xCORE C Library

A library providing a native C implementation of xCORE hardware features. This is effectively a bare-metal programming environment for using the xCORE. It assumes very good understanding of how the xCORE architecture works as it does not provide the usual protection that xC does to prevent incorrect use of the hardware.

Features

- Support for channel and streaming channels. This includes full interoperability with xC channels and the ability to write custom channel protocols
- Support for ports and clock blocks
- Support for timers
- Support for select events so that xC ‘select’ functionality can be implemented
- Support for interrupt events
- Support for hardware locks

Software version and dependencies

This document pertains to version 2.0.0 of this library. It is known to work on version 14.2.1 of the xTIMEcomposer tools suite, it may work on other versions.

This library depends on the following other libraries:

- lib_trycatch (>=1.0.0)
- lib_xassert (>=2.0.1)

1 Usage

All functions can be accessed via the xcore_c.h header:

```
#include "xcore_c.h"
```

You will also have to add lib_xcore_c to the USED_MODULES field of your application Makefile.

1.1 Using timers

The library provides support for xCORE hardware timers. They are allocated using:

```
hwtimer_t tmr;
hwtimer_alloc(&tmr);
```

A timer can then be read to get the current time by doing:

```
uint32_t time;
hwtimer_get_time(tmr, &time);
```

There are two functions provided to delay using a timer. The first waits for a specified time:

```
// The times are in 10ns, so 100000 timer ticks is 1ms
uint32_t now;
hwtimer_wait_until(tmr, time + 100000, &now); // Wait for (time + 1ms)
```

The second delays for a period of time in 100MHz timer ticks:

```
hwtimer_delay(tmr, 100000); // Delay for 1ms
```

When the timer is no longer required it can be released to be used by other cores by calling:

```
hwtimer_free(&tmr);
```

Each logical core is automatically allocated a hardware timer for use by xC code. If a task is not running xC code, or the xC code is not using timers, the core's hardware timer may be released back into the pool by calling:

```
// Start of task.
hwtimer_free_xc_timer();
...
hwtimer_realloc_xc_timer();
// End of task.
```

As the above code illustrates, the hardware timer must be reallocated before the logical core completes execution. **There must be a free hardware timer available when hwtimer_realloc_xc_timer() is called.**

1.2 Using channels

1.2.1 Local channels

Local channel connections on a tile are fully supported by the library. A channel connection is created using:

```
channel_t c;
chan_alloc(&c);
```

Data can then be sent and received using:

```
chan_out_word(c.end_a, 1);
chan_out_byte(c.end_a, 2);
```

with a corresponding block of code on another core to consume the data:

```
uint32_t i;
chan_in_word(c.end_b, &i);
uint8_t j;
chan_in_byte(c.end_b, &j);
```

When the channel is finished with then it is closed and the resources released using:

```
chan_free(&c);
```

1.2.2 Inter-tile channels

The use of inter-tile channels is supported by the library. However, the only way to create inter-tile channels is to use a top-level main() written in xC. Without the top-level main there is no way to automatically communicate the tile ID of multiple tiles within a system.

A basic top-level main would look like:

```
#include <platform.h>
#include "application.h"

int main()
{
  chan c;
  par {
    on tile[0]: {
      application_0(c);
    }
    on tile[1]: {
      application_1(c);
    }
  }
  return 0;
}
```

This uses xC to do all of the thread assignment and connecting of the initial channel. After that, the applications can use more channels on each tile and have enough information to know how to connect to each other.

A new channel-end can be allocated using:

```
void application_0(chanend c)
{
  chanend new_c;
  chanend_alloc(&new_c);
```

And a new connection established by passing this new channel-end over the existing link, receiving the destination link on the other tile and connecting the two. So, both applications can do:

```
chan_out_word(c, new_c);             // Send my new-chanend to other tile.
chanend new_dest;
chan_in_word(c, new_dest);           // Recieve other tile's new-chanend...
chanend_set_dest(new_c, new_dest);   // ... and connect it to my new-chanend.
```

When the channel-end is finished with then it is closed and the resources released using:

```
chanend_free(&new_c);
```

1.2.3 Streaming channels

Streaming channels can be used in a similar manner to standard channels. A streaming channel is created using:

```
streaming_channel_t c;
s_chan_alloc(&c);
```

Data can then be sent and received using:

```
s_chan_out_word(c.end_a, 1);
s_chan_out_byte(c.end_a, 2);
```

with a corresponding block of code on another core to consume the data:

```
uint32_t i;
s_chan_in_word(c.end_b, &i);
uint8_t j;
s_chan_in_byte(c.end_b, &j);
```

When the channel is finished with then it is closed and the resources released using:

```
s_chan_free(&c);
```

1.2.4 Channel transactions

The library has functions to support interacting with xC channel ends. This includes `master`/`slave` transactions. For example, a block of xC could use a `master` transaction to send a block of words syhchronised only at the beginning and end:

```
uint32_t data[10] = {...}
master {
  for (size_t i = 0; i < 10; i++) {
    c <: data[i];
  }
}
```

The C code to receive this data is of the form:

```
// we have a chanend 'c';
transacting_chanend_t tc;
chan_init_transaction_slave(&c, &tc);
uint32_t data[10];
for (size_t i = 0; i < 10; i++) {
  t_chan_in_word(tc, &data[i]);
}
chan_complete_transaction(&c, &tc);
```

There are additional functions to send and receive both bytes and blocks of data.

1.3 Using ports and clock blocks

The use of ports and clock blocks is fully supported in the library. This section of the document gives a brief example of how to use the library. For complete documentation of the functionality supported please see the API section.

1.3.1 Example

This example will show how to use the library to configure a clock block and port. The first thing to do is to configure the clock block. For example, if using clock block 1 to be clocked from a divided version of the reference clock:

```
clock c;
clock_alloc(&c, clock_1);
clock_set_source_clk_ref(c);
clock_set_divide(c, 1); // Configure to 50MHz (100Mhz / 2*1)
```

The port to be used can then be enabled, configured and connected to the clock:

```
port p;
port_alloc(&p, port_1A);
port_set_clock(p, c);
```

Starting the clock will reset the port counters on all connected ports. This is generally best done after all ports have been connected so that their counters will be synchronised:

```
clock_start(c);
```

The port can now be used to output or input data:

```
port_out(p, 1);
port_out(p, 0);
...
```

In order to clean up, both the port and clock block must be freed:

```
clock_free(&c);
port_free(&p);
```

1.3.2 Ready signals

Configuring ports to use ready signals is done using the `port_protocol_*` functions provided in `port_protocol.h`. All the basic functions needed to implement this functionality is provided, but the order of configuring a port as a strobed or handshaken port is critical and therefore best done using these wrapper functions.

For example, to create a data port which is controlled by a strobe then the following code sequence could be used:

```
port p;
port_alloc(&p, port_4A);
port p_ready;
port_alloc(&p_ready, port_1A);
clock clk;
clock_alloc(&clk, clock_1);

port_protocol_in_strobed_slave(p, p_ready, clk);
clock_start(clk);
```

After this, any data received on the port p will only be available when the valid signal (strobe on PORT_1A) is high.

1.4 Using hardware locks

The library provides support for xCORE hardware locks. They are allocated using:

```
lock_t l;
lock_alloc(&l);
```

To enter a mutex region the lock is then acquired:

```
lock_acquire(l);
```

After this function completes it is safe to use shared state that must only be used by one core at a time.

To leave the mutex region the lock is released:

```
lock_release(l);
```

The lock resource is released using:

```
lock_free(&l);
```

1.5 Using select events

The library provides the ability to re-create the equivalent of the xC `select` statement.

1.5.1 Example

As an example, take a function which receives data from two channels and handles whichever one is ready.

The function needs to have an *enum* containing an entry per resource that is part of the `select`:

```
typedef enum {
  EVENT_CHAN_C = ENUM_ID_BASE,
  EVENT_CHAN_D
} event_choice_t;
```

The function then starts by clearing all existing select triggers on resources owned by this core to ensure that they cannot trigger events:

```
void channel_example(chanend c, chanend d)
{
  select_disable_trigger_all();
```

The resources are each configured to trigger events and return a value from the *enum* specified above:

```
chanend_setup_select(c, EVENT_CHAN_C);
chanend_enable_trigger(c);
chanend_setup_select(d, EVENT_CHAN_D);
chanend_enable_trigger(d);
```

And then the rest of the function can simply use the `select_wait()` function to wait for events to be triggered by either resource:

```
while (1) {
  event_choice_t choice = select_wait();
  uint32_t x;
  switch (choice) {
    case EVENT_CHAN_C: {
      // Read value to clear the trigger
      chan_in_word(c, &x);
      ...
      break;
    }
    case EVENT_CHAN_D: {
      // Read value to clear the trigger
      chan_in_word(d, &x);
      ...
      break;
    }
  }
}
```

1.5.2 Select event handling with a default

In xC a `select` can have a `default` case which is executed if no events have triggered. This library provides the user with the ability to do this by using the `select_no_wait()` function. For example, the above example could be changed to add to the `enum` a no-event value:

```
typedef enum {
  EVENT_CHAN_C = ENUM_ID_BASE,
  EVENT_CHAN_D,
  EVENT_NONE
} event_choice_t;
```

And then to test for triggers but perform some background task if there is no data available on either channel:

```
while (1) {
  event_choice_t choice = select_no_wait(EVENT_NONE);
  uint32_t x;
  switch (choice) {
    case EVENT_CHAN_C: {
      // Read value and clear event
      chan_in_word(c, &x);
      ...
      break;
    }
    case EVENT_CHAN_D: {
      // Read value and clear event
      chan_in_word(d, &x);
      ...
      break;
    }
    case EVENT_NONE: {
      // Run background task
      ...
      break;
    }
  }
}
```

The argument that is passed to `select_no_wait()` is the value that will be returned if no events are ready.

1.5.3 Select event callback functions

This library also supports the ability to install select event callback functions. This allows the user to write code where events are not all handled within one `switch` statement. It makes it possible to write libraries which are completely self-contained.

For example, if the user writes a library to perform a real-time task based on a timer event the library initialisation would install a callback:

```
void lib_init(void *data)
{
  hwtimer_t tmr;
  hwtimer_alloc(&tmr);
  uint32_t time;
  hwtimer_get_time(tmr, &time);
  hwtimer_setup_select_callback(tmr, time + period,
                                data, SELECT_CALLBACK(hwtimer_callback_func));
  hwtimer_enable_trigger(tmr);
}
```

This code allocates a hardware timer and then gets the current time before registering callback function. The call to `hwtimer_setup_select_callback()` takes four arguments.

1. The timer to configure
2. The time at which the next event should fire
3. A `void*` which is user data that is passed to the handler
4. The macro generated select_callback_t function called when events are triggered by the timer

Note: There are similar functions for ports (`port_setup_select_callback()`) and channel ends (`chanend_setup_select_callback()`).

The callback function is passed the user data registered with that resource:

```
void hwtimer_callback_func(void *data);
```

However, we also need to generate a wrapping function, so we use the API's marcro to declare both at the same time:

```
DECLARE_SELECT_CALLBACK(hwtimer_callback_func, data);
```

data will usually be the resource's ID so that the callback can access the resource. If additional information is required, data may be a pointer to a struct:

```
typedef struct data_t {hwtimer_t tmr; uint32_t period;} data_t;

DEFINE_SELECT_CALLBACK(hwtimer_callback_func, data) {
  data_t *d = (data_t *)data;
  uint32_t time;
  hwtimer_get_time(d->tmr, &time);
  hwtimer_change_trigger_time(d->tmr, time + d->period);
```

When using select callback functions, the select_disable_trigger_all() function should not be called, otherwise any registered callback functions will be disabled. Instead, users should now clear any triggers it enables:

```
void handle_events(chanend c, chanend d)
{
  // Setup the channels to generate select events
  chanend_setup_select(c, EVENT_CHAN_C);
  chanend_enable_trigger(c);
  chanend_setup_select(d, EVENT_CHAN_D);
  chanend_enable_trigger(d);

  // Handle select events using select_wait() / select_no_wait()
  ...

  // Disable select events local to this function
  chanend_disable_trigger(c);
  chanend_disable_trigger(d);
  // The chanends keep their setup should you wish to re-enable their triggering.
}
```

After the handle_events() function has completed another equivalent function can be called in which the timer callback will continue to be called periodically.

When the timer select callbacks are no longer required then they can be disabled using the hwtimer_disable_trigger() function (or equivalent port/chanend functions):

```
hwtimer_disable_trigger(tmr);
```

1.5.4 Ordered select events

The xCORE hardware has implicit ordering enforced. Ports are highest priority, then timers, then channels. If there are multiple resources of the same type that are ready then the resource with the highest priority (lowest resource ID) will be selected.

If the user wants to enforce a different ordering from that provided by the hardware then they can use select_wait_ordered() (or the no wait equivalent select_no_wait_ordered()).

