XMOS XTC Tools - Programming Guide

Publication Date: 2025/12/11
Document Number: XM-014363-PC v15.3

XMOS

IN THIS DOCUMENT

Programming models

The logical cores of an XCore multicore processor can be programmed in extremely flexible ways:

"Hardware as Software"
Individual logical cores, or collaborations thereof, can respond within a few clock cycles to specific hardware events. Once activated, each logical core is given a guaranteed share of the processor cycles. With this programming model, parts of the XCore can be thought of as software-defined hardware peripherals.

"Vector accelerator"
Logical cores may act as a team operating on the same vector. This approach, combined with VPU acceleration, allows the greatest utilisation of the available compute.

"Application processor"
As well as responding to events, logical cores can respond to interrupts. This allows a logical core to behave like a traditional MCU running an RTOS.

These application architecture approaches may be combined and may even be chosen as appropriate at runtime. *The multicore programming model* explores these concepts in more depth.

Languages

XCore processors are, in the main, programmed using the C (or C++) language with special hardware features being accessed through system libraries. See *Programming an XCORE tile with C and lib_xcore* for more details.

Where a particularly performant piece of code is required, a developer might chose to program in assembly.

1 Architecture and hardware guide

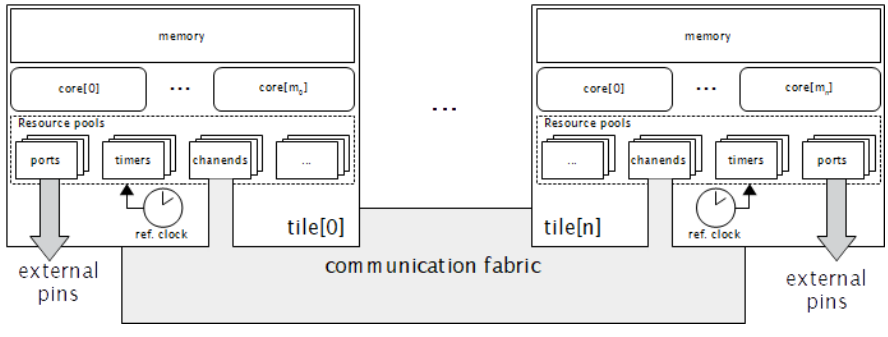
xcore is a multicore microprocessor which enables highly flexible and responsive I/O, whilst delivering high performance for applications. The architecture enables a programming model where many simple tasks run concurrently and communicate using hardware support. Multiple xcore processors can be interconnected, and tasks running on separate physical processors can communicate seamlessly. To utilise the platform effectively it is helpful to understand the hardware model; the purpose of this document is to provide an overview of the platform and its features, and an introduction to utilising them using the C programming language.

1.1 Overview

The xcore architecture facilitates scaling of applications over multiple physical packages in order to provide high performance as well as low-latency I/O. A complete xcore application targets a set of xcores and communication between them uses features of the xcore hardware (see below).

An xcore network is made up of one or more device *packages*; these are connected by the xCONNECT interconnect to allow high-speed, hardware-assisted communication.

- Each *package* contains one or more *nodes*
- Each *node* contains one or more *tiles* plus interconnect which allows communication within and between tiles.
- Each *tile* contains one or more *logical cores*, some memory, a reference clock, and a variety of *resources*.
- Each *logical core* is a hardware thread - it shares the tile's memory and *resources* with other cores, but each logical core has its own register set and can operate independently of the others.
- There are several *resources* types which exist on each tile. They can be claimed by a logical core, used and released for use by the same or another logical core. *chanend* resources are used to form channels for communication between logical cores. *timer* resources provide a logical core with a timestamp or a facility to wait for a time period, based on the reference clock. *Port* and *clock block* resources facilitate flexible GPIO.



1.1.1 Nodes

Each physical package typically contains one node but may contain more. Multiple packages may be connected using xLINKs, to provide a multi-node system. A node typically contains two tiles.

1.1.2 Tiles

Tiles are individual and independent processing units contained within nodes; each tile has its own memory, I/O subsystem, clock divider and other resources. Tiles within a node communicate using the communication fabric contained within that node, and can communicate with tiles in other packages using xLINKs.

1.1.3 Logical cores

Each tile has eight logical cores. Each logical core has its own registers and executes instructions independently of the other logical cores. However, all logical cores within a tile share access to that tile's resources and memory. The xcore pipeline has five stages and each stage takes one system clock cycle to complete. Almost every xcore instruction takes five cycles to execute using this pipeline. This makes it straightforward to calculate the duration of a straight-line instruction sequence. Five logical cores can operate in parallel, but staggered such that on a given clock cycle each will be using a different pipeline stage. These five will run independently and each gets one fifth of the MIPS (machine instructions per second) available to the entire tile. When more than five logical cores are active the relative rate of execution of each will drop to share the five pipeline stages between them. The xcore hardware scheduler uses round-robin to allocate each logical core a time slice.

A logical core may be put into a "paused state" if it is waiting for a resource (see below) to satisfy a specified condition (for example a timer reaching a required value). When a logical core is in the paused state it is removed from the list of logical core scheduled by the xcore hardware scheduler. Once the resource satisfies the required condition the logical core is put back on the list of logical cores to schedule.

1.1.4 I/O and pooled resources

Resources are shared between all cores in a tile. Many types of resource are available, and the exact types and numbers of resources vary between devices. Resources are general-purpose peripherals which help to accelerate real-time tasks and efficient software implementations of higher-level peripherals e.g. UART. Many of the available resource types are described in later sections. Due to the diverse nature of resources, their interfaces vary somewhat. However, most resources have some or all of the following traits:

- Pooled - the xcore tile maintains a pool of the resource type - a core can allocate a resource from the pool and free it when no longer required.
- Input/Output - values can be read from and/or written to the resource (for example, the value of a group of external pins).
- Event-raising - the resource can generate events when a condition occurs (on input resources, this will indicate that data is available to be read). Events can wake a core from a paused state.
- Configurable triggers - some 'event-raising' resources can be configured to generate events under programmable conditions.

Though available resources vary, all tiles have a number of common resource types:

- Ports - These provide input from and output to the physical pins attached to the tile. Ports are highly configurable and can automatically shift data in and out as well as generate events on reading certain values. As they have a fixed mapping to physical pins, ports are allocated explicitly (rather than from a pool), and have fixed widths which can be 1, 4, 8, 16 or 32 bits.

- Clock Blocks - Configurable clocks for controlling the rate at which a port shifts in/out data. These can divide the reference clock or be driven by a single-bit port.
- Timers - Provide a means of measuring time as well as generating events at fixed times in the future - this can be used to implement very precise delays.
- Chanends - An endpoint for communicating over the network fabric. A chanend (short for 'channel end') can communicate with any other chanend in the network (whether on the same tile, or on a different physical node).

1.1.5 Communication fabric

The communication fabric is a physical link between channel ends within a network, which allows any channel end to send data to any other channel end. When one channel end first sends data to another, a path through the network is established. This path persists until closed explicitly (usually as part of a transaction) and handles all traffic from the sender to the receiver during that time. Links are directed; so if channel end **A** sends data to channel end **B**, and then (without the link being closed) **B** sends data back to **A**, two links will be opened. These two links will not necessarily take the same route through the network. The communication capacity between channel ends within a single node is always enough for at least two links to be open. Between nodes, the capacity depends on the number of physical links which are connected.

2 Programming overview

Note

The reader is expected to already be familiar with:

- *XCore architecture fundamentals*

2.1 The multicore programming model

2.1.1 Parallel tasks of execution

A multicore program consists of several tasks. Typically, these tasks all start at the beginning of the program and run indefinitely - though the XCORE architecture does allow for transitory tasks too. Each task is an ordinary function. A task function usually consists of some one-off setup followed by a never-ending loop which continuously handles events from one or more resources. Each task in a program runs on its own core, meaning that there is a hard limit on the number of tasks which can execute on a single tile. Where there are too many conceptual 'event listeners' in a system to give each its own core, the XCORE architecture allows multiplexing and demultiplexing of events. This effectively allows multiple tasks to run cooperatively on a single core.

2.1.2 Channel-based communication

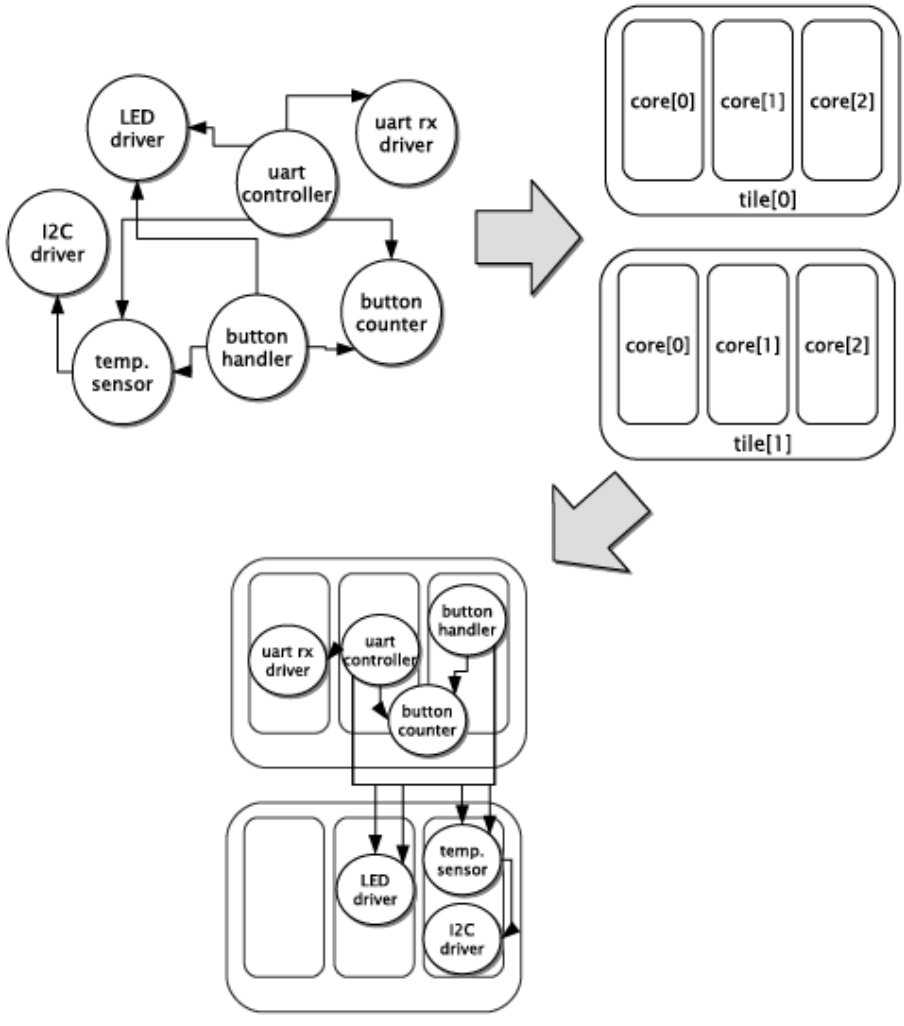
A key concept in the XCore architecture and the programming model is channel-based communication. In this model, when two tasks need to communicate, they are connected by a channel. In practice this means that each task takes control of a *chanend* resource which represents its end of the channel. Both tasks must enter a transacting state before any information is exchanged - that is, a send operation will block until the receive end is ready, and a receive operation will block until data is sent. A route through the network is automatically created when one channel end attempts to send data to another and is typically 'torn down' at the end of a transaction.

The following diagram shows how a system of tasks, connected by channels, might be placed on a pair of tiles:

2.1.3 Event-based programming

Typically, a task will act in response to one or more external events, such as a change in value on a GPIO pin or receiving data from another task via a channel. Events are supported at the hardware level in the XCore architecture; they are generated by event-raising resources and are conceptually similar to interrupts. Each event-raising resource has a trigger condition; often this condition is configurable. When a resource's trigger condition occurs, an event 'becomes available' on the resource. Each such resource has an 'event vector' which, like an interrupt vector, is the address of handling code for the event. Unlike an interrupt, control is only transferred to an event handler when the core enters an explicit wait-state. At this point, any available event may be taken, but any other waiting event remains available to be handled next time the core enters a waiting state. If no events are available when the core enters the wait state, it is paused until an event becomes available (which will be handled immediately).

This model simplifies application code as it strictly limits the points when event handling code may run. This additionally makes reasoning about execution time significantly simpler.



2.2 Programming an XCORE tile with C and lib_xcore

The XCORE compiler (`xcc`) supports targeting the XCORE using GNU C or C++. A C platform library lib_xcore_api provides access to XCORE-specific hardware features. *Lib_xcore* is available as a collection of system headers when using the XCC C compiler; the names of the headers all begin `xcore/`.

2.2.1 Parallel execution

The following code listing shows a multicore "hello world" application using lib_xcore. Note that, as usual, the program has a single `main` function as its entrypoint - `main` starts on a single core and lib_xcore must be used to start tasks on other cores.

```
1  #include <stdio.h>
2  #include <xcore/parallel.h>
3
4  DECLARE_JOB(say_hello, (int));
5  void say_hello(int my_id)
```

(continues on next page)

(continued from previous page)

```
{
  printf("Hello world from %d!\n", my_id);
}

int main(void)
{
  PAR_JOBS(
    PJOB(say_hello, (0)),
    PJOB(say_hello, (1)));
}
```

2.2.1.1 PAR_JOBS

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

DECLARE_JOB(say_hello, (int));
void say_hello(int my_id)
```

The PAR_JOBS macro, from `xcore/parallel.h`, is a lib_xcore construct which makes two or more function calls, each on its own core. Each invocation of the PJOB macro represents a task - the first argument is the function to call, the second the argument pack to call it with. Note that a function called by PAR_JOBS must be declared using DECLARE_JOB in the same translation unit as the PAR_JOBS - the parameters are the name of the function and a pack of typenames which represent the argument signature. Tasks always have a `void` return type.

