,52clk"/>
        <LinkEndpoint NodeId="3" Link="XL0" Delays="1clk,1clk"/>
      </Link>
    </Links>
  </Package>
</Packages>

<ExternalDevices>
  <Device NodeId="0" Tile="0" Class="SQIFlash" Name="bootFlash0" Type="S25LQ016B" PageSize="256"
↪SectorSize="4096" NumPages="8192">
    <Attribute Name="PORT_SQI_CS" Value="PORT_SQI_CS_0"/>
    <Attribute Name="PORT_SQI_SCLK" Value="PORT_SQI_SCLK_0"/>
    <Attribute Name="PORT_SQI_SIO" Value="PORT_SQI_SIO_0"/>
  </Device>
</ExternalDevices>

<JTAGChain>
  <JTAGDevice NodeId="0"/>
  <JTAGDevice NodeId="2"/>
</JTAGChain>

</Network>
```

3.2 Working with targets

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

3.2.1 Introduction

The XTC Tools are a suite of tools to run on a host PC for working with XMOS target systems. 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 `xcc` tool), may be downloaded into the target RAM and executed using the `xrun` tool. Symbolic debugging of a target program may be carried out using the `xgdb` tool. A target program may burned to non-volatile flash using the `xflash` tool. These tools require an XMOS xTAG to connect the target system to the host computer via a USB 2.0 interface.

3.2.2 Key target-based features

The following target-related 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

3.2.2.1 Targets

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

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

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

3.2.3 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) 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 (xLINK) 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`, `xflash` 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 hosts do not support this feature.

3.2.3.1 xTAG v3.0

xTAG v3.0 datasheet: 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 XCORE-200 target families. It supports only 3v3 target voltage levels.

3.2.3.2 xTAG v4.0

The xTAG v4.0 provides the xSYS2 target interface (Amphenol Minitek127 20-pin 0.05”) and supports XCORE-200 and xcore.ai target families. It supports 1v8 and 3v3 target voltages, automatically and independently for JTAG and xLINK interfaces.

The xLINK 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 IEEEE1149.1 JTAG signals need to be connected.

3.2.3.3 Using xTAGs with Windows hosts

On Windows 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 hosts 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 hosts 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.

3.2.4 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.

3.2.4.1 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. Download a target program to RAM and launch it, and allow a user-supplied host program capture xSCOPE data from the target and send data to the target
6. 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

3.2.4.1.1 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.

3.2.4.1.2 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 *xSCOPE* host-IO enabled:

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

3.2.4.2 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.

3.2.4.2.1 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.

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.

3.2.4.2.2 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.

3.2.5 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-IO.

3.2.5.1 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-IO data. When a tile enters debug mode all its logical cores will be paused.

3.2.5.2 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.

3.2.5.3 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.

3.2.5.4 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(), the xrun session will terminate with a successful exit code. If the program was launched using xgdb it will report "Program exited normally " on termination. (If xrun has been used to launch a program without either the *xrun --io* or *xrun --xscope* options, then it will detach after loading, and not await termination.)

If a tile calls exit() with a non-zero value, this will be returned by xrun as the process exit code. One tile calling exit() terminates the entire system. If launched by xgdb, the exit code will be shown on the console, and the *xgdb --return-child-result* option can be used to make xgdb return the exit code as its exit code.

3.2.6 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 `xrun --io` or `xrun --xscope` options, these calls will be redirected to act on the host computer's filesystem (or its console, for stdin, stdout and stderr). This is part of the "semihosting" system which runs all system calls on the host system when the target is connected via a debug interface. By default, data is sent to and from the target via the JTAG interface. Using the JTAG interface, the tile making the call enters 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`. The system calls will be skipped if a debugger is not connected, however much of the logic (such as the string formatting of printf) will still execute. If `xrun` or `xgdb` is later used with a such a system, the IO will be redirected to the host computer. 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`.

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
```

3.2.6.1 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");
```

3.2.6.2 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 }
```

3.2.6.3 Host-IO via the xCONNECT link (xSCOPE)

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 (xLINK) 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 xSCOPE 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 xSCOPE.

To use the xSCOPE protocol over xLINK, the following should be provided in a source file from which the target program is built:

```
#include <xscope.h>

// This user-implemented function will be called just before main()
void xscope_user_init() {
  // This enables using xSCOPE for write syscalls (such as when using printf)
  xscope_config_io(XSCOPE_IO_BASIC);
}
```

For full documentation of the xSCOPE runtime library, see *xSCOPE target library*.

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 as a static constructor)
2. The target program must be built with xSCOPE support, either by passing `xcc -fxscope` or an *xSCOPE config file* to xcc
3. One of the `--xscope` options must be supplied to *xrun* or the `xgdb` `connect` command.

3.2.7 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 called 'probes' are defined in the target program and each appears as a separate waveform trace.

The xLINK is always used to transport the data sent from the target.

xSCOPE 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.

Lossy or lossless data delivery may be selected using `xscope_mode_lossless()` and `xscope_mode_lossy()`.

The APIs are defined in *xscope.h*. An example to emit float and integer data for two probes is:

```
void xscope_user_init(void) {
    // Register 2 probes. The first implicitly has ID 0, the second has ID 1.
    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 target program must be built with xSCOPE support, either by passing `xcc -fxscope` or an *xSCOPE config file* to xcc

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

3.2.8 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 semihosting APIs (such as printf) 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. See *xSCOPE host library* for more information.

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>
```

3.2.8.1 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()`*

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.

3.2.8.2 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.

3.2.8.2.1 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
```

3.2.8.2.2 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
```

3.2.8.2.3 Host program

Listing 3.1: host.c

```
#include <signal.h>
#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)
{
  // Handle the registration of the probe with the id 'id'
  // For this example, we would expect a registration for:
  // - Probe id=0, name = "V"
  // - Probe id=1, name = "I"
  // See the xscope_register call in the target file below
}

/* 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 probe number used by the target in the xSCOPE API call */
    switch (id) {
      /* handle each probe id */
      case 0:
      /* do something for probe 0 */
      break;
      case 1:
      /* do something for probe 1 */
      break;
```

(continues on next page)

(continued from previous page)

```
    }
    /* Optional: 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 volatile int running;

/* Called when the target program terminates */
static void sigint_handler(int) {
    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);
    signal(SIGINT, &sigint_handler);

    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;
}
```

3.2.8.2.4 Target program

Listing 3.2: 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 3.3: test.c

```
void xscope_user_init(void) {
    xscope_register(2,
        XSCOPE_CONTINUOUS, "V", XSCOPE_INT, "mV", /* Probe 0 */
        XSCOPE_CONTINUOUS, "I", XSCOPE_INT, "mA", /* Probe 1 */
    );
}

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

    xscope_int(0, 45);              /* Send 45 to probe 0 */

    // 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(1, 578);             /* Send 578 to probe 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);
        }
    }
}
```

3.2.8.3 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
```

3.2.8.4 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
```

See *xSCOPE performance figures* for benchmarks on xSCOPE performance.

3.2.9 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.

3.2.9.1 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.

3.2.9.2 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
```

3.2.9.3 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" -ex "continue" 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.

3.2.10 Target errors and warnings

3.2.10.1 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()`

```
XScope connection was requested with '--xscope', but support does not appear to be
compiled into program (or symbols are missing). JTAG IO will be used. See -fxscope compiler option.
```

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

3.2.10.2 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 …
```

3.2.11 Command examples

3.2.11.1 XGDB examples

In the following examples *xgdb commands* are often supplied as a sequence of `-ex` options on the `xgdb` command line. But the 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.

3.2.11.1.1 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.

3.2.11.1.2 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
```

3.2.11.1.3 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 it 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.

3.2.11.1.4 Attaching to a running target

It is possible to perform symbolic debugging on a target which already has an application running. Such as if it has booted from flash, or if XRUN or XGDB had previously started a program then detached. The *attach* command can be used:

```
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. XSCOPE communication will not function (See *xSCOPE FAQ*).

3.2.11.1.5 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 --gprof --adapter-id pr1V0_15" -ex "continue"  led-flash.xe
```

3.2.11.2 xrun examples

3.2.11.2.1 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` command `connect` or to the `xflash` option `--adapter-id <>`
- The type of target connected to the xTAG. `XS2A` indicates an XCORE-200 XS2A architecture and `XS3A` indicates 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        XS3A[0]
  1     XMOS XTAG-4             BEJMRJ7V        XS2A[0]
  2     XMOS XTAG-3             KAKqyceA        XS2A[0]
  3     XMOS XTAG-4             pr1v0_20        XS3A[0]
```

A device is described as “In Use” if a currently running host tool 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” is 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.

3.2.11.2.2 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. They will be resumed after the state has been dumped (although, if they have hit an exception, xrun will not take them out of the exception.)

3.2.11.2.3 Reboot an xTAG which fails to respond

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

```
$ xgdb -ex "connect --xtag=reboot"
```

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

3.3 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*.

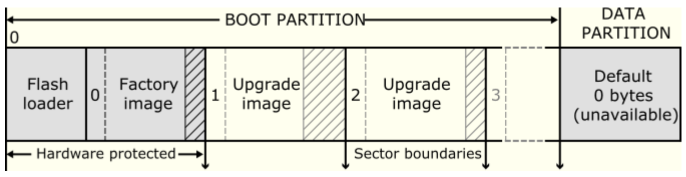


Fig. 3.1: Flash format diagram

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.

3.3.1 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> a.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.

3.3.2 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 a.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.

3.3.3 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.

3.3.3.1 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 */

fl_QSPIPorts QSPI = {
  PORT_SQI_CS,
  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));
 }
```

(continues on next page)

(continued from previous page)

```
 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 `fl...()` 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.

3.3.3.2 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 main.xc target.xn -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 main.xc target.xn -lquadflash -o latest-release.xe
   ```

   XCC compiles your program and links it against libquadflash.

2.
   ```
   $ xflash --upgrade <version-id> 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.

3.3.3.3 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 device are:

1. Include `flash.h` instead of `quadflash.h`
2. Replace the `fl_QSPIPorts` structure with an instance of `fl_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 */

fl_SPIPorts SPI = {
  PORT_SPI_MISO,
  PORT_SPI_SS,
  PORT_SPI_CLK,
  PORT_SPI_MOSI,
  XS1_CLKBLK_1
};

int upgrade(chanend c, int usize) {

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

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

 fl_connect(SPI);

...
```

Build for SPI, linking against libflash:

```
$ xcc main.xc target.xn -lflash -o first-release.xe
```

3.3.4 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;
}
```

3.3.4.1 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 loader.xc -o loader.o
   ```

   XCC compiles your functions for image selection, producing a binary object.

2.
   ```
   $ xflash a.xe --loader loader.o
   ```

   XFLASH writes a flash image containing the custom loader and factory image to the specified file.

3.3.4.2 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.

3.3.5 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

  ...
}
```

3.4 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.

3.4.1 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*.

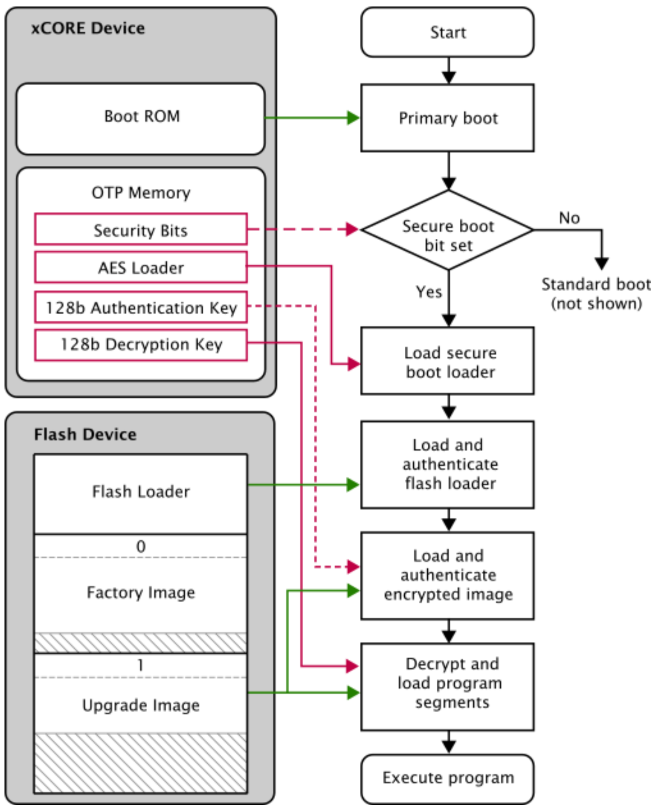


Fig. 3.2: Secure boot procedure used with the AES Module

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.

3.4.2 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 ID Devices
   -- ---- ---------- -------
   ```

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> a.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
```

3.4.3 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 a.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 ID Devices
   -- ---- ---------- -------
   ```

2.
   ```
   $ xflash --id <id> --target-file target.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.

3.4.4 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 ID Devices
   -- ---- ---------- -------
   ```

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.

3.4.5 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.

3.5 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.

3.5.1 Libflash device ID

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

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

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

3.5.2 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.

3.5.3 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 3.2: Table 4 of M25P10-A datasheet

Instruction	Description	One-byte instruction code		Address byte	Dummy bytes	Data bytes
WREN	Write Enable	0000 0110	06h	0	0	0
WRDI	Write Disable	0000 0100	04h	0	0	0
RDID	Read Identification	1001 1111	9Fh	0	0	1 to 3
RDSR	Read Status Register	0000 0101	05h	0	0	1 to infinity
WRSR	Write Status Register	0000 0001	01h	0	0	1
READ	Read Data Bytes	0000 0011	03h	3	0	1 to infinity
FAST_READ	Read Data Bytes at Higher Speed	0000 1011	0Bh	3	1	1 to infinity
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 infinity
	Release from Deep Power-down			0	0	0

3.5.4 Clock divider

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

Symbol	Alt.	Parameter	Min	Typ	Max	Unit
f_C	f_C	Clock frequency for the following instructions: FAST_READ, PP, SE, BE, DP, RES, WREN, WRDI, RDSR, WRSR	D.C.		25	MHz
f_R		Clock frequency for READ instructions	D.C.		20	MHz
t_{CH}	t_{CLH}	Clock High time	18			ns
t_{CL}	t_{CLL}	Clock Low time	18			ns

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

3.5.5 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 3.4: Table 5 of M25P10-A datasheet

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

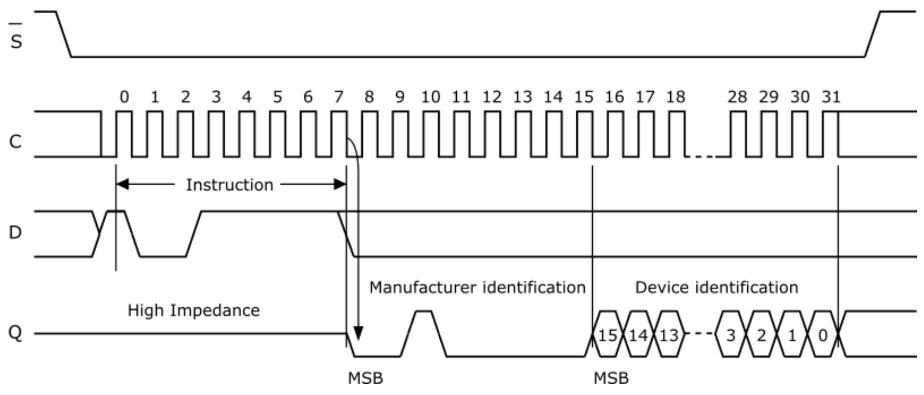


Fig. 3.3: Figure 9 of M25P10-A datasheet

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.

3.5.6 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.

3.5.7 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`.

3.5.8 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 3.5: Table 2 of M25P10-A datasheet

Status Register Content		Memory content	
BP1 bit	BP0 bit	Protected area	Unprotected area
0	0	none	All sectors (four sectors: 0, 1, 2 and 3)
0	1	Upper quarter (sector 3)	Lower three-quarters (three sectors: 0 to 2)
1	0	Upper half (two sectors: 2 and 3)	Lower half (sectors 0 and 1)
1	1	All sectors (four sectors: 0, 1, 2 and 3)	none

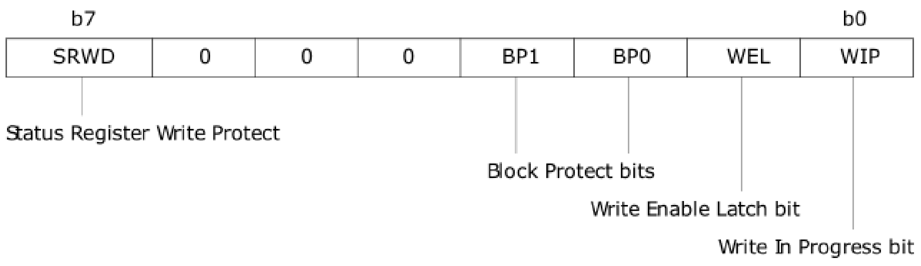


Fig. 3.4: Table 6 of M25P10-A datasheet

3.5.9 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 3.6: Table 7 of F25L004A datasheet.

Symbol	Parameter	Minimum	Units
$T_{\text{PU-READ}}$	V_{DD} Min to Read Operation	10	us
$T_{\text{PU-WRITE}}$	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 */
```

3.5.10 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.

3.5.11 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

On the M25P10-A datasheet, this can be found from the following paragraph in section 5:

> The memory is organized as:
>
> - 131,072 bytes (8 bits each)
> - 4 sectors (256 Kbits, 32768 bytes each)
> - 512 pages (256 bytes each).

The sector sizes is specified as part of a construction: `{a, {b, {c}}}`. For regular sector sizes, the size is specified in *a*. The values of *b* and *c* should be 0.

For irregular sector sizes, the size number of sectors is specified in *b*. The log base 2 of the number of pages in each sector is specified in c. The value of *a* should be 0. An example of a device with irregular sectors is the *AMIC A25L80P. :ref:'add_flash_support_tab2_a25l80p* reproduces the part of this datasheet that provides the sector information.