Events are set up as detailed above and a list is created with all the active resources. For example, if using two channels (c, d) and a timer (tmr) then a null-terminated list can be defined to ensure the channels are handled before the timer if they are ready:

```
resource_t ids[4] = {c, d, tmr, 0};
```

And then the core of the select event handling loop would be changed to pass this list of resource IDs to define the order in which events are enabled:

```
while (1) {
  event_choice_t choice = select_wait_ordered(ids);
  switch (choice) {
  ...
  }
}
```

1.6 Using interrupts

The library provides support for hardware interrupts from xCORE resources.

Interrupts can be raised by resources as an alternative to select events, and will be vectored to the provided callback function.

As interrupts can occur at any point during program execution there are certain requirements which must be adhered to ensure safe operation:

1. Resources must not have interrupts enabled whilst being configured, or the core must have interrupts masked if the resource has already been configured to raise interrupts.
2. The core must have interrupts masked when disabling interrupts for a resource.

1.6.1 Example

As an example, take a function which receives data from two channels and handles whichever one is ready.

We start by declaring the scope in which interrupts may occur - ‘the hosting function’. The hosting function will make space on its stack for a temporary kernel stack which will be used by the interrupts. Our ordinary ‘void test(chanend,chanend)’ is turned into a hosting function by wrapping it in the ‘DECLARE_INTERRUPT_PERMITTED’ function macro:

```
// xc top level file creating our logical cores.
DECLARE_INTERRUPT_PERMITTED(void, test, chanend c1, chanend c2);
int main() {
  chan c, d;
  par {
    INTERRUPT_PERMITTED(test)(c, d); // interrupts hosted on this core.

    // Start two other cores to create the interrupt events.
    {
      delay_ticks(5000);
      c <: 12;
      delay_ticks(5000);
      c <: 34;
    }
    {
      delay_ticks(10000);
      d <: 56;
      delay_ticks(10000);
      d <: 78;
    }
  }
  return 0;
}
```

and likewise the definition (see below for implementation):

```
DEFINE_INTERRUPT_PERMITTED(my_group, void, test, chanend c1, chanend c2)
{
  ...
}
```

The identifier ‘my_group’ tells the hosting function which interrupts it will be hosting, so it can calculate the stack requirements.

One piece of user data will be sent to the callback as a ‘void*’ argument. We will register a pointer to a structure:

```
typedef struct {
  chanend c;    // The resource that caused the interrupt.
  const char *message;
} chan_data_t;
```

We now define/declare the interrupt callback function, wrapping it in function macros and placing it in the same group:

```
volatile size_t received = 0;  // For the host to monitor events.

DEFINE_INTERRUPT_CALLBACK(my_group, my_handler, data)
{
  chan_data_t *cd = (chan_data_t*)data;
  uint32_t x;
  chan_in_word(cd->c, &x);
  debug_printf("%s received %d\n", cd->message, x);
  received++;
}
```

And finally we can set up the interrupt and enable them:

```
DEFINE_INTERRUPT_PERMITTED(my_group, void, test, chanend c1, chanend c2)
{
  // Set up interrupt.
  // We assume either the triggers are disabled or interrupts are masked.
  chan_data_t cd1 = {c1, "channel 1"};
  chanend_setup_interrupt_callback(cd1.c, (void*)&cd1,
                                   INTERRUPT_CALLBACK(my_handler));
  chan_data_t cd2 = {c2, "channel 2"};
  chanend_setup_interrupt_callback(cd2.c, (void*)&cd2,
                                   INTERRUPT_CALLBACK(my_handler));
  // Enable interrupts.
  chanend_enable_trigger(cd1.c);
  chanend_enable_trigger(cd2.c);
  interrupt_unmask_all();
```

And when we have finished, disable them:

```
  while (received < 4);

  // Disable interrupts.
  interrupt_mask_all(); // Mask before disabling.
  chanend_disable_trigger(cd1.c);
  chanend_disable_trigger(cd2.c);
}
```

2 API

2.1 Opaque types used by the library

Type	**resource_t**
Description	generic resource handle This is an opaque base of the types ‘chanend’, ‘port’ and ‘timer’. It is used to form a list to pass into select_wait_ordered() and select_no_wait_ordered(). Users must not access its raw underlying type.

Type	**chanend**
Description	Opaque type for use in C/C++ code. It enables a xC function prototyped as taking a parameter of type chanend to be called from C and vice versa. Users must not access its raw underlying type.

Type	**streaming_chanend_t**
Description	Opaque type for use in C/C++ code. It enables a xC function prototyped as taking a parameter of type streaming_chanend_t to be called from C and vice versa. Users must not access its raw underlying type.

Type	**transacting_chanend_t**
Description	An opaque type for handling transactions. Users must not access its raw underlying type.

Type	**clock**
Description	clock is an opaque type for use in C/C++ code. It enables a xC function prototyped as taking a parameter of type clock to be called from C and vice versa. Users must not access its raw underlying type.

Type	**lock_t**
Description	lock is an opaque type that denotes a hardware lock which provide a mutex function. Users must not access its raw underlying type.

Type	**port**
Description	port is an opaque type for use in C/C++ code. It enables a xC function prototyped as taking a parameter of type port to be called from C and vice versa. Users must not access its raw underlying type.

Type	**hwtimer_t**
Description	hwtimer_t is an opaque type. The hwtimer_t type can be used just like the timer type. It gives a unique hardware timer to use (as opposed to the default timer in xC which is allocated based on a shared hardware timer per logical core). Users must not access its raw underlying type.

Type	**select_callback_t**
Description	wrapped select callback function This is an opaque type returned by the SELECT_CALLBACK() macro. Users must not access its raw underlying type.

Type	**interrupt_callback_t**
Description	wrapped interrupt callback function This is an opaque type returned by the INTERRUPT_CALLBACK() macro. Users must not access its raw underlying type.

2.2 Errors and exception

Type	**xcore_c_error_t**
Description	Errors returned when XCORE_C_NO_EXCEPTION policy is active. All errors (apart from error_none) are caught hardware exceptions. See 'The XMOS XS1/XS2 Architecture' for details regarding the exceptions.

Values	`error_none`
	
	`error_link_error`
	
	`error_illegal_pc`
	
	`error_illegal_instruction`
	
	`error_illegal_resource`
	
	`error_load_store`
	
	`error_illegal_ps`
	
	`error_arithmetic`
	
	`error_ecall`
	
	`error_resource_dep`
	
	`error_kcall`

Macro	**XCORE_C_NO_EXCEPTION**

Continued on next page

Description	The exception policy for the library. If the user respects the resource type parameter of the library, checks for zero (fail) on allocation, passes in valid pointer addresses and does not access the same resource on multiple logical cores the library will not throw an exception. **Exceptions should be viewed as programming errors rather than runtime errors.** The default exception policy for the library is to throw exceptions. Setting the library to no_exceptions will increase the code size and run time of the resultant binary. XCORE_C_NO_EXCEPTION may be set by the user to a boolean constant or expression.

2.3 Chanends

Function	**s_chanend_alloc**
Description	Allocate a single streaming_chanend_t. This function allocates a hardware channel end. If there are no channel ends available the function returns 0. When the channel end is no longer required, s_chanend_free() must be called to deallocate it.

Type	`xcore_c_error_t`
	`s_chanend_alloc(streaming_chanend_t *c)`

Parameters	c streaming_chanend_t or zero if none are available. ET_LOAD_STORE invalid **c* argument.
Returns	error_none (or exception type if policy is XCORE_C_NO_EXCEPTION).

Function	**s_chanend_free**
Description	Deallocate a single streaming_chanend_t. This function frees the hardware streaming_chanend_t. The last transfer on the streaming_chanend_t must have been a CT_END token.

Type	`xcore_c_error_t`
	`s_chanend_free(streaming_chanend_t *c)`

Parameters	c streaming_chanend_t to free. ET_ILLEGAL_RESOURCE not an allocated streaming_chanend_t, an input/output is pending, or it has not received/sent a CT_END token. ET_RESOURCE_DEP another core is actively using the streaming_chanend_t. ET_LOAD_STORE invalid **c* argument.
Returns	error_none (or exception type if policy is XCORE_C_NO_EXCEPTION).

Function	**s_chanend_set_dest**
Description	Set the destination of a streaming_chanend_t.

Continued on next page

Type	`xcore_c_error_t s_chanend_set_dest(streaming_chanend_t c, streaming_chanend_t dst)`
Parameters	`c` streaming_chanend_t to set. `dst` Destination streaming_chanend_t. `ET_ILLEGAL_RESOURCE` not an allocated streaming_chanend_t. `ET_RESOURCE_DEP` another core is actively using the streaming_chanend_t.
Returns	error_none (or exception type if policy is XCORE_C_NO_EXCEPTION).

Function	**s_chanend_convert**
Description	Convert a chanend to a streaming_chanend_t. A chanend is always in a safe state for converting into a streaming_chanend_t.
Type	`streaming_chanend_t s_chanend_convert(chanend c)`
Parameters	`c` chanend to convert.
Returns	the streaming_chanend_t

Function	**chanend_alloc**
Description	Allocate a single chanend. This function allocates a hardware channel end. If there are no channel ends available the function returns 0. When the channel end is no longer required, chanend_free() must be called to deallocate it.
Type	`xcore_c_error_t chanend_alloc(chanend *c)`
Parameters	`c` chanend or zero if none are available. `ET_LOAD_STORE` invalid **c* argument.
Returns	error_none (or exception type if policy is XCORE_C_NO_EXCEPTION).

Function	**chanend_free**
Description	Deallocate a single chanend. This function frees the hardware chanend. The last transfer on the chanend must have been a CT_END token.
Type	`xcore_c_error_t chanend_free(chanend *c)`
Parameters	c chanend to free. ET_ILLEGAL_RESOURCE not an allocated chanend, an input/output is pending, or it has not received/sent a CT_END token. ET_RESOURCE_DEP another core is actively using the chanend. ET_LOAD_STORE invalid **c* argument.
Returns	error_none (or exception type if policy is XCORE_C_NO_EXCEPTION).

Function	**chanend_set_dest**
Description	Set the destination of a chanend.
Type	`xcore_c_error_t chanend_set_dest(chanend c, chanend dst)`
Parameters	c chanend to set. dst Destination chanend. ET_ILLEGAL_RESOURCE not an allocated chanend. ET_RESOURCE_DEP another core is actively using the chanend.
Returns	error_none (or exception type if policy is XCORE_C_NO_EXCEPTION).

Function	**chanend_convert**
Description	Convert a streaming_chanend_t to a chanend. **streaming_chanend_t must have completed any transaction with an end-token handshake before being converting into a chanend.**

Continued on next page

Type	`chanend`
	`chanend_convert(streaming_chanend_t c)`

Parameters	c streaming_chanend_t to convert.
Returns	the chanend

Function	**chanend_setup_select**
Description	Setup select events on a chan-end. Configures a chan-end to trigger select events when data is ready. It is used in combination with select_wait() et al functions, returning the enum_id when the event is triggered. Once the event is setup you need to call chanend_enable_trigger() to enable it.
Type	`xcore_c_error_t chanend_setup_select(chanend c, uint32_t enum_id)`
Parameters	c The chan-end to setup the select event on enum_id The value to be returned by select_wait() et al when the chan-end event is triggered. ET_ILLEGAL_RESOURCE not a valid chan-end. ET_RESOURCE_DEP another core is actively using the chan-end. ET_ECALL when xassert enabled, on XS1 bit 16 not set in enum_id.
Returns	error_none (or exception type if policy is XCORE_C_NO_EXCEPTION).

Function	**chanend_setup_select_callback**
Description	Setup select events on a chan-end where the events are handled by a function. Same as chanend_setup_select() except that a callback function is used rather than the event being passed back to the select_wait() et al functions. Once the event is setup you need to call chanend_enable_trigger() to enable it.

Type	`xcore_c_error_t`
	`chanend_setup_select_callback(chanend c,`
	`void *data,`
	`select_callback_t func)`

Continued on next page

Parameters	c	The chan-end to setup the select event on
	data	The value to be passed to the select_callback_t function
	func	The select_callback_t function to handle the event
	ET_ILLEGAL_RESOURCE	not a valid chan-end.
	ET_RESOURCE_DEP	another core is actively using the chan-end.
	ET_ECALL	when xassert enabled, on XS1 bit 16 not set in enum_id.
Returns	error_none (or exception type if policy is XCORE_C_NO_EXCEPTION).	

Function	**chanend_setup_interrupt_callback**
Description	Setup interrupt events on a chan-end . Once the event is setup you need to call chanend_enable_trigger() to enable it.