```
  PAR_JOBS(
    PJOB(say_hello, (0)),
    PJOB(say_hello, (1)));
```

When PAR_JOBS is executed, each of the 'PJobs' is executed in parallel, with each one executing on a different core of the current tile. There is an implicit 'join' at the end of this, meaning that execution does not continue beyond PAR_JOBS until all the launched tasks have returned. Due to each PJob running on its own core, at the point the PAR_JOBS executes there must be enough free cores available to run all its jobs - this means that there is a hard limit on the number of jobs which may run, which is the number of cores on the tile. In the case that some cores are already busy running tasks, the number of jobs which can run will be fewer. In the case that there are not enough cores available, the PAR_JOBS will cause a trap at runtime.

Stack size calculation In the previous examples, multiple tasks were launched by a single thread, and yet it was not necessary to specify where the stacks for those threads should be located. This is because the PAR_JOBS macro allocates a stack for each thread launched, as a 'sub stack' of the launching thread's stack. The size of the stack allocated is calculated by the XMOS tools, which also ensure that the stack allocated to the calling thread is sufficient for all its launched threads and callees.

It is possible to view the amount of RAM which has been allocated to stacks using the `-report` flag to the compiler:

```
$ xcc -target=XK-EVK-XU316 hello_world.c -report
Constraint check for tile[0]:
  Memory available:       524288,   used:      22408 .  OKAY
    (Stack: 1620, Code: 19528, Data: 1260)
Constraints checks PASSED.
```

To enable this calculation, the compiler adds special symbols for each function which describe their stack requirements. These symbols are detailed in the XCore ABI document. The tools are not always able to determine the stack requirement of a function - such as when it calls a function through a pointer, or when it is recursive. In the case that the compiler cannot deduce the stack size requirement of a function, it is necessary to provide a worst-case requirement manually. This is possible either by defining the symbols manually, or by annotating the code (this is discussed later in this document).

2.2.2 Hardware timers

Timers are a simple resource which can be read to retrieve a timestamp. Timers can be configured with a trigger time, which causes the read operation to block until that timestamp has been reached. This makes timers suitable for measuring time as well as delaying by a known period. Timers can be accessed using lib_xccore's `xcore/hwtimer.h` - this provides a number of functions for interacting with timer resources. Like all resource-related functions, lib_xcore's timers work on a resource handle. As timers are a pooled resource, the XCore keeps track of available timers and allocates handles as required. `hwtimer_alloc` allocates a timer from the pool and returns its handle. As there is a limited number of timers available, the allocation can fail - in which case it will return 0.

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

int main(void)
{
  hwtimer_t timer = hwtimer_alloc();

  printf("Timer value is: %ld\n", hwtimer_get_time(timer));
  hwtimer_delay(timer, 100000000);
  printf("Timer value is now: %ld\n", hwtimer_get_time(timer));

  hwtimer_free(timer);
}

```

2.2.3 Channel communications

The standard way of allowing tasks to communicate is to use the XCORE's 'chanend' resources to send data through the communication fabric. This has the advantage of allowing synchronised transactions even with tasks running on a different tile or package.

2.2.3.1 Channels

Lib_xcore provides a channel type in `xcore/channel.h` which is used to establish a channel between two tasks on the current tile. The following code listing is a program where the `sends_first` task sends one word of data to `receives_first`, which responds with a single byte. In this program, `channel_alloc` is called to establish a channel which can be used by two tasks to communicate. Notice that the returned `channel_t` is simply a structure containing the two ends of the channel. The channel is created before the tasks are started, and each task is passed a different end of the channel.

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

DECLARE_JOB(sends_first, (chanend_t));
void sends_first(chanend_t c)
{
  chan_out_word(c, 0x12345678);
  printf("Received byte: '%c'\n", chan_in_byte(c));
}

DECLARE_JOB(receives_first, (chanend_t));
void receives_first(chanend_t c)
{
  printf("Received word: 0x%lx\n", chan_in_word(c));
  chan_out_byte(c, 'b');
}

int main(void)
{
  channel_t chan = chan_alloc();

  PAR_JOBS(
    PJOB(sends_first, (chan.end_a)),
    PJOB(receives_first, (chan.end_b)));

  chan_free(chan);
}
```

This approach has the advantage that the two tasks are decoupled - as long as they implement the same application-level protocol, neither one needs knowledge of what it is communicating with or where the other end of the channel is executing within the network.

Within the tasks, `chan_` functions are used to send and receive data. These functions synchronise the tasks since `chan_out_word` will block until `chan_in_word` is called in the other task, and vice versa. It is imperative that the correct corresponding 'receive' function is called for each 'send' function invocation, otherwise the tasks will usually deadlock or raise a hardware exception.

2.2.3.2 Streaming channels

Regular channels implement handshaking for each send/receive pair. This has the advantage that it synchronises the participating threads, and additionally prevents tasks progressing if more or less data is sent than was expected by the receiving end. Whilst handshaking is desirable in most cases, it does incur a runtime penalty - in the time taken to perform the actual handshake, and often by the sending task being held up by synchronisation.

Streaming channels are provided to address this. For each `chan_` function in `xcore/channel.h`, there is a corresponding `s_chan_` function in `xcore/channel_streaming.h`. These have the same functionality as their non-streaming counterparts, except that handshaking is not performed. Each chanend has a buffer which can hold at least one word of data. Sending data over a steaming channel only blocks when there is insufficient space in the buffer for the outgoing data.

2.2.3.2.1 Routing capacity

The communication fabric within and between XCore packages has a limited capacity meaning that there is a limit to the number of routes between channel ends which can be in use at any given time. If no capacity is available in the network when a task attempts to send data, that task is queued until capacity becomes available. Regular channels establish and 'tear down' a route through the network on each transaction, as a side-effect of the handshake. This means that routes remain open for very limited times, and all tasks have an opportunity to utilise the network. However, as streaming channels do not perform handshaking, their routes remain open for the entire duration between their allocation and deallocation. If too many streaming channels are left open, this can starve other tasks of access to the network (this includes tasks using non-streaming channels). For this reason, it is advisable to leave streaming channels allocated for as little time as possible.

2.2.4 Basic port I/O

Ports are a resource which allow interaction with the external pins attached to an XCore package. Each tile has a variety of ports of different widths; these will often have pins in common, and the actual mapping of ports to pins varies by package. Ports are extremely flexible and can be used to implement I/O peripherals as software tasks.

Unlike pooled resources, ports are not allocated by the XCore (as a port mapped to the desired pins will always be required). Instead, port handles are available from `platform.h` and have the form `XS1_PORT_<W><I>` where **<W>** is the width of the port in bits, and **<I>** is a single letter to differentiate the port from others of the same width. E.g. `XS1_PORT_16A` is the first 16 bit port. As ports are not managed by a pool, they must be enabled explicitly before use, and disabled when no longer required.

Like timers, Ports have a configurable trigger which, when set, causes reading the port to become blocking until the trigger condition is met. For example, it is possible to configure a port so that input is blocking until its pins read a particular value. By default, this trigger persists so input will be non-blocking whilst the condition is met but will become blocking again once the trigger condition becomes false. Functions for interacting with ports - including input, output and configuring triggers - are available from `xcore/port.h`.

The following program lights an LED while a button is held. Each time the port is read, its trigger condition is updated to occur when its value is not equal to its current one - this means the port is only read once each time the value on its pins changes.

```
#include <platform.h>
#include <xcore/port.h>

int main(void)
{
  port_t button_port = XS1_PORT_4D,
         led_port = XS1_PORT_4C;

  port_enable(button_port);
  port_enable(led_port);

  while (1)
  {
    unsigned long button_state = port_in(button_port);
    port_set_trigger_in_not_equal(button_port, button_state);
    port_out(led_port, ~button_state & 0x1);
  }

  port_disable(led_port);
  port_disable(button_port);
}
```

2.2.5 Event handling

Events are an important concept in the XCore architecture as they allow resources to indicate that they are ready for an input operation. Lib_xore provides a *select* construct for waiting for an event and running handling code when it occurs.

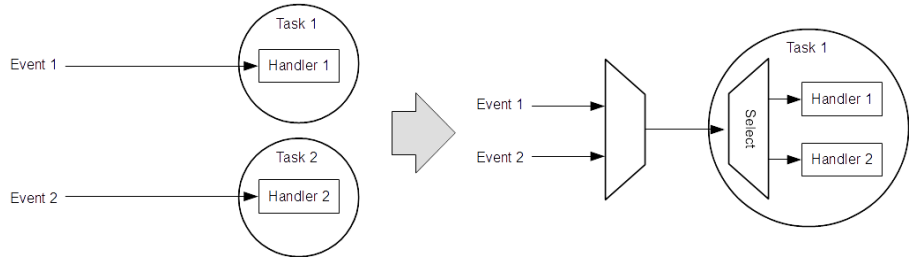
```
#include <platform.h>
#include <xcore/port.h>
#include <xcore/select.h>

int main(void)
{
  port_t button_port = XS1_PORT_4D,
         led_port = XS1_PORT_4C;

  port_enable(button_port);
  port_enable(led_port);

  SELECT_RES(
    CASE_THEN(button_port, on_button_change))
  {
  on_button_change: {
    unsigned long button_state = port_in(button_port);
    port_set_trigger_in_not_equal(button_port, button_state);
    port_out(led_port, ~button_state & 0x1);
    continue;
  }
  }

  port_disable(led_port);
  port_disable(button_port);
}
```

The above application shows a lib_xcore `SELECT_RES`, from `xcore/select.h`, which handles a single event. This is equivalent the example in *Basic port I/O*, except that the `port_in` will not block since it is not executed until the port's trigger condition is met (instead, the thread will block at the start of the `SELECT_RES`).

In this example, and previous examples in this document, the task blocks waiting for a single resource. Conceptually this is often a good design, and is the model generally encourage in the programming model. However, this is generally not an effective use of the available cores - most tasks will be paused in a wait state most of the time. It is also possible that a task needs to be able to accept input from multiple resources (e.g. a 'collector' task which listens to multiple channels).

For these reasons, the XCore allows a task to configure multiple resources to generate events, and then wait for any one of those events to occur. The lib_xcore *select* construct supports this - making it analogous to a Unix select. Conceptually, the XCore multiplexes events which are of interest to a task, and the *select* demultiplexes them.

The following listing introduces a timer which allows the LED to flash whilst the button is held. If an event occurs whilst another is being handled, the later event remains available on its respective resource and will be handled once the running handler continues.




```
#include <platform.h>
#include <xcore/hwtimer.h>
#include <xcore/port.h>
#include <xcore/select.h>

int main(void)
{
  port_t button_port = XS1_PORT_4D,
         led_port = XS1_PORT_4C;

  hwtimer_t timer = hwtimer_alloc();
  port_enable(button_port);
  port_enable(led_port);

  int led_state = 0;
  unsigned long button_state = port_in(button_port);
  port_set_trigger_in_not_equal(button_port, button_state);


  SELECT_RES(
    CASE_THEN(button_port, on_button_change),
    CASE_THEN(timer, on_timer))
  {
  on_button_change:
    button_state = port_in(button_port);
    port_set_trigger_value(button_port, button_state);
    continue;

  on_timer:
    hwtimer_set_trigger_time(timer, hwtimer_get_time(timer) + 20000000);
    if (~button_state & 0x1) {
      led_state = !led_state;
      port_out(led_port, led_state);
    }
    continue;
  }

  port_disable(led_port);
  port_disable(button_port);
  hwtimer_free(timer);
}
```

The SELECT_RES macro takes one or more arguments, which must be expansions of 'case specifiers' from the same header. The most common case specifier is CASE_THEN. This takes two parameters: the first being a resource to wait for an event on, and the second a label within the SELECT block to jump to when the event occurs. On entry to the SELECT_RES the task enters a wait state, when an event becomes available on one of the specified resources control is transferred to the label specified as the handler. Each handler must end with either `break` or `continue`; `break` causes control to jump the point after the SELECT block, whereas `continue` returns to the waiting state to handle another event.

In the preceding example the timer's trigger is adjusted into the future each time it is reached, causing it to trigger indefinitely at regular intervals. During the handler for the timer event the LED state is toggled only if the button is currently held. This structure means that the task will spend most of the time waiting for either of the two possible events. However, the timer event still occurs even when the button isn't held - so the handler runs even though it won't have any effect. The select construct has the facility to conditionally mask events which aren't always of interest - this is called a 'guard expression'. In the following listing, the guard expression prevents the `on_timer` event from being handled until its condition is met. The guard expression is re-evaluated each time

the SELECT enters its wait state; if an event has occurred whilst 'masked' by a guard expression, it will be handled once its condition is re-evaluated and true.

```
#include <platform.h>
#include <xcore/hwtimer.h>
#include <xcore/port.h>
#include <xcore/select.h>

int main(void)
{
  port_t button_port = XS1_PORT_4D,
         led_port = XS1_PORT_4C;

  hwtimer_t timer = hwtimer_alloc();
  port_enable(button_port);
  port_enable(led_port);

  int led_state = 0;
  unsigned long button_state = port_in(button_port);
  port_set_trigger_in_not_equal(button_port, button_state);


  SELECT_RES(
    CASE_THEN(button_port, on_button_change),
    CASE_GUARD_THEN(timer, ~button_state & 0x1, on_timer))
  {
  on_button_change:
    button_state = port_in(button_port);
    port_set_trigger_value(button_port, button_state);
    continue;

  on_timer:
    hwtimer_set_trigger_time(timer, hwtimer_get_time(timer) + 20000000);
    led_state = !led_state;
    port_out(led_port, led_state);
    continue;
  }

  port_disable(led_port);
  port_disable(button_port);
  hwtimer_free(timer);
}
```

2.2.6 Advanced port I/O

Ports are extremely flexible and 'soft-peripheral' implementations can often delegate significant portions of their work to them; this can drastically improve responsiveness. In particular, ports can be connected to configurable *Clock Block* resources which can be used to drive timings of port operations. Documenting the full capabilities and interfaces of ports and clock blocks is not within the scope of this document. Further information about these resources can be found in the lib_xcore port and clock APIs, and in the in the XCore instruction set documentation.

2.2.7 Guiding stack size calculation

As shown earlier in this document, the XMOS tools are able to calculate stack size requirements for many C/C++ functions.

Generally a function's stack requirement is calculable when its own frame is a fixed size and it only calls functions whose stack sizes are also calculable.

2.2.7.1 Function pointer groups One obvious case where it is not possible to determine a function's stack requirement is when it makes a call through a function pointer. It is usually impossible to automatically determine all functions which a given pointer may point to. For this reason, the XMOS tools allow indirect call sites to be annotated with a *function pointer group*. Functions can also be annotated with one or more groups to which they should belong. When the stack size calculator encounters an indirect call through an annotated function pointer, it takes the stack size requirement to be that of the hungriest callee in pointer's group.

The following snippet declares a function pointer `fp` which is a member of the group named `my_functions`:

```
__attribute__(( fptrgroup("my_functions") )) void(*fp)(void);
```

When a call through such a pointer appears in a function, that function's stack size will include a sufficient allocation for a call to any function which has been placed in `my_functions`. Pointer groups are empty until functions are added to the explicitly, so such a call is dangerous until all possible callees have been annotated with the correct group. In the following snippet, `func1` is annotated so that it is a member of the group `my_functions`:

```
_attribute__(( fptrgroup("my_functions") ))
void func1(void)
{
}
```

2.2.7.2 Explicitly setting stack size

In some cases it is necessary to explicitly set the stack size allocated for a function. This may be because a function does not have a fixed requirement (e.g. where a variable length array is used) or might be desirable when heavy use of indirect calls makes annotation infeasible.

In these cases it is possible to specify the number of words which should be allocated for calls to a particular function; the following snippet is an assembly directive which sets the stack requirement for a function named `task_main` to 1024 words:

```
.globl task_main.nstackwords
.set task_main.nstackwords,1024
```

Note that this is simply defining a symbol (whose name is based on the function name) which would be defined by the compiler for a calculable function.

This code can be assembled and linked as a separate object, or can be passed to `xcc` as an additional input file when compiling C/C++ code.

A limited set of binary operators is available for expressing stack size requirements; often this is useful for setting a function's requirements in terms of functions it calls. The available operators are:

- `+` and `*`
- `$M` - evaluates to the greater of its two operands (useful for finding the hungriest of a list of functions)
- `$A` - evaluates to its left operand rounded up to the next multiple of the right operand (useful for rounding up to the stack alignment requirement)

Parentheses (`(` and `)`) are also available.

The following directives set `task_main`'s stack requirement to 1024 words plus the greater of the requirements of `my_function0` and `my_function1`, all rounded to double word alignment:

```
.globl task_main.nstackwords
.set task_main.nstackwords, (1024 + (my_function0.nstackwords $M my_function1.nstackwords )) $A 2
```

When stack sizes are set this way, the compiler will still emit warnings for unannotated calls through pointers; this can be suppressed with the `-Wno-xcore-fptrgroup` option.

Care should be taken when manually setting stack requirements - a function's allocation must be sufficient for its 'local' usage as well as all functions it calls and threads it launches using `PAR_JOBS`.

3 Reference

3.1 Programming in C/XC

3.1.1 Calling between C/C++ and XC

In certain cases, it is possible to pass arguments of one type in XC to function parameters that have different types in C/C++, and vice versa.

To help simplify the task of declaring common functions between C/C++ and XC, the system header file `xccompat.h` contains various type definitions and macro defines. See the header file for documentation.

3.1.1.1 Passing arguments from XC to C/C++ A function defined in C/C++ with a parameter of type `unsigned int` can be declared in XC as taking a parameter of type `port`, `chanend` or `timer`.

A function defined in C/C++ with a parameter of type "pointer to T" can be declared in XC as taking a parameter of type "reference to T" or "nullable reference to T."

A function defined in C/C++ with a parameter of type "pointer to T" can be declared in XC as taking a parameter of type "array of T."

3.1.1.2 Passing arguments from C/C++ to XC A function defined in XC with a parameter of type `port`, `chanend` or `timer` can be declared in C/C++ as taking a parameter of type `unsigned int`.

A function defined in XC with a parameter of type "reference to T" or "nullable reference to T" can be declared in C/C++ as taking a parameter of type "pointer to T."

A function defined in XC with a parameter of type "array of T" can be declared in C/C++ as taking a parameter of type "pointer to type T." In this case, the xCORE linker inserts code to add an implicit array bound parameter equal to the maximum value of the `unsigned int` type.

3.1.2 XC implementation-defined behavior

A conforming XC implementation is required to document its choice of behavior for all parts of the language specification that are designated implementation-defined. In the following section, all choices that depend on an externally determined application binary interface are listed as "determined by ABI," and are documented in the *Application Binary Interface Specification*.

- The value of a multi-character constant (Section 1.5.2).

  The value of a multi-character constant is the same as the value of its first character; all other characters are ignored.

- Whether identical string literals are distinct (Section 1.6).

  Identical string literals are not distinct; they are implemented in a single location in memory.

- The available range of values that may be stored into a `char` and whether the value is signed (Section 3.2).

  The size of `char` is 8 bits. Whether values stored in a `char` variable are signed or not is determined by the ABI.

- The number of pins interfaced to a port and used for communicating with the environment; and the value of a port or clock not declared **extern** and not explicitly initialized (Section 3.2,7.7).

  The number of pins connected to a port for communicating with the environment is defined either by the explicit initializer for its declarator. If no initializer is provided, the compiler produces an error message.

- The notional transfer type of a port, the notional counter type of a port, and the notional counter type of a timer (Section 3.2).

  The notional types are determined by the ABI.

- The value of an integer converted to a signed type such that its value cannot be represented in the new type (Section 5.2).

  When any integer is converted to a signed type and its value cannot be represented in the new type, its value is truncated to the width of the new type and sign extended.

- The handing of overflow, divide check, and other exceptions in expression evaluation (Section 6).

  All conditions (divide by zero, mod zero and overflow of signed divide / mod) result in undefined behaviour.

- The notion of alignment (Section 6.3.4).

  An alignment of 2 *n* guarantees that the least significant *n* bits of the address in memory are 0. The specific alignment of the types is determined by the ABI.

- The value and the type of the result of **sizeof** (Section 6.4.6).

  The value of the result of the **sizeof** operator is determined by the ABI. The type of the result is **unsigned int**.

- Attempted run-time division by zero (Section 6.6).

  Attempted run-time division by zero results in a trap.

- The extent to which suggestions made by using the **inline** function specifier are effective (Section 7.3).

  The **inline** function specifier is taken as a hint to inline the function. The compiler tries to inline the function at all optimization levels above **-O0**.

- The extent to which suggestions made by using the **register** storage class specifier are effective (Section 7.7.4).

  The **register** storage class specifier causes the register allocator to try and place the variable in a register within the function. However, the allocator is not guaranteed to place it in a register.

- The supported predicate functions for input operations (Section 8.3).

- The meaning of inputs and outputs on ports (Section 8.3.2).

  The inputs and outputs on ports map to in and out instructions on port resources, the behaviour of which is defined in the XS1 Ports Specification.

- The extent to which the underlying communication protocols are optimized for transaction communications (Section 8.9).

  The communication protocols are determined by the ABI.

- Whether a transaction is invalidated if a communication occurs such that the number of bytes output is unequal to the number of byte input, and the value communicated (Section 11).

  This is determined by the ABI.

- The behavior of an invalid operation (Section 12).

  Except as described below, all invalid operations are either reported as compilation errors or cause a trap at run-time.

  - The behavior of an invalid master transaction statement is undefined; an invalid slave transaction always traps.
  - The *~~unsafe~~ pragma* can be used to disable specific safety checks, resulting in undefined behavior for invalid operations.

3.1.3 C implementation-defined behavior

A conforming C99 implementation is required to document its choice of behavior for all parts of the language specification that are designated *implementation-defined*. The tools implementation-defined behavior matches that of GCC 4.2.1 except for the choices listed below.

The following section headings refer to sections in the *C99 specification* and all choices that depend on an externally determined application binary interface are listed as "determined by ABI," and are documented in the *Application Binary Interface Specification*.

Only the supported C99 features are documented.

3.1.3.1 Environment

- The name and type of the function called at program startup in a freestanding environment (5.1.2.1).

  A hosted environment is provided.

- An alternative manner in which the `main` function may be defined (5.1.2.2.1).

  There is no alternative manner in which `main` may be defined.

- The values given to the strings pointed to by `argv` argument to `main` (5.1.2.2.1).

  The value of `argc` is equal to zero. `argv[0]` is a null pointer. There are no other array members.

- What constitutes an interactive device (5.1.2.3).

  All streams are refer to interactive devices.

- Signal values other than `SIGFPE`, `SIGILL`, and `SIGSEGV` that correspond to a computational exception (7.14.1.1).

  No other signal values correspond to a computational exception.

- Signal values for which is equivalent of `signal(sig, SIG_IGN);` is executed at program startup (7.14.1.1).

  At program startup the equivalent of `signal(sig, SIG_DFL);` is executed for all signals.

- The set of environment names and the method for altering the environment list used by the `getenv` function (7.20.4.5).

  The set of environment names is empty. There is no method for altering the environment list used by the `getenv` function.

- The manner of execution of the string by the `system` function used by the `getenv` function (7.20.4.6).

  This is determined by the execution environment.

3.1.3.2 Identifiers

- The number of significant initial characters in an identifier (5.2.4.1, 6.4.1).

  All characters in identifiers (with or without external linkage) are significant.

3.1.3.3 Characters

- The value of the members of the execution character set (5.2.1).

  This is determined by the ASCII character set.

- The unique value of the member of the execution character set produced for each of the standard alphabetic escape sequences (5.2.2).

  This is determined by the ASCII character set.

- The value of a `char` object into which has been stored any character other than a member of the basic execution set (6.2.5).

  The value of any character other than a member of the basic execution set is determined by the ASCII character set.

- The mapping of members of the source character set (in character constants and string literals) to members of the execution character set (6.4.4.4, 5.1.1.2).

  The source character set is required to be the ASCII character set. Each character in the source character set is mapped to the same character in the execution character set.

- The value of an integer character constant containing more than one character or containing a character or escape sequence that does not map to a single-byte execution character (6.4.4.4).

  The value of an integer character constant containing more than one character is equal to the value of the last character in the character constant. The value of an integer character constant containing a character or escape sequence that does not map to a single-byte execution character is equal to the value reduced modulo 2 *n* to be within range of the `char` type, where *n* is the number of bits in a `char`.

- The value of a wide character constant containing more than one multibyte character, or containing a multibyte character or escape multibyte character, or containing a multibyte character or escape sequence not represented in the extended execution character set (6.4.4.4).

  Wide character constants must not contain multibyte characters.

- The current locale used to convert a wide character constant consisting of a single multibyte character that maps to a member of the extended execution character set into a corresponding wide character code (6.4.4.4).

  Wide character constants must not contain multibyte characters.

- The value of a string literal containing a multibyte character or escape sequence not represented in the execution character set (6.4.5).

  String literals must not contain multibyte characters. If an escape sequence not represented in the execution character set is used in a string literal, the value of the cor-

responding character in the string is the same as the value that would be given to an integer character constant consisting of that escape sequence.

3.1.3.4 Floating point

- The accuracy of the floating-point operations and of the library functions in `<math.h>` and `<complex.h>` that return floating-point results (5.2.4.2.2).

  This is intentionally left undocumented.

- Additional floating-point exceptions, rounding modes, environments, and classifications, and their macros names (7.6, 7.12).

  No additional floating-point exceptions, rounding modes, environments or classifications are defined.

3.1.3.5 Hints

- The extent to which suggestions made by using the `register` storage-class specifier are effective (6.7.1).

  The `register` specifier is ignored except when used as part of the register variable extension.

3.1.3.6 Preprocessing directives

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

  This is documented in *XCC pragma directives*.

3.1.3.7 Library functions

- Any library facilities available to a freestanding program, other than the minimal set required by clause 4 (5.1.2.1).

  A hosted environment is provided.

- The format of the diagnostic printed by the `assert` macro (7.2.1.1).

  The `assert` macro uses the format "Assertion failed: *expression*, file *filename*, line *line number*, function: *function*." where *expression* is the text of the argument, *filename* is the value of `__FILE__`, *line number* is the value of `__LINE__` and *function* is the name of the current function. If the name of the current function cannot be determined, this part of the message is omitted.

- The representation of the floating-point status flags stored by the `fegetexceptflag` function (7.6.2.2).

  The function `fegetexceptflag` is not supported.

- Whether the `feraiseexcept` function raises the "inexact" floating-point exception in addition to the "overflow" and "underflow" floating-point exception (7.6.2.3).

  The function `feraiseexcept` is not supported.

- Strings other than `"C"` and `""` that may be passed as the second argument to the `setlocale` function (7.11.1.1).

  No other strings may be passed as the second argument to the `setlocale` function.

- The types defined for `float_t` and `double_t` when the value of the `FLT_EVAL_METHOD` macro is less than 0 or greater than 2 (7.12).

  No other values of the `FLT_EVAL_METHOD` macro are supported.

- Domain errors for the mathematics functions, other than those required by this International Standard (7.12.1).

  This is intentionally left undocumented.

- The values returned by the mathematics functions on domain errors (7.12.1).

  This is intentionally left undocumented.

- The values returned by the mathematics functions on underflow range errors, whether `errno` is set to the value of the macro `ERANGE` when the integer expression `math_errhandling & MATH_ERRNO` is nonzero, and whether the "underflow" floating-point exception is raised when the integer expression `math_errhandling & MATH_ERREXCEPT` is nonzero (7.12.1).

  This is intentionally left undocumented.

- Whether a domain error occurs or zero is returned when an `fmod` function has a second argument of zero (7.12.10.1).

  A domain error occurs when an `fmod` function has a second argument of zero.

- The base-2 logarithm of the modulus used by the `remquo` functions in reducing the quotient (7.12.10.3).

  The quotient is reduced modulo 2^7.

- Whether the equivalent of `signal(sig, SIG_DFL);` is executed prior to the call of a signal handler, and, if not, the blocking of signals that is performed (7.14.1.1).

  The equivalent of `signal(sig, SIG_DFL);` is executed prior to the call of a signal handler.

- The null pointer constant to which the macro `NULL` expands (7.17).

  `NULL` is defined as `((void *)0)`.

- Whether the last line of a text stream requires a terminating new-line character (7.19.2).

  This is determined by the execution environment.

- Whether space characters that are written out to a text stream immediately before a newline character appear when read in (7.19.2).

  This is determined by the execution environment.

- The number of null characters that may be appended to data written to a binary stream (7.19.2).

  This is determined by the execution environment.

- Whether the file position indicator of an append-mode stream is initially positioned at the beginning or end of a file (7.19.3).

  This is determined by the execution environment.

- Whether a write on a text stream causes the associated file to be truncated beyond that point (7.19.3).

  This is determined by the execution environment.

- The characteristics of file buffering (7.19.3).

  A buffered output stream saves characters until the buffer is full and then writes the characters as a block. A line buffered output stream saves characters until the line is complete or the buffer is full and then writes the characters as a block. An unbuffered output stream writes characters to the destination file immediately.

- Whether a zero-length file actually exists (7.19.3).

  This is determined by the execution environment.

- The rules for composing valid file names (7.19.3).

  This is determined by the execution environment.

- Whether the same file can be simultaneously opened multiple times (7.19.3).

  This is determined by the execution environment.

- The nature and choice of encodings used for multibyte characters in files (7.19.3).

  The execution character set must not contain multibyte characters.

- The effect of the `remove` function on an open file (7.19.4.1).

  This is determined by the execution environment.

- The effect if a file with the new name exists prior to a call to the `rename` function (7.19.4.1).

  This is determined by the execution environment.

- Whether an open temporary file is removed upon abnormal program termination (7.19.4.3).

  Temporary files are not removed on abnormal program termination.

- Which changes of mode are permitted (if any), and under what circumstances (7.19.5.4).

  The file cannot be given a more permissive access mode (for example, a mode of "w" will fail on a read-only file descriptor), but can change status such as append or binary mode. If modification is not possible, failure occurs.

- The style used to print an infinity or *NaN*, and the meaning of any n-char or n-wchar sequence printed for a *NaN* (7.19.6.1, 7.24.2.1).

  A `double` argument representing infinity is converted in the style `[-]inf`. A `double` argument representing a *NaN* is converted in the style as `nan`.

- The output for `%p` conversion in the `fprintf` or `fwprintf` function (7.19.6.1, 7.24.2.1).

  The value of the pointer is converted to unsigned hexadecimal notation in the style *dddd*; the letters `abcdef` are used for the conversion. The precision specifies the minimum number of digits to appear; if the value being converted can be represented in fewer digits, it is expanded with leading zeros. The default precision is 1. The characters `0x` are prepended to the output.

  The `fwprintf` function is unsupported.

- The interpretation of a `-` character that is neither the first nor the last character, nor the second where a `^` character is the first, in the scanlist for `%[` conversion in the `fscanf` or `fwscanf` function (7.19.6.2, 7.24.2.1).

  The `-` character is considered to define a range if the character following it is numerically greater than the character before it. Otherwise the `-` character itself is added to the scanset.

  The `fwscanf` function is unsupported.

- The set of sequences matched by a `%p` conversion and the interpretation of the corresponding input item in the `fscanf` or `fwscanf` function (7.19.6.2, 7.24.2.2).

`%p` matches the same format as `%x`. The corresponding input item is converted to a pointer.

The `fwscanf` function is unsupported.

- The meaning of any n-char or n-wchar sequence in a string representing *NaN* that is converted by the `strtod`, `strtof`, `strtold`, `wcstod`, `wcstof` or `wcstold` function (7.20.1.3, 7.24.4.1.1).

  The functions `wcstod`, `wcstof` and `wcstold` are not supported. A n-char sequence in a string representing *NaN* is scanned in hexadecimal form. Any characters which are not hexadecimal digits are ignored.

- Whether or not the `strtod`, `strtof`, `strtold`, `wcstod`, `wcstof` or `wcstold` function sets `errno` to `ERANGE` when underflow occurs (7.20.1.3, 7.24.4.1.1).

  The functions `wcstod`, `wcstof` and `wcstold` are not supported. The functions `strtod`, `strtof` and `strtold` do not set `errno` to `ERANGE` when and return 0 when underflow occurs.

- Whether the `calloc`, `malloc`, and `realloc` functions return a null pointer or a pointer to an allocated object when the size requested is zero (7.20.3).

  The functions `calloc`, `malloc` and `realloc` functions all return a pointer to an allocated object when the size requested is zero.

- Whether open streams with unwritten buffered data are flushed, open streams are closed, or temporary files are removed when the `abort` or `_Exit` function is called (7.20.4.1, 7.20.4.3, 7.20.4.4).

  When the `abort` function or `_Exit` function is called, temporary files are not removed, buffered files are not flushed and open streams are left open.

- The termination status returned to the host environment by the `abort`, `exit` or `_Exit` function (7.20.3).

  The function `abort` causes a software exception to be raised. The termination status returned to the host environment by the functions `exit` and `_Exit` is determined by the execution environment.

- The value returned by the `system` function when its argument is not a null pointer (7.20.4.6).

  This is determined by the execution environment.

- The range and precision of times representable in `clock_t` and `time_t` (7.23.1).

  The precision of times representable in `time_t` is defined by the execution environment. `time_t` designates an `unsigned long`. The actual range of times representable by `time_t` is defined by the execution environment.

  The macro `CLOCKS_PER_SEC` is defined as `1000`. `clock_t` designates an unsigned long.

- The era for the `clock` function (7.23.2.1).

  The `clock` function always returns the value `(clock_t)(-1)` to indicate that the processor time used is not available.

- The replacement string for the `%Z` specifier to the `strftime` and `wcsftime` functions in the `"C"` locale (7.23.3.5, 7.24.5.1).

  The `%Z` specifier is replaced with the string “GMT”.

3.1.3.8 Locale-Specific Behavior

- Additional members of the source and execution character sets beyond the basic character set (5.2.1).

  Both the source and execution character sets include all members of the ASCII character set.

- The presence, meaning, and representation of additional multibyte characters in the execution character set beyond the basic character set (5.2.1.2).

  The execution character set does not contain multibyte characters.

- The shift states used for the encoding of multibyte characters (5.2.1.2).

  The source and execution character sets does not contain multibyte characters.

- The direction of writing of successive printing characters (5.2.2).

  Characters are printed from left to right.

- The decimal-point character (7.1.1).

  The decimal-point character is '`.`'.

- The set of printing characters (7.4, 7.25.2).

  This is determined by the ASCII character set.

- The set of control characters (7.4, 7.25.2).

  This is determined by the ASCII character set.

- The set of characters tested for by the `isalpha`, `isblank`, `islower`, `ispunct`, `isspace`, `isupper`, `iswalpha`, `iswblank`, `iswlower`, `iswpunct`, `iswspace`, or `iswupper` functions (7.4.1.2, 7.4.1.3, 7.4.1.7, 7.4.1.9, 7.4.1.10, 7.4.1.11, 7.25.2.1.2, 7.25.2.1.3, 7.25.2.1.7, 7.25.2.1.9, 7.25.2.1.10, 7.25.2.1.11).

  The functions `isblank`, `iswalpha`, `iswblank`, `iswlower`, `iswpunct`, `iswspace` and `iswupper` are not supported.

  `islower` tests for the characters '`a`' to '`z`'. `isupper` tests for the characters '`A`' to '`Z`'. `isspace` tests for the characters ' ', '`\f`', '`\n`', '`\r`', '`\t`' and '`\v`'. `isalpha` tests for upper and lower case characters. `ispunct` tests for all printable characters except space and alphanumeric characters.

- The native environment (7.11.1.1).

  The native environment is the same as the minimal environment for C translation.

- Additional subject sequences accepted by the numerical conversion functions (7.20.1, 7.24.4.1).

  No additional subject sequences are accepted by the numerical conversion functions.

- The collation sequence of the execution character set (7.21.4.3, 7.24.4.4.2).

  The comparison carried out by the function `strcoll` is identical to the comparison carried out by the function `strcmp`.

- The contents of the error message strings set up by the `strerror` function (7.21.4.3, 7.24.4.4.2).

  The contents of the error message strings are given in *Error message strings*.

Table 1: Error message strings

Value	String
`EPERM`	Not owner
`ENOENT`	No such file or directory
`EINTR`	Interrupted system call
`EIO`	I/O error
`ENXIO`	No such device or address
`EBADF`	Bad file number
`EAGAIN`	No more processes
`ENOMEM`	Not enough space
`EACCES`	Permission denied
`EFAULT`	Bad address
`EBUSY`	Device or resource busy
`EEXIST`	File exists
`EXDEV`	Cross-device link
`ENODEV`	No such device
`ENOTDIR`	Not a directory
`EISDIR`	Is a directory
`EINVAL`	Invalid argument
`ENFILE`	Too many open files in system
`EMFILE`	Too many open files
`ETXTBSY`	Text file busy
`EFBIG`	File too large
`ENOSPC`	No space left on device
`ESPIPE`	Illegal seek
`EROFS`	Read-only file system
`EMLINK`	Too many links
`EPIPE`	Broken pipe
`EDOM`	Math argument
`ERANGE`	Result too large
`ENAMETOOLONG`	File or path name too long
`ENOSYS`	Function not implemented
`ENOTEMPTY`	Directory not empty
`ELOOP`	Too many symbolic links

- Character classifications that are supported by the `iswctype` function (7.25.1).

  The character classifications supported by `iswctype` are given in *Wide character mappings*.

Table 2: Wide character mappings

Value	Description
`WCT_TOLOWER`	Convert to lower case
`WCT_TOUPPER`	Convert to upper case

3.1.4 C and C++ language reference

XMOS does not produce documentation for C and C++ standard language features as high quality documentation is readily available.

3.1.4.1 Standards

- ISO/IEC 9899:1989: Programming Languages — C. (C89). International Organization for Standardization.
- ISO/IEC 9899:1999: Programming Languages — C. (C99). International Organization for Standardization.
- ISO/IEC 14882:2011: Programming Languages — C++ (C++ Standard). International Organization for Standardization.

3.1.4.2 Books

- The C Programming Language (second edition), by Brian W. Kernighan and Dennis M. Ritchie, published by Prentice-Hall, Upper Saddle River, NJ, USA, 1988. ISBN-10: 0131103628

3.1.4.3 Online

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

3.1.5 XCC pragma directives

The tools supports the following pragmas.

`#pragma unsafe arrays`

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

`#pragma loop unroll (n)`

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

`#pragma stackfunction n`

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

`#pragma stackcalls n`

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

`#pragma ordered`

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

`#pragma select handler`

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

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

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

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

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

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

`#pragma fallthrough`

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

3.1.6 XC to C cheat sheet

This cheat sheet is intended to act as a quick reference for experienced XC programmers wanting to migrate to using C as per *Programming an XCORE tile with C and lib_xcore*.

3.1.6.1 Parallel tasks & channels

XC