Table 3.7: Table 2 of A25L80P datasheet

Sector	Sector Size (Kb)	Address Range	
15	64	F0000h	FFFFFh
14	64	E0000h	EFFFFh
13	64	D0000h	DFFFFh
12	64	C0000h	CFFFFh
11	64	B0000h	BFFFFh
10	64	A0000h	AFFFFh
9	64	90000h	9FFFFh
8	64	80000h	8FFFFh
7	64	70000h	7FFFFh
6	64	60000h	6FFFFh
5	64	50000h	5FFFFh
4	64	40000h	4FFFFh
3	64	30000h	3FFFFh
2	64	20000h	2FFFFh
1	64	10000h	1FFFFh
0-4	32	08000h	0FFFFh
0-3	16	04000h	07FFFh
0-2	8	02000h	03FFFh
0-1	4	01000h	01FFFh
0-0	4	00000h	00FFFh

The corresponding entry in the specification file is:

```
SECTOR_LAYOUT_IRREGULAR,
    {0,{20,{4,4,5,6,7,8,8,8,8,8,8,8,8,8,8,8,8,8,8,8}}},
```

3.5.12 Status register bits

```
0x05,                   /* 21. RDSR cmd */
0x01,                   /* 22. WRSR cmd */
0x01,                   /* 23. WIP bit mask */
```

Most flash devices provide instructions to read and write a status register, including a write-in-progress bit mask.

Table 4 of M25P10-A datasheet reproduces the part of the M25P10-A datasheet that documents the RDSR and WRSR commands. The diagram in *Table 6 of M25P10-A datasheet* shows that the WIP bit is in bit position 0 of the SR, resulting in a bit mask of `0x01`.

3.5.13 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 3.8: Quad Enable method

Value	Method	Description	Read-modify-write
0	Legacy libquadflash behavior	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.

3.5.14 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.

3.5.15 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.

3.6 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.

3.6.1 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)

(continued from previous 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.

3.6.2 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:

3.6.2.1 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.

3.6.2.2 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:

3.6.2.2.1 external.c

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

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

3.6.2.2.2 external.h

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

3.6.2.2.3 main.c

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

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

3.6.3 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)

3.6.3.1 LPDDR clock frequency specification

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

3.6.3.1.1 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.

3.6.3.1.2 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_out = (Oscillator x SecondaryPllFeedbackDiv/2) / (SecondaryPllInputDiv x 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>
```

3.6.4 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 repatedly evicted by another logical core and must be re-loaded).

3.7 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.

3.7.1 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.

3.7.2 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` andf `xcore/swmem_evict.h`.

3.7.3 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.

3.7.4 Compiling for software-defined memory

See *compiling for external memory (LPDDR) and software-defined memory*.

3.7.5 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 is flash.

3.7.5.1 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 sofware fill handler, which is triggerred 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 the level 1 cache. The symbol `__swmem_address` must be defined and intialised 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:

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

Listing 3.4: 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_teardown(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_teardown(fill_handle);
}
```

Listing 3.5: 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;
}
```

3.7.5.2 Evict handler example

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

Listing 3.6: 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);
}
```

3.7.6 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 intialiser 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 swap is done by the `--reverse` option to `xflash`.

3.8 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 3.7: 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`.

3.9 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.

3.9.1 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.

3.9.2 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`.

3.10 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 *XOBJDUMP* to explore the contents of an XE file
- How *XRUN* and *XGDB* coordinate the loading and execution of an XE file

3.10.1 Prepare an XE file

First, build a.xe as per the example in *Targeting multiple tiles*.

3.10.2 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: PROGINFO sector
12: XSCOPE sector
13: LASTSEC sector
```

Why are there four ELFs 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 ELFs 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 ELFs because:

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

Note: Single-tile applications on multi-tile targets

If you create the single-tile application:

Listing 3.8: single-tile.c

```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 ELFs will still be produced; two setup ELFs and two application ELFs. 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].

3.10.3 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 ELFs:

```
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 ELFs:

```
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.
```

4 Reference

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

4.1 Command line tools

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

4.1.1 XRUN

4.1.1.1 Synopsis

```
xrun [options]
xrun [options] xe-file
xrun [options] --args xe-file arg1 arg2 .. argn
```

4.1.1.2 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).

4.1.1.3 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.

4.1.1.3.1 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)

(continued from previous page)

```
--    ----                   ----------      -------
0     XMOS XTAG-3            V0JhnXmh        XS2A[0]
1     XMOS XTAG-3            wfF.G58J        XS3A[0]
```

The adapters are ordered by their Adapter IDs.

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

--version
Displays the version number and copyrights.

4.1.1.3.2 Run options

If only one xtag device is present, it is selected 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 `<serial-number>`
Specifies the serial number of the adapter connected to the target hardware.

--verbose
Prints detailed information about what commands are going to be run using *XGDB*, and print all XGDB output to the terminal (without this flag, it is heavily filtered). This can be especially useful when diagnosing a problem, or to provide a starting point for moving from using XRUN to XGDB for an advanced use case.

--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 $25/(n+2)$ 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. See *xSCOPE* for more info on xSCOPE.

`--xscope`

Enables an xSCOPE connection to the target for fast io and data collection.

`--xscope-file` `<filename>`

Specifies the filename for xSCOPE data collection.

`--xscope-port` `<ip:port>`

Specifies the IP address and port to host a server for realtime data collection.

`--xscope-limit` `<limit>`

Specifies the record limit for xSCOPE data collection. After the limit is hit, no further records from probes will be processed.

`--xscope-io-only`

Alias for `--xscope-limit 0`. Probes will not be processed: xSCOPE will only be used for io.

4.1.1.4 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.

4.1.2 XSIM

4.1.2.1 Synopsis

```
xsim xe-file
xsim [options]
```

4.1.2.2 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.

4.1.2.3 Options

4.1.2.3.1 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.

4.1.2.3.2 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.

4.1.2.3.3 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();
   p1 <: 1;
   p1 <: 0;
   _traceStop();

   p1 <: 1;
   p1 <: 0;

   return 0;
}
```

4.1.2.3.4 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*.

4.1.2.3.5 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'
```

4.1.2.3.6 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>

Limts the xSCOPE output records to the given number.

For example, the following will run xSIM with xSCOPE enabled in offline mode:

```
xsim app.xe --xscope "-offline xscope.xmt"
```

For example, the following will run xSIM with xSCOPE enabled in reatime mode:

```
xsim app.xe --xscope "-realtime localhost:12345"
```

4.1.3 XCC

4.1.3.1 Synopsis

```
xcc [-c|-S|-E] [-g] [-O<level>]
    [-W<warn>...]
    [-I<dir>...] [-L<dir>...]
    [-D<macro>[=<defn>]]
    [-f<option>...]
    [-target=<platform>]
    [-mcmodel=<model>]
    [-o <outfile>] infile
```

Only the most useful options are listed here; see below for the remainder.

4.1.3.2 Description

XCC is the compiler driver tool and drives the underlying XC, C and C++ language compilers, the assembler and the linker.

XCC will normally run all the steps to take a set of source files and produce an executable file (.xe), which may be run on the simulator (using `xsim`), run on a target (using `xrun`) or converted to a format to be stored on flash memory (using `xflash`).

The linker is invoked multiple times to produce an Elf executable for each tile which are combined into the final *XE file*.

Options such as *-c*, *-S* and *-E* prevent certain compilation and linkage steps running, and instead produce an intermediate output.

A target platform description file conforming to the *XN specification* must be specified. This is done via *XCC_DEFAULT_TARGET*, *-target* or listed as one of the input files.

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

4.1.3.3 Options

4.1.3.3.1 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 4.1: File extensions recognized by XCC and their meaning

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

-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.

-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
```

-mcmodel=<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, uart is not supported). The XN file of the target must contain an xSCOPE link. The option should be passed both when compiling and linking. This is an alternative to passing an *xSCOPE configuration file*, and is equivalent to a config file containing only `<xSCOPEconfig ioDest="link" ioMode="none" enabled="true">`. Thus, this flag can be used to enable xSCOPE and the *xSCOPE target library* without writing an xSCOPE configuration file.

-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.

4.1.3.3.2 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.

4.1.3.3.3 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 seperated 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".

4.1.3.3.4 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.

4.1.3.3.5 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 overriden with *-MF*.

-MMD

The same as *-MD* expect 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.

4.1.3.3.6 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 `lib`*`library`*`.a`.

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

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

-nostartfiles

Do not link with the system startup files.

-nodefaultlibs

Do not link with the system libraries.

-nostdlib

Do not link with the system startup files or system libraries.

-s

Removes all symbol table and relocation information from the executable.

-default-clkblk `<clk>`

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

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

-Wm,<option>

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

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

-Xmapper `<option>`

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

-report

Prints a summary of resource usage.

4.1.3.3.7 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*.

4.1.3.4 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 seperater 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.

4.1.4 XOBJDUMP

4.1.4.1 Synopsis

```
xobjdump [options] xe-file
```

4.1.4.2 Description

The xobjdump tool is used to examine and manipulate the contents of *XMOS Executable (XE)* files.

4.1.4.3 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.

4.1.4.3.1 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`**.

4.1.5 XGDB

4.1.5.1 Synopsis

```
xgdb [options] xe-file
xgdb [options] --args xe-file arg1 arg2 .. argn
```

4.1.5.2 Description

XGDB is an extended version of the open-source GDB debugger. The extensions allow XGDB to debug multi-tile XCore applications in the form of *XE files*.

XGDB communicates with the target via XGDBSERVER, which is able to attach to multiple types of target including:

- Real hardware via an xTAG debug adapter
- Simulated hardware provided by XSIM

XGDBSERVER can be started from within the XGDB environment using either the *`connect`* or *`attach`* command.

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.

For a quick-start guide on how to use the tool, see *Debugging with XGDB*.

4.1.5.3 Concepts

XGDB uses many features of GDB that may not be familiar, in particular the “multi-inferior” feature, which allows simultaneous debugging of multiple tiles. This section aims to give an overview of these concepts and some more advanced topics.

4.1.5.3.1 Inferiors & threads

In an XMOS system, there are multiple tiles, which are independent processors with their own independent memory and address space. Each of these tiles contains multiple hardware threads (logical cores), which all share the same memory and other resources on the tile they belong to.

In GDB, each tile is represented as a unique "inferior". An inferior in GDB is a program which runs in its own context and address space. Inferior numbers start at 1 and increment by 1 for each new inferior. You can switch between different inferiors using the *`inferior`* or *`tile`* command (`inferior` is a standard GDB command, `tile` is an XMOS extension).

Note that there is no particular mapping between a tile ID and the inferior number. The tile ID is a property of the hardware system - each tile has a unique ID. XGDB assigns an incrementing, unique inferior number to each new inferior it sees. Also noteworthy is that tile IDs start at zero, whereas inferior numbers start at 1. This leads to the situation that usually tile[0] is inferior 1, tile[1] is inferior 2, etc.

All inferiors in the system can be listed using the *`info inferiors`* command.

Each logical core in a system is represented by a thread in GDB. Threads in GDB belong to an inferior, and their thread ID represents this. For instance, the first thread of inferior 1 will have thread id 1.1. The third thread of inferior 2 will have thread id 2.3. You can switch between different threads in the system using the *`thread`* command. Again, there is no guaranteed correspondence between the thread id in GDB and the logical core index on the hardware, but usually tile[0] core[0] is thread 1.1, tile[0] core[1] is thread 1.2, etc.

All threads in the system can be listed using the *`info threads`* command.

4.1.5.3.2 Targets, connections and XGDBSERVER

The 'target' is the platform on which the program runs. In the case of XGDB, the target is usually either actual hardware, or the XSIM simulator. XGDB connects to the target by using the 'remote' target connection mechanism to *XGDBSERVER*, using a well-defined protocol over a TCP pipe between the 2 programs.

The 'target program' is the program which is running on the target.

XGDBSERVER then handles actually connecting to the hardware, or starting up the simulator, so that GDB has a target to control. You may recognise this pattern from other ecosystems which use other gdbserver implementations such as OpenOCD.

XGDB provides helper commands *`connect`* and *`attach`*, which start up an instance of XGDBSERVER as a child process, then immediately connect to it. This is very convenient for connecting to a target which is plugged into the same host PC, or quickly starting the simulator.

For more advanced situations, it may be more desirable to run XGDB and XGDBSERVER separately, such as:

- Running XGDBSERVER in a separate terminal window, so the program output is separate from the XGDB interface.
- Connecting to a target board via an xTAG which is plugged in to a different host machine.

Note that all system calls, including print messages are handled by XGDBSERVER. This means, for example, that if the target program creates a file, that file will be created on the system running XGDBSERVER, relative to the working directory of the XGDBSERVER program.

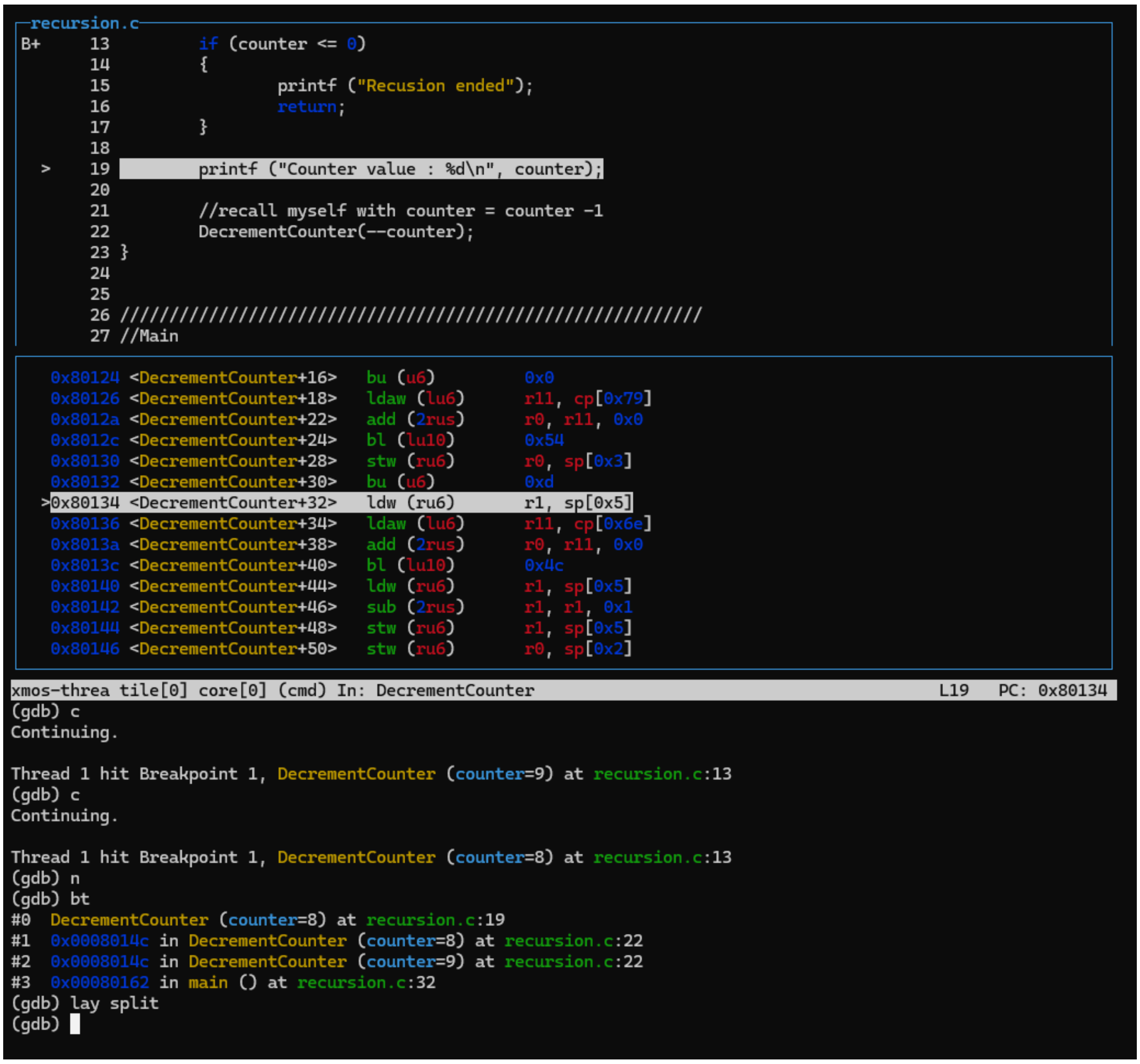
If running XGDBSERVER separately, you must specify the TCP port XGDBSERVER should host the server at using the *`xgdbserver --port`* option. Then XGDB can be used to connect to that server using the `target remote` command.

If using the *`connect`* or *`attach`* command, note that XGDBSERVER is started as a subprocess, so care must be taken when forcefully killing it. If using SIGKILL on XGDB, for instance, it may take some additional time for XGDBSERVER to notice that it has shut down and clean up after itself. Using SIGTERM (on POSIX) or `quit` allows XGDB to properly clean up after itself before terminating. This also applies to cleaning up temporary files.

4.1.5.3.3 TUI as an visual interface

XGDB includes a text user interface (TUI), which provides a more visual, IDE-like experience when debugging target programs.