Type	`xcore_c_error_t`
	`chanend_setup_interrupt_callback(chanend c,`
	`void *data,`
	`interrupt_callback_t func)`

Parameters	c — The chan-end to setup the events on data — The value to be passed to the interrupt_callback_t function func — The interrupt_callback_t function to handle events ET_ILLEGAL_RESOURCE — not a valid chan-end. ET_RESOURCE_DEP — another core is actively using the chan-end. ET_ECALL — when xassert enabled, on XS1 bit 16 not set in enum_id.
Returns	error_none (or exception type if policy is XCORE_C_NO_EXCEPTION).

Function	**chanend_enable_trigger**

Continued on next page

Description	Enable select & interrupt events on a chan-end. Prior to enabling, chanend_setup_select(), chanend_setup_select_callback() or chanend_setup_interrupt_callback() must have been called. Events can be temporarily disabled and re-enabled using chanend_disable_trigger() and chanend_enable_trigger(). When the event fires, the value must be read from the chan-end to clear the event.
Type	`xcore_c_error_t chanend_enable_trigger(chanend c)`
Parameters	c The chan-end to enable events on ET_ILLEGAL_RESOURCE not a valid chan-end. ET_RESOURCE_DEP another core is actively using the chan-end.
Returns	error_none (or exception type if policy is XCORE_C_NO_EXCEPTION).

Function	**chanend_disable_trigger**
Description	Disable select & interrupt events for a given chan-end. This function prevents any further events being triggered by a given chan-end.
Type	`xcore_c_error_t chanend_disable_trigger(chanend c)`
Parameters	c The chan-end to disable events on ET_ILLEGAL_RESOURCE not a valid chan-end. ET_RESOURCE_DEP another core is actively using the chan-end.
Returns	error_none (or exception type if policy is XCORE_C_NO_EXCEPTION).

2.4 Channels

Type	channel_t
Description	Helper type for passing around both ends of a channel.

Function	chan_alloc
Description	Allocate a channel. This function allocates two hardware chan-ends and joins them. If there are not enough chan-ends available the function returns a channel_t with both fields set to 0. When the channel_t is no longer required, chan_free() must be called to deallocate it. **The chan-ends must be accessed on the same tile**
Type	`xcore_c_error_t chan_alloc(channel_t *c)`
Parameters	c channel_t variable holding the two initialised and joined chan-ends or 0s. ET_LOAD_STORE invalid *c* argument.
Returns	error_none (or exception type if policy is XCORE_C_NO_EXCEPTION).

Function	chan_free
Description	Deallocate a channel. This function frees the two hardware chan-ends.
Type	`xcore_c_error_t chan_free(channel_t *c)`
Parameters	c channel_t to free ET_ILLEGAL_RESOURCE not an allocated chan-end, or channel handshaking corrupted. ET_RESOURCE_DEP another core is actively using the chanend. ET_LOAD_STORE invalid *c* argument.
Returns	error_none (or exception type if policy is XCORE_C_NO_EXCEPTION).

Function	**chan_out_word**
Description	Output a word over a channel.
Type	`xcore_c_error_t chan_out_word(chanend c, uint32_t data)`

Parameters	`c` The chan-end
	
	`data` The word to be output
	
	`ET_LINK_ERROR`
	chan-end destination is not set.
	
	`ET_ILLEGAL_RESOURCE`
	not an allocated chan-end, or channel handshaking corrupted.
	
	`ET_RESOURCE_DEP`
	another core is actively using the chan-end.

Returns	error_none (or exception type if policy is XCORE_C_NO_EXCEPTION).

Function	**chan_out_byte**
Description	Output a byte over a channel.
Type	`xcore_c_error_t chan_out_byte(chanend c, uint8_t data)`

Parameters	`c` The chan-end
	
	`data` The byte to be output
	
	`ET_LINK_ERROR`
	chan-end destination is not set.
	
	`ET_ILLEGAL_RESOURCE`
	not an allocated chan-end, or channel handshaking corrupted.
	
	`ET_RESOURCE_DEP`
	another core is actively using the chan-end.

Returns	error_none (or exception type if policy is XCORE_C_NO_EXCEPTION).

Function	**chan_out_buf_word**
Description	Output a block of data over a channel.

Continued on next page

Type	`xcore_c_error_t chan_out_buf_word(chanend c, const uint32_t buf[], size_t n)`
Parameters	c — The chan-end buf — A pointer to the buffer containing the data to send n — The number of words to send ET_LINK_ERROR — chan-end destination is not set. ET_ILLEGAL_RESOURCE — not an allocated chan-end, or channel handshaking corrupted. ET_RESOURCE_DEP — another core is actively using the chan-end. ET_LOAD_STORE — invalid *buf[]* argument.
Returns	error_none (or exception type if policy is XCORE_C_NO_EXCEPTION).

Function	**chan_out_buf_byte**
Description	Output a block of data over a channel.
Type	`xcore_c_error_t chan_out_buf_byte(chanend c, const uint8_t buf[], size_t n)`

Continued on next page

Parameters	c — The chan-end buf — A pointer to the buffer containing the data to send n — The number of bytes to send ET_LINK_ERROR chan-end destination is not set. ET_ILLEGAL_RESOURCE not an allocated chan-end, or channel handshaking corrupted. ET_RESOURCE_DEP another core is actively using the chan-end. ET_LOAD_STORE invalid *buf[]* argument.
Returns	error_none (or exception type if policy is XCORE_C_NO_EXCEPTION).

Function	**chan_in_word**
Description	Input a word from a channel.
Type	`xcore_c_error_t chan_in_word(chanend c, uint32_t *data)`
Parameters	c — The chan-end data — The inputted word ET_ILLEGAL_RESOURCE not an allocated chan-end, or channel handshaking corrupted. ET_RESOURCE_DEP another core is actively using the chan-end. ET_LOAD_STORE invalid **data* argument.
Returns	error_none (or exception type if policy is XCORE_C_NO_EXCEPTION).

Function	**chan_in_byte**
Description	Input a byte from a channel.
Type	`xcore_c_error_t chan_in_byte(chanend c, uint8_t *data)`

Continued on next page

Parameters	c — The chan-end data — The inputted byte ET_ILLEGAL_RESOURCE — not an allocated chan-end, or channel handshaking corrupted. ET_RESOURCE_DEP — another core is actively using the chan-end. ET_LOAD_STORE — invalid **data* argument.
Returns	error_none (or exception type if policy is XCORE_C_NO_EXCEPTION).

Function	**chan_in_buf_word**
Description	Input a block of data from a channel.
Type	`xcore_c_error_t chan_in_buf_word(chanend c, uint32_t buf[], size_t n)`
Parameters	c — The chan-end buf — A pointer to the memory region to fill n — The number of words to receive ET_ILLEGAL_RESOURCE — not an allocated chan-end, or channel handshaking corrupted. ET_RESOURCE_DEP — another core is actively using the chan-end. ET_LOAD_STORE — invalid *buf[]* argument.
Returns	error_none (or exception type if policy is XCORE_C_NO_EXCEPTION).

Function	**chan_in_buf_byte**
Description	Input a block of data from a channel.
Type	`xcore_c_error_t chan_in_buf_byte(chanend c, uint8_t buf[], size_t n)`

Continued on next page

Parameters	c	The chan-end
	buf	A pointer to the memory region to fill
	n	The number of bytes to receive
	ET_ILLEGAL_RESOURCE	not an allocated chan-end, or channel handshaking corrupted.
	ET_RESOURCE_DEP	another core is actively using the chan-end.
	ET_LOAD_STORE	invalid *buf[]* argument.
Returns	error_none (or exception type if policy is XCORE_C_NO_EXCEPTION).	

2.5 Streaming channels

Type	**streaming_channel_t**
Description	Helper type for passing around both ends of a streaming channel.

Fields	`streaming_chanend_t end_a`
	
	`streaming_chanend_t end_b`

Function	**s_chan_alloc**
Description	Allocate a streaming_channel_t. This function allocates two hardware chan-ends and joins them. If there are not enough chan-ends available the function returns a streaming_channel_t with both fields set to 0. When the streaming_channel_t is no longer required, s_chan_free() must be called to deallocate it. **The chan-ends must be accessed on the same tile**
Type	`xcore_c_error_t s_chan_alloc(streaming_channel_t *c)`
Parameters	c — streaming_channel_t variable holding the two initialised and joined chan-ends or 0s. ET_LOAD_STORE — invalid **c* argument.
Returns	error_none (or exception type if policy is XCORE_C_NO_EXCEPTION).

Function	**s_chan_free**
Description	Deallocate a streaming_channel_t. This function frees the two hardware chan-ends.
Type	`xcore_c_error_t s_chan_free(streaming_channel_t *c)`

Continued on next page

Parameters	c — streaming_channel_t to free. ET_LINK_ERROR — a chan-end destination is not set. ET_ILLEGAL_RESOURCE — not an allocated channel, or an input/output is pending. ET_RESOURCE_DEP — another core is actively using the channel. ET_LOAD_STORE — invalid **c* argument.
Returns	error_none (or exception type if policy is XCORE_C_NO_EXCEPTION).

Function	**s_chan_out_word**
Description	Output a word over a streaming_channel_t.

Type	`xcore_c_error_t`
	`s_chan_out_word(streaming_chanend_t c,`
	`                uint32_t data)`

Parameters	c — The streaming chan-end data — The word to be output ET_LINK_ERROR — chan-end destination is not set. ET_ILLEGAL_RESOURCE — not an allocated chan-end. ET_RESOURCE_DEP — another core is actively using the chan-end.
Returns	error_none (or exception type if policy is XCORE_C_NO_EXCEPTION).

Function	**s_chan_out_byte**
Description	Output an byte over a streaming_channel_t.

Type	`xcore_c_error_t`
	`s_chan_out_byte(streaming_chanend_t c,`
	`                uint8_t data)`

Continued on next page

Parameters	c	The streaming chan-end
	data	The byte to be output
	ET_LINK_ERROR	chan-end destination is not set.
	ET_ILLEGAL_RESOURCE	not an allocated chan-end.
	ET_RESOURCE_DEP	another core is actively using the chan-end.
Returns	error_none (or exception type if policy is XCORE_C_NO_EXCEPTION).	

Function	**s_chan_out_buf_word**	
Description	Output a block of data over a streaming_channel_t.	

Type	`xcore_c_error_t`
	`s_chan_out_buf_word(streaming_chanend_t c,`
	`                    const uint32_t buf[],`
	`                    size_t n)`	

Parameters	c	The streaming chan-end
	buf	A pointer to the buffer containing the data to send
	n	The number of words to send
	ET_LINK_ERROR	chan-end destination is not set.
	ET_ILLEGAL_RESOURCE	not an allocated chan-end.
	ET_RESOURCE_DEP	another core is actively using the chan-end.
	ET_LOAD_STORE	invalid *buf[]* argument.
Returns	error_none (or exception type if policy is XCORE_C_NO_EXCEPTION).	

Function	**s_chan_out_buf_byte**
Description	Output a block of data over a streaming_channel_t.

Type	`xcore_c_error_t`
	`s_chan_out_buf_byte(streaming_chanend_t c,`
	`                    const uint8_t buf[],`
	`                    size_t n)`

Parameters	`c` The streaming chan-end `buf` A pointer to the buffer containing the data to send `n` The number of bytes to send `ET_LINK_ERROR` chan-end destination is not set. `ET_ILLEGAL_RESOURCE` not an allocated chan-end. `ET_RESOURCE_DEP` another core is actively using the chan-end. `ET_LOAD_STORE` invalid *buf[]* argument.
Returns	error_none (or exception type if policy is XCORE_C_NO_EXCEPTION).

Function	**s_chan_in_word**
Description	Input a word from a streaming_channel_t.

Type	`xcore_c_error_t`
	`s_chan_in_word(streaming_chanend_t c,`
	`               uint32_t *data)`

Continued on next page

Parameters	`c` The streaming chan-end `data` The inputted word. `ET_ILLEGAL_RESOURCE` not an allocated chan-end, or has pending control token. `ET_RESOURCE_DEP` another core is actively using the chan-end. `ET_LOAD_STORE` invalid _*data_ argument.
Returns	error_none (or exception type if policy is XCORE_C_NO_EXCEPTION).

Function	**s_chan_in_byte**
Description	Input a byte from a streaming_channel_t.

Type	`xcore_c_error_t`
	`s_chan_in_byte(streaming_chanend_t c,`
	`               uint8_t *data)`

Parameters	`c` The streaming chan-end `data` The inputted byte `ET_ILLEGAL_RESOURCE` not an allocated chan-end, or has pending control token. `ET_RESOURCE_DEP` another core is actively using the chan-end. `ET_LOAD_STORE` invalid _*data_ argument.
Returns	error_none (or exception type if policy is XCORE_C_NO_EXCEPTION).

Function	**s_chan_in_buf_word**
Description	Input a block of data from a streaming_channel_t.