```
void task1(chanend c) { ... }
void task2(chanend c, int count) { ... }
void task3(void) { ... }

int main(void)
{
  int count = 5;
  chan c;
  par {
    task1(c);
    task2(c, count);
    task3();
  }

}
```

C

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

DECLARE_JOB(task1, (chanend_t));
void task1(chanend_t c) { ... }

DECLARE_JOB(task2, (chanend_t, int));
void task2(chanend_t c, int count) { ... }

DECLARE_JOB(task3, (void));
void task3(void) { ... }

int main(void)
{
  int count = 5;
  channel_t c = chan_alloc();
  PAR_JOBS(
    PJOB(task1, (c.end_a)),
    PJOB(task2, (c.end_b, count)),
    PJOB(task3, ())
  );
  chan_free(c);
}
```

```
int a = 5;
c <: a;
c :> a;
```

```
int a = 5;
chan_out_word(c, a);
a = chan_in_word(c);
```

```
unsigned char a = 5;
c <: a;
c :> a;
```

```
unsigned char a = 5;
chan_out_byte(c, a);
a = chan_in_byte(c);
```

NA

```
uint32_t words[5] = {1,2,3,4,5};
chan_out_buf_word(c, words, 5);
chan_in_buf_word(c, words, 5);
```

NA

```
unsigned char bytes[4] = {1,2,3,4};
chan_out_buf_byte(c, bytes, 4);
chan_in_buf_byte(c, bytes, 4);
```

```
streaming chan c;
```

```
#include <xcore/channel_streaming.h>

streaming_channel_t c = s_chan_alloc();
// use streaming channel
s_chan_free();
```

```
int a = 5;
c <: a;
c :> a;
```

```
int a = 5;
s_chan_out_word(c, a);
a = s_chan_in_word(c);
```

3.1.6.2 Ports

XC

```
#include <platform.h>

port my_port = XS1_PORT_1J;

void port_user(port p) { ... }

int main(void)
{
  port_user(my_port);
}
```

```
int a;
p :> a;
p <: a;
```

C

```
#include <platform.h>
#include <xcore/port.h>

void port_user(port_t p) { ... }

int main(void)
{
  port_t my_port = XS1_PORT_1J;
  port_enable(my_port);
  port_user(my_port);
  port_disable(my_port);
}
```

```
int a;
a = port_in(p);
port_out(p, a);
```

3.1.6.3 Timers

XC

```
int main()
{
  [[hwtimer]]
  timer t;

  unsigned v;
  t :> v;
}
```

C

```
#include <xcore/hwtimer.h>

int main(void)
{
  hwtimer_t t = hwtimer_alloc();

  unsigned v;
  v = hwtimer_get_time(t);

  hwtimer_free(t);
}
```

3.1.6.4 'Selecting' events

3.1.6.4.1 Select Blocks

XC

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

int main(void)
{
  [[hwtimer]]
  timer t;
  unsigned long now;
  t :> now;

  while (1)
  {
    select
    {
    case t when timerafter(now + 10000000) :> now:
      printf("Timer handler at time: %lu\n",
        now);
      break;
    default:
      puts("Nothing happened...");
      break;
    }
  }
}
```

```
[[ordered]]
select{ // case statements }
```

C

```
#include <stdio.h>
#include <xcore/hwtimer.h>
#include <xcore/select.h>

int main(void)
{
  hwtimer_t t = hwtimer_alloc();
  unsigned long now = hwtimer_get_time(t);

  hwtimer_set_trigger_time(t,
    now + 10000000);

  SELECT_RES(
    CASE_THEN(t, timer_handler),
    DEFAULT_THEN(default_handler))
  {
  timer_handler:
    now = hwtimer_get_time(t);
    hwtimer_change_trigger_time(t,
      now + 10000000);
    printf("Timer handler at time: %lu\n",
      now);
    continue;
  default_handler:
    puts("Nothing happened...");
    continue;
  }

  hwtimer_free(t);
}
```

```
SELECT_RES_ORDERED( // case specifiers ){}
```

XC's `[[combine]]` and `[[combinable]]` keywords are not available in C. Combine the functions manually.

3.1.6.4.2 Case specifiers

XC

Guarded case:

```
case guard_flag => t :> now:
```

Guarded default case:

```
guard_flag => default:
```

Inverted guarded case:

```
case !guard_flag => t :> now:
```

Inverted guarded default case:

```
!guard_flag => default:
```

C

```
CASE_GUARD_THEN(t, guard_flag, timer_handler)
```

```
timer_handler:
  now = hwtimer_get_time(t);
```

```
DEFAULT_GUARD_THEN(guard_flag, default_handler)
```

```
CASE_NGUARD_THEN(t, guard_flag, timer_handler)
```

```
timer_handler:
  now = hwtimer_get_time(t);
```

```
DEFAULT_NGUARD_THEN(guard_flag, default_handler)
```


3.1.6.4.3 Event handlers Event handlers are denoted by a label which must be immediately followed by a read from the event-generating resource, even if the value is not required.

Event handlers can terminated with the following:

`break`
Exit implicit select block loop

`continue`
Continue implicit select block loop; re-initialise as appropriate

With care, and depending on the likely presence or guaranteed absence of a nested select block, `continue` in the outer select block may be replaced with:

`SELECT_CONTINUE_RESET`
Continue; force re-initialisation

`SELECT_CONTINUE_NO_RESET`
Continue; force no re-initialisation

3.1.6.5 Locks

XC	C

NA	`#include <xcore/lock.h>`
	
	`int task_func(void)`
	`{`
	`  lock_t l = lock_alloc();`
	
	`  lock_acquire(l);`
	
	`  // Critical section`
	
	`  lock_release(l);`
	
	`  lock_free(l);`
	`}`

Locks are not 'eventable' and therefore cannot be referenced within a select block case specifier.

3.1.6.6 Function pointers Assist stack size calculation by annotating function pointers:

```
__attribute__(( fptrgroup("my_functions") ))
void func1_small(void) { char cs[64]; cs[0] = 0; }

__attribute__(( fptrgroup("my_functions") ))
void func2_big(void) { char cs[256]; cs[0] = 0; }

int main(void)
{
  __attribute__(( fptrgroup("my_functions") ))
  void(*fp)(void);

  fp = func1_small; fp();
  fp = func2_big;   fp();
}
```

Membership of the correct group is not checked at build-time. Enable runtime checking in the above using the fragment below when the function pointer is declared:

```
__attribute__(( fptrgroup("my_functions", 1) ))
void(*fp)(void);
```

Use of the same fragment on the function declaration has no effect.

3.1.6.7 Targeting multiple tiles

Deploying applications onto multiple tiles still requires the use of a minimal `multitile.xc`:

Listing 1: 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;
}
```

Avoid all procedural code within `multitile.xc`. Instead, place it within the C code:

Listing 2: main.c

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

void main_tile0(chanend_t c)
{
  printf("Tile 0 prints first\n");

  chan_out_word(c, 1);
}

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

  printf("Tile 1 prints second\n");
}
```

3.1.7 Memory models

The XCore C/C++ compiler provides memory models to handle:

- different total sizes of statically allocated application data on any one tile
- different maximum ranges of jumps and sizes of loops
- using different memories (internal, external, software-defined).

Memory model is specified per source file (translation unit). It controls the machine code generated from that source file for accessing data and calling functions. Usually all source files of an application will be compiled under the same model.

3.1.7.1 Small (default)

```
xcc test.c ...
xcc test.c -mcmodel=small ...
```

Machine code is generated assuming that both:

1. All static data on a tile fits within a contiguous 256KB area.

2. The maximum range of a branch is 128KB (single-issue) or 256KB (dual-issue).

If either turns out to be untrue, the linker issues an error. The small model generates more efficient machine code than the other models.

3.1.7.2 Large

```
xcc test.c -mcmodel=large ...
```

Data need not fit within contiguous 256KB. Different data may reside in different memories, for example internal RAM, external DDR, and software-defined memory. Functions anywhere in memory may be called. Compared with the small model, the large model generates less efficient machine code to access data and to call functions, so use the default model when possible.

3.1.7.3 Hybrid

```
xcc test.c -mcmodel=hybrid ...
```

If the compiler considers it safe to use small-model data access in a particular case, efficient small-model code is generated for that case. Otherwise large-model access is used. Large-model function calls are generated in code. So the hybrid model allows an application to combine efficient access to a contiguous data area (usually in internal RAM) with access to other memories, such as external memory. However, the cases suitable for small-model data access are limited. For example, C `extern` variables will be accessed with less efficient large-model code. So use the default model when possible.

3.1.7.4 Indirect access to memory The data access described above is direct access to a data object by name. An application will link under the small memory model only if direct accesses to code and data meet the contiguous size conditions. But indirect addressing will always work. For example, in C, a pointer could be obtained to an object anywhere in memory, and used in code compiled under the small model. See example.

3.2 Programming in assembly

3.2.1 Inline assembly

The `asm` statement can be used to embed code written in assembly inside a C or XC function. For example, the add instruction can be written as follows:

```
asm("add %0, %1, %2" : "=r"(result) : "r"(a), "r"(b));
```

Colons separate the assembler template, the output operands and the input operands. Commas separate operands within a group. Each operand is described by an operand constraint string followed by an expression in parentheses. The "`r`" in the operand constraint string indicates that the operand must be located in a register. The "`=`" in the operand constraint string indicates that the operand is written.

Each output operand expression must be an lvalue and must have "`=`" in its constraint.

The location of an operand may be referred to in the assembler template using an escape sequence of the form `%num` where `num` is the operand number. The escape sequence "`%=`" can be used to emit a number that is unique to each expansion of an `asm` statement. This can be useful for making local labels. To produce a literal "`%`" you must write "`%%`".

If code overwrites specific registers this can be described by using a third colon after the input operands, followed by the names of the clobbered registers as a comma-separated list of strings. For example:

```
asm ("get r11, id\n\tmov %0, r11"
     : "=r"(result)
     : /* no inputs */
     : "r11");
```

The compiler ensures none of the input or output operands are placed in clobbered registers.

If an `asm` statement has output operands, the compiler assumes the statement has no side effects apart from writing to the output operands. The compiler may remove the `asm` statement if the values written by the `asm` statement are unused. To mark an `asm` statement as having side effects add the `volatile` keyword after `asm`. For example:

```
asm volatile("in %0, res[%1]" : "=r"(result) : "r"(lock));
```

If the `asm` statement accesses memory, add "`memory`" to the list of clobber registers. For example:

```
asm volatile("stw %0, dp[0]"
     : /* no outputs */
     : "r"(value));
```

This prevents the compiler caching memory values in registers around the `asm` statement.

The *earlyclobber* constraint modifier "**&**" can be used to specify that an output operand is modified before all input operands are consumed. This prevents the compiler from placing the operand in the same register as any of the input operands. For example:

```
asm("or %0, %1, %2\n"
    "or %0, %0, %3\n"
    : "=&r"(result)
    : "r"(a), "r"(b), "r"(c));
```

Jumps from one `asm` statement to another are not supported. `asm` statements must not be used to modify the event enabled status of any resource.

An input operand can be tied to an output operand by specifying the number of the output operand in the input operand's constraint. For example:

```
::
```

asm("zext %0, 8n"
: "=r"(result) : "0"(x));

Operands that are tied together will be allocated to the same register.

3.2.2 Make assembly programs compatible with the XMOS XS1 ABI

The XMOS XS1 Application Binary Interface (ABI) defines the linking interface for objects compiled from C/C++, XC and assembly code. This tutorial explains how to write functions in assembly code that can be linked against objects generated by the XMOS compiler.

3.2.2.1 Symbols As the assembler parses an assembly file, it maintains a current address which it increments every time it allocates storage.

Symbols are used to associate names to addresses. Symbols may be referenced in directives and instructions, and the linker patches the corresponding address onces its value is calculated.

The program below defines a symbol with name `f` that refers to the value of the current address. It also makes the symbol globally visible from other files, which can reference the symbol by its name.

```
# Give the symbol f the value of the current address.
f:
# Mark the symbol f as global.
  .globl f
```

The symbol is defined by writing its name followed by a colon. The `.globl` directive makes the symbol visible from outside of the file.

3.2.2.2 Alignment

The XS1 ABI specifies minimum alignment requirements for code and data. The start of a function must be 2-byte aligned, and data must be word-aligned. An address is aligned by placing the `.align` directive before the definition of a symbol.

The program below defines a symbol `f` that is defined to be the next 2-byte aligned address.

```
# Force 2 byte alignment of the next address.
  .align 2
f:
```

3.2.2.3 Sections

Each object file may contain multiple sections. When combined by the linker, sections with the same name in each object file are placed together at consecutive addresses. This allows different types of code or data to be grouped together in the final executable.

The XS1 ABI requires functions to be placed in the `.text` section, read-only data in the `.cp.rodata` section and writable data in the `.dp.data` section. The default section is the `.text` section, and the current section can be changed using one of the following directives.

Table 3: Sections supported by the XMOS linker

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

3.2.2.3.1 Data

The example program below defines a 4-byte writeable object, initialized with the value 5, and aligned on a 4-byte boundary.

```
  .section .dp.data, "awd", @progbits
  .align 4
x:
  .word 5
```

You can use the following directives to emit different types of data.

Table 4: Directives for emitting different types of data

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

3.2.2.3.2 Arrays

The program below defines a global array that is 42 bytes in size.

```
  .section .dp.data, "awd", @progbits
  .globl a
  .align 4
a:
  .space 42
  .globl a.globound
  .set a.globound, 42
```

The XS1 ABI requires that for each global array `f` there is a corresponding global symbol `f.globound` which is initialized with the number of elements of the first dimension of the array. You can use the `.set` directive to perform the initialization. Note that this value is used for array bounds checking if the variable is used by an XC function.

3.2.2.4 Functions

The XS1 ABI specifies rules for passing parameters and return values between functions. It also defines symbols for specifying the amount of hardware resources required by the function.

3.2.2.4.1 Parameters and return values

Scalar values of up to 32 bits are passed as 32 bit values. The first four parameters are passed in registers *r0*, *r1*, *r2* and *r3*, and any additional parameters are passed on the stack. Similarly, the first four return values are returned in the registers *r0*, *r1*, *r2* and *r3*, and any additional values are returned on the stack.

In the XC function prototype below, the parameters a and b are passed in registers *r0* and *r1*, as are the return values.

```
{int, int} swap(int a, int b);
```

An assembly implementation of this function is shown below.

```
  .globl swap
  .align 2
swap:
  mov r2, r0
  mov r0, r1
  mov r1, r2
  retsp 0
```

3.2.2.4.2 Caller and callee save registers

The XS1 ABI specifices that the registers *r0*, *r1*, *r2*, *r3* and *r11* are *caller-save*, and all other registers are *callee-save*.

Before a function is called, the contents of all caller-save registers whose values are required after the call must be saved. Upon returning from a function, the contents of all callee-save registers must be the same as on entry to the function.

The following example shows the prologue and epilogue for a function that uses the callee-save registers *r4*, *r5* and *r6*. The prologue copies the register values to the stack, and the epilogue restores the values from the stack back to the registers.

```
# Prologue
  entsp 4
  stw r4, sp[1]
  stw r5, sp[2]
  stw r6, sp[3]

# Main body of function goes here
# ...

# Epilogue
  ldw r4, sp[1]
  ldw r5, sp[2]
  ldw r6, sp[3]
  retsp 4
```

3.2.2.4.3 Resource usage The linker attempts to calculate the amount of resources required by each function, including its memory requirements, and the number of threads, channel ends and timers it uses. This allows the linker to check that the resource usage of the final executable does not exceed that available on the target device.

For a function `f`, the resource usage symbols defined by the XS1 ABI are as follows.

Table 5: Resource usage symbols defined by the XS1 ABI

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

You can define resource usage symbols using the `.linkset` directive. If a function is global, you should also make the resource usage symbols global.

The example program below defines resource usage symbols for a function `f` that uses 4 words of stack, 2 threads, 0 timers and 2 channel ends.

```
.globl f
.globl f.nstackwords
.linkset f.nstackwords, 5
.globl f.maxthreads
.linkset f.maxthreads, 2
.globl f.maxtimers
.linkset f.maxtimers, 0
.globl f.maxchanends
.linkset f.maxchanends, 2
```

In more complex cases, you can use the maximum (`$M`) and addition (`+`) operators to build expressions for the resource usage that are evaluated by the linker. If two functions are called in sequence, you should compute the maximum for the two functions, and if called in parallel you should compute the sum for the two functions.

The example program below defines resource usage symbols for a function `f` that extends the stack by 10 words, allocates two timers and calls functions **g** and **h** in sequence before freeing the timer and returning.

```
.globl f
.globl f.nstackwords
.linkset f.nstackwords, 10 + (g.nstackwords $M h.nstackwords)
.globl f.maxthreads
.linkset f.maxthreads, 1 + ((g.maxthreads-1) $M (h.maxthreads-1))
.globl f.maxtimers
.linkset f.maxtimers, 2 + (g.maxtimers $M h.maxtimers)
.globl f.maxchanends
.linkset f.maxchanends, 0 + (g. maxchanends $M h.maxchanends)
```

You can omit the definition of a resource usage symbol if its value is unknown, for example if the function makes an indirect call through a function pointer. If the value of the symbol is required to satisfy a relocation in the program, however, the program will fail to link.

3.2.2.4.4 Side effects The XC language requires that functions used as boolean guards in `select` statements have no side effects. It also specifies that functions called from within a transaction statement do not declare channels. By default, a function `f` is assumed to be side-effecting and to declare channels unless you explicitly set the following symbols to zero.

Table 6: Symbols for denoting side-effects

Symbol	Description
`f.locnoside`	Specifies whether the function is side effecting
`f.locnochandec`	Specifies whether the function allocates a channel end

3.2.2.5 Elimination blocks The linker can eliminate unused code and data. Code and data must be placed in elimination blocks for it to be a candidate for elimination. At final link time, if all of the symbols inside an elimination block are unreferenced, the block is removed from the final image.

The example program below declares a symbol within an elimination block.

```
  .cc_top f.function, f
f:
  .cc_bottom f.function
```

The first argument to the `.cc_top` directive and the `.cc_bottom` directive is the name of the elimination block. The `.cc_top` directive takes an additional argument, which is a symbol on which the elimination of the block is predicated on. If the symbol is referenced, the block is not eliminated.

Each elimination block must be given a name which is unique within the assembly file.

3.2.2.6 Typestrings A typestring is a string used to describe the type of a function or variable. The encoding of type information into a typestring is specified by the XS1 ABI. The following directives are used to associate a typestring with a symbol.

Table 7: Typestring directives

Binding	Directive
Global	`.globl name, "typestring"`
External	`.extern name, "typestring"`
Local	`.locl name, "typestring"`

When a symbol from one object file is matched with a symbol with the same name in another object, the linker checks whether the typestrings are compatible. If the typestrings are compatible linking continues as normal. If the typestrings are function types which differ only in the presence of array bound parameters the linker generates a thunk and replaces uses of the symbol with this thunk to account for the difference in arguments. The linker errors on all other typestring mismatches. This ensures that programs that are compiled from multiple files are as robust as those compiled from a single file.

If you fail to emit a typestring for a symbol, comparisons against this symbol are assumed to be compatible. If you are implementing a function which takes an array of unknown size, you should emit a typestring to allow it to be called from both C and XC. In other cases, typestrings can be omitted, but error checking is not performed.

3.2.2.7 Example The program below prints the words “Hello world” to standard output.