It can display several different windows, such as "source", "commands", "disassembly", "registers". Custom windows can also be created: see the official GDB TUI documentation for more information.



```
┌recursion.c─────────────────────────────────────────────────────────────
│B+     13         if (counter <= 0)
│       14         {
│       15                 printf ("Recusion ended");
│       16                 return;
│       17         }
│       18
│  >    19         printf ("Counter value : %d\n", counter);
│       20
│       21         //recall myself with counter = counter -1
│       22         DecrementCounter(--counter);
│       23 }
│       24
│       25
│       26 ///////////////////////////////////////////////////////////
│       27 //Main

    0x80124 <DecrementCounter+16>   bu (u6)         0x0
    0x80126 <DecrementCounter+18>   ldaw (lu6)      r11, cp[0x79]
    0x8012a <DecrementCounter+22>   add (2rus)      r0, r11, 0x0
    0x8012c <DecrementCounter+24>   bl (lu10)       0x54
    0x80130 <DecrementCounter+28>   stw (ru6)       r0, sp[0x3]
    0x80132 <DecrementCounter+30>   bu (u6)         0xd
   >0x80134 <DecrementCounter+32>   ldw (ru6)       r1, sp[0x5]
    0x80136 <DecrementCounter+34>   ldaw (lu6)      r11, cp[0x6e]
    0x8013a <DecrementCounter+38>   add (2rus)      r0, r11, 0x0
    0x8013c <DecrementCounter+40>   bl (lu10)       0x4c
    0x80140 <DecrementCounter+44>   ldw (ru6)       r1, sp[0x5]
    0x80142 <DecrementCounter+46>   sub (2rus)      r1, r1, 0x1
    0x80144 <DecrementCounter+48>   stw (ru6)       r1, sp[0x5]
    0x80146 <DecrementCounter+50>   stw (ru6)       r0, sp[0x2]

xmos-threa tile[0] core[0] (cmd) In: DecrementCounter                        L19   PC: 0x80134
(gdb) c
Continuing.

Thread 1 hit Breakpoint 1, DecrementCounter (counter=9) at recursion.c:13
(gdb) c
Continuing.

Thread 1 hit Breakpoint 1, DecrementCounter (counter=8) at recursion.c:13
(gdb) n
(gdb) bt
#0  DecrementCounter (counter=8) at recursion.c:19
#1  0x0008014c in DecrementCounter (counter=8) at recursion.c:22
#2  0x0008014c in DecrementCounter (counter=9) at recursion.c:22
#3  0x00080162 in main () at recursion.c:32
(gdb) lay split
(gdb) █
```

Fig. 4.1: Example TUI session

To switch focus between different windows, use `Ctrl-X` followed by `O`. This allows, for instance, changing between using the arrow keys to move up and down the source code, and using the arrow keys to scroll through command history. It is not currently possible to scroll through command output history.

To enable the TUI, you can use the `tui enable` command. Or the `layout src/layout split` commands can be used to simultaneously enable the TUI and jump into a specific layout. To disable the TUI and return to normal GDB, you can use `tui disable`.

Sometimes to prompt GDB to actually print the source of the current location, it may be necessary to use the `list` command, or select a stack frame for which the source is actually available. For instance, this may happen if GDB does not have debug line information for the current program counter.

Note: On Windows, GDB does not automatically resize when the terminal is resized. Use `Ctrl-L` to refresh the screen and resize to the correct dimensions. This will also clear any other non-TUI output that has been written to the terminal.

4.1.5.3.4 Using XGDB in scripts

XGDB is primarily a tool for interactive debugging, but it can also be used from scripts in order to run programs and debug them according to a script. See the following section for guidance on how to script XGDB itself.

When using XGDB in a script, it is essential to pass the *`xgdb --batch`* option. This option means that XGDB will terminate if the script encounters an error, or once all commands have been executed. It will also return an error code if it terminates unexpectedly.

The *`xgdb --return-child-result`* option is also sometimes useful. This will make XGDB return the exit code of the target program as its own exit code when it terminates. If the script terminates before the program is complete, then XGDB will return 1.

The options *`xgdb -ex`* and *`xgdb -x`* allow commands or command files to be passed to XGDB on the command line. This allows XGDB to be controlled in a non-interactive way.

See *Scripted debugging* for an example.

4.1.5.3.5 Scripting GDB

XGDB can be extended and customised. It can be reconfigured to user preference, or expanded with additional commands & functionality as required.

The simplest method of customising XGDB is via the `.gdbinit` file which can be created in the `$HOME` directory. The `.gdbinit` file will automatically be read and executed every time that XGDB is started. Some configuration commands that may be desired are:

- `set confirm off`: Disable XGDB prompting for confirmation when running certain commands.
- `set pagination off`: Disable XGDB pausing midway for confirmation when printing long command output.
- `define`: Define simple procedures which can be run as commands.
- `set history save on` and `set history size 500`: Enable command history to be saved between sessions. This creates a `.gdb-history` file in each directory you start XGDB in.
- `set auto-load local-gdbinit` and `set auto-load safe-path /`: Enable project-specific `.gdbinit` files to be created and automatically run if XGDB is started in the same directory as them.

Note: On Windows, the `HOME` environment variable is not set by default, so you will need to create and define a `HOME` environment variable to a suitable path in order to use a `.gdbinit` file.

On top of the initialisation files, it is possible to arbitrarily source a command file into the current session using the `source` command.

XGDB contains a built-in Python interpreter (Currently Python version 3.8), which can be used for in-process scripting. This enables extensive scripting capability far beyond what is possible just using GDB commands. See the Official GDB documentation for more information, but to summarise, the Python interpreter can be used for many extensions, such as:

- Implementing custom commands
- Implementing “pretty printers” which nicely print custom datatypes
- Implementing custom windows for the TUI

- Advanced access to the inferior & target state
- Running arbitrary Python scripts

4.1.5.4 Options

The options for XGDB are 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.

`-x` `<command file>`

Execute all of the commands in the given command file. Lines prepended with # will be ignored.

`--batch`

Run XGDB in a manner more suitable for use in scripts. XGDB will exit after all commands specified on the command line have been run, and return an erroneous exit code if anything fails.

`--return-child-result`

Configure XGDB to return the exit code of the target program as its own exit code when it terminates. If the script terminates before the program is complete, then XGDB will return 1. This can be particularly useful when running XGDB from a script.

`--help`

List all available command line options

4.1.5.5 XMOS XGDB commands

The following commands are all provided as part of the XGDB extension to GDB. Available commands can be listed from within the XGDB environment using either the `help` or `apropos` commands.

For some examples of how to use the commands in practice, see *XGDB examples*.

Some of these commands are already GDB commands, but have been overridden to work better with XMOS multi-inferior devices and XE files. This 'override' behaviour can be turned off for advanced use cases, see `set cosmetic-command`

`listdevices`

List and enumerate all xTAG adapters connected to your PC.

`connect`

Usage: `connect [<options>...]`

Connects to a target, resetting it to kill the running program first. If no options are supplied, it will connect to the single xTAG adapter connected to your PC.

This command starts a local instance of XGDBSERVER to interact with the target. For additional options which can be passed to connect, see *XGDBSERVER Options*, as any arguments to `connect` will be passed to the local instance of XGDBSERVER.

Common options:

- `connect --adapter-id` Select a specific xTAG to connect to. See `listdevices`.
- `connect --xscope` Enable the use of xSCOPE.
- `connect -s` Connect to XSIM instead of real hardware.

`attach`

Usage: `attach [<options>...]`

Connects to a target without resetting it - although it is halted and put in debug mode for inspection.

This command is useful in order to debug a program that is already loaded on the device, whereas *`connect`* is more appropriate when loading and running a new program.

This command is identical to *`connect`* other than it doesn't reset the target, so the same *XGDBSERVER Options* can be passed to it.

`detach`

Detach from a target and allow it to continue running, free from the debugger. This will detach every node and every tile in the system, allowing them to continue running without the debugger.

`load`

Load the binary (and run any setup ELFs as necessary) but do not start it. This will reset the target.

It is usually more appropriate to use one of *`run`*, *`start`* or *`starti`*, which are higher level GDB commands with additional functionality. They all implicitly run `load` to re-initialise the target and program.

`starti`

Load the binary but halt on the first instruction. This will reset the target.

Any arguments passed to this command will be forwarded to the target program when it is started. A program must be built with *`xcc -fcmdline-buffer-bytes`* in order to support command line arguments.

This command is functionally equivalent to *`load`* if arguments are not passed.

`run`

Load the binary and start it. This will reset the target.

Any arguments passed to this command will be forwarded to the target program when it is started. A program must be built with *`xcc -fcmdline-buffer-bytes`* in order to support command line arguments.

`start`

Load the binary and start it, halting at the start of main. This will reset the target.

Any arguments passed to this command will be forwarded to the target program when it is started. A program must be built with *`xcc -fcmdline-buffer-bytes`* in order to support command line arguments.

`tile`

Usage: `tile <tile number>`

Change the focus of subsequent commands to tile `<tile number>`. This is an alternative to the default GDB command *`inferior`*.

Running the command with no arguments will print the current inferior description.

`tile apply all`

`tiaa`

Usage: `tile apply all <command>`

Apply a command to every tile in the system, in the same spirit as the default GDB `thread apply all` command. Example:

```
(gdb) tiaa p $pswitch_device_id0
Inferior 1 (tile[0] core[0]):
$1 = [ version=0 revision=4 node=0 pid=0 ]

Inferior 2 (tile[1] core[0]):
$2 = [ version=0 revision=4 node=0 pid=1 ]
```

This command will work whether or not a target is actually connected (only static analysis is possible when there is no target connected).

through

Step through a channel operation, halting on the receiving thread on the other side of a channel.

This command causes GDB to continue until it hits an `OUT` instruction on a `chanend`. When it does, it locates the other side of the channel, and halts on it once that thread reaches the corresponding `IN` instruction.

monitor

Usage: `monitor <command>`

Once connected, the `monitor` command can be used to access *XGDBSERVER's monitor commands*. These are typically more advanced/niche commands which interact directly with the target. The command `monitor help` will print all the available monitor commands for the current session.

4.1.5.5.1 Standard GDB commands

These are documented further in the actual GDB documentation, or by using the `help` command, but are included here because they are referenced from this document.

help

Usage: `help [<command>]`

Print help information. If a command is specified, then print help information for that command.

apropos

Usage: `apropos <query>`

Search all commands for the query, and print the short help page for anything that matches.

inferior

Usage: `inferior <inferior number>`

Change the focus of subsequent commands to *inferior* `<inferior number>`. This is an alternative to the *`tile`* command.

Running the command with no arguments will print the current inferior description.

info inferiors

List the current inferiors in the system, and information about them (such as if they are running). Example:

```
(gdb) info inferiors
  Num  Description       Connection           Executable
  1    tile[0]           1 (remote :39211)    /tmp/.xgdb/593782/0/MultiNodeBasic.xe.i0_n0_t0.elf
* 2    tile[1]           1 (remote :39211)    /tmp/.xgdb/593782/0/MultiNodeBasic.xe.i1_n0_t1.elf
```

The * denotes the current inferior.

thread

Usage: `thread <thread id>`

Change the focus of subsequent commands to *thread* `<thread id>`.

Running the command with no arguments will print the current thread description.

info threads

List the current threads in the system, across all inferiors. Example:

```
(gdb) info threads
  Id   Target Id              Frame
  1.1  tile[0] core[0] (xsim) run (inC=3842, outC=258) at snip.xc:8
  1.2  tile[0] core[1] (xsim) 0x000802a8 in __main__main_tile_0_combined_tile_0_u1
↪(frame=0xfff24) at snip.xc:30
```

(continues on next page)

(continued from previous page)

```
  1.3  tile[0] core[2] (xsim) 0x000802d2 in __main__main_tile_0_combined_tile_0_u2
↪(frame=0xfff24) at snip.xc:31
  2.1  tile[1] core[0] (xsim) start (inC=69378, outC=65538) at snip.xc:16
  2.2  tile[1] core[1] (xsim) 0x00080126 in run (inC=66306, outC=66562) at snip.xc:6
* 2.3  tile[1] core[2] (xsim) run (inC=66818, outC=67074) at snip.xc:7
  2.4  tile[1] core[3] (xsim) run (inC=67330, outC=67586) at snip.xc:8
```

The `*` denotes the current thread.

`call`

Usage: `call <function expression>`

The call command can be used to call a function on the current thread on the target. It will set up the program counter to run the function, run it, then restore the previous state and print the result.

> **Warning:** When using this function, it is important to ensure that there is enough stack space for the function to run, as the call command does not check this, and the XTC compilation tools allocate as little space is possible for each thread's stack. If there is not enough stack space, it will silently overrun the stack, potentially corrupting other data.

`break`

Usage: `break <expression> [inferior <num>] [if <condition>]`

Set a software breakpoint at the location expressed by 'expression'. An unlimited number of these can be placed, but they can not be placed in ROM code, and will be overwritten if the system boots from flash or uses self-modifying code.

The `if` option can be used to only halt the system if a certain condition is met.

The `inferior` option can be used to only set the breakpoint on a single inferior if it would otherwise be set on multiple.

Note that both of these may cause the system to halt momentarily while the condition is evaluated, before being resumed. Realtime behaviour may be broken.

For example:

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

`hbreak`

Identical to *`break`*, except it uses a hardware breakpoint rather than injecting a software breakpoint. A maximum of 3 of these can be used per tile.

`watch`

Usage: `watch [-location] <expression>`

Set a watchpoint at a location described by the given expression. A watchpoint (also known as a data breakpoint) will halt the target when the target location is written to. On xCORE, up to 4 hardware watchpoints are supported at a time (any more and they can be emulated with single stepping, which is extremely slow).

For example, to watch for changes in the `my_counter` variable:

```
(gdb) watch my_counter
```

Using the `-location` flag will mean that the expression will be treated as an address, rather than an object. The following 2 examples are equivalent:

```
(gdb) watch *((char *) 0x80250)
(gdb) watch --location 0x80250
```

To watch more than a single word, it is possible to watch an address which is cast to an array of the correct size:

```
(gdb) watch (int[50]) *0x80250
```

`awatch`

Identical to *`watch`*, except it will halt the program on all accesses to the watched location including read (load), not just write (store) accesses.

4.1.5.5.2 Advanced commands

These commands should not be required for normal use, but may be useful for certain situations.

`set cosmetic-command`

`get cosmetic-command`

Usage: `set cosmetic-command <on/off>` or `get cosmetic-command`

Cosmetic commands are the commands which are standard GDB commands, that XMOS has overridden with new behaviour. This 'override' behaviour can be enabled or disabled. In most cases, the overrides are to make the commands feel more natural in a multi-tile XMOS system. For example, the override behaviour for *`load`* means that all tiles in the system are configured and loaded, not just the currently selected one.

The `set/get cosmetic-command` commands either sets or gets the current state of cosmetic commands, which override certain built-in GDB commands with commands that are more appropriate for use with XMOS multi-tile systems and *XE files*.

Cosmetic commands are enabled by default in XGDB (the overrides are active).

If they are disabled by the user, it is still possible to access the xmos-override versions of the commands by using the always-available name:

Table 4.2: Cosmetic command overrides

Cosmetic override	Always-available name	Description of XMOS changes
`file`	`xefile`	Allows opening of *XE files*, creating a new inferior for each tile in the system.
`load`	`xeload`	Loads every tile in the system, and uses the 'setup elfs' from the XE file in order to initialise the system.
`run`	`xerun`	Run every tile in the system, not just the currently selected one.
`start`	`xestart`	Start every tile in the system, not just the currently selected one.
`starti`	`xestarti`	Start every tile in the system, not just the currently selected one.
`detach`	`xedetach`	Detach every tile in the system, not just the currently selected one.
`attach`	(none)	Behave the same as the *`connect`* command, without resetting the target.

Note that the cosmetic commands only override default behaviour when working with *XE files*. If working directly with ELF files instead of XE files, then XGDB will act according to traditional GDB behaviour and these overrides will not apply. This is out-of-scope of this manual.

`set xcore-catchpoints`

get xcore-catchpoints

Usage: `set xcore-catchpoints <on/off>` or `get xcore-catchpoints`

Either set or get the current state of the xCORE exception catchpoints. When enabled, a breakpoint is installed on the target, which will halt the program and print the error whenever there is an unhandled xCORE exception.

A 'catchpoint' is a type of breakpoint that occurs on a certain kind of program event, rather than on a specific address, symbol or function.

These are enabled by default in XGDB.

It may be useful to disable this in order to prevent the whole system from being halted once an unhandled exception is hit (which would disrupt real-time behaviour).

set schedule-multiple

get schedule-multiple

Usage: `set schedule-multiple <on/off>` or `get schedule-multiple`

This command affects whether or not *all* inferiors (tiles) are run when using the `continue`, `step` or `next`` commands. If `schedule-multiple` is:

- `on`, then all inferiors will be continued when any of the continue commands are used.
- `off`, then only the currently selected inferior will be resumed when any of the continue commands are used.

For XGDB, the default is `on`.

This command can be useful, for example, if you want to freeze all other tiles while just debugging the currently selected tile. For instance if the secondary tile would create a lot of stdout messages, or drive signals to connected hardware.

See also: *`set scheduler-locking`*.

set scheduler-locking

get scheduler-locking

Usage: `set scheduler-locking <on/off/step>` or `get scheduler-locking`

This command affects whether or not *all* threads (logical cores) are run when using the `continue`, `step` or `next` commands. If `scheduler-locking` is:

- `off`, then all threads will run when any continue command is used.
- `on`, then only the current thread will run when any continue command is used.
- `step`, then when stepping only the current thread will run, but for any other continue command all of the threads will be run.

For XGDB, the default is `off`.

Note that changing this may result in threads becoming unable to progress when inter-thread synchronising instructions are called.

This command can be useful, for example, if you want to freeze all other threads while just debugging the currently selected thread. For instance if debugging and stepping through an algorithmic part of the code that does not interface with other tiles, it can be convenient to just not run them.

See also: *`set schedule-multiple`*.

4.1.5.6 Environment

NO_COLOR

If the NO_COLOR environment variable is set to a non-empty value, then all colour output of XGDB will be suppressed, reverting to plain monochrome font. Note that the suppression of colour output is automatic if XGDB detects that it is not outputting to a TTY (such as when it is being piped to a file).

To change the shade of the colours that GDB is using, check the settings of your terminal, as GDB is using ANSI escape sequences to ask the terminal to display from a palette of colours, which can usually be reconfigured by the user.

4.1.6 XGDBSERVER

4.1.6.1 Synopsis

```
xgdbserver [options]
```

4.1.6.2 Description

XGDBSERVER is an XMOS implementation of a GDB server (also known as a stub or RSP server). XGDB-SERVER hosts a server which handles communication with the debug target. XGDB can then be used to interface with the XGDBSERVER server.