Type	`xcore_c_error_t`
	`s_chan_in_buf_word(streaming_chanend_t c,`
	`                   uint32_t buf[],`
	`                   size_t n)`

Continued on next page

Parameters	c The streaming chan-end buf A pointer to the memory region to fill n The number of words to receive ET_ILLEGAL_RESOURCE not an allocated chan-end, or has pending control token. ET_RESOURCE_DEP another core is actively using the chan-end. ET_LOAD_STORE invalid *buf[]* argument.
Returns	error_none (or exception type if policy is XCORE_C_NO_EXCEPTION).

Function	**s_chan_in_buf_byte**
Description	Input a block of data from a streaming_channel_t.

Type	`xcore_c_error_t`
	`s_chan_in_buf_byte(streaming_chanend_t c,`
	`                   uint8_t buf[],`
	`                   size_t n)`

Parameters	c The streaming chan-end buf A pointer to the memory region to fill n The number of bytes to receive ET_ILLEGAL_RESOURCE not an allocated chan-end, or has pending control token. ET_RESOURCE_DEP another core is actively using the chan-end. ET_LOAD_STORE invalid *buf[]* argument.
Returns	error_none (or exception type if policy is XCORE_C_NO_EXCEPTION).

Function	s_chan_out_ct
Description	Output a control token onto a streaming_channel_t.
Type	`xcore_c_error_t s_chan_out_ct(streaming_chanend_t c, uint8_t ct)`
Parameters	`c` The streaming chan-end `ct` Control token to be output. Legal control tokens that can be used are 0 or any value in the range 3..191 inclusive. ET_LINK_ERROR chan-end destination is not set or token is reserverd. ET_ILLEGAL_RESOURCE not an allocated chan-end. ET_RESOURCE_DEP another core is actively using the chan-end.
Returns	error_none (or exception type if policy is XCORE_C_NO_EXCEPTION).

Function	s_chan_out_ct_end
Description	Output a CT_END control token onto a streaming_channel_t. Outputting a CT_END control token informs the communication network and the other chan-end that the current transaction has completed. Thus freeing the communication network for other channels to use. The streaming_channel_t remains allocated, awaiting another transaction or deallocation.

Type	`xcore_c_error_t`
	`s_chan_out_ct_end(streaming_chanend_t c)`

Parameters	`c` The streaming chan-end ET_LINK_ERROR chan-end destination is not set. ET_ILLEGAL_RESOURCE not an allocated chan-end. ET_RESOURCE_DEP another core is actively using the chan-end.
Returns	error_none (or exception type if policy is XCORE_C_NO_EXCEPTION).

Function	s_chan_check_ct
Description	Check that a specific control token on a streaming_channel_t. This function blocks until a token is available on the streaming_channel_t. If the available token is a control token and has the value `ct`, then the token is input and discarded. Otherwise an exception is raised.

Type	`xcore_c_error_t`
	`s_chan_check_ct(streaming_chanend_t c, uint8_t ct)`

Parameters	`c` The streaming chan-end `ct` Control token that is expected on the streaming_channel_t. ET_ILLEGAL_RESOURCE not an allocated chan-end, or does not contain expected token. ET_RESOURCE_DEP another core is actively using the chan-end.
Returns	error_none (or exception type if policy is XCORE_C_NO_EXCEPTION).

Function	s_chan_check_ct_end
Description	Check for a CT_END token on a streaming_channel_t. This function blocks until a token is available on the streaming_channel_t. If the available token is a CT_END token the token is input and discarded. Otherwise an exception is raised.

Type	`xcore_c_error_t`
	`s_chan_check_ct_end(streaming_chanend_t c)`

Parameters	`c` The streaming chan-end ET_ILLEGAL_RESOURCE not an allocated chan-end, or does not contain CT_END token. ET_RESOURCE_DEP another core is actively using the chan-end.
Returns	error_none (or exception type if policy is XCORE_C_NO_EXCEPTION).

2.6 Channels with transactions

Function	**chan_init_transaction_master**
Description	Start a transaction (master). This initiates a transaction on a channel. A transacting_chanend_t is used to temporarily open a transaction route through a channel. During the transaction, you can use transaction channel operations for increased efficiency. You can create a transacting chanend from a normal *chanend* using chan_init_transaction_master() and chan_init_transaction_slave(). This called must be matched by a call to chan_init_transaction_slave() on the other end of the channel. A transaction must be closed with chan_complete_transaction().

Type	`xcore_c_error_t`
	`chan_init_transaction_master(chanend *c,`
	`                             transacting_chanend_t *tc)`

Parameters	`c` chan-end to initialize the transaction on. chanend is invalidated `tc` the intialized master transacting_chanend_t `ET_LINK_ERROR` chan-end destination is not set. `ET_ILLEGAL_RESOURCE` not an allocated chan-end. `ET_RESOURCE_DEP` another core is actively using the chan-end. `ET_LOAD_STORE` invalid **c* or **tc* argument.
Returns	error_none (or exception type if policy is XCORE_C_NO_EXCEPTION).

Function	**chan_init_transaction_slave**
Description	Start a transaction (slave). This initiates a transaction on a channel. This called must be matched by a call to chan_init_transaction_master() on the other end of the channel. A transaction must be closed with chan_complete_transaction(). The original *chanend* must not be used until the transaction is closed.

Type	`xcore_c_error_t`
	`chan_init_transaction_slave(chanend *c,`
	`                            transacting_chanend_t *tc)`

Continued on next page

Parameters	c — chan-end to initialize the transaction on. chanend is invalidated tc — the intialized slave transacting_chanend_t ET_ILLEGAL_RESOURCE not an allocated chan-end, or does not contain CT_END token. ET_RESOURCE_DEP another core is actively using the chan-end. ET_LOAD_STORE invalid **c* or **tc* argument.
Returns	error_none (or exception type if policy is XCORE_C_NO_EXCEPTION).

Function	**chan_complete_transaction**
Description	Complete a transaction. This function completes a transaction. After this call the route between the two ends of the channel is freed allowing other channels to use the communication network. Whilst the transacting_chanend_t is now invalid, the channel remains allocated, awaiting another transaction or deallocation. This call must be accompanied by a call to chan_complete_transaction() on the other end of the channel.

Type	`xcore_c_error_t`
	`chan_complete_transaction(chanend *c,`
	`                          transacting_chanend_t *tc)`

Parameters	c — The original chan-end. chanend is made valid again. tc — Transacting chan-end to close. transacting_chanend_t is invalidated ET_LINK_ERROR chan-end destination is not set. ET_ILLEGAL_RESOURCE not an allocated chan-end, or channel handshaking corrupted. ET_RESOURCE_DEP another core is actively using the chan-end. ET_LOAD_STORE invalid **c* or **tc* argument.

Function	**t_chan_out_word**
Description	Output a word over a transacting chan-end.

Type	`xcore_c_error_t`
	`t_chan_out_word(transacting_chanend_t *tc,`
	`                uint32_t data)`

Parameters	`tc` Transacting chan-end
	
	`data` Word to be output
	
	`ET_LINK_ERROR`
	chan-end destination is not set.
	
	`ET_ILLEGAL_RESOURCE`
	not an allocated chan-end, or channel handshaking corrupted.
	
	`ET_RESOURCE_DEP`
	another core is actively using the chan-end.
	
	`ET_LOAD_STORE`
	invalid **tc* argument.

Returns	error_none (or exception type if policy is XCORE_C_NO_EXCEPTION).

Function	**t_chan_out_byte**
Description	Output an byte over a transacting chan-end.

Type	`xcore_c_error_t`
	`t_chan_out_byte(transacting_chanend_t *tc,`
	`                uint8_t data)`

Continued on next page

Parameters	tc Transacting chan-end data Byte to be output ET_LINK_ERROR chan-end destination is not set. ET_ILLEGAL_RESOURCE not an allocated chan-end, or channel handshaking corrupted. ET_RESOURCE_DEP another core is actively using the chan-end. ET_LOAD_STORE invalid **tc* argument.
Returns	error_none (or exception type if policy is XCORE_C_NO_EXCEPTION).

Function	**t_chan_out_buf_word**
Description	Output a block of data over a transacting chan-end.

Type	`xcore_c_error_t`
	`t_chan_out_buf_word(transacting_chanend_t *tc,`
	`                    const uint32_t buf[],`
	`                    size_t n)`

Parameters	tc Transacting chan-end buf Pointer to the buffer containing the data to send n Number of words to send ET_LINK_ERROR chan-end destination is not set. ET_ILLEGAL_RESOURCE not an allocated chan-end, or channel handshaking corrupted. ET_RESOURCE_DEP another core is actively using the chan-end. ET_LOAD_STORE invalid **tc* or *buf[]* argument.
Returns	error_none (or exception type if policy is XCORE_C_NO_EXCEPTION).

Function	**t_chan_out_buf_byte**
Description	Output a block of data over a transacting chan-end.

Type	`xcore_c_error_t`
	`t_chan_out_buf_byte(transacting_chanend_t *tc,`
	`                    const uint8_t buf[],`
	`                    size_t n)`

Parameters	`tc` Transacting chan-end
	
	`buf` Pointer to the buffer containing the data to send
	
	`n` Number of bytes to send
	
	`ET_LINK_ERROR`
	chan-end destination is not set.
	
	`ET_ILLEGAL_RESOURCE`
	not an allocated chan-end, or channel handshaking corrupted.
	
	`ET_RESOURCE_DEP`
	another core is actively using the chan-end.
	
	`ET_LOAD_STORE`
	invalid **tc* or *buf[]* argument.

Returns	error_none (or exception type if policy is XCORE_C_NO_EXCEPTION).

Function	**t_chan_in_word**
Description	Input a word from a transacting chan-end.

Type	`xcore_c_error_t`
	`t_chan_in_word(transacting_chanend_t *tc,`
	`               uint32_t *data)`

Continued on next page

Parameters	`tc` Transacting chan-end `data` Inputted integer `ET_LINK_ERROR` chan-end destination is not set. `ET_ILLEGAL_RESOURCE` not an allocated chan-end, or channel handshaking corrupted. `ET_RESOURCE_DEP` another core is actively using the chan-end. `ET_LOAD_STORE` invalid **tc* or **data* argument.
Returns	error_none (or exception type if policy is XCORE_C_NO_EXCEPTION).

Function	**t_chan_in_byte**
Description	Input a byte from a transacting chan-end.

Type	`xcore_c_error_t`
	`t_chan_in_byte(transacting_chanend_t *tc,`
	`               uint8_t *data)`

Parameters	`tc` Transacting chan-end `data` Inputted byte `ET_LINK_ERROR` chan-end destination is not set. `ET_ILLEGAL_RESOURCE` not an allocated chan-end, or channel handshaking corrupted. `ET_RESOURCE_DEP` another core is actively using the chan-end. `ET_LOAD_STORE` invalid **tc* or **data* argument.
Returns	error_none (or exception type if policy is XCORE_C_NO_EXCEPTION).

Function	**t_chan_in_buf_word**
Description	Input a block of data from a transacting chan-end.

Type	`xcore_c_error_t`
	`t_chan_in_buf_word(transacting_chanend_t *tc,`
	`                   uint32_t buf[],`
	`                   size_t n)`

Parameters	`tc` Transacting chan-end
	
	`buf` Pointer to the memory region to fill
	
	`n` The number of words to receive
	
	`ET_LINK_ERROR`
	chan-end destination is not set.
	
	`ET_ILLEGAL_RESOURCE`
	not an allocated chan-end, or channel handshaking corrupted.
	
	`ET_RESOURCE_DEP`
	another core is actively using the chan-end.
	
	`ET_LOAD_STORE`
	invalid **tc* or *buf[]* argument.

Returns	error_none (or exception type if policy is XCORE_C_NO_EXCEPTION).

Function	**t_chan_in_buf_byte**
Description	Input a block of data from a transacting chan-end.

Type	`xcore_c_error_t`
	`t_chan_in_buf_byte(transacting_chanend_t *tc,`
	`                   uint8_t buf[],`
	`                   size_t n)`

Continued on next page

Parameters	`tc` Transacting chan-end
	
	`buf` Pointer to the memory region to fill
	
	`n` The number of bytes to receive
	
	`ET_LINK_ERROR`
	chan-end destination is not set.
	
	`ET_ILLEGAL_RESOURCE`
	not an allocated chan-end, or channel handshaking corrupted.
	
	`ET_RESOURCE_DEP`
	another core is actively using the chan-end.
	
	`ET_LOAD_STORE`
	invalid **tc* or *buf[]* argument.

Returns	error_none (or exception type if policy is XCORE_C_NO_EXCEPTION).

2.7 Clock blocks

Type	**clock_id_t**
Description	A clock block identifier. Clock resources must be allocated by name rather than from a pool.