```
const char str[] = "Hello world";

int main() {
  printf(str);
  return 0;
}
```

The assembly implementation below complies with the XS1 ABI.

```
   .extern printf, "f{si}(p(c:uc),va)"
   .section .cp.rodata, "ac", @progbits
   .globl str, "a(12:c:uc)"
   .cc_top str.data, str
   .align 4
str :
   .asciiz "Hello world"
.cc_bottom str.data
   .globl str.globound
   .set str.globound, 12

   .text
   .globl main, "f{si}(0)"
.cc_top main.function, main
   .align 2
main:
  entsp 1
  ldaw r11, cp[str]
  mov r0, r11
  bl printf
  ldc r0, 0
  retsp 0
.cc_bottom main.function
   .globl main.nstackwords
   .linkset main .nstackwords, 1 + printf.nstackwords
   .globl main.maxthreads
   .linkset main.maxthreads, printf.maxthreads
   .globl main.maxtimers
   .linkset main.maxtimers, 0 + printf.maxtimers
   .globl main.maxchanends
   .linkset main.maxchanends, 0 + printf.maxchanends
   .linkset main.locnochandec, 1
   .linkset main.locnoside, 1
```

By defining symbols for resource usage, the linker can check whether the program fits on a target device. By providing typestrings, the linker can check type compatibilty when different object files are linked. The linker can eliminate unused code and data since it is placed in elimination blocks.

3.2.3 Assembly programming manual

The XMOS assembly language supports the formation of objects in the Executable and Linkable Format (ELF) with DWARF 3 debugging information. Extensions to the ELF format are documented in the *XMOS Application Binary Interface*.

3.2.3.1 Lexical conventions There are six classes of tokens: symbol names, directives, constants, operators, instruction mnemonics and other separators. Blanks, tabs, formfeeds and comments are ignored except as they separate tokens.

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

3.2.3.1.2 Symbol names A symbol name begins with a letter or with one of the characters '`.`' or '`_`', followed by an optional sequence of letters, digits, periods, underscores and dollar signs. Upper and lower case letters are different.

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

3.2.3.1.4 Constants A constant is either an integer number, a character constant or a string literal.

- A binary integer is `0b` or `0B` followed by zero or more of the digits `01`.
- An octal integer is `0` followed by zero or more of the digits `01234567`.
- A decimal integer is a non-zero digit followed by zero or more of the digits `0123456789`.
- A hexadecimal integer is `0x` or `0X` followed by one or more of the digits and letters `0123456789abcdefABCDEF`.
- A character constant is a sequence of characters surrounded by single quotes.
- A string literal is a sequence of characters surrounded by double quotes.

The C escape sequences may be used to specify certain characters.

3.2.3.2 Sections and relocations Named ELF sections are specified using directives (see *section, pushsection, popsection*). In addition, there is a unique unnamed "absolute" section and a unique unnamed "undefined" section. The notation {*secname* X} refers to an "offset X into section *secname*."

The values of symbols in the absolute section are unaffected by relocations. For example, address {absolute 0} is "relocated" to run-time address 0. The values of symbols in the undefined section are not set.

The assembler keeps track of the current section. Initially the current section is set to the text section. Directives can be used to change the current section. Assembly instructions and directives which allocate storage are emitted in the current section. For each section, the assembler maintains a location counter which holds the current offset in the section. The *active location counter* refers to the location counter for the current section.

3.2.3.3 Symbols Each symbol has exactly one name; each name in an assembly program refers to exactly one symbol. A local symbol is any symbol beginning with the characters "`.L`". A local symbol may be discarded by the linker when no longer required for linking.

3.2.3.3.1 Attributes

Each symbol has a *value*, an associated section and a *binding*. A symbol is assigned a value using the **set** or **linkset** directives (see *set, linkset*), or through its use in a label (see *Labels*). The default binding of symbols in the undefined section is *global*; for all other symbols the default binding is *local*.

3.2.3.4 Labels

A label is a symbol name immediately followed by a colon (**:**). The symbol's value is set to the current value of the active location counter. The symbol's section is set to the current section. A symbol name must not appear in more than one label.

3.2.3.5 Expressions

An expression specifies an address or value. The result of an expression must be an absolute number or an offset into a particular section. An expression is a *constant expression* if all of its symbols are defined and it evaluates to a constant. An expression is a simple expression if it is one of a constant expression, a symbol, or a symbol +/- a constant. An expression may be encoded in the ELF-extended expression section and its value evaluated by the linker (see *set, linkset*); the encoding scheme is determined by the ABI. The syntax of an expression is:

```
expression ::= unary-exp
             | expression infix-op unary-exp
             | unary-exp ``?`` unary-exp ``$:`` unary-exp
             | function-exp

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

argument ::= symbol
           | constant
           | ``(`` expression ``)``

function-exp ::= ``$overlay_region_ptr`` ``(`` symbol ``)``
               | ``$overlay_index`` ``(`` symbol ``)``
               | ``$overlay_physical_addr`` ``(`` symbol ``)``
               | ``$overlay_virtual_addr`` ``(`` symbol ``)``
               | ``$overlay_num_bytes`` ``(`` symbol ``)``

infix-op ::= *one of*
             ``+`` ``-`` ``<`` ``>`` ``<=`` ``>=`` ``||`` ``<<`` ``>>`` ``*`` ``$M`` ``$A`` ``&`` ``/``

prefix-op ::= *one of*
              ``-`` ``~`` ``$D``
```

Symbols are evaluated to {*section x*} where *section* is one of a named section, the absolute section or the undefined section, and *x* is a signed 2's complement 32-bit integer.

Infix operators have the same precedence and behavior as C, and operators with equal precedence are performed left to right. In addition, the **$M** operator has lowest precedence, and the **$A** operator has the highest precedence.

For the **+** and **-** operators, the set of valid operations and results is given in *Valid operations for + and - operators*. For the **$D** operator, the argument must be a symbol; the result is 1 if the symbol is defined and 0 otherwise.

Table 8: Valid operations for **+** and **-** operators

Op	Left Operand	Right Operand	Result
+	{*section* x}	{absolute y}	{*section* x+y}
+	{absolute x}	{*section* y}	{*section* x+y}
+	{absolute x}	{absolute y}	{absolute x+y}
-	{*section* x}	{*section* y}	{absolute x-y}
-	{*section* x}	{absolute y}	{*section* x-y}
-	{absolute x}	{absolute y}	{absolute x-y}

The **?** operator is used to select between symbols: if the first operand is non-zero then the result is the second operand, otherwise the result is the third operand.

The operators **$overlay_region_ptr**, **$overlay_index**, **$overlay_physical_addr**, **$overlay_virtual_addr** and **$overlay_num_bytes** can be used to query properties of the overlay containing the overlay roots with the specified overlay key symbol (see *Overlay directives*). The set of results of these operators is given in *Operators for querying properties of overlays.*.

Table 9: Operators for querying properties of overlays.

Operator	Result
$overlay_region_ptr	Virtual address of the overlay region containing the overlay.
$overlay_index	Index of the overlay in the overlay region.
$overlay_physical_addr	Physical address of the overlay.
$overlay_virtual_addr	Virtual (runtime) address of the overlay.
$overlay_num_bytes	Size of the overlay in bytes.

For all other operators, both arguments must be absolute and the result is absolute. The **$M** operator returns the maximum of the two operands and the **$A** operator returns the value of the first operand aligned to the second.

Wherever an absolute expression is required, if omitted then {absolute 0} is assumed.

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

3.2.3.6.1 add_to_set The **add_to_set** directive adds an expression to a set of expressions associated with a key symbol. Its syntax is:

```
add-to-set-directive ::= ``.add_to_set`` symbol ``,`` expression
                       | ``.add_to_set`` symbol ``,`` expression ``,`` symbol
```

An optional predicate symbol may be specified as the 3rd argument. If this argument is specified the expression will only be added to the set if the predicate symbol is not eliminated from the linked object.

3.2.3.6.2 max_reduce, sum_reduce The **max_reduce** directive computes the maximum of the values of the expressions in a set. The **sum_reduce** directive computes the sum of the values of the expressions in a set.

```
max-reduce-directive ::= ``.max_reduce`` symbol ``,`` symbol ``,`` expression
sum-reduce-directive ::= ``.sum_reduce`` symbol ``,`` symbol ``,`` expression
```

The first symbol is defined using the value computed by the directive. The second symbol is the key symbol identifying the set of expressions (see *add_to_set*). The expression specifies the initial value for the reduction operation.

3.2.3.6.3 align The **align** directive pads the active location counter section to the specified storage boundary. Its syntax is:

```
align-directive ::= ``.align`` expression
```

The expression must be a constant expression; its value must be a power of 2. This value specifies the alignment required in bytes.

3.2.3.6.4 ascii, asciiz

The **ascii** directive assembles each string into consecutive addresses. The **asciiz** directive is the same, except that each string is followed by a null byte.

```
ascii-directive ::= ``.ascii`` string-list
                  | ``.asciiz`` string-list

string-list ::= string-list ``,`` string
              | string
```

3.2.3.6.5 byte, short, int, long, word

These directives emit, for each expression, a number that at run-time is the value of that expression. The byte order is determined by the endianness of the target architecture. The size of numbers emitted with the word directive is determined by the size of the natural word on the target architecture. The size of the numbers emitted using the other directives are determined by the sizes of corresponding types in the ABI.

```
value-directive ::= value-size exp-list

value-size ::= ``.byte``
             | ``.short``
             | ``.int``
             | ``.long``
             | ``.word``

exp-list ::= exp-list ``,`` expression
           | expression
```

3.2.3.6.6 file

The **file** directive has two forms.

```
file-directive ::= ``.file`` string
                 | ``.file`` constant string
```

When used with one argument, the **file** directive creates an ELF symbol table entry with type STT_FILE and the specified string value. This entry is guaranteed to be the first entry in the symbol table.

When used with two arguments the **file** directive adds an entry to the DWARF 3 **.debug_line** file names table. The first argument is a unique positive integer to use as the index of the entry in the table. The second argument is the name of the file.

3.2.3.6.7 loc

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

```
loc-directive ::= ``.loc`` constant constant <constant>?
                | ``.loc`` constant constant constant <loc-option>*

loc-option ::= ``basic_block``
             | ``prologue_end``
             | ``epilogue_begin``
             | ``is_stmt`` constant
             | ``isa`` constant
```

The address register is set to active location counter. The first two arguments set the file and line registers respectively. The optional third argument sets the column register. Additional arguments set further registers in the **.debug_line** state machine.

basic_block
: Sets *basic_block* to true.

prologue_end
: Sets *prologue_end* to true.

epilogue_begin
: Sets *epilogue_begin* to true.

is_stmt
Sets *is_stmt* to the specified value, which must be 0 or 1.

isa
Sets *isa* to the specified value.

3.2.3.6.8 weak

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

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

3.2.3.6.9 globl, global, extern, locl, local

The `globl` directive makes the specified symbols visible to other objects during linking. The `extern` directive specifies that the symbol is defined in another object. The `locl` directive specifies a symbol has local binding.

```
visibility ::= ``.globl``
             | ``.extern``
             | ``.locl``
             | ``.global``
             | ``.extern``
             | ``.local``

vis-directive ::= visibility symbol
                | visibility symbol ``,`` string
```

If the optional string is provided, an `SHT_TYPEINFO` entry is created in the ELF-extended type section which contains the symbol and an index into the string table whose entry contains the specified string. (If the string does not already exist in the string table, it is inserted.) The meaning of this string is determined by the ABI.

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

3.2.3.6.10 globalresource

```
globalresource-directive ::= ``.globalresource`` expression ``,`` string
                           | ``.globalresource`` expression ``,`` string ``,`` string
```

The `globalresource` directive causes the assembler to add information to the binary to indicate that there was a global port or clock declaration. The first argument is the resource ID of the port. The second argument is the name of the variable. The optional third argument is the tile the port was declared on. For example:

```
.globalresource 0x10200, p, tile[0]
```

specifies that the port `p` was declared on `tile[0]` and initialized with the resource ID `0x10200`.

3.2.3.6.11 typestring

The `typestring` adds an `SHT_TYPEINFO` entry in the ELF-extended type section which contains the symbol and an index into the string table whose entry contains the specified string. (If the string does not already exist in the string table, it is inserted.) The meaning of this string is determined by the ABI.

```
typestring-directive ::= ``.typestring`` symbol ``,`` string
```

3.2.3.6.12 ident, core, corerev

Each of these directives creates an ELF note section named "`.xmos_note`."

```
info-directive ::= ``.ident`` string
                 | ``.core`` string
                 | ``.corerev`` string
```


The contents of this section is a (name, type, value) triplet: the name is **xmos**; the type is either **IDENT**, **CORE** or **COREREV**; and the value is the specified string.

3.2.3.6.13 section, pushsection, popsection

The section directives change the current ELF section (see *Sections and relocations*).

```
section-directive ::= sec-or-push name
                    | sec-or-push name ``,`` flags <sec-type>?
                    | ``.popsection``

sec-or-push ::= ``.section``
              | ``.pushsection``

flags ::= string

sec-type ::= type
           | type ``,`` flag-args

type ::= ``@progbits``
       | ``@nobits``

flag-args ::= string
```

The code following a **section** or **pushsection** directive is assembled and appended to the named section. The optional flags may contain any combination of the following characters.

a	section is allocatable
c	section is placed in the global constant pool
d	section is placed in the global data region
w	section is writable
x	section is executable
M	section is mergeable
S	section contains zero terminated strings

The optional type argument **progbits** specifies that the section contains data; **nobits** specifies that it does not.

If the **M** symbol is specified as a flag, a type argument must be specified and an integer must be provided as a flag-specific argument. The flag-specific argument represents the entity size of data entries in the section. For example:

```
.section .cp.const4, "M", @progbits, 4
```

Sections with the **M** flag but not S flag must contain fixed-size constants, each *flag-args* bytes long. Sections with both the **M** and **S** flags must contain zero-terminated strings, each character *flag-args* bytes long. The linker may remove duplicates within sections with the same name, entity size and flags.

Each section with the same name must have the same type and flags. The **section** directive replaces the current section with the named section. The **pushsection** directive pushes the current section onto the top of a *section stack* and then replaces the current section with the named section. The **popsection** directive replaces the current section with the section on top of the section stack and then pops this section from the stack.

3.2.3.6.14 text

The **text** directive changes the current ELF section to the **.text** section. The section type and attributes are determined by the ABI.