Tip: For most cases, it is more convenient to start XGDBSERVER via the XGDB `connect` or `attach` commands, which start and connect to an instance of XGDBSERVER from within the XGDB environment.

For advanced use cases, it is possible to start an instance of XGDBSERVER independently and attach to it with the XGDB `target remote` command.

4.1.6.3 XGDBSERVER Options

XGDBSERVER accepts many different options, which can be passed either directly to the `xgdbserver` program, or to the XGDB `connect` or `attach` commands, which start up a local XGDBSERVER.

4.1.6.3.1 General options

-h, **--help**

Print help information for all of the available options.

--version

Display the version of XGDBSERVER.

-b <backend-type>, **--backend** <backend-type>

Backend type to use to connect to the target. Valid values:

- `hardware` (default) Connect to hardware via an xTAG.
- `simulator` Connect to an XSIM instance via simulator backdoors.
- `debugrom-simulator` Connect to an XSIM instance, using a fully simulated (slower) debugrom interface.

The backend-type string may be shortened if non-ambiguous.

-s [--simulator]
: Shortcut for `--backend=simulator`

--xe-file <filepath>
: Path to the XE file to debug. This is required when using a simulator backend, but note that when using *`connect`* or *`attach`*, XGDB will provide this automatically. Note that this only uses the XE file for its system specification, it doesn't actually load its components into simulated memory.

: **Note:** When starting XGDBSERVER via the XGDB *`connect`* or *`attach`* commands, this option will automatically be provided, so the user does not need to repeat it.

--args <args>
: Single string containing all of the arguments to pass to the target program when it starts. This string will be split into individual arguments by POSIX convention.

: A program must be built with *`xcc -fcmdline-buffer-bytes`* in order to support command line arguments. See the compiler documentation for more info.

: This is equivalent to the *`monitor args set`* command.

: **Note:** From a shell, if multiple arguments are to be passed, wrap them in single quotes so that they are not treated as arguments to XGDBSERVER itself.

--trace
: Enable maximum log level (shortcut for `--log-level=trace`).

--log-file <filepath>
: Send all log output to a file instead of to stderr. Target program output will not be redirected.

--log-level <logspec>
: Set the log level for XGDBSERVER for internal diagnostic purposes. Log level can be set either broadly, or on a per-log-channel basis. Log levels are specified as pairs, while the global log level (lowest priority) is by itself. The syntax is like `rsp=debug,xdbg=trace,warning`, which means:

- `rsp` messages of `debug` or greater will be printed (including sub-channels, like `rsp::packet`)
- All `xdbg` messages will be printed
- Any other channel will have messages printed if they are `warning` or greater.

: The available log levels (in order of verbosity) are: `none`, `error`, `warning`, `info`, `debug` and `trace`. The available channels are dynamic and can be seen in the output log. Some primary log channels are `rsp`, `xdbg`, `xdbg::usb`, `service`.

4.1.6.3.2 Connection options

These options allow specifying how the server part of XGDBSERVER should behave. These options **cannot be used** when starting XGDBSERVER via the XGDB *`connect`* or *`attach`* commands, since XGDB automatically specifies them itself.

-p <portnum>, **--port** <portnum>
: Port to listen on for TCP connections from XGDB. 0 for ephemeral (see *`--portfile`*). Port defaults to 2331 if not specified.

--portfile <filepath>
: XGDBSERVER will write the port number it is using to host the TCP server to this file - useful if an ephemeral port is being used.

--server-timeout <seconds>

Give up waiting for a connection to the server after this many seconds and terminate. The default (0) means no timeout.

4.1.6.3.3 Backend-specific options

These options change the behaviour of the particular 'backend' (see *--backend*) which is in use, but are not necessarily available for use with every backend.

--adapter-id <id>

Specify the xTAG adapter to connect to by its serial number. Use *--list-devices* to get connected device serial numbers. (Supported by: **hardware**)

--adapter-index <index>

Specify the xTAG adapter to connect to by its index in the *--list-devices* list. (Supported by: **hardware**)

--list-devices

List all devices which are available for connection, then exit. (Supported by: **hardware**)

Example:

```
% xgdbserver --list-devices
Available XMOS Devices
----------------------
  ID    Name                    Adapter ID      Devices
  --    ----                    ----------      -------
  0     XMOS XTAG-3             .BT0u0X2        XS3A[0]
  1     XMOS XTAG-4             BEJMRJ7V        XS2A[0]
```

- The `ID` column indicates an index which can be used with the *xgdbserver --adapter-index* option.
- The `Name` column indicates the name of the debug adapter.
- The `Adapter ID` column indicates a unique string which can be used with the *xrun --adapter-id* option.
- The `Devices` column indicates the target that is connected to the debug adapter. This includes the architecture string, and the IDs of the nodes in square brackets. A 2-chip XS3A system would have `XS3A[0-1]` in the devices column, for instance.

--reset

Reset the target upon connection. Without this flag, the system will be left in its current state for XGDB to observe. With this flag, the system will be reset and in a fresh state for XGDB to initialise and control.

XGDB automatically injects this argument when *connect* is used, and does not use it when *attach* is used.

(Supported by: **hardware**) (Implicit for: **simulator**, **debugrom-simulator**)

--xscope

Connect to the target with xSCOPE for IO and data collection. Supplying any other `--xscope` option will cause this to be automatically implied.

(Supported by: **hardware**)

--xscope-limit <limit>

Limit the number of (non-IO) xSCOPE records which will be processed. If not specified, an unlimited number will be allowed.

Note: Set to 0 to restrict xSCOPE to only be used for IO. (Supported by: **hardware**)

`--xscope-file` `<filepath>`

Send xSCOPE data entries to a file. Argument is the file to write xSCOPE data entries to. Both a `file.vcd` file for the VCD trace and a `file.gtkw` for easy loading into GTKWave will be created. The maximum number of entries can be controlled by *`--xscope-limit`*. (Supported by: **hardware**)

`--xscope-port` `<bind address>`

Start a local xSCOPE server at the specified port, which will serve xSCOPE messages. The argument is of the form `<local_ip>:<local_port>`. If `<local_ip>` is omitted, the server will be bound to `localhost`. If `<local_port>` is omitted, an ephemeral port will be used.

Note that packets sent using this mechanism may be received out-of-order, so if ordering is important for your application, you should reorder them on the client side. See *xSCOPE host library* for more information on the capability of the host application.

(Supported by: **hardware**)

`--xscope-port-blocking`

Wait for a connection to the *`--xscope-port`* xSCOPE server to be made before completing initialisation of XGDBSERVER.This will prevent XGDB from connecting to XGDBSERVER until the xSCOPE server connection has been made, which is useful for preventing race conditions which would cause initial packets to be lost. Note waiting too long (several minutes) before forming the connection may trigger timeouts within XGDB. (Supported by: **hardware**)

`--gprof`

Capture gprof profiling information from the running target to a file in the current directory. One output file will be created for each tile.

See *Sample-based profiling of the target program* for an example of how to use this feature. (Supported by: **hardware**)

`--sim-args` `<arg>`

Single string containing all of the arguments to pass to a simulator backend.

See *`xsim --help`* for arguments which can be passed into XSIM by this mechanism.

Note: From a shell, if multiple arguments are to be passed, wrap them in single quotes so that they are not treated as arguments to XGDBSERVER itself.

(Supported by: **simulator**, **debugrom-simulator**)

`--jtag-speed` `<divider>`

Set the JTAG TCK divider, where the JTAG TCK frequency is `(25/(<divider>+2))` MHz. The default frequency is 12.5MHz. (Supported by: **hardware**)

`--xtag` `<command>`

Instruct the xTAG to do something before continuing. Some of the commands result in XGDBSERVER terminating instead of waiting for a connection. Available commands:

- `reboot` Reboot the xTAG before forming a connection.
- `rebootandexit` Reboot the xTAG, then exit XGDBSERVER without awaiting a GDB connection.
- `upgrade` Upgrade the xTAG bootloader to the newest one compiled into the xTAG firmware, then exit XGDBSERVER.
- `downgrade` Downgrade the xTAG bootloader to the one which was programmed in the factory, then exit XGDBSERVER.

(Supported by: **hardware**)

4.1.6.4 Monitor commands

The following commands are all provided as part of the XGDBSERVER, and can be accessed via the XGDB `monitor` command while connected.

They can be listed within XGDB by issuing `monitor help` at the XGDB interactive prompt. Not every monitor command is available for every backend or target type.

Note: Every command documented in this section can be run from within XGDB using the `monitor` command once connected to XGDBSERVER (such as with the `connect` command).

For example, to run the command `help`, type `monitor help` into the XGDB command prompt.

`help`

Usage: `monitor help [<query>]`

Print the help information for all monitor commands. If `query` is specified, then only print commands that contain matching text.

`args`

Usage: `monitor args`

Get the arguments which will be passed to the target program when it starts.

Usage: `monitor args set <arg>`

Set the arguments which will be passed to the target program when it starts. This is equivalent to the `xgdbserver --args` option.

It is usually preferable to pass arguments to a GDB command such as `run` or use the `xgdb --args` option.

A program must be built with `xcc -fcmdline-buffer-bytes` in order to support command line arguments. See the compiler documentation for more info.

`sswitch read`

Usage: `monitor sswitch read <address>`

Read the node configuration register at 'address'.

Node configuration registers exist in the SSwitch (System Switch).

`sswitch write`

Usage: `monitor sswitch write <address> <value>`

Set the node configuration register at 'address' to equal 'value'.

Node configuration registers exist in the SSwitch (System Switch).

Warning: Node configuration registers can cause system instability, such as changing the PLL configuration while it is being used - care should be taken to avoid this.

`pswitch read`

Usage: `monitor pswitch read <address>`

Read the tile configuration register at 'address'.

Tile configuration registers exist in the PSwitch (Processor Switch)

`reset`

Usage: `monitor reset [<reset type>]`

Reset the target. Some of these reset types are internal and shouldn't really need to be used directly. To restart a program which has already been loaded by XGDB, it is usually more appropriate to just use the XGDB *run* command.

> **Warning:** When resetting the target using this command, XGDB will not be aware that the target state has changed, so it may be necessary to run `maintenance flush register-cache` on the XGDB command line to force it to refresh the state.

Table 4.3: Reset types

Reset Type	Description
`full-reboot`	**The default**. Fully reboot the chip/simulation environment. Note that this is a full reset of the chip, including (for instance) starting to load the program again from flash. The target will be halted after some short period of time (this may be long enough to fully boot from flash and start if the program is small.) This can be useful if you want to restart the program that is currently in flash, so it can be debugged after starting up the same way as it will in production.
`detach`	Fully reboot the chip/simulation environment (same as full-reboot), then immediately detach the debugger without halting the target. This can be useful if you want to restart the program exactly as it will boot in production, without affecting its realtime performance at all, because the debugger will not be attached. In order to debug the target after resetting it in this way, the *attach* command should be used.
`pre-load`	(internal use) Reset the chip to a state where it is ready to have a new program loaded. This reboots the target into a halted state where it is waiting for XGDB to load a program to it - it will not start booting from flash. This reset type is used automatically when the *--reset* option, or *load*, *run* or similar commands are used.
`post-load`	(internal use) Reset the ancillary parts of the system after a program has been loaded. This does **not** restart the target itself, it is just for synchronising the debug adapter systems.
`post-attach`	(internal use) Reset/Resynchronise the ancillary parts of the system after XGDB has attached. This does **not** restart the target itself, it is just for synchronising the debug adapter systems, such as xSCOPE.

`coordinates`

Usage: `monitor coordinates [-m]`

Get the coordinates to the current thread in the form `node[0] tile[1] thread[2]`. Add the `-m` option to provide it in a machine readable `0,1,2` format instead.

It is usually more appropriate to use the XGDB *info threads* command.

`resource read`

Usage: `monitor resource read <type> <number> [<port width>]`

Low level command to read a resource register, where `<type>` is the type of resource and `<number>` is the resource number. `<portwidth>` is the port width in bits, and must be specified for port resources, and not for other resource types.

Table 4.4: Resource types

`<type>`	Description
`port`	Port resource - must specify portwidth.
`timer`	Timer resource.
`chanend`	Channel end resource. Note that this does not read the value received on a channel end, it reads the channel end configuration.
`synchroniser`	Synchroniser resource.
`thread`	Thread resource.
`lock`	Lock resource.
`clkblk`	Clock Block resource.
`swmem`	Software Memory resource.
`ps`	Processor State Register resource (see processor status configuration in the datasheet).
`config`	Configuration resource.
`raw`	Raw Resource ID, for instance as would be allocated by a `GETR` instruction. The resource ID encodes the type, resource number, register ID and port width into a single integer.

4.1.6.5 Environment

`XDBG_LOG_LEVEL`

This environment variable can be set to increase the log level of XGDBSERVER, as an alternative to the *`xgdbserver --log-level`* option. It can be used in 2 different ways.

Firstly, it can accept a log level specification in the same form as is documented in the *`xgdbserver --log-level`* option, which allows for a string-based configuration of precise log levels.

Alternatively, for legacy compatibility, it can be set to an 8 bit unsigned integer value; each bit set will enable a logging feature associated with traffic to and from the xTAG. Bits 0 to 3 control system debug and USB tracing and bits 4 to 8 control xSCOPE trace processing.

For example, the following enables all logging:

```
$ XDGB_LOG_LEVEL=trace xgdb --ex "connect" --ex "run" a.xe
```

or this enables all logging except for the USB interface (which will only report errors):

```
$ XDGB_LOG_LEVEL=trace,xdbg::usb=error xgdb --ex "connect" --ex "run" a.xe
```

4.1.7 XFLASH

4.1.7.1 Synopsis

```
xflash xe-file
xflash [options]
```

4.1.7.2 Description

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

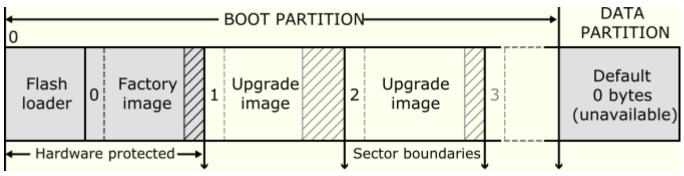


Fig. 4.2: Flash format diagram

Basic usage in the form **`xflash a.xe`** causes XFLASH to convert the application in `a.xe` to the XTC flash image format. This image format is loaded into memory by the XTC flash loader, which XFLASH also generates and customises for the target configuration.

The combined file with loader and image is transferred to the target using XGDB. This includes a small program loaded directly into RAM that runs on the target to perform the interaction with the flash device.

Once flash programming is complete, the system is reset and boots according to the boot source pins. If configured correctly the ROM will chain-load the XTC flash loader, which chain-loads the application.

XFLASH can also be used in conjunction with XBURN to secure the target system, providing a root of trust using OTP memory. Under this scheme, the device is permanently configured to boot an executable that resides in OTP, which verifies the signature of the flash loader. The flash loader also enforces similar signature verification of candidate application images.

The process of generating the overall flash image requires XFLASH to know certain properties of the flash device, such as number of pages. This information can be specified in the *XN file* as attributes of the flash device, but if unspecified, XFLASH will try to determine these automatically by running an additional program on the target called the 'device inquirer'. The inquirer connects to the flash device and will make use of SFDP to discover the necessary information.

XFLASH also exposes generic flash read, write and erase facilities that do not invoke the image generation component. These can be used for backing up the contents of the flash or programming an externally or separately generated image file. The flash writer additionally includes a feature to compress 'sparse' images in order to more quickly transmit the data from the host to the target - see the *`--load-format`* option for details.

4.1.7.3 Options

4.1.7.3.1 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, 15.2 and 15.3. 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 devices, the data partition of each device 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.

If no factory image is specified (meaning the flash is not used for booting), this option acts as an alias for *`--write-all`* for the target flash device, treating the entire flash array as raw storage for arbitrary data with no partitioning. If a factory image is specified, this option will use the data within the specified file to form the data partition.

This option can target multiple flash devices connected to the same tile if described as such in the XN file. This requires at least independent chip select (`SS` or `CS`) ports for each flash device, but the other ports may be multiplexed. Multiple flash devices can also be connected to different tiles or nodes but this requires more ports allocated to flash.

`--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, **-f**

Disables interactive prompts for user confirmation of any action. This can be used to bypass target warnings without user interaction.

--verbose, **-v**

Prints additional information about the program when loaded onto the target system.

--help, **-h**

Prints a description of the supported command line options.

--version

Displays the version number and copyrights.

4.1.7.3.2 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 host 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 <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. This is the default behaviour.

`--force-xscope`

Will instruct the program to communicate over xSCOPE. This results in much faster flash programming but is more likely to experience signal integrity issues on some hardware configurations.

`--force-pll-reset`

Will force a 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 $25/(n+2)$ MHz. The default value of the divider for the JTAG clock is 0, representing 12.5 MHz.

`--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 boundry 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 boundry within the boot partition. This is the default search mechanism as of 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.

4.1.7.3.3 Security Options

The following options are used in conjunction with the *AES Module*.

--key `<keyfile>`, **-k** `<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.

4.1.7.3.4 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.

--make-exec `<xe-file>`

Places an executable in `xe-file` that when run on an xCORE device, performs the specified flash operation.

The XE file can be run later using XRUN.

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...]`, **--spi-cmd** `<...>`

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.

--spi-interactive

Interactive variant of the above option - the target requests commands to issue to the flash device via host I/O in the same text format as in `--spi-command`. 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.

4.1.7.3.5 Advanced Options

--load-offset `<offset>`

Causes the flash programmer to treat the provided value as the start address of the flash array to access for any read/write/erase operation. By default this is zero.

--load-size `<size>`

Causes the flash programmer to treat the provided value as the data range of the flash array to access for any read/write/erase operation. This means an end address of `<offset> + <size>` used in combination with `--load-offset`. By default this is the size of the flash device for read or erase operations, or the size of the generated or specified image for write operations.