Values	`clock_ref`
	
	`clock_1`
	
	`clock_2`
	
	`clock_3`
	
	`clock_4`
	
	`clock_5`

Function	**clock_alloc**
Description	Allocates a clock. This function enables a specified clock block and returns a clock variable denoting the clock.
Type	`xcore_c_error_t clock_alloc(clock *clk, clock_id_t id)`
Parameters	`clk` Clock variable representing the initialised clock `id` The id of the clock to allocate ET_ILLEGAL_RESOURCE not a valid clock. ET_RESOURCE_DEP another core is actively changing the clock. ET_LOAD_STORE invalid **clk* argument.
Returns	error_none (or exception type if policy is XCORE_C_NO_EXCEPTION).

Function	**clock_free**
Description	Deallocate a clock.
Type	`xcore_c_error_t clock_free(clock *clk)`
Parameters	`clk` The clock to be freed ET_ILLEGAL_RESOURCE not a valid clock. ET_RESOURCE_DEP another core is actively changing the clock. ET_LOAD_STORE invalid **clk* argument.
Returns	error_none (or exception type if policy is XCORE_C_NO_EXCEPTION).

Function	**clock_start**
Description	Start a clock. This function starts a clock running.
Type	`xcore_c_error_t clock_start(clock clk)`
Parameters	`clk` The clock to start running ET_ILLEGAL_RESOURCE not a valid clock. ET_RESOURCE_DEP another core is actively changing the clock.
Returns	error_none (or exception type if policy is XCORE_C_NO_EXCEPTION).

Function	**clock_stop**
Description	Stop a clock. This function waits until the clock is low and then pauses a clock.
Type	`xcore_c_error_t clock_stop(clock clk)`

Continued on next page

Parameters	`clk` The clock to stop `ET_ILLEGAL_RESOURCE` not a valid clock. `ET_RESOURCE_DEP` another core is actively changing the clock.
Returns	error_none (or exception type if policy is XCORE_C_NO_EXCEPTION).

Function	**clock_set_source_port**
Description	Configure a clock's source to a 1-bit port. A clock can be a 1-bit port, the reference clock or the xCORE clock. Note that if the xCORE clock is used then a non-zero divide must be used for ports to function correctly.
Type	`xcore_c_error_t clock_set_source_port(clock clk, port p)`
Parameters	`clk` The clock to configure p The 1-bit port to set as the clock input. Attempting to set a port which is not 1-bit as the input will cause an exception. `ET_ILLEGAL_RESOURCE` not a valid clock or port, or the clock is running, or p not a one bit port. `ET_RESOURCE_DEP` another core is actively changing the clock.
Returns	error_none (or exception type if policy is XCORE_C_NO_EXCEPTION).

Function	**clock_set_source_clk_ref**
Description	Configure a clock's source to be the 100MHz reference clock.
Type	`xcore_c_error_t clock_set_source_clk_ref(clock clk)`
Parameters	`clk` The clock to configure `ET_ILLEGAL_RESOURCE` not a valid clock, or the clock is running. `ET_RESOURCE_DEP` another core is actively changing the clock.

Continued on next page

Returns	error_none (or exception type if policy is XCORE_C_NO_EXCEPTION).

Function	**clock_set_source_clk_xcore**
Description	Configure a clock's source to be the xCORE clock. *Note*: When using the xCORE clock as the clock input a divide of > 0 must be used for the ports to function correclty.

Type	`xcore_c_error_t`
	`clock_set_source_clk_xcore(clock clk)`

Parameters	`clk` The clock to configure
	
	`ET_ILLEGAL_RESOURCE`
	not a valid clock, or the clock is running.
	
	`ET_RESOURCE_DEP`
	another core is actively changing the clock.

Returns	error_none (or exception type if policy is XCORE_C_NO_EXCEPTION).

Function	**clock_set_divide**
Description	Configure the divider for a clock. A clock can divide its input signal by an integer value which this function specifies. The XS2 architecture supports dividing the signal from a 1-bit port while the XS1 architecture will raise a trap if a non-zero divide is used with a 1-bit port input. If the clock has been started then this will raise an exception. If the divide is 0 then the value signal will be passed through the clock. If the value is non-zero then the clock output will be divided by 2*divide.
Type	`xcore_c_error_t clock_set_divide(clock clk, uint8_t divide)`

Parameters	`clk` The clock to configure
	
	`divide` The divide value
	
	`ET_ILLEGAL_RESOURCE`
	not a valid clock, or the clock is running.
	
	`ET_RESOURCE_DEP`
	another core is actively changing the clock.

Returns	error_none (or exception type if policy is XCORE_C_NO_EXCEPTION).

Function	**clock_set_ready_src**
Description	Sets a clock to use a 1-bit port for the ready-in signal. If the port is not a 1-bit port then an exception is raised. The ready-in port controls when data is sampled from the pins.
Type	`xcore_c_error_t clock_set_ready_src(clock clk, port ready_source)`
Parameters	`clk` The clock to configure. `ready_source` The 1-bit port to use for the ready-in signal. ET_ILLEGAL_RESOURCE not a valid clock, or ready_source not a one bit port. ET_RESOURCE_DEP another core is actively changing the clock.
Returns	error_none (or exception type if policy is XCORE_C_NO_EXCEPTION).

2.8 Locks

Function	**lock_alloc**
Description	Allocates a lock. This function allocates a hardware lock. If there are no locks availble the function returns 0. When the lock is no longer required, lock_free() must be called to deallocate it.
Type	`xcore_c_error_t lock_alloc(lock_t *l)`
Parameters	l — lock_t variable representing the initialised lock ET_LOAD_STORE — invalid *l argument.
Returns	error_none (or exception type if policy is XCORE_C_NO_EXCEPTION).

Function	**lock_free**
Description	Deallocate a lock. This function frees the hardware lock. The lock must be released prior to calling this function.
Type	`xcore_c_error_t lock_free(lock_t *l)`
Parameters	l — The lock_t to be freed ET_ILLEGAL_RESOURCE — not an allocated lock, or the lock has not been released. ET_RESOURCE_DEP — another core is actively changing the lock. ET_LOAD_STORE — invalid *l argument.
Returns	error_none (or exception type if policy is XCORE_C_NO_EXCEPTION).

Function	**lock_acquire**
Description	Acquire a lock. Only one core at a time can acquire a lock. This provides a hardware mutex which have very low overheads. If another thread has already acquired this lock then this function will pause until the lock is released and this core becomes the owner.

Continued on next page

Type	`xcore_c_error_t lock_acquire(lock_t l)`
Parameters	`l` The lock_t to acquire `ET_ILLEGAL_RESOURCE` not an allocated lock. `ET_RESOURCE_DEP` another core is actively changing the lock.
Returns	error_none (or exception type if policy is XCORE_C_NO_EXCEPTION).

Function	**lock_release**
Description	Release a lock. This releases the lock and allocates the next owner from the list of waiting cores in round-robin manner. *Note*: there are no checks that the core releasing the lock is the current owner.
Type	`xcore_c_error_t lock_release(lock_t l)`
Parameters	`l` The lock_t to use release `ET_ILLEGAL_RESOURCE` not an allocated lock. `ET_RESOURCE_DEP` another core is actively changing the lock.
Returns	error_none (or exception type if policy is XCORE_C_NO_EXCEPTION).

2.9 Ports

Type	**port_id_t**
Description	A port identifier. Port resources must be allocated by name rather than from a poll.

Values	`port_1A`
	`port_1B`
	`port_1C`
	`port_1D`
	`port_1E`
	`port_1F`
	`port_1G`
	`port_1H`
	`port_1I`
	`port_1J`
	`port_1K`
	`port_1L`
	`port_1M`
	`port_1N`
	`port_1O`
	`port_1P`
	`port_4A`
	`port_4B`
	`port_4C`

Continued on next page

	port_4D port_4E port_4F port_8A port_8B port_8C port_8D port_16A port_16B port_16C port_16D port_32A port_32B

Type	**port_type_t**
Description	Enumeration to declare how the port was set up.
Values	PORT_UNBUFFERED PORT_BUFFERED

Function	**port_alloc**

Continued on next page

Description	Allocates a port. Either this function or port_alloc_buffered() must be called once for each variable of type `port` before use. port_free() must be called afterwards. The port's state is set to: input, unbuffered, inout_data, no_invert, rising_edge, master, no_ready, no triggers, clocked by XS1_CLKBLK_REF.
Type	`xcore_c_error_t port_alloc(port *p, port_id_t id)`
Parameters	p: Port variable representing the initialised port `id`: Value that identifies which port to drive ET_ILLEGAL_RESOURCE: not a valid port. ET_RESOURCE_DEP: another core is actively changing the port. ET_LOAD_STORE: invalid **p* argument.
Returns	error_none (or exception type if policy is XCORE_C_NO_EXCEPTION).

Function	**port_reset**
Description	Reset a port. Clears a ports settings back to the default state at port_alloc().
Type	`xcore_c_error_t port_reset(port p)`
Parameters	p: The port to be reset ET_ILLEGAL_RESOURCE: not a valid port. ET_RESOURCE_DEP: another core is actively changing the port. ET_LOAD_STORE: invalid **p* argument.
Returns	error_none (or exception type if policy is XCORE_C_NO_EXCEPTION).

Function	port_alloc_buffered
Description	Allocates a port to buffer and serialise/deserialise data. Either this function or port_alloc() must be called once for each variable of type `port` before use. port_free() must be called afterwards.

Type	`xcore_c_error_t port_alloc_buffered(port *p,`
	`port_id_t id,`
	`size_t transfer_width)`

Parameters	p: Port variable representing the initialised port `id`: Value that identifies which port to drive `transfer_width`: Number of bits to serialise; must be 1, 4, 8, or 32. The number of bits must be >= to the physical port width. ET_ILLEGAL_RESOURCE: not a valid port, or is not legal width for the port. ET_RESOURCE_DEP: another core is actively changing the port. ET_LOAD_STORE: invalid **p* argument.
Returns	error_none (or exception type if policy is XCORE_C_NO_EXCEPTION).

Function	port_free
Description	Deallocate a port.
Type	`xcore_c_error_t port_free(port *p)`
Parameters	p: The port to be freed ET_ILLEGAL_RESOURCE: not a valid port. ET_RESOURCE_DEP: another core is actively changing the port. ET_LOAD_STORE: invalid **p* argument.
Returns	error_none (or exception type if policy is XCORE_C_NO_EXCEPTION).

Function	**port_set_transfer_width**
Description	Change the transfer width of a port. The default transfer width is the same as the physical port width. **A port must have been set to buffered if the width is different from the physical port width**

Type	`xcore_c_error_t port_set_transfer_width(port p,`
	`size_t transfer_width)`

Parameters	p — The port to change the transfer width of `transfer_width` — Number of bits to serialise; must be 1, 4, 8, or 32. The number of bits must be >= to the physical port width. ET_ILLEGAL_RESOURCE — not a valid port, or is not legal width for the port, or the port is unbuffered. ET_RESOURCE_DEP — another core is actively changing the port.
Returns	error_none (or exception type if policy is XCORE_C_NO_EXCEPTION).

Function	**port_set_buffered**
Description	Sets a port to be buffered. Configures a port into buffered mode where it can automatically serialise or deserialise data.
Type	`xcore_c_error_t port_set_buffered(port p)`
Parameters	p — The port to set as buffered ET_ILLEGAL_RESOURCE — not a valid port. ET_RESOURCE_DEP — another core is actively changing the port.
Returns	error_none (or exception type if policy is XCORE_C_NO_EXCEPTION).

Function	**port_set_unbuffered**
Description	Sets a port to be unbuffered (default state). Configures a port into unbuffered mode. Note that before this is called, a a port needs to have its transfer width equal to the port width and be configured as a master port.
Type	`xcore_c_error_t port_set_unbuffered(port p)`

Parameters	`p` The port to set as unbuffered
	
	`ET_ILLEGAL_RESOURCE`
	not a valid port.
	
	`ET_RESOURCE_DEP`
	another core is actively changing the port.

Returns	error_none (or exception type if policy is XCORE_C_NO_EXCEPTION).

Function	**port_set_clock**
Description	Set the clock clocking a port. This function changes the clock used for a port's control functions. The default clock is 'XS1_CLKBLK_REF'.
Type	`xcore_c_error_t port_set_clock(port p, clock clk)`

Parameters	`p` Port whose clock is being changed.
	
	`clk` Clock to attach to the port.
	
	`ET_ILLEGAL_RESOURCE`
	not a valid port, clock, or clock is running.
	
	`ET_RESOURCE_DEP`
	another core is actively changing the port.

Returns	error_none (or exception type if policy is XCORE_C_NO_EXCEPTION).

Function	**port_set_inout_data**
Description	Set a port drive out the data value (default state).
Type	`xcore_c_error_t port_set_inout_data(port p)`

Continued on next page

Parameters	p — Port to change the mode of ET_ILLEGAL_RESOURCE — not a valid port. ET_RESOURCE_DEP — another core is actively changing the port.
Returns	error_none (or exception type if policy is XCORE_C_NO_EXCEPTION).