```
text-directive ::= ``.text``
```

3.2.3.6.15 set, linkset

A symbol is assigned a value using the `set` or `linkset` directive.

```
set-directive ::= set-type symbol ``,`` expression

set-type ::= ``.set``
           | ``.linkset``
```

The `set` directive defines the named symbol with the value of the expression. The expression must be either a constant or a symbol: if the expression is a constant, the symbol is defined in the absolute section; if the expression is a symbol, the defined symbol inherits its section information and other attributes from this symbol.

The `linkset` directive is the same, except that the expression is not evaluated; instead one or more SHT_EXPR entries are created in the ELF-extended expression section which together form a tree representation of the expression.

Any symbol used in the assembly code may be a target of an SHT_EXPR entry, in which case its value is computed by the linker by evaluating the expression once values for all other symbols in the expression are known. This may happen at any incremental link stage; once the value is known, it is assigned to the symbol as with set and the expression entry is eliminated from the linked object.

3.2.3.6.16 cc_top, cc_bottom

The `cc_top` and `cc_bottom` directives are used to mark the beginning and end of elimination blocks.

```
cc-top-directive ::= ``.cc_top`` name ``,`` predicate
                   | ``.cc_top`` name

cc-directive ::= cc-top-directive
               | ``.cc_bottom`` name

name ::= symbol

predicate ::= symbol
```

`cc_top` and `cc_bottom` directives with the same name refer to the same elimination block. An elimination block must have precisely one `cc_top` directive and one `cc_bottom` directive. The top and bottom of an elimination block must be in the same section. The elimination block consists of the data and labels in this section between the `cc_top` and `cc_bottom` directives. Elimination blocks must be disjoint; it is illegal for elimination blocks to overlap.

An elimination block is retained in final executable if one of the following is true:

- A label inside the elimination block is referenced from a location outside an elimination block.
- A label inside the elimination block is referenced from an elimination block which is not eliminated
- The predicate symbol is defined outside an elimination block or is contained in an elimination block which is not eliminated.

If none of these conditions are true the elimination block is removed from the final executable.

3.2.3.6.17 scheduling

The `scheduling` directive enables or disables instruction scheduling. When scheduling is enabled, the assembler may reorder instructions to

minimize the number of FNOPs. The default scheduling mode is determined by the command-line option `xcc -fschedule`.

```
scheduling-directive ::= ``.scheduling`` scheduling-mode

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

3.2.3.6.18 issue_mode

The `issue_mode` directive changes the current issue mode assumed by the assembler. See *Instructions* for details of how the issue mode affects how instructions are assembled.

```
issue-mode-directive ::= ``.issue_mode`` issue-mode

issue-mode ::= ``single``
             | ``dual``
```

3.2.3.6.19 syntax

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

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

syntax ::= ``default``
         | ``architectural``
```

3.2.3.6.20 assert

```
assert-directive ::= ``.assert`` constant ``,`` symbol ``,`` string
```

The `assert` directive requires an assertion to be tested prior to generating an executable object: the assertion fails if the symbol has a non-zero value. If the constant is 0, a failure should be reported as a warning; if the constant is 1, a failure should be reported as an error. The string is a message for an assembler or linker to emit on failure.

3.2.3.6.21 Overlay directives

The overlay directives control how code and data is partitioned into overlays that are loaded on demand at runtime.

```
overlay-directive ::= ``.overlay_reference`` symbol ``,`` symbol
                    | ``.overlay_root`` symbol ``,`` symbol
                    | ``.overlay_root`` symbol
                    | ``.overlay_subgraph_conflict`` sym-list

sym-list ::= sym-list ``,`` symbol
           | symbol
```

- The `overlay_root` directive specifies that the first symbol should be treated as an overlay root. The optional second symbols specifies a overlay key symbol. If no overlay key symbol is explictly specified the overlay root symbol is used as the key symbol. Specifying the same overlay key symbol for multiple overlay roots forces the overlay roots into the same overlay.
- The `overlay_reference` directive specifies that linker should assume that there is a reference from the first symbol to the second symbol when it partitions the program into overlays.
- The `overlay_subgraph_conflict` directive specifies that linker should not place any code or data reachable from one the symbols into an overlay that is mapped an overlay region that contains another overlay containing code or data reachable from one of the other symbols.

3.2.3.6.22 Language directives

The language directives create entries in the ELF-extended expression section; the encoding is determined by the ABI.

```
xc-directive ::= globdir symbol ``,`` string
               | globdir symbol ``,`` symbol ``,`` range-args ``,`` string
               | ``.globpassesref`` symbol ``,`` symbol ``,`` string
               | ``.call`` symbol ``,`` symbol
               | ``.par`` symbol ``,`` symbol ``,`` string

range-args ::= expression ``,`` expression

globdir ::= ``.globread``
          | ``.globwrite``
          | ``.parwrite``
          | ``.globpassesref``
```

For each directive, the string is an error message for the assembler or linker to display on encountering an error attributed to the directive.

call
: Both symbols must have function type. This directive sets the property that the first function may make a call to the second function.

par
: Both symbols must have function type. This directive sets the property that the first function is invoked in parallel with the second function.

globread
: The first symbol must have function type and the second directive must have object type. This directive sets the property that the function may read the object. When a range is specified, the first expression is the offset from the start of the variable in bytes of the address which is read and the second expression is the size of the read in bytes.

globwrite
: The first symbol must have function type and the second directive must have object type. This directive sets the property that the function may write the object. When a range is specified, the first expression is the offset from the start of the variable in bytes of the address which is written and the second expression is the size of the write in bytes.

parwrite
: The first symbol must have function type and the second directive must have object type. This directive set the property that the function is called in an expression which writes to the object where the order of evalulation of the write and the function call is undefined. When a range is specified, the first expression is the offset from the start of the variable in bytes of the address which is written and the second expression is the size of the write in bytes.

globpassesref
: The first symbol must have function type and the second directive must have object type. This directive sets the property that the object may be passed by reference to the function.

3.2.3.6.23 uleb128, sleb128

The following directives emit, for each expression in the comma-separated list of expressions, a value that encodes either an unsigned or signed DWARF little-endian base 128 number.

```
leb-directive ::= ``.uleb128`` exp-list
                | ``.sleb128`` exp-list
```

3.2.3.6.24 space, skip

The `space` directive emits a sequence of bytes, specified by the first expression, each with the fill value specified by the second expression. Both expressions must be constant expressions.

```
space-or-skip ::= ``.space``
                | ``.skip``

space-directive ::= space-or-skip expression
                  | space-or-skip expression ``,`` expression
```

The `skip` directive is a synonym for the space directive. It is provided for compatibility with other assemblers.

3.2.3.6.25 type

The `type` directive specifies the type of a symbol to be either a function symbol or an object symbol.

```
type-directive ::= ``.type`` symbol ``,`` symbol-type

symbol-type ::= ``@function``
              | ``@object``
```

3.2.3.6.26 size

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

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

3.2.3.6.27 jmptable, jmptable32

The `jmptable` and `jmptable32` directives generate a table of unconditional branch instructions. The target of each branch instruction is the next label in the list. The size of the each branch instruction is 16 bits for the `jmptable` directive and 32 bits for the `jmptable32` directive.

```
jmptable-directive ::= ``.jmptable`` <jmptable-list>?
                     | ``.jmptable32`` <jmptable-list>?

jmptable-list ::= symbol
                | jmptable-list symbol
```

Each symbol must be a label. A maximum of 32 labels maybe specified. If the unconditional branch distance does not fit into a 16-bit branch instruction, a branch is made to a trampoline at the end of the table, which performs the branch to the target label.

3.2.3.7 Instructions

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

```
instruction ::= mnemonic <instruction-args>?

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

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

To target the dual issue execution mode of xCORE-200 and xcore.ai devices, instructions may be put in bundles:

```
separator ::= newline
            | ``;``

instruction-bundle ::= ``{`` <separator>* bundle-contents <separator>* ``}``

bundle-contents ::= instruction <separator>+ instruction
                  | instruction
```

The current issue mode, as specifed by the issue_mode directive (see *issue_mode*), affects how the assembler assembles instructions. Initially the current issue mode is single and instruction bundles cannot be used. If the current issue mode is changed to dual then:

- Instruction bundles can be specified.
- 16-bit instructions not in an instruction bundle are implicitly placed in an instruction bundle alongside a NOP instruction.

- The encoding of some operands may change. For example the assembler applies a different scaling factor to the immediate operand of relative branch instructions to match the different scaling factor that the processor uses at runtime when the instruction is executed in dual issue mode.

The order in which instructions are listed in an instruction bundle is not significant. The assembler may reorder the instructions in the bundle to satisfy architectural constraints.

The assembly instructions are summarized below using the default assembly syntax. The XMOS XS1 Architecture documents the architectural syntax of the instructions. The syntax directive is used to switch the syntax mode.

The following notation is used:

`bitp`	one of: 1, 2, 3, 4, 5, 6, 7, 8, 16, 24 and 32
`b`	register used as a base address
`c`	register used as a conditional operand
`d, e`	register used as a destination operand
`i`	register used as a index operand
`r`	register used as a resource identifier
`s`	register used as a source operand
`t`	register used as a thread identifier
`u` ~~s~~	small unsigned constant in the range 0… 11
`u` ~~x~~	unsigned constant in the range 0… (2 ~~x~~ -1)
`v, w, x, y`	registers used for two or more source operands

A register is one of: `r0`, `r1`, `r2`, `r3`, `r4`, `r5`, `r6`, `r7`, `r8`, `r9`, `r10`, `r11`, `sp`, `dp`, `cp` and `lr`. The instruction determines which of these registers are permitted.

Where there is choice of instruction formats, the assembler chooses the format with the smallest size. To force a specific format, specify a mnemonic of the form `INSTRUCTION_format` where the instruction and format names are as described in the architecture manual. For example the `LDWCP_ru6` mnemonic specifies the ru6 format of the `LDWCP` instruction.

3.2.3.7.1 Data access

Mnemonic	Operands	Meaning
`ld16s`	`d, b[i]`	Load signed 16 bits
`ld8u`	`d, b[i]`	Load unsigned 8 bits
`lda16`	`d, b[i]`	Add to 16-bit address
`lda16`	`d, b[-i]`	Subtract from 16-bit address
`ldap`	`r11, u` ~~20~~	Load pc-relative address
`ldap`	`r11, -u` ~~20~~	Load pc-relative address
`ldaw`	`d, b[i]`	Add to a word address
`ldaw`	`d, b[-i]`	Subtract from a word address
`ldaw`	`d, b[u` ~~s~~ `]`	Add to a word address immediate
`ldaw`	`d, b[-u` ~~s~~ `]`	Subtract from a word address immediate

continues on next page

Table 10 – continued from previous page

Mnemonic	Operands	Meaning
ldaw	r11, cp[u16]	Load address of word in constant pool
ldaw	d, dp[u16]	Load address of word in data pool
ldaw	d, sp[u16]	Load address of word on stack
ldd	e, d, b[i]	Load double word
ldd	e, d, b[u~~s~~]	Load double word immediate
ldd	e, d, sp[u~~s~~]	Load double from the stack
ldw	et, sp[4]	Load ET from the stack
ldw	sed, sp[3]	Load SED from the stack
ldw	spc, sp[1]	Load SPC from the stack
ldw	ssr, sp[2]	Load SSR from the stack
ldw	d, b[i]	Load word
ldw	d, b[u~~s~~]	Load word immediate
ldw	d, cp[u16]	Load word from constant pool
ldw	r11, cp[u~~20~~]	Load word from constant pool
ldw	d, dp[u16]	Load word from data pool
ldw	d, sp[u16]	Load word from stack
set	cp, s	Set constant pool
set	dp, s	Set data pointer
set	sp, s	Set the stack pointer
st16	s, b[i]	16-bit store
st8	s, b[i]	8-bit store
std	e, d, b[i]	Store double word
std	e, d, b[u~~s~~]	Store double word immediate
std	y, x, sp[u~~s~~]	Store double word on the stack
stw	sed, sp[3]	Store SED on the stack
stw	et, sp[4]	Store ET on the stack
stw	spc, sp[1]	Store SPC on the stack
stw	ssr, sp[2]	Store SSR on the stack
stw	s, b[i]	Store word
stw	s, b[u~~s~~]	Store word immediate
stw	s, dp[u16]	Store word in data pool
stw	s, sp[u16]	Store word on stack

3.2.3.7.2 Branching, jumping and calling

Mnemonic	Operands	Meaning
bau	s	Branch absolute unconditional

continues on next page

Table 11 – continued from previous page

Mnemonic	Operands	Meaning
bf	c, u 16	Branch relative if false
bf	c, -u 16	Branch relative if false
bl	u ~~20~~	Branch and link relative
bl	-u ~~20~~	Branch and link relative
bla	s	Branch and link absolute via register
bla	cp[u ~~20~~]	Branch and link absolute via CP
blat	u 16	Branch and link absolute via table
bru	s	Branch relative unconditional via register
bt	c, u 16	Branch relative if true
bt	c, -u 16	Branch relative if true
bu	u 16	Branch relative unconditional
bu	-u 16	Branch relative unconditional
dualentsp	u 16	Adjust stack, save link register and enable dual issue
entsp	u 16	Adjust stack and save link register and enable single issue
extdp	u 16	Extend data pointer
extsp	u 16	Extend stack pointer
retsp	u 16	Return