--load-format `<format>`

Causes the flash programmer to use the specified image format for write operations, where `<format>` is either `BIN` or `PPB`. `BIN` means a raw, flat binary while `PPB` is an S-Record supported sparse image file. By default the flash programmer uses whichever format will transmit the least data over the debug interface to minimise the write duration, but both formats are expected to result in identical data programmed to the flash device.

4.1.7.3.6 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-cmd 0x06 --spi-cmd 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-cmd 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-cmd 0x06 --spi-cmd 0xC7 \
    --spi-cmd 0x80000005 1 --spi-cmd 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-cmd 0x82000000 \
    --spi-cmd 0x44EB 258 0x00 0x00 0x00 0x00 --spi-cmd 0x82000001 \
    --spi-cmd 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.

Further worked example(s) concerning specific flash device quirks can be found under *libquadflash devices*.

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.

4.1.8 XBURN

4.1.8.1 Synopsis

```
xburn xe-file
xburn [options]
```

4.1.8.2 Description

XBURN creates OTP images, and programs images into the OTP memory of xCORE devices.

Usage of the form **`xburn a.xe`** burns the application in `a.xe` directly into OTP. This is typically undesirable as it's an irreversible process, meaning that the application cannot be removed or upgraded, and OTP memory space is limited and cannot accommodate large applications.

Instead, XBURN is most commonly used to provision a device for secure boot and set the overall security configuration of the system. Secure boot typically works in conjunction with XFLASH - a small bootloader is installed to OTP memory which enforces signature verification of upgradable software loaded from flash memory. This bootloader is installed into OTP using the XBURN option `--lock`, which burns an executable generated by XBURN rather than one specified from an external file.

For more advanced use cases, XBURN exposes *low-level programming* of individual OTP rows with supplementary data as required by the application, such as device-unique IDs. This can be read back at runtime using the *OTP library*.

4.1.8.3 Options

4.1.8.3.1 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.

`<csv-file>`

Specifies OTP `tile,address,value` tuples from *csv-file*. This may be anything from a handful of special values to program for device provisioning or identification, to a complete OTP specification generated using `--dump-otp-spec`.

See *CSV Format*.

`--version`

Displays the version number and copyrights.

`--help`, `-h`

Prints a description of the supported command-line options.

Note: It's advised to consistently use a single version of XBURN on a given device. Bootloader images and redundancy mechanisms may subtly change between releases of the XTC Tools. By consistently using only a single version, any (perhaps accidental) double-programming attempt will generally behave as a no-op, whereas a difference in content to be programmed may cause irreversible damage to the device.

4.1.8.3.2 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 ID Devices
-- ---- ---------- -------
```

The adapters are ordered by their serial numbers.

--id `<id>`

Specifies the adapter connected to the target hardware.

--adapter-id `<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 $25/(n + 2)$ MHz. The default value is 0 (12.5 MHz).

--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 $100/(2 * n)$ MHz. The default value is 20 (2.5MHz).

This option is only valid with `--lock`.

4.1.8.3.3 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 does not perform any programming or target interactions.

--lock `<keyfile>`

Specifies the *XCORE AES boot module* and a *default set of security bits*.

4.1.8.3.4 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 4.5: Default security bits written by XBURN

Security Bit	XE Image	OTP Image	AES Module (`--lock`)
OTP Boot	Enabled	As per OTP image file	Enabled
JTAG Access	Enabled		Disabled
Plink Access	Enabled		Enabled
Global Debug	Enabled		Disabled
Master Lock	Disabled		Enabled
Secure Config Access	Enabled		Enabled

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-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`
: Enables/disables access to the security register.

Note: xcore.ai devices do not support the usage style of progressively setting more security bits to upgrade the device's security in phases by rewriting the security register. The security register can only be burned once on xcore.ai, so must be programmed with the intended final configuration from the start. This diverges from the behaviour of XCORE-200 and previous devices which do allow progressively setting more bits.

4.1.8.3.5 Programming Options

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

`--force`, **`-f`**
: Do not prompt for interactive confirmation 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.

`--dump-otp-spec`
: Dump the OTP images to CSV specification, disabling programming.

  The CSV file can be edited by hand and/or provided to XBURN for programming later in a second pass. If the CSV file has been unmodified, programming in a second pass is identical to single pass behaviour.

  See *CSV Format*.

`--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.

4.1.8.4 CSV Format

XBURN supports OTP programming from a CSV format specification. This is an advanced option that can be used for programming additional supplementary data out-of-band, with respect to the majority of OTP memory which is typically occupied by the AES Module generated by XBURN. This option can be used for provisioning of device serial numbers or similar information. Providing a CSV file to XBURN during programming will result in the tool incorporating the OTP data records in the CSV file into the overall image that will be programmed.

XBURN can also generate a complete CSV specification of all OTP data instead of directly programming it. This is done using the `--dump-otp-spec` option, and the resulting file can be provided later to XBURN to perform the actual programming of the device.

The CSV format is provided in the form `tile,address,value` where each row programs a 32-bit word of OTP memory. The `tile` field is typically `0` or `1` indicating `tile[0]` or `tile[1]`, but may also be the character `*` to apply to all tiles in the system. The address provided is a word address, so programming two adjacent 32-bit words requires incrementing the address by 1.

```
0,0x1FE,0xFFFF0000
1,0x1FE,0xFFFF0001
*,0x1FF,0xDEADBEEF
```

xcore.ai example: sets OTP virtual address `0x1FE` to `0xFFFF0000` on `tile[0]`, `0xFFFF0001` on `tile[1]` and address `0x1FF` to `0xDEADBEEF` on both tiles. This programs two words at the end of each tile's OTP, using eight physical rows when accounting for hardware redundancy.

Due to physical implementation details of each device, the format between the XCORE-200 family and xcore.ai family have a few differences. XCORE-200 considers addresses to be physical addresses, with redundant records expressed explicitly within the CSV file. On xcore.ai the addresses are considered to be 'virtual' addresses where hardware redundancy is always enabled, and therefore the redundant records are not specified explicitly in the CSV file.

```
0,0x7FE,0xFFFF0000
1,0x7FE,0xFFFF0001
*,0x7FF,0xDEADBEEF
```

XCORE-200 example: sets OTP physical address `0x7FE` to `0xFFFF0000` on `tile[0]`, `0xFFFF0001` on `tile[1]` and address `0x7FF` to `0xDEADBEEF` on both tiles. This programs two words at the end of each tile's OTP, using four physical rows, with two rows used in each physical OTP.

If exporting a CSV specification on the XCORE-200 family using `--dump-otp-spec`, an optional fourth CSV column will be present on some rows indicating that the executable (typically the AES Module) participates in the software redundancy mechanism used to improve yield. This column is currently unused by the xcore.ai family.

An example use case for the XCORE-200 family is provided in the tools release under the `examples/ExampleBurning/XCORE-200` directory. The script provided serves as a replacement for the retired XBURN `--serial-number`, `--board-id` and `--mac-address` options, and will generate a CSV file that can be provided to XBURN to provision the device with those values.

For a visual confirmation of the effect of the provided CSV entries, the option `--dump-otp-images` can be provided to XBURN and the resulting images viewed in a hex editor.

Caution: The CSV format allows programming any OTP region with no restrictions. It requires reasonable knowledge of the OTP memory map of the target device family to manipulate it correctly. See your device's datasheet for more information.

Be careful when using this option, particularly to ensure that the provided CSV doesn't instruct XBURN to corrupt or overwrite existing data structures, such as the AES Module. It's advisable to program custom application data towards the end of the OTP array as this area is unused by XBURN internally. The contiguous block at the start of the OTP array is reserved for boot functions.

4.1.9 XMAKE

4.1.9.1 Synopsis

```
xmake [options] [target] ...
```

4.1.9.2 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.

4.1.9.3 Options

--help, -h

Display all options.

4.2 File formats and data descriptions

This section collates and describes all the various data formats used or generated by the tools.

4.2.1 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.

4.2.1.1 Binary format

The following sections explain the common elements of the binary format. All data is encoded as little endian.

4.2.1.1.1 XE header

An XE file must start with an XE header. It has the following format:

Table 4.6: 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.

4.2.1.1.2 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 4.7: Sector header

Byte offset	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 4.8: 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 4.9: 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.

4.2.1.1.2.1 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.

4.2.1.1.2.2 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 4.10: 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.

4.2.1.1.2.3 XN sector

The XN sector contains a *XN description* of the system.

4.2.1.1.2.4 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 4.11: 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.

4.2.1.1.2.5 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 4.12: 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.

4.2.1.1.2.6 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.

4.2.1.1.2.7 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.

4.2.1.2 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 an xcore.ai device.

Table 4.13: 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	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	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*.

4.2.2 XN specification

4.2.2.1 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
                        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
```

4.2.2.2 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>
```

4.2.2.3 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 4.14: XN Package element

Attribute	Required	Type	Description
Id	Yes	String	A name for the package. All package names in the network must be unique.
Type	Yes	String	The name of the XML package. The tools search for the file `<type>.pkg` in the path specified by `XCC_DEVICE_PATH`.

Example

```
<Package id="0" Type="XS3-UnA-1024-QF60A">
```

This package is described in the file `XS3-UnA-1024-QF60A.pkg`.

4.2.2.4 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 4.15: 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 `periph:XS1-SU` the node is a XS1-SU peripheral node. Otherwise the type specifies the name of an XML file that describes the node. The tools search for the file `config_<type>.xml` in the path specified by `XCC_DEVICE_PATH`.
Reference	Yes	String	Associates the node with a xCORE Tile indentifer specified in a `Declaration`. This attribute is only valid on nodes with type `periph:XS1-SU`.
RoutingId	No	Integer	The routing identifier on the xCONNECT Link network.
InPackageId	Yes	String	Maps the node to an element in the package file.
Oscillator	No	String	The PLL oscillator input frequency, specified as a number followed by either `MHz`, `KHz` or `Hz`.
OscillatorSrc	No	String	The name of the node which supplies the PLL oscillator input.
SystemFrequency	No	String	The system frequency, specified as a number followed by either `MHz`, `KHz` or `Hz`. Defaults to 400MHz if not set.
PllFeedbackDivMin	No	Integer	The minimum allowable PLL feedback divider. Defaults to 1 if not set.
ReferenceFrequency	No	String	A reference clock frequency, specified as a number followed by either `MHz`, `KHz` or `Hz`. Defaults to 100MHz if not set.
PllDividerStageOneReg	No	Integer	The PLL divider stage 1 register value.
PllMultiplierStageReg	No	Integer	The PLL multiplier stage register value.
PllDividerStageTwoReg	No	Integer	The PLL divider stage 2 register value.
SecondaryPllInputDiv	No	Integer	The secondary PLL input divider register value
SecondaryPllOutputDiv	No	Integer	The secondary PLL output divider register value
SecondaryPllFeedbackDiv	No	Integer	The secondary PLL feedback divider register value
RefDiv	No	Integer	`SystemFrequency` / `RefDiv` = `ReferenceFrequency`

The PLL registers can be configured automatically using the attributes `SystemFrequency`, `PllFeedbackDivMin` and `ReferenceFrequency`, or can be configured manually using the attributes `PllDividerStageOneReg`, `PllMultiplierStageReg`, `PllDividerStageTwoReg` and `RefDiv`. If any of the first three attributes are provided, none of the last four attributes may be provided, and vice versa.

The PLL oscillator input frequency may be specifed using the `Oscillator` or `OscillatorSrc` attribute. If the `Oscillator` attribute is provided the `OscillatorSrc` attribute must not be provided, and vice versa.

If manual configuration is used, the attributes `PllDividerStageOneReg`, `PllMultiplierStageReg`, `PllDividerStageTwoReg` and `RefDiv` must be provided and the PLL oscillator input frequency must be specifed. The tools use these values to set the PLL registers and reference clock divider. Information on the PLL dividers can be found in the device datasheets.

If the oscillator frequency is specifed and none of the manual PLL attributes are provided, automatic configuration is used. The tools attempt to program the PLL registers such that the target system frequency is achieved, the PLL feedback divider is greater than or equal to the minimum value and the target reference clock frequency is achieved. If any of these constraints cannot be met, the tools issue a warning and report the actual values used.

If the oscillator frequency is not specified, the tools do not attempt to configure the 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">

4.2.2.4.1 Tile

A `Tile` element describes the properties of a single xCORE Tile.

Table 4.16: XN `Tileref` 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]">
```

4.2.2.4.2 Port

A `Port` element provides a symbolic name for a port.

Table 4.17: XN `Port` element

Attribute	Required	Type	Description
Location	Yes	String	A port identifier defined in the standard header file `<xs1.h>`. The ports are described in the relevant device architecture manual and datasheet.
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"/>
```

4.2.2.4.3 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.

Tip: The XMOS tools require a `Boot` element to be able to *boot programs from flash memory*.

4.2.2.4.4 Source

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

Table 4.18: XN `Source` element

Attribute	Required	Type	Description
Location	Yes	String	Has the form `SPI:` or `LINK`. The `device-name` must be declared in the set of `Device` elements.
BootMode	No	Integer	Required for `Location` of type `LINK`, and maps to an entry in the "boot source pins" table in the "boot procedure" chapter of the datasheet. Valid values are 4 through 7.

Example

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

4.2.2.4.5 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 4.19: XN `Bootee` element

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

Example

```
<Bootee NodeId="Secondary">
```

4.2.2.4.6 Bit

A `Bit` element specifies the direction for messages whose first mismatching bit matches the specified bit number.

Table 4.20: 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"/>
```

4.2.2.4.7 Link

When it appears within a `RoutingTable` element, a `Link` element specifies the direction of an xCONNECT Link (xLINK).

Table 4.21: 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"/>
```

4.2.2.4.8 Service

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

Table 4.22: XN `Service` element

Attribute	Required	Type	Description
Proto	Yes	String	The prototype for the service function, excluding the `service` keyword. This prototype is exported to the header file `<platform.h>` for use in XC programs.

Example

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

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>
```

4.2.2.4.9 Chanend

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

Table 4.23: XN `Service` element

Attribute	Required	Type	Description
Indentifier	Yes	String	The identifier for the chanend argument in the service function prototype.
end	Yes	Integer	The number of the channel end on the current node.
remote	Yes	Integer	The number of the remote channel end that is connected to the channel end on the current node.

Example

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

4.2.2.5 Link

A Link element describes the characteristics of an xCONNECT Link. It must contain exactly two *~~LinkEndpoint~~ children*.

Table 4.24: XN `Link` element

Attribute	Required	Type	Description
Encoding	Yes	String	Must be either `2wire` or `5wire`.
Delays	Yes	String	Of the form `<x>clk,<y>clk` where <x> specifies the inter-token delay value for the endpoint, and <y> specifies the intra-token delay value for the endpoint. <x> and <y> are specified as a number of `SystemFrequency` cycles. If , `<y>clk` is omitted, the <x>clk value is used for the intra-token delay value.
Flags	No	String	Specifies additional properties of the link. Use the value `XSCOPE` to specify a link used to send xSCOPE trace information.

Example

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

4.2.2.5.1 LinkEndpoint

A `LinkEndpoint` describes one end of an xCONNECT Link, the details of which can be found in the device datasheet. Each endpoint associates a node identifier to a physical xCONNECT Link.

Table 4.25: XN `LinkEndpoint` element

Attribute	Required	Type	Description
NodeID	No	String	A valid node identifier.
Link	No	String	A link identifier in the form `XL<n>` where <n> denotes a link index. See the corresponding device datasheet for available link pinouts.
RoutingId	No	Integer	The routing identifier on the xCONNECT Link network.
Chanend	No	Integer	A channel end.

An endpoint is usually described as a combination of a node identifier and link identifier. For a streaming debug link, one of the endpoints must be described as a combination of a routing identifier and a channel end. For example:

```
<LinkEndpoint NodeId="0" Link="XL0"/>
<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	Link specifier
0	XL0
1	XL1
2	XL2
3	XL3
4	XL4
5	XL5
6	XL6
7	XL7
8	XL8

The following example demonstrates how Node 0 Link number 4 is connected to Node 1 Link number 7:

```
<Links>
  <Link Encoding="5wire" Delays="4,4">
    <LinkEndpoint NodeId="0" Link="XL4"/>
    <LinkEndpoint NodeId="1" Link="XL7"/>
  </Link>
</Links>
```

4.2.2.6 Device

A `Device` element describes a device attached to an xCORE Tile that is not connected directly to an xCONNECT Link.

Table 4.26: 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.

4.2.2.6.1 Attribute

An `Attribute` element describes one aspect of a ~~Device~~.

Table 4.27: 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"/>
  </Device>
</ExternalDevices>
```

4.2.2.7 JTAGChain

The JTAGChain element describes a device in the JTAG chain. The order of these elements defines their order in the JTAG chain.

Table 4.28: XN `JTAGChain` element

Attribute	Required	Type	Description
JTAGSpeed	No	String	Sets the JTAG clock speed.

4.2.2.7.1 JTAGDevice

Table 4.29: 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">
```

4.2.3 XSIM trace output

XSIM trace output is produced by using *`xsim -t`*, *`xsim --trace`* or *`xsim --trace-to`*.

The following example shows an except of the trace output from a program built for an xcore.ai device. It shows 8 logical cores running on tile 0. For each logical core, indentations using . to the addess field and subsequent fields is applied based on the logical core number.