Function	**port_set_out_clock**
Description	Set a port to drive out its clocking signal. This function configures the port to drive the clock signal instead of its own data values. The clock signal that is driven out is configured using the port_set_clock() function.
Type	`xcore_c_error_t port_set_out_clock(port p)`
Parameters	p — Port to change the mode of ET_ILLEGAL_RESOURCE — not a valid port. ET_RESOURCE_DEP — another core is actively changing the port.
Returns	error_none (or exception type if policy is XCORE_C_NO_EXCEPTION).

Function	**port_set_out_ready**
Description	Set a port to drive out the ready signal of another port. This function configures the port to drive the ready signal of another port instead of its own data values.
Type	`xcore_c_error_t port_set_out_ready(port p, port ready_source)`

Continued on next page

Parameters	p — Port to change the mode of. This must be a 1-bit port or this function will trap. ready_source — The port whose ready signal will be output ET_ILLEGAL_RESOURCE — not a valid port. or p not a one bit port. ET_RESOURCE_DEP — another core is actively changing the port.
Returns	error_none (or exception type if policy is XCORE_C_NO_EXCEPTION).

Function	**port_set_invert**
Description	Set the port to invert its data. This function configures a port to invert the data on the pin. This can be reverted by calling port_set_no_invert().
Type	`xcore_c_error_t port_set_invert(port p)`
Parameters	p — Port to set its data to be inverted. This must be a 1-bit port or a trap will be raised. ET_ILLEGAL_RESOURCE — not a valid port, or p not a one bit port. ET_RESOURCE_DEP — another core is actively changing the port.
Returns	error_none (or exception type if policy is XCORE_C_NO_EXCEPTION).

Function	**port_set_no_invert**
Description	Set the port to not invert its data (default state). This function configures a port to not invert the data on the pin.
Type	`xcore_c_error_t port_set_no_invert(port p)`

Continued on next page

Parameters	p Port to set the data to not be inverted. ET_ILLEGAL_RESOURCE not a valid port. ET_RESOURCE_DEP another core is actively changing the port.
Returns	error_none (or exception type if policy is XCORE_C_NO_EXCEPTION).

Function	**port_set_sample_falling_edge**
Description	Set the port to sample on the falling edge. The default is for a port to sample data on the rising edge of the clock. This function changes the port to sample on the falling edge instead. This change can be reverted by calling port_set_sample_rising_edge().
Type	`xcore_c_error_t port_set_sample_falling_edge(port p)`
Parameters	p Port to change its sampling edge. ET_ILLEGAL_RESOURCE not a valid port. ET_RESOURCE_DEP another core is actively changing the port.
Returns	error_none (or exception type if policy is XCORE_C_NO_EXCEPTION).

Function	**port_set_sample_rising_edge**
Description	Set the port to sample on the rising edge (default state). This function restores a port to sampling data on the rising edge of the clock.
Type	`xcore_c_error_t port_set_sample_rising_edge(port p)`
Parameters	p Port to change its sampling edge. ET_ILLEGAL_RESOURCE not a valid port. ET_RESOURCE_DEP another core is actively changing the port.
Returns	error_none (or exception type if policy is XCORE_C_NO_EXCEPTION).

Function	**port_set_master**
Description	Set the port to master mode (default state). This function configures a port to be a master. This is only relevant when using ready signals (port_set_ready_strobed() / port_set_ready_handshake()). It is highly recommended to use the `port_protocol_*` functions to put a port into its desired mode as the order of operations is critical.
Type	`xcore_c_error_t port_set_master(port p)`
Parameters	p Port to set as a master ET_ILLEGAL_RESOURCE not a valid port. ET_RESOURCE_DEP another core is actively changing the port.
Returns	error_none (or exception type if policy is XCORE_C_NO_EXCEPTION).

Function	**port_set_slave**
Description	Set the port to slave mode. This function configures a port to be a master. This is only relevant when using a ready strobe (port_set_ready_strobed()) *Note*: the port must be set to use a ready strobe, otherwise this function will raise an exception. It is highly recommended to use the `port_protocol_*` functions to put a port into its desired mode as the order of operations is critical.
Type	`xcore_c_error_t port_set_slave(port p)`
Parameters	p Port to set as a slave ET_ILLEGAL_RESOURCE not a valid port. ET_RESOURCE_DEP another core is actively changing the port.
Returns	error_none (or exception type if policy is XCORE_C_NO_EXCEPTION).

Function	**port_set_no_ready**

Continued on next page

Description	Set the port to use no ready signals (default state). This function changes a port to not use ready signals. A port can be configured to use strobes or handshaking signals using port_set_ready_strobed() or port_set_ready_handshake(). *Note*: the port must be a `master` port otherwise this function will raise an exception. It is highly recommended to use the `port_protocol_*` functions to put a port into its desired mode as the order of operations is critical.
Type	`xcore_c_error_t port_set_no_ready(port p)`
Parameters	p Port to change to not use ready signals ET_ILLEGAL_RESOURCE not a valid port. ET_RESOURCE_DEP another core is actively changing the port.
Returns	error_none (or exception type if policy is XCORE_C_NO_EXCEPTION).

Function	**port_set_ready_strobed**
Description	Set the port to use a single strobe. This function changes a port to not use ready signals. A port can be configured to use strobes or handshaking signals using port_set_ready_strobed() or port_set_ready_handshake(). *Note*: the port must be a buffered port otherwise this function will raise an exception. It is highly recommended to use the `port_protocol_*` functions to put a port into its desired mode as the order of operations is critical.
Type	`xcore_c_error_t port_set_ready_strobed(port p)`
Parameters	p Port to change to not use ready signals ET_ILLEGAL_RESOURCE not a valid port. ET_RESOURCE_DEP another core is actively changing the port.
Returns	error_none (or exception type if policy is XCORE_C_NO_EXCEPTION).

Function	**port_set_ready_handshake**

Continued on next page

Description	Set the port to be fully handshaken. This function changes a port to use both a ready input and drive a ready output in order to control when data is sampled or written. *Note*: the port must be a master buffered port otherwise this function will raise an exception. It is highly recommended to use the `port_protocol_*` functions to put a port into its desired mode as the order of operations is critical.
Type	`xcore_c_error_t port_set_ready_handshake(port p)`
Parameters	p Port to change to not use ready signals ET_ILLEGAL_RESOURCE not a valid port. ET_RESOURCE_DEP another core is actively changing the port.
Returns	error_none (or exception type if policy is XCORE_C_NO_EXCEPTION).

Function	**port_get_trigger_time**
Description	Get the timestamp of the last operation on a port. This function gets the timestamp of the last input or output operation on a port.
Type	`xcore_c_error_t port_get_trigger_time(port p, int16_t *t)`
Parameters	p The port to get the timestamp from t The timestamp of the last operation ET_ILLEGAL_RESOURCE not a valid port. ET_RESOURCE_DEP another core is actively changing the port. ET_LOAD_STORE invalid **t* argument.
Returns	error_none (or exception type if policy is XCORE_C_NO_EXCEPTION).

Function	**port_set_trigger_time**

Continued on next page

Description	Set the timestamp at which the port will input/output data. This function sets the time condition for the next input or output on a port. If the port is unbuffered or the buffer is empty/full a call to port_in() or port_out() will pause until the specified time. The trigger is cleared by a input/output or by calling port_clear_trigger_time().
Type	`xcore_c_error_t port_set_trigger_time(port p, int16_t t)`
Parameters	p The port to set the condition on t The port timestamp to match ET_ILLEGAL_RESOURCE not a valid port. ET_RESOURCE_DEP another core is actively using the port.
Returns	error_none (or exception type if policy is XCORE_C_NO_EXCEPTION).

Function	**port_clear_trigger_time**
Description	Clear the timestamp trigger on a port. This function clears any trigger_time condition on the port so the next input or output will happen unconditionally in respect to the timestamp. This function does not clear the trigger_in condition on the port.
Type	`xcore_c_error_t port_clear_trigger_time(port p)`
Parameters	p the port to clear the trigger_time on ET_ILLEGAL_RESOURCE not a valid port. ET_RESOURCE_DEP another core is actively changing the port.
Returns	error_none (or exception type if policy is XCORE_C_NO_EXCEPTION).

Function	**port_set_trigger_in_equal**
Description	Setup an event to trigger on a port when its input value matches. On a unbuffered port the trigger will apply to all future inputs until the trigger is set again. On an buffered port the trigger will only hold for the next input after which the trigger_in_equal will be cleared.

Continued on next page

Type	`xcore_c_error_t port_set_trigger_in_equal(port p, uint32_t v)`
Parameters	p — The port to set the trigger on v — The value to match ET_ILLEGAL_RESOURCE not a valid port. ET_RESOURCE_DEP another core is actively changing the port.
Returns	error_none (or exception type if policy is XCORE_C_NO_EXCEPTION).

Function	**port_set_trigger_in_not_equal**
Description	Setup an event to trigger on a port when its input value does not matches. On a unbuffered port the trigger will apply to all future inputs until the trigger is set again. On an buffered port the trigger will only hold for the next input after which the trigger_in_not_equal will be cleared.

Type	`xcore_c_error_t`
	`port_set_trigger_in_not_equal(port p, uint32_t v)`

Parameters	p — The port to set the trigger on v — The value not to match ET_ILLEGAL_RESOURCE not a valid port. ET_RESOURCE_DEP another core is actively changing the port.
Returns	error_none (or exception type if policy is XCORE_C_NO_EXCEPTION).

Function	**port_clear_trigger_in**
Description	Clear the in trigger on a port. This function clears any trigger_in condition on the port so the next input will happen unconditionally in respect to the input value. This function does not clear the trigger_time condition on the port.
Type	`xcore_c_error_t port_clear_trigger_in(port p)`

Continued on next page

Parameters	p The port to clear the trigger_in on ET_ILLEGAL_RESOURCE not a valid port. ET_RESOURCE_DEP another core is actively changing the port.
Returns	error_none (or exception type if policy is XCORE_C_NO_EXCEPTION).

Function	**port_peek**
Description	Peek at the value on a a port. Peeking a port returns the current value on the pins of a port, regardless of whether the port is an output or input and without affecting its direciton. Peek will not pause, regardless of any triggers that have been set.
Type	`xcore_c_error_t port_peek(port p, uint32_t *data)`
Parameters	p Port to be peeked data The current value on the pins ET_ILLEGAL_RESOURCE not a valid port. ET_LOAD_STORE invalid **data* argument.
Returns	error_none (or exception type if policy is XCORE_C_NO_EXCEPTION).

Function	**port_out**
Description	Outputs a value onto a port. In the case of an unbuffered port, the value will be driven on the pins on the next clock cycle. In the case of a buffered port, the data will be stored in the buffer, and be serialised onto the output pins. If there is a time trigger setup and the port is unbuffered or the buffer is full the call will pause until the specified time.
Type	`xcore_c_error_t port_out(port p, uint32_t data)`

Continued on next page

Parameters	p — Port to output to data — Value to output ET_ILLEGAL_RESOURCE not a valid port. ET_RESOURCE_DEP another core is actively changing the port.
Returns	error_none (or exception type if policy is XCORE_C_NO_EXCEPTION).

Function	**port_in**
Description	Input a value from a port. For unbuffered port with no trigger, the data will be whatever is on the input pins. For unbuffered port with a trigger, the data will be the value read when the trigger fired. The call will pause if the trigger has not yet fired. For buffered port, this function will pause until the buffer is filled up with deserialised data.
Type	`xcore_c_error_t port_in(port p, uint32_t *data)`
Parameters	p — Port to input from data — The inputted data ET_ILLEGAL_RESOURCE not a valid port. ET_RESOURCE_DEP another core is actively changing the port. ET_LOAD_STORE invalid **data* argument.
Returns	error_none (or exception type if policy is XCORE_C_NO_EXCEPTION).

Function	**port_out_shift_right**
Description	Outputs a value onto a port and shift the output data. In the case of an unbuffered port, the value will be driven on the pins on the next clock cycle. In the case of a buffered port, the data will be stored in the buffer, and be serialised onto the output pins. If there is a time trigger setup and the port is unbuffered or the buffer is full the call will pause until the specified time.

Continued on next page

Type	`xcore_c_error_t port_out_shift_right(port p, uint32_t *data)`
Parameters	p — Port to output to data — data is shifted right by the transfer width of the port, with the bits shifting out onto the port. The remaining shifted bits are returned in data. ET_ILLEGAL_RESOURCE — not a valid port. ET_RESOURCE_DEP — another core is actively changing the port. ET_LOAD_STORE — invalid **data* argument.
Returns	error_none (or exception type if policy is XCORE_C_NO_EXCEPTION).