3.2.3.7.3 Data manipulation

Mnemonic	Operands	Meaning
add	d, x, y	Add
add	d, x, u ~~s~~	Add immediate
and	d, x, y	Bitwise and
andnot	d, s	And not
ashr	d, x, y	Arithmetic shift right
ashr	d, x, bitp	Arithmetic shift right immediate
bitrev	d, s	Bit reverse
byterev	d, s	Byte reverse
clz	d, s	Count leading zeros
crc32	d, r, p	Word CRC
crc32_inc	d, e, x, y, bitp	Word CRC with address increment
crc8	r, o, d, p	8-step CRC
crcn	d, x, p, n	Variable step CRC
divs	d, x, y	Signed division
divu	d, x, y	Unsigned division
eq	c, x, y	Equal
eq	c, x, u ~~s~~	Equal immediate
ladd	e, d, x, y, v	Long unsigned add with carry

continues on next page

Table 12 – continued from previous page

Mnemonic	Operands	Meaning
ldc	d, u 16	Load constant
ldivu	d, e, v, x, y	Long unsigned divide
lextract	d, x, y, u, bitp	Bitfield extraction from register pair
linsert	d, e, x, y, bitp	Inserts a bitfield into a pair of registers
lmul	d, e, x, y, v, w	Long multiply
lsats	d, x, y	Saturate signed
lss	c, x, y	Less than signed
lsu	c, x, y	Less than unsigned
lsub	e, d, x, y, v	Long unsigned subtract
maccs	d, e, x, y	Mulitply and accumulate signed
maccu	d, e, x, y	Multiply and accumulate unsigned
mkmsk	d, s	Make mask
mkmsk	d, bitp	Make mask immediate
mul	d, x, y	Multiply
neg	d, s	Two's complement negate
not	d, s	Bitwise not
or	d, x, y	Bitwise or
rems	d, x, y	Signed remainder
remu	d, x, y	Unsigned remainder
sext	d, s	Sign extend
sext	d, bitp	Sign extend immediate
shl	d, x, y	Shift left
shl	d, x, bitp	Shift left immediate
shr	d, x, y	Shift right
shr	d, x, bitp	Shift right immediate
sub	d, x, y	Subtract
sub	d, x, u ~~s~~	Subtract immediate
unzip	d, e, x	Unzips a pair of registers
xor	d, x, y	Bitwise exclusive or
xor4	d, e, x, y, v	Bitwise exclusive or of four words
zext	d, s	Zero extend
zext	s, bitp	Zero extend immediate
zip	d, e, x	Zips together a pair of registers

3.2.3.7.4 Concurrency and thread synchronization

Mnemonic	Operands	Meaning
freet		Free unsynchronized thread
get	r11, id	Get thread ID

continues on next page

Table 13 – continued from previous page

Mnemonic	Operands	Meaning
getst	d, res[r]	Get synchronized thread
mjoin	res[r]	Master synchronize and join
msync	res[r]	Master synchronize
ssync		Slave synchronize
init	t[r]:cp, s	Initialize thread's CP
init	t[r]:dp, s	Initialize thread's DP
init	t[r]:lr, s	Initialize thread's LR
init	t[r]:pc, s	Initialize thread's PC
init	t[r]:sp, s	Initialize thread's SP
set	t[r]:d, s	Set register in thread
start	t[r]	Start thread
tsetmr	d, s	Set register in master thread

3.2.3.7.5 Communication

Mnemonic	Operands	Meaning
chkct	res[r], s	Test for control token
chkct	res[r], u ~~s~~	Test for control token immediate
getn	d, res[r]	Get network
in	d, res[r]	Input data
inct	d, res[r]	Input control token
int	d, res[r]	Input token of data
out	res[r], s	Output data
outct	res[r], s	Output control token
outct	res[r], u ~~s~~	Output control token immediate
outt	res[r], s	Output token of data
setn	res[r], s	Set network
testlcl	d, res[r]	Test local
testct	d, res[r]	Test for control token
testwct	d, res[r]	Test for position of control token

3.2.3.7.6 Resource operations

Mnemonic	Operands	Meaning
clrpt	res[r]	Clear port time
elate	s	Throw exception if too late
endin	d, res[r]	End a current input
freer	res[r]	Free a resource
getd	d, res[r]	Get resource data
getr	d, u ~~s~~	Allocate resource

continues on next page

Table 15 – continued from previous page

Mnemonic	Operands	Meaning
gettime	d	Get the reference time
getts	d, res[r]	Get port timestamp
in	d, res[r]	Input data
inpw	d, res[r], bitp	Input a part word
inshr	d, res[r]	Input and shift right
out	res[r], s	Output data
outpw	res[r], s, bitp	Output a part word
outpw	res[r], s, w	Output a part word immediate
outshr	res[r], s	Output data and shift
peek	d, res[r]	Peek at port data
setc	res[r], s	Set resource control bits
setc	res[r], u_{16}	Set resource control bits immediate
setclk	res[r], s	Set clock for a resource
setd	res[r], s	Set data
setev	res[r], r11	Set environment vector
setpsc	res[r], s	Set the port shift count
setpt	res[r], s	Set the port time
setrdy	res[r], s	Set ready input for a port
settw	res[r], s	Set transfer width for a port
setv	res[r], r11	Set event vector
syncr	res[r]	Synchronize a resource

3.2.3.7.7 Event handling

Mnemonic	Operands	Meaning
clre		Clear all events
clrsr	u_{16}	Clear bits in SR
edu	res[r]	Disable events
eef	d, res[r]	Enable events if false
eet	d, res[r]	Enable events if true
eeu	res[r]	Enable events
getsr	r11, u_{16}	Get bits from SR
setsr	u_{16}	Set bits in SR
waitef	c	Wait for event if false
waitet	c	Wait for event if true
waiteu		Wait for event

3.2.3.7.8 Interrupts, exceptions and kernel calls

Mnemonic	Operands	Meaning
clrsr	u_{16}	Clear bits in SR
ecallf	c	Raise exception if false
ecallt	c	Raise exception if true
get	r11, ed	Get ED into r11
get	r11, et	Get ET into r11
get	r11, kep	Get the kernel entry point
get	r11, ksp	Get the kernel stack pointer
getsr	r11, u_{16}	Get bits from SR
kcall	s	Kernel call
kcall	u_{16}	Kernel call immediate
kentsp	u_{16}	Switch to kernel stack
krestsp	u_{16}	Restore stack pointer from kernel stack
kret		Kernel return
set	kep, r11	Set the kernel entry point
setsr	u_{16}	Set bits in SR

3.2.3.7.9 Debugging

Mnemonic	Operands	Meaning
dcall		Cause a debug interrupt
dentsp		Save and modify stack pointer for debug
dgetreg	s	Debug read of another thread's register
drestsp		Restore non debug stack pointer
dret		Return from debug interrupt
get	d, ps[r]	Get processor state
set	ps[r], s	Set processor state

3.2.3.7.10 Pseudo instructions In the default syntax mode, the assembler supports a small set of pseudo instructions. These instructions do not exist on the processor, but are translated by the assembler into xCORE instructions.

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

3.2.3.8 Assembly program An assembly program consists of a sequence of statements.

```
program ::= <statement>*

statement ::= <label-list>? <dir-or-inst>? separator

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

dir-or-inst ::= directive
              | instruction
```

(continues on next page)

(continued from previous page)

```
              | instruction-bundle

directive ::= align-directive
            | ascii-directive
            | value-directive
            | file-directive
            | loc-directive
            | weak-directive
            | vis-directive
            | text-directive
            | set-directive
            | cc-directive
            | scheduling-directive
            | syntax-directive
            | assert-directive
            | xc-directive
            | space-directive
            | type-directive
            | size-directive
            | jmptable-directive
            | globalresource-directive
```

3.3 Programming for XCORE Devices

3.3.1 xcore data types

The size and alignment of C and XC's data types are not specified by the language. This allows the size of `int` to be set to the natural word size of the target device, ensuring the fastest possible performance for many computations. It also allows the alignment to be set wide enough to enable efficient memory loads and stores. *Size and alignment of data types on xcore devices* represents the size and alignment of the data types specified by the *xCORE Application Binary Interface*, which provides a standard interface for linking objects compiled from both C and XC.

Table 19: Size and alignment of data types on xcore devices

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

In addition:

- The `char` type is by default unsigned.
- The types `char`, `short` and `int` may be specified in a bit-field's declaration.
- A zero-width bit-field forces padding until the next bit-offset aligned with the bit-field's declared type.

- The notional transfer type of a port is `unsigned int` (32 bits).
- The notional counter type of a port is `unsigned short` (16 bits).
- The notional counter type of a timer is `unsigned int` (32 bits).

3.3.2 xcore port-to-pin mapping

On xcore devices, pins are used to interface with external components via ports and to construct links to other devices over which channels are established. The ports are multiplexed, allowing the pins to be configured for use by ports of different widths. xs1_port_to_pin_mapping_table gives the XCORE port-to-pin mapping, which is interpreted as follows:

- The name of each pin is given in the format X<n>D<pq> where n is a valid xCORE Tile number for the device and pq exists in the table. The physical position of the pin depends on the packaging and is given in the device datasheet.
- Each link is identified by a letter A-D. The wires of a link are identified by means of a superscripted digit 0-4.
- Each port is identified by its width (the first number 1, 4, 8, 16 or 32) and a letter that distinguishes multiple ports of the same width (A-P). These names correspond to port identifiers in the header file `<xs1.h>` (for example port 1A corresponds to the identifier `XS1_PORT_1A`). The individual bits of the port are identified by means of a superscripted digit 0-31.
- The table is divided into six rows (or *banks*). The first four banks provide a selection of 1, 4 and 8-bit ports, with the last two banks enabling the single 32-bit port. Different packaging options may export different numbers of banks; the 16-bit and 32-bit ports are not available on small devices.

The ports used by a program are determined by the set of XC port declarations. For example, the declaration:

```
on tile [0] : in port p = XS1_PORT_1A
```

uses the 1-bit port 1A on xcore Tile 0, which is connected to pin X0D00.

Usually the designer should ensure that there is no overlap between the pins of the declared ports, but the precedence has been designed so that, if required, portions of the wider ports can be used when overlapping narrower ports are used. The ports to the left of the table have precedence over ports to the right. If two ports are declared that share the same pin, the narrower port takes priority. For example:

```
on tile[2] : out port p1 = XS1_PORT_32A;
on tile[2] : out port p2 = XS1_PORT_8B;
on tile[2] : out port p3 = XS1_PORT_4C;
```

In this example:

- I/O on port **p1** uses pins X2D02 to X2D09 and X2D49 to X2D70.
- I/O on port **p2** uses pins X2D16 to X2D19; inputting from **p2** results in undefined values in bits 0, 1, 6 and 7.
- I/O on port **p3** uses pins X2D14, X2D15, X2D20 and X2D21; inputting from **p1** results in undefined values in bits 28-31, and when outputting these bits are not driven.

Table 20: Available ports and links for each pin

Pin	Highest precedence link	1-bit ports	4-bit ports	8-bit ports	16-bit ports	Lowest precedence 32-bit port
XnD00		1A				
XnD01	A^4 out	1B				
XnD02	A^3 out		$4A^0$	$8A^0$	$16A^0$	$32A^{20}$
XnD03	A^2 out		$4A^1$	$8A^1$	$16A^1$	$32A^{21}$
XnD04	A^1 out		$4B^0$	$8A^2$	$16A^2$	$32A^{22}$
XnD05	A^0 out		$4B^1$	$8A^3$	$16A^3$	$32A^{23}$
XnD06	A^0 in		$4B^2$	$8A^4$	$16A^4$	$32A^{24}$
XnD07	A^1 in		$4B^3$	$8A^5$	$16A^5$	$32A^{25}$
XnD08	A^2 in		$4A^2$	$8A^6$	$16A^6$	$32A^{26}$
XnD09	A^3 in		$4A^3$	$8A^7$	$16A^7$	$32A^{27}$
XnD10	A^4 in	1C				
XnD11		1D				
XnD12		1E				
XnD13	B^4 out	1F				
XnD14	B^3 out		$4C^0$	$8B^0$	$16A^8$	$32A^{28}$
XnD15	B^2 out		$4C^1$	$8B^1$	$16A^9$	$32A^{29}$
XnD16	B^1 out		$4D^0$	$8B^2$	$16A^{10}$	
XnD17	B^0 out		$4D^1$	$8B^3$	$16A^{11}$	
XnD18	B^0 in		$4D^2$	$8B^4$	$16A^{12}$	
XnD19	B^1 in		$4D^3$	$8B^5$	$16A^{13}$	
XnD20	B^2 in		$4C^2$	$8B^6$	$16A^{14}$	$32A^{30}$
XnD21	B^3 in		$4C^3$	$8B^7$	$16A^{15}$	$32A^{31}$
XnD22	B^4 in	1G				
XnD23		1H				
XnD24		1I				
XnD25		1J				
XnD26			$4E^0$	$8C^0$	$16B^0$	
XnD27			$4E^1$	$8C^1$	$16B^1$	
XnD28			$4F^0$	$8C^2$	$16B^2$	
XnD29			$4F^1$	$8C^3$	$16B^3$	
XnD30			$4F^2$	$8C^4$	$16B^4$	
XnD31			$4F^3$	$8C^5$	$16B^5$	
XnD32			$4E^2$	$8C^6$	$16B^6$	
XnD33			$4E^3$	$8C^7$	$16B^7$	
XnD34		1K				
XnD35		1L				
XnD36		1M		$8D^0$	$16B^8$	
XnD37		1N		$8D^1$	$16B^9$	
XnD38		1O		$8D^2$	$16B^{10}$	
XnD39		1P		$8D^3$	$16B^{11}$	
XnD40				$8D^4$	$16B^{12}$	
XnD41				$8D^5$	$16B^{13}$	
XnD42				$8D^6$	$16B^{14}$	
XnD43				$8D^7$	$16B^{15}$	

continues on next page

Table 20 – continued from previous page

Pin	Highest precedence link	1-bit ports	4-bit ports	8-bit ports	16-bit ports	Lowest precedence 32-bit port
XnD49	C^4 out					$32A^0$
XnD50	C^3 out					$32A^1$
XnD51	C^2 out					$32A^2$
XnD52	C^1 out					$32A^3$
XnD53	C^0 out					$32A^4$
XnD54	C^0 in					$32A^5$
XnD55	C^1 in					$32A^6$
XnD56	C^2 in					$32A^7$
XnD57	C^3 in					$32A^8$
XnD58	C^4 in					$32A^9$
XnD61	D^4 out					$32A^{10}$
XnD62	D^3 out					$32A^{11}$
XnD63	D^2 out					$32A^{12}$
XnD64	D^1 out					$32A^{13}$
XnD65	D^0 out					$32A^{14}$
XnD66	D^0 in					$32A^{15}$
XnD67	D^1 in					$32A^{16}$
XnD68	D^2 in					$32A^{17}$
XnD69	D^3 in					$32A^{18}$
XnD70	D^4 in					$32A^{19}$

3.3.3 XCORE 32-bit application binary interface

Information on the *xcore 32-bit application binary interface (ABI)*, the *XE file format* and the *System call interface* is available in the Tools Development Guide.