```
tile[0]@0- -SI A-a-a-a-p-p-p-p-.----000803a8 (iprintf             + 18) : add     r1(0x81e04),
↪r11(0x81e04), 0x0 @307446
tile[0]@1- -DI p-A-a-a-a-p-p-p-.----.00080390 (iprintf             +  0) : entsp   0x0 S[0x81f24]
↪@307447
tile[0]@2- -DI p-p-A-a-a-a-p-p-.----..00080390 (iprintf             +  0) : entsp   0x0 S[0x81f24]
↪@307448
tile[0]@3- -DI p-p-p-A-a-a-a-p-.----...00080390 (iprintf             +  0) : entsp   0x0 S[0x81f24]
↪@307449
tile[0]@4- -DI p-p-p-p-A-a-a-a-.----....00080390 (iprintf             +  0) : entsp   0x0 S[0x81f24]
↪@307450
tile[0]@5- -DI a-p-p-p-p-A-a-a-.----.....00080390 (iprintf             +  0) : entsp   0x0
↪S[0x81f24] @307451
tile[0]@6- -DI a-a-p-p-p-p-A-a-.----......00080390 (iprintf             +  0) : entsp   0x0
↪S[0x81f24] @307452
tile[0]@7- -DI a-a-a-p-p-p-p-A-.----.......00080390 (iprintf             +  0) : entsp   0x0
↪S[0x81f24] @307453
```

Each row represents the execution of a single instruction. For the first line in the above example, the fields show the following information:

- `tile[0]`: the tile executing the instruction
- `@0` : the logical core executing the instruction
- `-  -SI` : the I0, I1, I2, I3 and I4 fields - see below

- `A-a-a-a-p-p-p-p-` : the S0 and S1 fields for the eight active logical cores
- `000803a8`: program counter address
- `(iprintf + 18)`: the function name symbol and offset from it
- `add r1(0x81e04), r11(0x81e04), 0x0 @307446`: instruction and operands
- `@3261`: tile clock cycle count

When an instruction makes a load or store to memory the address will be shown, for example (`L[0x42000]`).

Further detail on the fields and the values they may contain is shown in the table below:

Table 4.30: Trace output field definitions

Tile and core	Core State (single character fields)									
Name from XN	I0	I1	I2	I3	I4	S0	S1	M	S	K	N

Table 4.31: Trace output field definitions (continued)

Address		Instruction		Mem	Cycle
PC	(sym+offset):	name	operands	access	@val

For each field, the possible values and their meanings are shown below. Note the S0 and S1 character fields are repeated for each active core.

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
I3: `S` Logical core in single issue mode
I3: `D` Logical core in dual issue mode
I4: `I` Always shows `I`
S0: `-` Logical core not in use
S0: `a` Logical core active
S0: `A` Logical core active (the instruction being traced belongs to this core)
S0: `i` Logical core active with ININT bit set
S0: `I` Logical core active with ININT bit set (belongs to this core)
S0: `p` Logical core paused due to instruction fetch
S0: `m` Logical core paused with MSYNC bit set
S0: `s` Logical core paused with SSYNC bit set
S0: `w` Logical 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
operands: `rn(val)` Value `val` of register `n`
operands: `res[id]` Resource identifier `id`
access: `L/S[addr]` Load from/Store to address `addr`
@val: tile clock cycle count

4.2.4 xSCOPE config file

An xSCOPE config file is an XML file, with a suffix of `.xscope`.

It is used for the configuration of xSCOPE, such as to enable printing over xSCOPE, or to configure probes to send data to the host. Config files are the preferred mechanism to configure xSCOPE, however it is also possible to use the *xSCOPE target library*.

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>
```

Here is a more complex example:

```
<xSCOPEconfig ioDest="link" ioMode="basic" enabled="true">
  <Probe name="Probe 1" type="DISCRETE" datatype="UINT" units="Bytes" enabled="true"/>
  <Probe name="Probe 2" type="DISCRETE" datatype="UINT" units="Bytes" enabled="false"/>
  <Probe name="Really Interesting Probe 3" shortname="PROBE3" type="DISCRETE" datatype="UINT" units=
↪"Bytes" enabled="true"/>
  <Module name="Ethernet" id="ETH">
    <Probe name="Packet Count" type="DISCRETE" datatype="UINT" units="Bytes" enabled="true"/>
  </Module>
</xSCOPEconfig>
```

If compiling and linking separately, the xSCOPE config files must be provided to XCC at every stage, since they are used both to autogenerate definitions at compile time, and provide functionality at link time.

Multiple xSCOPE config files can be provided to xcc, but they must be provided in the same order for every xcc command, and care must be taken: later files can override information specified in earlier files. As good practice, avoid specifying the same probe or module more than once, and only specify the root node's attributes in a single file.

4.2.4.1 `xscopeconfig` element

The `xscopeconfig` element is required. It specifies the top level configuration for xSCOPE.

Attributes are:

`iomode`
Configure *write* syscalls (such as `printf` or `fwrite`) to be sent over the much faster xSCOPE transport rather than over JTAG. Can be set to:

Value	Description
`none`	Use JTAG for write syscalls (slow)
`basic`	Use xSCOPE for write syscalls.
`timed`	Use xSCOPE for write syscalls, and prepend each write with a timestamp. Note while this is a useful shortcut to get timestamps while printf debugging, it causes *all* writes, including writes to files to be prepended with a timestamp. This may not be desirable.

Note that if xSCOPE is configured for io, then *`xrun --xscope`* must be specified when using xrun/xgdb or the output will not be shown. It is not possible to attach to an already running program and see the xSCOPE output: it must be freshly loaded for the session with xrun/xgdb.

`enabled`
May be set to `true` to enable or `false` to disable xSCOPE. If a `.xscope` file is not provided, or this attribute is not specified, then xSCOPE is disabled by default.

4.2.4.2 `probe` elements

The `probe` elements are optional, depending on the desired number of named xSCOPE "probes". Each `probe` element represents a single probe channel which can be used to send reports to the host.

Attributes are:

`name`
Set to a string representing the name of the probe. This can consist of alphanumeric characters, spaces and underscores.

`type`
May be set to:

Value	Description
`STARTSTOP`	Event which gets a 'start' and a 'stop' representing a block of execution (e.g a function call).
`CONTINUOUS`	A probe which will receive continuous data which can be interpolated.
`DISCRETE`	A probe which will receive data which should not be interpolated.
`STATEMACHINE`	Similar to `DISCRETE` - a probe which represents different states of a state machine, and each report represents moving to a new state.

`datatype`
May be set to:

Value	Description
`NONE`	No data. The event will be reported but there is no associated data.
`UINT`	Unsigned bit integer (1, 2, 4 or 8 bytes)
`INT`	Signed 32 bit integer (1, 2, 4 or 8 bytes)
`FLOAT`	Floating point number (4 or 8 bytes)

`units`
Set to a string representing the units of measurement for the probe

enabled
: May be set to `true` to enable the probe or `false` to disable it. Defaults to `false`.

shortname
: The name of the define which will be used for the auto-generated define name. See *Auto-generated header file* for more detail. This defaults to a fully capitalised, underscored version of `name`. This can consist of alphanumeric characters and underscores, as it becomes a macro name available in the C program.

module
: Name of the module to add this probe to, which will affect the name of the define visible in C code. It is preferred to not use this attribute, and instead group module probes together in `<Module>` elements.

4.2.4.3 `module` element

The `module` element can be used to group several related probes together. Probe elements can be children of the module element.

The auto-generated probe defines will be prefixed with `MODULE_ID_` where `MODULE_ID` is the contents of the `id` attribute.

name
: Set to a string representing the name of the probe.

id
: Set to a string to prefix the autogenerated probe define names in the C header. See *Auto-generated header file* for more detail.

4.2.4.4 Auto-generated header file

The xSCOPE config file is used to automatically generate some preprocessor definitions for the purpose of referencing the probes from code.

These definitions can be used with the *xSCOPE target library*.

These preprocessor definitions are accessible by `#include <xscope.h>`.

The definition name is of the form `SHORTNAME` or `MODULE_ID_SHORTNAME` where `SHORTNAME` is the value of the `shortname` attribute of the `probe` element, and `MODULE_ID` is the contents of the `id` attribute of the `module` element if the probe is part of a module.

The probe definition values are equal to the unique ID of the probe, or -1 if the probe is disabled.

To see the autogenerated file, *`xcc -save-temps`* can be used. The auto-generated file will be dumped to the current directory as `xscope_probes.h`.

For example, the *earlier example* autogenerates the following output:

```
// This file is autogenerated.
#ifndef __XSCOPE_PROBES_H__
#define __XSCOPE_PROBES_H__
#define PROBE_1 (0)
#define PROBE_2 (-1)
#define PROBE3 (1)
#define ETH_PACKET_COUNT (2)
#endif // __XSCOPE_PROBES_H__
```

4.3 xSCOPE

xSCOPE is a high speed protocol between the host and the target which runs over an xLINK.

It enables high-speed communication with two primary uses:

- Probes, where data values are sent to probes as 'records' which can then be plotted on a graph
- IO, where the 'write' syscall (used for functions such as printf) sends data over xSCOPE for faster printing

4.3.1 xSCOPE command line options

In order to use xSCOPE, the various tools need to be configured to make use of it. This section links to the relevant chapters which are required.

4.3.1.1 xcc options

To enable xSCOPE support when using xcc, one of the following methods must be used:

1. Use *`xcc -fxscope`* for every xcc call

or

2. Pass an *xSCOPE configuration file* to every xcc call.

Note that to enable xSCOPE, support must be indicated in the target's xn file. See *XN specification* for more information.

4.3.1.2 xrun options

These options can be supplied to xrun. Note that if a program is compiled with xSCOPE support, xSCOPE support must also be enabled when using xrun. If probes or records are being used, then either *`xrun --xscope-file`* or option:*xrun –xscope-port* must be provided for those records to be recorded.

Attempting to use any xSCOPE argument on a program that has not been compiled with xSCOPE support will fail.

Table 4.32: xrun options

Option	Translated to xgdbserver
`xrun --xscope`	`--xscope`
`xrun --xscope-io-only`	`--xscope-limit 0`
`xrun --xscope-limit`	`--xscope-limit arg`
`xrun --xscope-file`	`--xscope-file arg`
`xrun --xscope-port`	`--xscope-port-blocking --xscope-port ip:port`

Supplying any of these xSCOPE options will automatically imply `--xscope`.

4.3.1.3 xgdbserver options

These options can be supplied either to the xgdb `connect` or `attach` command, or xgdbserver directly.

Table 4.33: xgdbserver options

Option	Short description
`xgdbserver --xscope`	Connect to the target with xSCOPE for io and data collection.
`xgdbserver --xscope-limit`	Limit the number of (non-ID) xSCOPE records which will be processed.
`xgdbserver --xscope-file`	Send xSCOPE data entries to a file.
`xgdbserver --xscope-port`	Start a local xSCOPE server at the provided port, which will serve xS-COPE messages.
`xgdbserver --xscope-port-blocking`	Wait for a connection to the –xscope-port xSCOPE server to be made before completing initialisation of xgdbserver.

Supplying any of these xSCOPE options will automatically imply `--xscope`.

4.3.2 xSCOPE performance figures

4.3.2.1 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 approcimately 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 4.34: 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.

4.3.2.2 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 may drop trace data records.

4.3.2.3 Host/target data throughput benchmarks

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

Table 4.35: 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.

4.3.3 xSCOPE target library

The xSCOPE target library is used for communication between the host and the target over the fast xSCOPE transport. It can be used for data logging and targeted profiling.

The xSCOPE target library is automatically linked if your application is built with xSCOPE (see *xcc options*), so just include the header:

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

Including this also makes the probes which were specified in the *xSCOPE config file* available to the program.

This library is not accessible if a program is not built with xSCOPE support.

4.3.3.1 Example

This file contains functions to access xSCOPE. Example:

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

void xscope_user_init(void) {
  xscope_register(1, XSCOPE_CONTINUOUS, "Continuous Value 1", XSCOPE_UINT, "Value");
}

int main (void) {
  par {
    on tile[0]: {
      for (int i = 0; i < 100; i++) {
        xscope_int(0, i*i);
      }
    }
  }
  return 0;
}
```

This example defines a probe “Continuous Value 1”, then sends 100 numbers as records over that probe.

4.3.3.2 Configuration API

The configuration API is an alternative mechanism of configuring xSCOPE to *xSCOPE config file*.

For most purposes, the *xSCOPE config file* should be preferred over using this, as it is a richer, less error-prone mechanism. They can also be used in combination, such as defining the probes with a config file, but using this interface for the more advanced functionality.

Enums

enum **xscope_EventType**

Kind of event that an xSCOPE probe can receive.

Values:

enumerator **XSCOPE_STARTSTOP**

Start/Stop - Event gets a start and stop value representing a block of execution.

enumerator **XSCOPE_CONTINUOUS**

Continuous - Only gets an event start, single timestamped “ping”.

enumerator **XSCOPE_DISCRETE**

Discrete - Event generates a discrete block following on from the previous event.

enumerator **XSCOPE_STATEMACHINE**

State Machine - Create a new event state for every new data value.

enumerator **XSCOPE_HISTOGRAM**

enumerator **XSCOPE_STARTSTOP**

Start/Stop - Event gets a start and stop value representing a block of execution.

enumerator **XSCOPE_CONTINUOUS**

Continuous - Only gets an event start, single timestamped "ping".

enumerator **XSCOPE_DISCRETE**

Discrete - Event generates a discrete block following on from the previous event.

enumerator **XSCOPE_STATEMACHINE**

State Machine - Create a new event state for every new data value.

enumerator **XSCOPE_HISTOGRAM**

enumerator **XSCOPE_STARTSTOP**

Start/Stop - Event gets a start and stop value representing a block of execution.

enumerator **XSCOPE_CONTINUOUS**

Continuous - Only gets an event start, single timestamped "ping".

enumerator **XSCOPE_DISCRETE**

Discrete - Event generates a discrete block following on from the previous event.

enumerator **XSCOPE_STATEMACHINE**

State Machine - Create a new event state for every new data value.

enumerator **XSCOPE_HISTOGRAM**

enum **xscope_UserDataType**

C data types an xSCOPE probe can receive.

Values:

enumerator **XSCOPE_NONE**

No user data.

enumerator **XSCOPE_UINT**

Unsigned int user data.

enumerator **XSCOPE_INT**

Signed int user data.

enumerator **XSCOPE_FLOAT**

Floating point user data.

enumerator **XSCOPE_NONE**

No user data.

enumerator **`XSCOPE_UINT`**

Unsigned int user data.

enumerator **`XSCOPE_INT`**

Signed int user data.

enumerator **`XSCOPE_FLOAT`**

Floating point user data.

enumerator **`XSCOPE_NONE`**

No user data.

enumerator **`XSCOPE_UINT`**

Unsigned int user data.

enumerator **`XSCOPE_INT`**

Signed int user data.

enumerator **`XSCOPE_FLOAT`**

Floating point user data.

enum **`xscope_IORedirectionMode`**

Methods of I/O redirection over the xSCOPE connection.

This applies to all write syscalls.

Values:

enumerator **`XSCOPE_IO_NONE`**

Output is not redirected.

enumerator **`XSCOPE_IO_BASIC`**

Output is redirected over xSCOPE.

enumerator **`XSCOPE_IO_TIMED`**

Output is redirected over xSCOPE, with prepended timestamp.

enumerator **`XSCOPE_IO_NONE`**

I/O is not redirected.

enumerator **`XSCOPE_IO_BASIC`**

Basic I/O redirection.

enumerator **`XSCOPE_IO_TIMED`**

Timed I/O redirection.

Functions

void **`xscope_user_init`**( )

User-implemented function to allow xSCOPE event registration.

This allows the code on the device to synchronize with the host. This may be defined anywhere in the application code and (if present) will be called before main(). It can be used to register probes which can then be used later in the application with the other functions defined in this header.

Note: A weak stub version of this function is defined, but it is intended to be overridden by a user implementation.

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

Registers the trace probes with the host system.

First parameter is the number of probes that will be registered. Further parameters are in groups of four.

a. event type (*xscope_EventType*)
b. probe name
c. user data type (*xscope_UserDataType*)
d. user data name.

This must be called from inside the *xscope_user_init()* function.

Examples:

```
xscope_register(1, XSCOPE_DISCRETE, "A probe", XSCOPE_UINT, "value"); ``
xscope_register(2, XSCOPE_CONTINUOUS, "Probe", XSCOPE_FLOAT, "Level",
                   XSCOPE_STATEMACHINE, "State machine", XSCOPE_NONE, "no name");
```

Note that the 'id' of each of these probes is implicit, starting from 0 and incrementing by 1 for each additional probe registered. In the above example:

- "A probe" would have id 0.
- "Probe" would have id 1.
- "State machine" would have id 2.

If an *xSCOPE config file* file is specified on the compile line, that takes priority, so this call will be ignored.

For these reasons, it is recommended to not use this function, and instead use the *xSCOPE config file* on the command line to allocate probes. xSCOPE config file probe IDs are available as preprocessor constants in the compiled program.

Parameters

- **`num_probes`** – Number of probes that will be specified.

void **`xscope_enable`**( )

Enable the xSCOPE event capture on the local xCORE tile.

This is not necessary if an *xSCOPE config file* is used to enable xSCOPE.

void **`xscope_disable`**( )

Disable the xSCOPE event capture on the local xCORE tile.

void **`xscope_config_io`**(unsigned int mode)

Configures xSCOPE I/O redirection.

This is not necessary if an *xSCOPE config file* is used to enable xSCOPE io.

Parameters

- **`mode`** – I/O redirection mode. See *xscope_IORedirectionMode*

void **`xscope_ping`**( )

Generate an xSCOPE ping system timestamp event.

void **`xscope_char`**(unsigned char id, unsigned char data)

Send a trace event for the specified xSCOPE probe of type char.

Parameters

- **`id`** – xSCOPE probe id.
- **`data`** – User data value (char).

void **`xscope_short`**(unsigned char id, unsigned short data)

Send a trace event for the specified xSCOPE probe of type short.

Parameters

- **`id`** – xSCOPE probe id.
- **`data`** – User data value (short).

void **`xscope_int`**(unsigned char id, unsigned int data)

Send a trace event for the specified xSCOPE probe of type int.

Parameters

- **`id`** – xSCOPE probe id.
- **`data`** – User data value (int).

void **`xscope_longlong`**(unsigned char id, unsigned long long data)

Send a trace event for the specified xSCOPE probe of type long long.

Parameters

- **`id`** – xSCOPE probe id.
- **`data`** – User data value (long long).

void **`xscope_float`**(unsigned char id, float data)

Send a trace event for the specified xSCOPE probe of type float.

Parameters

- **`id`** – xSCOPE probe id.
- **`data`** – User data value (float).

void **`xscope_double`**(unsigned char id, double data)

Send a trace event for the specified xSCOPE probe of type double.

Parameters

- **`id`** – xSCOPE probe id.
- **`data`** – User data value (double).

void **`xscope_bytes`**(unsigned char id, unsigned int size, const unsigned char data[])

Send a trace event for the specified xSCOPE probe with a byte array.

Parameters

- **`id`** – xSCOPE probe id.
- **`size`** – User data size.
- **`data`** – User data bytes (char[]).

void **`xscope_start`**(unsigned char id)

Start a trace block for the specified xSCOPE probe.

For use with XSCOPE_STARTSTOP probes.

Parameters

- **`id`** – xSCOPE probe id.

void **`xscope_stop`**(unsigned char id)

Stop a trace block for the specified xSCOPE probe.

For use with XSCOPE_STARTSTOP probes.

Parameters

- **`id`** – xSCOPE probe id.

void **`xscope_start_int`**(unsigned char id, unsigned int data)

Start a trace block for the specified xSCOPE probe and capture a value of type int.

For use with XSCOPE_STARTSTOP probes.

Parameters

- **`id`** – xSCOPE probe id.
- **`data`** – User data value (int).

void **`xscope_stop_int`**(unsigned char id, unsigned int data)

Stop a trace block for the specified xSCOPE probe and capture a value of type int.

For use with XSCOPE_STARTSTOP probes.

Parameters

- **`id`** – xSCOPE probe id.
- **`data`** – User data value (int).

void **`xscope_core_char`**(unsigned char id, unsigned char data)

Send a trace event for the specified xSCOPE probe of type char with logical core info.

Parameters

- **`id`** – xSCOPE probe id.
- **`data`** – User data value (char).

void **`xscope_core_short`**(unsigned char id, unsigned short data)

Send a trace event for the specified xSCOPE probe of type short with logical core info.

Parameters

- **`id`** – xSCOPE probe id.
- **`data`** – User data value (short).

void **`xscope_core_int`**(unsigned char id, unsigned int data)

Send a trace event for the specified xSCOPE probe of type int with logical core info.

Parameters

- **`id`** – xSCOPE probe id.
- **`data`** – User data value (int).

void **`xscope_core_longlong`**(unsigned char id, unsigned long long data)

Send a trace event for the specified xSCOPE probe of type long long with logical core info.

Parameters

- **`id`** – xSCOPE probe id.
- **`data`** – User data value (long long).

void **`xscope_core_float`**(unsigned char id, float data)

Send a trace event for the specified xSCOPE probe of type float with logical core info.

Parameters

- **`id`** – xSCOPE probe id.
- **`data`** – User data value (float).

void **`xscope_core_double`**(unsigned char id, double data)

Send a trace event for the specified xSCOPE probe of type double with logical core info.

Parameters

- **`id`** – xSCOPE probe id.
- **`data`** – User data value (double).

void **`xscope_core_bytes`**(unsigned char id, unsigned int size, const unsigned char data[])

Send a trace event for the specified xSCOPE probe with a byte array with logical core info.

Parameters

- **`id`** – xSCOPE probe id.
- **`size`** – User data size.
- **`data`** – User data bytes (char[]).

void **`xscope_core_start`**(unsigned char id)

Start a trace block for the specified xSCOPE probe with logical core info.

For use with XSCOPE_STARTSTOP probes.

Parameters

- **`id`** – xSCOPE probe id.

void **`xscope_core_stop`**(unsigned char id)

Stop a trace block for the specified xSCOPE probe with logical core info.

For use with XSCOPE_STARTSTOP probes.

Parameters

- **`id`** – xSCOPE probe id.

void **`xscope_core_start_int`**(unsigned char id, unsigned int data)

Start a trace block for the specified xSCOPE probe with logical core info and capture a value of type int For use with XSCOPE_STARTSTOP probes.

Parameters

- **`id`** – xSCOPE probe id.
- **`data`** – User data value (int).

void **`xscope_core_stop_int`**(unsigned char id, unsigned int data)

Stop a trace block for the specified xSCOPE probe with logical core info and capture a value of type int For use with XSCOPE_STARTSTOP probes.

Parameters

- **`id`** – xSCOPE probe id.
- **`data`** – User data value (int).

void **xscope_mode_lossless**( )

Put xSCOPE into a lossless mode where timing is no longer guaranteed.

If the logical core tries to send an xSCOPE packet which the xTAG does not have buffers for, then the logical core will wait until the xTAG does have buffers, stalling for a non-deterministic amount of time.

See also:

xscope_mode_lossy

void **xscope_mode_lossy**( )

Put xSCOPE into a lossy mode where timing is not impacted, but data is lossy.

This is the default xSCOPE mode. If the logical core tries to send an xSCOPE packet which the xTAG does not have buffers for, then the entire packet will be dropped.

See also:

xscope_mode_lossless

void **xscope_data_from_host**(unsigned int c, char buf[256], int *n)

Receive data from the host over xSCOPE.

This function receives an xSCOPE packet from the host and puts the data contained within it into the supplied buffer. See *Example host program and target programs* for an example.

See also:

xscope_connect_data_from_host

Parameters

- **c** – The xSCOPE chanend which has been configured with xscope_connect_data_from_host
- **buf** – The user-supplied buffer to fill with data from the host
- **n** – A pointer(c) or reference(xc) to an int which will be filled with the number of bytes put into the buffer

void **xscope_connect_data_from_host**(unsigned int from_host)

Connect to the xSCOPE chanend to receive data packets from the host.

Note that it is only possible to use this function once in the system, as there is only a single xSCOPE chanend available to receive from the host. i.e. it can only be called on a single tile. See *Example host program and target programs* for an example.

See also:

xscope_data_from_host

Parameters

- **from_host** – The chanend to receive on, obtained using the xscope_host_data() call from a multi-tile main.

__attribute__((deprecated)) static inline void **xscope_probe**(unsigned char id)

4.3.3.3 Tracing API

The tracing API is the mechanism to actually send event data to the host for a particular probe. It uses the high-speed xSCOPE link.

The 'id' for each of these functions is the id of the probe that the data is being sent on, which is either:

- A preprocessor definition for **config file probes**, as explained in *Auto-generated header file*
- Implicit for **dynamically defined probes**, as explained in *`xscope_register()`*

group **`xscope_trace_functions`**

Functions

void **`xscope_char`**(unsigned char id, unsigned char data)

Send a trace event for the specified xSCOPE probe of type char.

Parameters

- **`id`** – xSCOPE probe id.
- **`data`** – User data value (char).

void **`xscope_short`**(unsigned char id, unsigned short data)

Send a trace event for the specified xSCOPE probe of type short.

Parameters

- **`id`** – xSCOPE probe id.
- **`data`** – User data value (short).

void **`xscope_int`**(unsigned char id, unsigned int data)

Send a trace event for the specified xSCOPE probe of type int.

Parameters

- **`id`** – xSCOPE probe id.
- **`data`** – User data value (int).

void **`xscope_longlong`**(unsigned char id, unsigned long long data)

Send a trace event for the specified xSCOPE probe of type long long.

Parameters

- **`id`** – xSCOPE probe id.
- **`data`** – User data value (long long).

void **`xscope_float`**(unsigned char id, float data)

Send a trace event for the specified xSCOPE probe of type float.

Parameters

- **`id`** – xSCOPE probe id.
- **`data`** – User data value (float).

void **`xscope_double`**(unsigned char id, double data)

Send a trace event for the specified xSCOPE probe of type double.

Parameters

- **`id`** – xSCOPE probe id.
- **`data`** – User data value (double).

void **`xscope_bytes`**(unsigned char id, unsigned int size, const unsigned char data[])

Send a trace event for the specified xSCOPE probe with a byte array.

Parameters

- **`id`** – xSCOPE probe id.
- **`size`** – User data size.
- **`data`** – User data bytes (char[]).

void **`xscope_start`**(unsigned char id)

Start a trace block for the specified xSCOPE probe.

For use with XSCOPE_STARTSTOP probes.

Parameters

- **`id`** – xSCOPE probe id.

void **`xscope_stop`**(unsigned char id)

Stop a trace block for the specified xSCOPE probe.

For use with XSCOPE_STARTSTOP probes.

Parameters

- **`id`** – xSCOPE probe id.

void **`xscope_start_int`**(unsigned char id, unsigned int data)

Start a trace block for the specified xSCOPE probe and capture a value of type int.

For use with XSCOPE_STARTSTOP probes.

Parameters

- **`id`** – xSCOPE probe id.
- **`data`** – User data value (int).

void **`xscope_stop_int`**(unsigned char id, unsigned int data)

Stop a trace block for the specified xSCOPE probe and capture a value of type int.

For use with XSCOPE_STARTSTOP probes.

Parameters

- **`id`** – xSCOPE probe id.
- **`data`** – User data value (int).

void **`xscope_core_char`**(unsigned char id, unsigned char data)

Send a trace event for the specified xSCOPE probe of type char with logical core info.

Parameters

- **`id`** – xSCOPE probe id.
- **`data`** – User data value (char).

void **`xscope_core_short`**(unsigned char id, unsigned short data)

Send a trace event for the specified xSCOPE probe of type short with logical core info.

Parameters

- **`id`** – xSCOPE probe id.
- **`data`** – User data value (short).

void **`xscope_core_int`**(unsigned char id, unsigned int data)

Send a trace event for the specified xSCOPE probe of type int with logical core info.

Parameters

- **`id`** – xSCOPE probe id.
- **`data`** – User data value (int).

void **`xscope_core_longlong`**(unsigned char id, unsigned long long data)

Send a trace event for the specified xSCOPE probe of type long long with logical core info.

Parameters

- **`id`** – xSCOPE probe id.
- **`data`** – User data value (long long).

void **`xscope_core_float`**(unsigned char id, float data)

Send a trace event for the specified xSCOPE probe of type float with logical core info.

Parameters

- **`id`** – xSCOPE probe id.
- **`data`** – User data value (float).

void **`xscope_core_double`**(unsigned char id, double data)

Send a trace event for the specified xSCOPE probe of type double with logical core info.

Parameters

- **`id`** – xSCOPE probe id.
- **`data`** – User data value (double).

void **`xscope_core_bytes`**(unsigned char id, unsigned int size, const unsigned char data[])

Send a trace event for the specified xSCOPE probe with a byte array with logical core info.

Parameters

- **`id`** – xSCOPE probe id.
- **`size`** – User data size.
- **`data`** – User data bytes (char[]).

void **`xscope_core_start`**(unsigned char id)

Start a trace block for the specified xSCOPE probe with logical core info.

For use with XSCOPE_STARTSTOP probes.

Parameters

- **`id`** – xSCOPE probe id.

void **`xscope_core_stop`**(unsigned char id)

Stop a trace block for the specified xSCOPE probe with logical core info.

For use with XSCOPE_STARTSTOP probes.

Parameters

- **`id`** – xSCOPE probe id.

void **`xscope_core_start_int`**(unsigned char id, unsigned int data)

Start a trace block for the specified xSCOPE probe with logical core info and capture a value of type int For use with XSCOPE_STARTSTOP probes.

Parameters

- **`id`** – xSCOPE probe id.

- **`data`** – User data value (int).

void **`xscope_core_stop_int`**(unsigned char id, unsigned int data)

Stop a trace block for the specified xSCOPE probe with logical core info and capture a value of type int For use with XSCOPE_STARTSTOP probes.

Parameters

- **`id`** – xSCOPE probe id.
- **`data`** – User data value (int).

4.3.4 xSCOPE host library

The xSCOPE host library is used for communication between the host and the target over the fast xSCOPE transport using a custom user host program. It connects to the "xSCOPE server", and communicates while the program is running. For a full example, see *User-supplied host program*.

The communication takes place using "probes", which can be defined using *xSCOPE config file* or using the *xSCOPE target library*.

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.dll` on Windows hosts, using C linkage. These libraries are provided by the XTC Tools in the `lib` sub-directory of the installation root.

4.3.4.1 Example

For an example, see *Host program*.

4.3.4.2 API

xSCOPE host API

This file contains functions to communicate with an xSCOPE server.

Defines

`XSCOPE_EP_SUCCESS`

Status code: Function was successful.

`XSCOPE_EP_FAILURE`

Status code: Function failed.

Typedefs

typedef void (***xscope_ep_register_fptr**)(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)

Function pointer which will be called when the target registers new probes.

Probes registered using an xSCOPE config file will trigger this callback upon initial connection. Probes registered via calls to `xscope_register()` will trigger this callback when they are called.

Most of these parameters are configured directly by the target code/config file. The RGB color is currently chosen arbitrarily by the server.

Param id
The unique ID of the probe, which has been allocated by the server.

Param type
The type of the probe. See `xscope_EventType`

Param r
Red color value from 0-255 to visually represent the probe

Param g
Green color value from 0-255 to visually represent the probe

Param b
Blue color value from 0-255 to visually represent the probe

Param name
String representing the name of the probe

Param unit
String representing the unit of time being used (e.g. 'ps')

Param data_type
Type of the data (signed, unsigned or float). See `xscope_UserDataType`

Param data_name
String representing the unit of measurement of the probe (e.g. 'mV')

typedef void (***xscope_ep_record_fptr**)(unsigned int id, unsigned long long timestamp, unsigned int length, unsigned long long dataval, unsigned char *databytes)

Function pointer which will be called when a record for a probe is received from the target.

Param id
ID value which has previously been registered with a *xscope_ep_register_fptr* call

Param timestmp
Timestamp of the received record, in the units given in the xscope_ep_register_fptr call

Param length
0 if the value received is in dataval, otherwise it is the length of the data in databytes

Param dataval
The value received for the record. Only valid if the length is zero. The value should be cast based on the data_type argument provided by the *xscope_ep_register_fptr* call

Param databytes
The data buffer received for the record. Only valid if length is nonzero. The target can send this kind of message using `xscope_bytes()`

typedef void (***xscope_ep_stats_fptr**)(int id, unsigned long long average)

Function pointer which will be called with stats when requested using *xscope_ep_request_stats*.

Warning: The server does not implement this request.

Param id
Not implemented: always zero

Param average
Not implemented: value of 'data' from the server. Always 0xdeadbeef.

typedef void (*`xscope_ep_print_fptr`)(unsigned long long timestamp, unsigned int length, unsigned char *data)

Function pointer which will be called when the target executes a write syscall (such as a print)

Warning: This function gets called for all write syscalls, not just prints to stdout.

Param timestamp
The timestamp of the print record that has been received

Param length
The number of characters in the data buffer

Param data
Data buffer containing the data the target is writing.

typedef void (*`xscope_ep_exit_fptr`)()

Function pointer which will be called when *xscope_ep_disconnect* is called.

Warning: This does not automatically get called by anything internally. Only *xscope_ep_disconnect* calls this when it is called manually.

Functions

int `xscope_ep_set_register_cb`(*xscope_ep_register_fptr* registration)

Register a callback for receiving probe registration information.

Parameters

- `registration` – Callback

Return values

- `XSCOPE_EP_SUCCESS` – Success
- `XSCOPE_EP_FAILURE` – Failure, such as endpoint is already connected.

int `xscope_ep_set_record_cb`(*xscope_ep_record_fptr* record)

Register a callback for receiving probe record data.

Parameters

- `record` – Callback

Return values

- `XSCOPE_EP_SUCCESS` – Success
- `XSCOPE_EP_FAILURE` – Failure, such as endpoint is already connected.

int **`xscope_ep_set_stats_cb`**(*xscope_ep_stats_fptr* stats)

Register a callback for getting statistics.

> **Warning:** This system is not implemented.

Parameters

- **`stats`** – Callback

Return values

- **`XSCOPE_EP_SUCCESS`** – Success
- **`XSCOPE_EP_FAILURE`** – Failure, such as endpoint is already connected.

int **`xscope_ep_set_print_cb`**(*xscope_ep_print_fptr* print)

Register a callback for receiving data to print to the user.

> **Warning:** This callback is not implemented.

Parameters

- **`print`** – Callback

Return values

- **`XSCOPE_EP_SUCCESS`** – Success
- **`XSCOPE_EP_FAILURE`** – Failure, such as endpoint is already connected.

int **`xscope_ep_set_exit_cb`**(*xscope_ep_exit_fptr* exit)

Register a callback which will be called when *xscope_ep_disconnect* is called.

Parameters

- **`exit`** – Callback

Return values

- **`XSCOPE_EP_SUCCESS`** – Success
- **`XSCOPE_EP_FAILURE`** – Failure, such as endpoint is already connected.

int **`xscope_ep_connect`**(const char *ipaddr, const char *port)

Connect to an xSCOPE server which is running and waiting for a client to connect.

Parameters

- **`ipaddr`** – IPv4 address of the xSCOPE server or a host name which will be resolved to one
- **`port`** – Port of the xSCOPE server

Return values

- **`XSCOPE_EP_SUCCESS`** – Success
- **`XSCOPE_EP_FAILURE`** – Failure, such as endpoint is already connected, or failed to connect.

int **`xscope_ep_disconnect`**(void)

Disconnect from the connected xSCOPE server, and clean up.

Return values

- **XSCOPE_EP_SUCCESS** – Success
- **XSCOPE_EP_FAILURE** – Failure

int **xscope_ep_request_registered**(void)

No-op, unimplemented.

This is not required. The *xscope_ep_register_fptr* callback will be called as probe registrations are made, regardless of whether this function is called.

Warning: This is not implemented

Returns
Status value

int **xscope_ep_request_stats**(void)

Request stats from the xSCOPE server, and trigger any registered *xscope_ep_stats_fptr* callback.

Warning: This is not implemented

Returns
Status value

int **xscope_ep_request_upload**(unsigned int length, const unsigned char *data)

Send data to the target.

This will be received by `xscope_data_from_host()` on the target.

Parameters

- **length** – Length of the data buffer, in bytes. Must be 256 bytes or fewer
- **data** – The data buffer to send to the target

Return values

- **XSCOPE_EP_SUCCESS** – Success
- **XSCOPE_EP_FAILURE** – Failure, such as data too long, or endpoint not connected.

4.3.5 Related

- *xSCOPE config file*
- *Using xSCOPE for fast "printf debugging"*

4.3.5.1 xSCOPE FAQ

- **How do I just use xSCOPE for fast printing?**
  See the quick start guide: *Using xSCOPE for fast "printf debugging"* for basic xSCOPE usage.
- **Do I need both the -fxscope compiler option and an xSCOPE config file?**
  If you are not using a `config.xscope` file, you must specify the `-fxscope` option on the command line to use xSCOPE (e.g. when using *xSCOPE target library*). You do not need both.

- **What is xSCOPE "offline mode"?**
  xSCOPE "offline mode", which is also known as "xSCOPE file" mode, is the mode of xSCOPE which sends all xSCOPE records to a .vcd file to be viewed later with a VCD viewer such as GTKWave. This does not affect the "fast printf" xSCOPE messages, which still get printed to stdout. It only affects probes.
- **What is xSCOPE "realtime xSCOPE"?**
  xSCOPE "realtime mode", also known as "xSCOPE server" refers to the the mode of xSCOPE which sends all received records to an xSCOPE client implemented by the user. Note that the realtime property of the server is as opposed to "offline". It does not possess any hard real-time behaviour.
- **What happens if the throughput of messages is too high for xSCOPE to keep up?**
  That depends on the configuration of xSCOPE. By default, they will get dropped. This can be changed by calling *`xscope_mode_lossless()`*. In lossless mode, the transmitting thread will be stalled until there is bandwidth in the system to send more data.

  If the throughput is extremely high and the host system is very overloaded, it is possible for the host USB stack to drop packets.

4.3.5.2 Common & Known Issues

- **Cannot simultaneously use the xSCOPE config file and xSCOPE API xscope_register function:**
  It is not possible to use both the *xSCOPE config file* and the xSCOPE API *`xscope_register()`* function simultaneously. The dynamic registration will be ignored if an xSCOPE config file is used. You should see a warning similar to "System is already registered, new registration call ignored" when running with `xrun` or `xgdb`.
- **Cannot attach to a running program:**
  xSCOPE cannot be attached to a running program. The xSCOPE connection must be initialized and synchronized when the program is initially loaded with `xrun/xgdb`.
- **No output without `--xscope` on xrun/xgdb:**
  If you build your program with xSCOPE I/O enabled but do not specify the `--xscope` option in the `xrun` or `xgdb connect` command, you will not see any xSCOPE output.
- **Cannot print when using xSCOPE server**
  If the client of the xSCOPE server supports the 'print' capability (as the default implementation does), then the `xgdb/xrun` executable will no longer handle printing the messages itself: the client is expected to print them.
- **xSCOPE io only affects write syscalls**
  xSCOPE ioMode only improves the speed of write syscalls. Other sycalls such as *read* still go over JTAG, so those will not see a speed increase. The xSCOPE fileio library offers a system to improve this performance.
- **Using both xSCOPE io and the xSCOPE server will break file writing.**
  All write syscalls will get forwarded to the server, without the file descriptor, and not handled by the syscall code which would handle the syscall on the host. Consider using xSCOPE fileio if you need to use xSCOPE io and the xSCOPE server simultaneously.
- **Using xSCOPE timed io will prepend the timestamp to all write syscalls, not just print messages**
  The timed io is a convenience feature, which will not work for all applications. Disable it if you want precise control over the exact content that gets written to file.
- **Signal integrity issues**
  xSCOPE is very fast, but is more likely to experience signal integrity issues on some hardware configurations, such as with long cables.
- **Trying to use xSCOPE when the physical connection is not present will silently fail**
  The xSCOPE systems do not check that the connection is functional before relying on it, so this is up to the individual developer to ensure the connection is available before trying to use it.

4.4 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
```