Function	**port_in_shift_right**
Description	Input a value from a port and shift the data. For unbuffered port with no trigger, the data will be whatever is on the input pins. For unbuffered port with a trigger, the data will be the value read when the trigger fired. The call will pause if the trigger has not yet fired. For buffered port, this function will pause until the buffer is filled up with deserialised data.
Type	`xcore_c_error_t port_in_shift_right(port p, uint32_t *data)`
Parameters	p — Port to input from data — The input data shifted right by the transfer width of the port ET_ILLEGAL_RESOURCE — not a valid port. ET_RESOURCE_DEP — another core is actively changing the port. ET_LOAD_STORE — invalid **data* argument.
Returns	error_none (or exception type if policy is XCORE_C_NO_EXCEPTION).

Function	port_out_at_time
Description	Outputs a value onto a port at a specified port timestamp. In the case of an unbuffered port, the value will be driven on the pins when on the clock cycle that moves the port timestamp to the specified time. In the case of a buffered port, the data will be stored in the buffer, and be serialised onto the output pins at the point that the time is reached.
Type	`xcore_c_error_t port_out_at_time(port p, int16_t t, uint32_t data)`
Parameters	p — Port to output to t — The timestamp to do the output on data — Value to output ET_ILLEGAL_RESOURCE — not a valid port. ET_RESOURCE_DEP — another core is actively changing the port.
Returns	error_none (or exception type if policy is XCORE_C_NO_EXCEPTION).

Function	port_in_at_time
Description	Input data from a port when its counter is at a specific time. In the case of an unbuffered port, the data will be inputted when the counter reaches time `t`. In the case of a buffered port, an input will wait until the given time and then will start capturing data, returning a value when the buffer is full.
Type	`xcore_c_error_t port_in_at_time(port p, int16_t t, uint32_t *data)`
Parameters	p — Port to input from t — The timestamp to do input on data — The inputted data ET_ILLEGAL_RESOURCE — not a valid port. ET_RESOURCE_DEP — another core is actively changing the port. ET_LOAD_STORE — invalid **data* argument.

Continued on next page

Returns	error_none (or exception type if policy is XCORE_C_NO_EXCEPTION).

Function	**port_out_shift_right_at_time**
Description	Outputs a value onto a port at a specified time and shifts the output data. In the case of an unbuffered port, the value will be driven on the pins when on the clock cycle that moves the port counter to the specified time. In the case of a buffered port, the data will be stored in the buffer, and be serialised onto the output pins at the point that the time is reached.

Type	`xcore_c_error_t port_out_shift_right_at_time(port p,`
	`                                               int16_t t,`
	`                                               uint32_t *data)`

Parameters	p — Port to output to t — The timestamp of the output data — data is shifted right by the transfer width of the port, with the bits shifting out onto the port. The remaining shifted bits are returned in data. ET_ILLEGAL_RESOURCE — not a valid port. ET_RESOURCE_DEP — another core is actively changing the port. ET_LOAD_STORE — invalid **data* argument.
Returns	error_none (or exception type if policy is XCORE_C_NO_EXCEPTION).

Function	**port_in_shift_right_at_time**
Description	Input data from a port at a specific time and shift the data. In the case of an unbuffered port, the data will be inputted when the counter reaches time `t`. In the case of a buffered port, an input will wait until the given time and then will start capturing data, returning a value when the buffer is full.

Type	`xcore_c_error_t port_in_shift_right_at_time(port p,`
	`                                              int16_t t,`
	`                                              uint32_t *data)`

Continued on next page

Parameters	`p` Port to input from `t` The timestamp to do input on `data` The input data shifted right by the transfer width of the port ET_ILLEGAL_RESOURCE not a valid port. ET_RESOURCE_DEP another core is actively changing the port. ET_LOAD_STORE invalid **data* argument.
Returns	error_none (or exception type if policy is XCORE_C_NO_EXCEPTION).

Function	**port_in_when_pinseq**
Description	Input data from a port when its pins match a specific value. In the case of an unbuffered port, the data inputted be identical to the value. In the case of a buffered port, an input will wait until the value appears on the pins and then return that value and some previous values that have been deserialised.
Type	`xcore_c_error_t port_in_when_pinseq(port p, port_type_t pt, uint32_t value, uint32_t *data)`
Parameters	`p` Port to input from `pt` If port is buffered or unbuffered. `value` The value to match against the pins `data` The inputted data ET_ILLEGAL_RESOURCE not a valid port. ET_RESOURCE_DEP another core is actively changing the port. ET_LOAD_STORE invalid **data* argument.
Returns	error_none (or exception type if policy is XCORE_C_NO_EXCEPTION).

Function	**port_in_when_pinsneq**
Description	Input data from a port when its pins do not match a specific value. In the case of an unbuffered port, the inputted data will be the non-matching pin values. In the case of a buffered port, this macro will wait until a non matching value appears on the pins, and then return that value and previous values that have been deserialised.
Type	`xcore_c_error_t port_in_when_pinsneq(port p, port_type_t pt, uint32_t value, uint32_t *data)`
Parameters	`p` Port to input from `pt` If port is buffered or unbuffered. `value` The value to match against the pins `data` The inputted data `ET_ILLEGAL_RESOURCE` not a valid port. `ET_RESOURCE_DEP` another core is actively changing the port. `ET_LOAD_STORE` invalid **data* argument.
Returns	error_none (or exception type if policy is XCORE_C_NO_EXCEPTION).

Function	**port_in_shift_right_when_pinseq**
Description	Input data from a port when its pins match a specific value and shift the data. In the case of an unbuffered port, the data inputted be identical to the value. In the case of a buffered port, an input will wait until the value appears on the pins and then return that value and some previous values that have been deserialised.
Type	`xcore_c_error_t port_in_shift_right_when_pinseq(port p, port_type_t pt, uint32_t value, uint32_t *data)`

Continued on next page

Parameters	p	Port to input from
	pt	If port is buffered or unbuffered.
	value	The value to match against the pins
	data	The input data shifted right by the transfer width of the port
	ET_ILLEGAL_RESOURCE	not a valid port.
	ET_RESOURCE_DEP	another core is actively changing the port.
	ET_LOAD_STORE	invalid **data* argument.
Returns	error_none (or exception type if policy is XCORE_C_NO_EXCEPTION).	

Function	**port_in_shift_right_when_pinsneq**
Description	Input data from a port when its pins do not match a specific value and shift the data. In the case of an unbuffered port, the inputted data will be the non-matching pin values. In the case of a buffered port, this macro will wait until a non matching value appears on the pins, and then return that value and previous values that have been deserialised.

Type	`xcore_c_error_t`
	`port_in_shift_right_when_pinsneq(port p,`
	`                                 port_type_t pt,`
	`                                 uint32_t value,`
	`                                 uint32_t *data)`

Continued on next page

Parameters	p — Port to input from pt — If port is buffered or unbuffered. value — The value to match against the pins data — The input data shifted right by the transfer width of the port ET_ILLEGAL_RESOURCE — not a valid port. ET_RESOURCE_DEP — another core is actively changing the port. ET_LOAD_STORE — invalid **data* argument.
Returns	error_none (or exception type if policy is XCORE_C_NO_EXCEPTION).

Function	**port_clear_buffer**
Description	Clears the buffer used by a port. Any data sampled by the port which has not been input by the processor is discarded. Any data output by the processor which has not been driven by the port is discarded. If the port is in the process of serialising output, it is interrupted immediately. If a pending output would have caused a change in direction of the port then that change of direction does not take place. If the port is driving a value on its pins when this function is called then it continues to drive the value until an output statement changes the value driven.
Type	`xcore_c_error_t port_clear_buffer(port p)`
Parameters	p — The port whose buffer is to be cleared ET_ILLEGAL_RESOURCE — not a valid port. ET_RESOURCE_DEP — another core is actively changing the port.
Returns	error_none (or exception type if policy is XCORE_C_NO_EXCEPTION).

Function	**port_endin**

Continued on next page

Description	Ends the current input on a buffered port. The number of bits sampled by the port but not yet input by the processor is returned. This count includes both data in the transfer register and data in the shift register used for deserialisation. Subsequent inputs on the port return transfer-width bits of data until there is less than one transfer-width bits of data remaining. Any remaining data can be read with one further input, which returns transfer-width bits of data with the remaining buffered data in the most significant bits of this value.
Type	`xcore_c_error_t port_endin(port p, size_t *num)`
Parameters	p The port to end the current input on num The number of bits of data remaining ET_ILLEGAL_RESOURCE not a valid port. ET_RESOURCE_DEP another core is actively changing the port. ET_LOAD_STORE invalid **num* argument.
Returns	error_none (or exception type if policy is XCORE_C_NO_EXCEPTION).

Function	**port_force_input**
Description	Force an input on a buffered port. The function will perform an input on a buffered port even if the buffer is only partially full.
Type	`xcore_c_error_t port_force_input(port p, size_t *num, uint32_t *data)`

Continued on next page

Parameters	p — The port to do the input on num — A variable to be filled with number of bits inputted data — The inputted data ET_ILLEGAL_RESOURCE not a valid port. ET_RESOURCE_DEP another core is actively changing the port. ET_LOAD_STORE invalid **num* or **data* argument.
Returns	error_none (or exception type if policy is XCORE_C_NO_EXCEPTION).

Function	**port_setup_select**
Description	Setup select events on a port. Configures a port to trigger select events when ready. By default a port will trigger when there is data available. The trigger event can be changed using the port_set_trigger_*() function. Once the select event is setup you need to call port_enable_trigger() to enable it.
Type	`xcore_c_error_t port_setup_select(port p, uint32_t enum_id)`
Parameters	p — The port to setup the select event on enum_id — The value to be returned by select_wait() et al when the port event is triggered. On XS1 bit 16 must be set (see ENUM_ID_BASE) ET_ILLEGAL_RESOURCE not a valid port. ET_RESOURCE_DEP another core is actively using the port. ET_ECALL — when xassert enabled, on XS1 bit 16 not set in enum_id.
Returns	error_none (or exception type if policy is XCORE_C_NO_EXCEPTION).

Function	**port_setup_select_callback**

Continued on next page

Description	Setup select events on a port where the events are handled by a function. Same as port_setup_select() except that a callback function is used rather than the event being passed back to the select_wait() et al functions. Once the event is setup you need to call port_enable_trigger() to enable it.

Type	`xcore_c_error_t port_setup_select_callback(port p,`
	`void *data,`
	`select_callback_t func)`

Parameters	`p` The port to setup the select event on `data` The value to be passed to the select_callback_t function On XS1 bit 16 must be set (see ENUM_ID_BASE) `func` The select_callback_t function to handle events `ET_ILLEGAL_RESOURCE` not a valid port. `ET_RESOURCE_DEP` another core is actively using the port. `ET_ECALL` when xassert enabled, on XS1 bit 16 not set in data.
Returns	error_none (or exception type if policy is XCORE_C_NO_EXCEPTION).

Function	**port_setup_interrupt_callback**
Description	Setup interrupt event on a port. Once the event is setup you need to call port_enable_trigger() to enable it.

Type	`xcore_c_error_t`
	`port_setup_interrupt_callback(port p,`
	`void *data,`
	`interrupt_callback_t func)`

Continued on next page

Parameters	p — The port to setup the interrupt event on data — The value to be passed to the interrupt_callback_t function On XS1 bit 16 must be set (see ENUM_ID_BASE) func — The interrupt_callback_t function to handle events ET_ILLEGAL_RESOURCE — not a valid port. ET_RESOURCE_DEP — another core is actively using the port. ET_ECALL — when xassert enabled, on XS1 bit 16 not set in data.
Returns	error_none (or exception type if policy is XCORE_C_NO_EXCEPTION).

Function	**port_enable_trigger**
Description	Enable select & interrupt events on a port. Prior to enabling, port_setup_select(), port_setup_select_callback() or port_setup_interrupt_callback() must have been called. Events can be temporarily disabled and re-enabled using port_disable_trigger() and port_enable_trigger(). When the event fires, the value must be read from the port to clear the event.
Type	`xcore_c_error_t port_enable_trigger(port p)`
Parameters	p — The port to enable events on ET_ILLEGAL_RESOURCE — not a valid port. ET_RESOURCE_DEP — another core is actively using the port.
Returns	error_none (or exception type if policy is XCORE_C_NO_EXCEPTION).

Function	**port_disable_trigger**
Description	Disable select & interrupt events for a given port. This function prevents any further events being triggered by a given port.
Type	`xcore_c_error_t port_disable_trigger(port p)`

Continued on next page

Parameters	p — The port to disable events on ET_ILLEGAL_RESOURCE — not a valid port. ET_RESOURCE_DEP — another core is actively using the port.
Returns	error_none (or exception type if policy is XCORE_C_NO_EXCEPTION).