4.4.1 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`.

4.4.2 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>
```

4.4.3 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.

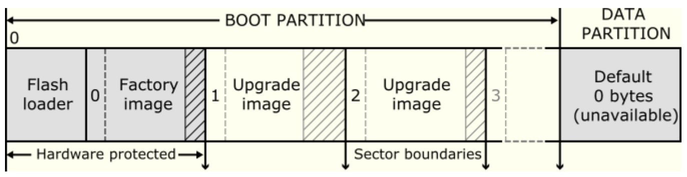


Fig. 4.3: Flash format diagram

All functions are prototyped in the header file `<flash.h>`. Except where otherwise stated, functions return 0 on success and non-zero on failure.

4.4.3.1 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 `fl_connect` and `fl_connectToDevice` require an argument of type `fl_SPIPorts`, which defines the four ports and clock block used to connect to the device.

```
typedef struct {
  in buffered port:8 spiMISO;
  out port spiSS;
  out port spiCLK;
  out buffered port:8 spiMOSI;
  clock spiClkblk;
} fl_SPIPorts;
```

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

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

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

> `fl_connectToDevice` opens a connection to an SPI device. It iterates through an array of *n* SPI device specifications, attempting to connect using each specification until one succeeds.

unsigned **`fl_getJedecId`**(void)

> `fl_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 **`fl_getFlashType`**(void)

> `fl_getFlashType` returns an `enum` value for the flash device. The enumeration of devices known to libflash is given below.

```
typedef enum {
  UNKNOWN = 0,
  ALTERA_EPCS1,
  ATMEL_AT25DF041A,
  ATMEL_AT25FS010,
  ST_M25PE10,
  ST_M25PE20,
  WINBOND_W25X40
} fl_FlashId;
```

> If the function call `fl_connectToDevice(p, spec, n)` is used to connect to a flash device, `fl_getFlashType` returns the parameter value `spec[i].flashId` where `i` is the index of the connected device.

unsigned **`fl_getFlashSize`**(void)

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

void **`fl_copySpec`**(fl_DeviceSpec *dest)

> `fl_copySpec` exports the completed SPI specification in use for the current connection.

unsigned **`fl_getLibraryStatus`**(void)

> `fl_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 **`fl_command`**(unsigned int cmd, unsigned char input[], unsigned int num_in, unsigned char output[], unsigned int num_out)

`fl_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 **`fl_disconnect`**(void)

`fl_disconnect` closes the connection to the SPI device.

4.4.3.2 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. Repeated calling of this function is required to iterate over all the sectors needed by the image and erase them.

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. Repeated calling of this function is required to iterate over all the sectors needed by the image and erase them.

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. Repeated calling of this function is required to iterate over all the sectors needed by the image and erase them.

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

> `fl_writeImagePage` waits until the SPI device is able to accept a request and then outputs the next page of data to the device. Attempting to write past the maximum size passed to `fl_startImageReplace`, `fl_startImageAdd` or `fl_startImageAddAt` is invalid.

int **`fl_writeImageEnd`**(void)

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

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

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

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

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

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

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

4.4.3.3 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 **`fl_getDataPartitionSize`**(void)

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

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

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

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

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

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

> `fl_writeData` writes the array `src` to the specified offset in the data partition. It uses the array `buffer` to preserve page data that must be re-written.

4.4.3.3.1 Page-level functions

unsigned **`fl_getPageSize`**(void)

> `fl_getPageSize` returns the page size in bytes.

unsigned **`fl_getNumDataPages`**(void)

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

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

> `fl_writeDataPage` writes the array data to the *n*-th page in the data partition. The data array must be at least as big as the page size; if larger, the highest elements are ignored.

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

> `fl_readDataPage` reads the *n*-th page in the data partition and writes it to the array data. The size of data must be at least as large as the page size.

4.4.3.3.2 Sector-level functions

unsigned **`fl_getNumDataSectors`**(void)

> `fl_getNumDataSectors` returns the number of sectors in the data partition.

unsigned **`fl_getDataSectorSize`**(unsigned n)

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

unsigned **`fl_eraseDataSector`**(unsigned n)

> `fl_eraseDataSector` erases the *n*-th sector in the data partition.

unsigned **`fl_eraseAllDataSectors`**(void)

> `fl_eraseAllDataSectors` erases all sectors in the data partition.

4.4.4 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.

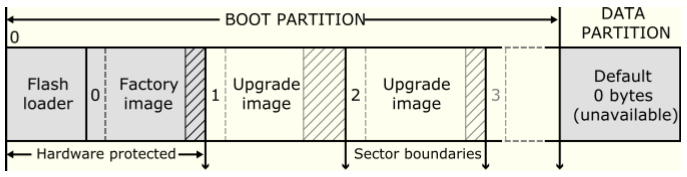


Fig. 4.4: Flash format diagram

All functions are prototyped in the header file `<quadflash.h>`. Except where otherwise stated, functions return 0 on success and non-zero on failure.

4.4.4.1 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 `fl_connect` and `fl_connectToDevice` require an argument of type `fl_QSPIPorts`, which defines the four ports and clock block used to connect to the device.

```
typedef struct {
  out port qspiCS;
  out port qspiSCLK;
  out buffered port:32 qspiSIO;
  clock qspiClkblk;
} fl_QSPIPorts;
```

int **`fl_connect`**(fl_QSPIPorts *SPI)

`fl_connect` opens a connection to a Quad-SPI device.

This API requires a flash device that supports SFDP.

int **`fl_connectToDevice`**(fl_QSPIPorts *SPI, fl_QuadDeviceSpec spec[], unsigned n)

`fl_connectToDevice` opens a connection to a Quad-SPI device. It iterates through an array of *n* Quad-SPI device specifications, attempting to connect using each specification until one succeeds.

This API makes use of SFDP if available on the target flash device.

int **`fl_connectToDeviceLight`**(fl_QSPIPorts *SPI, fl_QuadDeviceSpec spec[], unsigned n)

`fl_connectToDeviceLight` opens a connection to a Quad-SPI device. It iterates through an array of *n* Quad-SPI device specifications, attempting to connect using each specification until one succeeds.

This API does not use SFDP and expects a complete device specification to be provided. It is provided for a reduced memory footprint.

unsigned **`fl_getJedecId`**(void)

`fl_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`.

unsigned **`fl_getFlashSize`**(void)

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

void **`fl_copySpec`**(fl_QuadDeviceSpec *dest)

`fl_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 **`fl_getLibraryStatus`**(void)

`fl_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,
} fl_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 **`fl_command`**(unsigned int cmd, unsigned char input[], unsigned int num_in, unsigned char output[], unsigned int num_out)

`fl_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 **`fl_disconnect`**(void)

`fl_disconnect` closes the connection to the Quad-SPI device.

4.4.4.2 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. Repeated calling of this function is required to iterate over all the sectors needed by the image and erase them.

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. Repeated calling of this function is required to iterate over all the sectors needed by the image and erase them.

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. Repeated calling of this function is required to iterate over all the sectors needed by the image and erase them.

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

> `fl_writeImagePage` waits until the Quad-SPI device is able to accept a request and then outputs the next page of data to the device. Attempting to write past the maximum size passed to `fl_startImageReplace`, `fl_startImageAdd` or `fl_startImageAddAt` is invalid.

int **`fl_writeImageEnd`**(void)

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

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

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

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

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

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

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

4.4.4.3 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 **`fl_getDataPartitionSize`**(void)

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

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

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

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

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

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

> `fl_writeData` writes the array `src` to the specified offset in the data partition. It uses the array `buffer` to preserve page data that must be re-written.

4.4.4.3.1 Page-level functions

unsigned **`fl_getPageSize`**(void)

> `fl_getPageSize` returns the page size in bytes.

unsigned **`fl_getNumDataPages`**(void)

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

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

> `fl_writeDataPage` writes the array data to the *n*-th page in the data partition. The data array must be at least as big as the page size; if larger, the highest elements are ignored.

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

> `fl_readDataPage` reads the *n*-th page in the data partition and writes it to the array data. The size of data must be at least as large as the page size.

4.4.4.3.2 Sector-level functions

unsigned **`fl_getNumDataSectors`**(void)

> `fl_getNumDataSectors` returns the number of sectors in the data partition.

unsigned **`fl_getDataSectorSize`**(unsigned n)

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

unsigned **`fl_eraseDataSector`**(unsigned n)

> `fl_eraseDataSector` erases the *n*-th sector in the data partition.

unsigned **`fl_eraseAllDataSectors`**(void)

> `fl_eraseAllDataSectors` erases all sectors in the data partition.

4.4.5 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 4.36: 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.

4.4.6 List of devices natively supported by libquadflash

libquadflash supports a wide range of flash devices available in the market. Devices with SFDP (Serial Flash Discoverable Parameter) capability expecting 6 dummy clock cycles and capable of 20MHz speeds typically need the least (or no) configuration for use with the tools and the XMOS boot loader.

For devices with an alternate number of dummy cycles, it may be possible to reprogram the device using a non-volatile status register or equivalent to the required value of 6 for compatibility with XMOS boot. This would be indicated in the flash device's datasheet and can be performed using the XFLASH *SPI Command Option*.

For devices lacking SFDP support, the tools allow for manual configuration using a Quad-SPI specification file. Refer to the device's datasheet and *Add support for a new flash device* for information on the specification file format and advice.

Quad SPI devices may need a non-volatile QE (Quad Enable) bit to be set to be supported by XMOS boot. In some cases the bit may already be set by the manufacturer. If the device supports SFDP the tools will set it automatically when programming the flash.

Historically, libquadflash included a built-in list of natively supported devices similar to the *libflash SPI library*. This has been removed in favour of SFDP support, which all previously supported devices are capable of.

Below is a table of Quad SPI devices that libquadflash has recently been verified against which can be recommended, but this list is not exhaustive - for example, devices in the same family should also be compatible.

Table 4.37: Examples of flash devices compatible with libquadflash

Manufacturer	Part Number	Comment
ADESTO	AT25FF161A	Factory-programmed default dummy cycles may be incompatible, but can be reprogrammed using Write Status Register 5 command 71h. XFLASH example: `--spi-cmd 0x06 --spi-cmd 0x71 0 0x05 0x20` This sets the fifth register to value 20h, indicating 6 dummy cycles - see the flash device's datasheet.
	AT25FF321A	Ditto.
ISSI	IS25LP016D	
	IS25LP032	
	IS25LP064	
	IS25LP080D	
	IS25LP128	
	IS25LQ016B	
	IS25LQ032B	
	IS25LQ080B	
SPANSION	S25FL116K	
	S25FL132K	
	S25FL164K	
WINBOND	W25Q128JV	
	W25Q16JV	
	W25Q32JV	
	W25Q64JV	

4.4.7 lib_otp

The lib_otp library provides functions for reading from and programming data to the OTP (One Time Programmable) memory available on each tile.

Two implementations of the library are provided. The original `lib_otp` implements the API for the XCORE-200 family of devices and earlier, while `lib_otp3` implements it for the xcore.ai family.

To use the library, include either the `otp.h` or `otp3.h` header file and link your project using either `-lotp` or `-lotp3`.

Danger: Incorrect use of this library can result in irreversible damage to the device. It is documented for completeness, and its usage is unnecessary in the vast majority of applications. The most common operations can be performed using XBURN instead, which internally drives this library.

Note: xcore.ai does not have a "special register", and so `lib_otp3` will trap if either `otp_read_special_register` or `otp_program_special_register` are called.

4.5 XCommon CMake build system

The XCommon CMake system is a build and dependency management tool which is based on CMake. It provides utility functions for the standard CMake tool to accelerate xcore application development without requiring knowledge of the CMake language.

For full documentation, please refer to the XCommon CMake documentation.

4.6 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.

Note: The XCOMMON build system is deprecated and users who require this functionality should use XCommon CMake instead.

4.6.1 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.

4.6.1.1 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_`.

4.6.1.1.1 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`.

4.6.1.2 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.

4.6.1.3 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.

4.6.2 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/
          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).

4.6.2.1 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
```

4.6.2.2 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/
         support_fns.xc
       include/
         my_library.h
```

4.6.2.3 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.

4.7 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.

4.7.1 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 futher information on communicating sequential processes see references.

4.7.1.1 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*.

4.7.1.2 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 availabilty 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 implicitly by the language itself. These are all normal considerations for a C programmer.

The following XC to C conversion sheet cheat is provided.

4.7.2 Graphical IDE

The XTC Tools no longer contain a graphical IDE within the installation package. Instead, the tools are intended to be used with VS Code as shown in *Using VS Code*.

Viewing of offline XSCOPE output is delegated to the user's preferred VCD viewer. Use of GTKWave is demonstrated in *Using xSCOPE for fast "printf debugging"*, and other viewers are available.

4.7.3 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.

4.7.3.1 Flash-related changes

XFLASH and the libquadflash library now support the SFDP standard. This reduces and in some cases eliminates manual configuration that previously had to be specified by the developer. Manual configuration is now considered as an override to the result of the SFDP query, whether supplied by XN file or SPI-spec file. Beware that if values for page size and number of pages are not set via XN file or SPI-spec file, XFLASH will have to interrogate the flash device to generate a flash image, which requires the flash build process to have the target device attached.

This may have consequences for migrating older xTIMEComposer projects to this release, due to the change in memory footprint of the library. Other aspects of the library have been further optimised for size to offset this but in the worst case scenario `fl_connectToDeviceLight()` can be used to restore the xTIMEComposer / Tools 14 behaviour and further decrease memory utilisation.

4.7.3.1.1 XFLASH option changes

The version of XFLASH supplied with the XTC Tools 15 series 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.

4.7.3.1.2 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`*.

4.7.3.2 OTP-related changes

The `xburn --no-compression` option has similarly been removed, along with the update check facility, so likewise the option `xburn --no-update-check` is no longer provided.

In addition, the XCORE-200 `xburn --serial-number` family of options has been removed from the XBURN host tool - compatible data structures can be generated instead in CSV format using the example Python script referenced in *CSV Format* and subsequently programmed via XBURN.

4.7.3.3 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.