2.9.1 Port protocol helpers

Function	**port_protocol_in_handshake**
Description	Configure a port to be a clocked input port in handshake mode. If the ready-in or ready-out ports are not 1-bit ports, an exception is raised. The ready-out port is asserted on the falling edge of the clock when the port's buffer is not full. The port samples its pins on its sampling edge when both the ready-in and ready-out ports are asserted. By default the port's sampling edge is the rising edge of clock. This can be changed by the function port_set_sample_falling_edge(). *Note*: A handshaken port must be buffered, so this function will also make the port buffered.

Type	`xcore_c_error_t port_protocol_in_handshake(port p,`
	`port readyin,`
	`port readyout,`
	`clock clk)`

Parameters	`p` The port to configure `readyin` A 1-bit port to use for the ready-in signal `readyout` A 1-bit port to use for the ready-out signal `clk` The clock used to configure the port ET_ILLEGAL_RESOURCE not a valid port/clock or clock is running, or readyin/readyout not a one bit port. ET_RESOURCE_DEP another core is actively changing a port/clock
Returns	error_none (or exception type if policy is XCORE_C_NO_EXCEPTION).

Function	**port_protocol_out_handshake**

Continued on next page

Description	configures a port to be a clocked output port in handshake mode. if the ready-in or ready-out ports are not 1-bit ports an exception is raised. the port drives the initial value on its pins until an output statement changes the value driven. the ready-in port is read on the sampling edge of the port. outputs are driven on the next falling edge of the clock where the previous value read from the ready-in port was high. on the falling edge of the clock the ready-out port is driven high if data is output on that edge, otherwise it is driven low. by default the port's sampling edge is the rising edge of clock. this can be changed by the function port_set_sample_falling_edge(). *note*: a handshaken port must be buffered, so this function will also make the port buffered.
Type	`xcore_c_error_t port_protocol_out_handshake(port p, port readyin, port readyout, clock clk, uint32_t initial)`
Parameters	`p` the port to configure `readyin` a 1-bit port to use for the ready-in signal `readyout` a 1-bit port to use for the ready-out signal `clk` the clock used to configure the port `initial` the initial value to output on the port ET_ILLEGAL_RESOURCE not a valid port/clock or clock is running, or readyin/readyout not a one bit port. ET_RESOURCE_DEP another core is actively changing a port/clock
Returns	error_none (or exception type if policy is XCORE_C_NO_EXCEPTION).

Function	**port_protocol_in_strobed_master**
Description	configures a port to be a clocked input port in strobed master mode. if the ready-out port is not a 1-bit port, an exception is raised. the ready-out port is asserted on the falling edge of the clock when the port's buffer is not full. the port samples its pins on its sampling edge after the ready-out port is asserted. by default the port's sampling edge is the rising edge of clock. this can be changed by the function `set_port_sample_delay()`. *note*: a strobed port must be buffered, so this function will also make the port buffered.

Continued on next page

Type	`xcore_c_error_t`
	`port_protocol_in_strobed_master(port p,`
	`                                port readyout,`
	`                                clock clk)`

Parameters	`p` the port to configure `readyout` a 1-bit port to use for the ready-out signal `clk` the clock used to configure the port ET_ILLEGAL_RESOURCE not a valid port/clock or clock is running, or readyout not a one bit port. ET_RESOURCE_DEP another core is actively changing a port/clock
Returns	error_none (or exception type if policy is XCORE_C_NO_EXCEPTION).

Function	**port_protocol_out_strobed_master**
Description	configures a port to be a clocked output port in strobed master mode. if the ready-out port is not a 1-bit port, an exception is raised. the port drives the initial value on its pins until an output statement changes the value driven. outputs are driven on the next falling edge of the clock. on the falling edge of the clock the ready-out port is driven high if data is output on that edge, otherwise it is driven low. *note*: a strobed port must be buffered, so this function will also make the port buffered.

Type	`xcore_c_error_t`
	`port_protocol_out_strobed_master(port p,`
	`                                 port readyout,`
	`                                 clock clk,`
	`                                 uint32_t initial)`

Parameters	`p` the port to configure `readyout` a 1-bit port to use for the ready-out signal `clk` the clock used to configure the port `initial` the initial value to output on the port ET_ILLEGAL_RESOURCE not a valid port/clock or clock is running, or readyout not a one bit port. ET_RESOURCE_DEP another core is actively changing a port/clock

Continued on next page

Returns	error_none (or exception type if policy is XCORE_C_NO_EXCEPTION).

Function	**port_protocol_in_strobed_slave**
Description	configures a port to be a clocked input port in strobed slave mode. if the ready-in port is not a 1-bit port, an exception is raised. the port samples its pins on its sampling edge when the ready-in signal is high. by default the port's sampling edge is the rising edge of clock. this can be changed by the function `set_port_sample_delay()`. *note*: a strobed port must be buffered, so this function will also make the port buffered.

Type	`xcore_c_error_t`
	`port_protocol_in_strobed_slave(port p,`
	`                               port readyin,`
	`                               clock clk)`

Parameters	`p` the port to configure `readyin` a 1-bit port to use for the ready-in signal `clk` the clock used to configure the port `ET_ILLEGAL_RESOURCE` not a valid port/clock or clock is running, or readyin not a one bit port. `ET_RESOURCE_DEP` another core is actively changing a port/clock
Returns	error_none (or exception type if policy is XCORE_C_NO_EXCEPTION).

Function	**port_protocol_out_strobed_slave**
Description	configures a port to be a clocked output port in strobed slave mode. if the ready-in port is not a 1-bit port, an exception is raised. the port drives the initial value on its pins until an output statement changes the value driven. the ready-in port is read on the port's sampling edge. outputs are driven on the next falling edge of the clock where the previous value read from the ready-in port is high. by default the port's sampling edge is the rising edge of clock. this can be changed by the function `set_port_sample_delay()`. *note*: a strobed port must be buffered, so this function will also make the port buffered.

Continued on next page

Type	`xcore_c_error_t`
	`port_protocol_out_strobed_slave(port p,`
	`                                port readyin,`
	`                                clock clk,`
	`                                uint32_t initial)`

Parameters	`p` the port to configure `readyin` a 1-bit port to use for the ready-in signal `clk` the clock used to configure the port `initial` the initial value to output on the port ET_ILLEGAL_RESOURCE not a valid port/clock or clock is running, or readyin not a one bit port. ET_RESOURCE_DEP another core is actively changing a port/clock

2.10 Timers

Function	**hwtimer_free_xc_timer**
Description	Deallocate the xC timer resource for a thread. This function deallcoates the hardware timer automatically allocated for xC use. Each logical core is allocated a hardware timer that is multiplexed and used by the xC 'timer' interface. This multiplexed timer is not accessible from C. If the logical core is not running any xC code, or any xC code is not making use of the 'timer' resource type, the allocated hardware timer may be retrieved for use as a hwtimer_t. **This call must be paired with a call to :c:func:'hwtimer_realloc_xc_timer' prior to the logical core completing its tasks** **The xScope link also requires a hardware timer**
Type	`xcore_c_error_t hwtimer_free_xc_timer(void)`
Parameters	ET_ILLEGAL_RESOURCE timer has already been deallocated.
Returns	error_none (or exception type if policy is XCORE_C_NO_EXCEPTION).

Function	**hwtimer_realloc_xc_timer**
Description	Reallocate the xC timer resource for a thread. This function reallcoates a logical core's xC hardware timer that was deallocated by a call to hwtimer_free_xc_timer(). **There must be an available hw timer when this call is made, otherwise an exception will be raised when the logical core completes**
Type	`xcore_c_error_t hwtimer_realloc_xc_timer(void)`
Parameters	ET_ECALL no available hw timer, reallocation failed.
Returns	error_none.

Function	**hwtimer_alloc**
Description	Allocates and initialise a timer. This function allocates a hardware timer. If there are no timers available, then the function will return 0. This macro is to be called once on every variable of the type `timer`. When the timer is no longer required, hwtimer_free() must be called to deallocate it.
Type	`xcore_c_error_t hwtimer_alloc(hwtimer_t *t)`

Continued on next page

Parameters	t Timer variable representing the initialised timer ET_LOAD_STORE invalid *t argument.
Returns	error_none (or exception type if policy is XCORE_C_NO_EXCEPTION).

Function	**hwtimer_free**
Description	Deallocate a timer. This function frees the hardware timer.
Type	`xcore_c_error_t hwtimer_free(hwtimer_t *t)`
Parameters	t The timer to be freed ET_ILLEGAL_RESOURCE not an allocated timer. ET_RESOURCE_DEP another core is actively using the timer. ET_LOAD_STORE invalid *t argument.
Returns	error_none (or exception type if policy is XCORE_C_NO_EXCEPTION).

Function	**hwtimer_get_time**
Description	Get the current time from the timer. If there is a trigger time setup, the call will stall until after the trigger time. For select and interrupt event, calling hwtimer_get_time() will clear the event.
Type	`xcore_c_error_t hwtimer_get_time(hwtimer_t t, uint32_t *now)`

Continued on next page

Parameters	`t` The timer on which to input `now` The time value (a 32-bit value) `ET_ILLEGAL_RESOURCE` not an allocated timer. `ET_RESOURCE_DEP` another core is actively using the timer. `ET_LOAD_STORE` invalid **now* argument.
Returns	error_none (or exception type if policy is XCORE_C_NO_EXCEPTION).

Function	**hwtimer_set_trigger_time**
Description	Setup an event trigger on a timer. This will cause hwtimer_get_time() to pause until the specified time. The trigger may be cleared using hwtimer_clear_trigger_time(). **:c:func:'hwtimer_wait_until', :c:func:'hwtimer_delay', :c:func:'hwtimer_setup_select' :c:func:'hwtimer_setup_select_callback' and :c:func:'hwtimer_setup_interrupt_callback' call :c:func:'hwtimer_set_trigger_time'**

Type	`xcore_c_error_t`
	`hwtimer_set_trigger_time(hwtimer_t t,`
	`                         uint32_t time)`

Parameters	`t` The timer to setup a event trigger on. `time` The time at which the timer will trigger an event. The default timer ticks are at a 10ns resolution. `ET_ILLEGAL_RESOURCE` not a valid timer. `ET_RESOURCE_DEP` another core is actively using the timer.
Returns	error_none (or exception type if policy is XCORE_C_NO_EXCEPTION).

Function	**hwtimer_change_trigger_time**
Description	Change the time at which a timer trigger will fire. This function modifies the time at which a previously setup triggers fire. It is used to set a new trigger time after a select or interrupt event has occurred.

Continued on next page

Type	`xcore_c_error_t`
	`hwtimer_change_trigger_time(hwtimer_t t,`
	`                            uint32_t time)`

Parameters	`t` The timer to change
	
	`time` The time at which the timer will trigger an event. The default timer ticks are at a 10ns resolution.
	
	`ET_ILLEGAL_RESOURCE`
	not a valid timer.
	
	`ET_RESOURCE_DEP`
	another core is actively using the timer.

Returns	error_none (or exception type if policy is XCORE_C_NO_EXCEPTION).

Function	**hwtimer_clear_trigger_time**
Description	Clear an event trigger on a timer. Makes sure no triggers are setup on a timer. Should be called when a timer is no longer being used for select and interrupt events. Both hwtimer_wait_until() and hwtimer_delay() call hwtimer_clear_trigger_time().

Type	`xcore_c_error_t`
	`hwtimer_clear_trigger_time(hwtimer_t t)`

Parameters	`t` The timer to tear down events on
	
	`ET_ILLEGAL_RESOURCE`
	not a valid timer.
	
	`ET_RESOURCE_DEP`
	another core is actively using the timer.

Returns	error_none (or exception type if policy is XCORE_C_NO_EXCEPTION).

Function	**hwtimer_wait_until**
Description	Wait until after a specified time. **This will destroy any select or interrupt event triggers set on this resource**

Type	`xcore_c_error_t hwtimer_wait_until(hwtimer_t t,`
	`                                   uint32_t until,`
	`                                   uint32_t *now)`

Continued on next page

Parameters	`t` The timer to use for timing `until` The time to wait until `now` The time we actually waited until ET_ILLEGAL_RESOURCE not an allocated timer. ET_RESOURCE_DEP another core is actively using the timer. ET_LOAD_STORE invalid **now* argument.
Returns	error_none (or exception type if policy is XCORE_C_NO_EXCEPTION).

Function	**hwtimer_delay**
Description	Delay for a specified time using a specific timer. **This will destroy any select or interrupt event triggers set on this resource**
Type	`xcore_c_error_t hwtimer_delay(hwtimer_t t, uint32_t period)`
Parameters	`t` The timer resource to use `period` The amount of time to wait (in reference time ticks, usually 10ns steps) ET_ILLEGAL_RESOURCE not an allocated timer. ET_RESOURCE_DEP another core is actively using the timer. ET_LOAD_STORE invalid **now* argument.
Returns	error_none (or exception type if policy is XCORE_C_NO_EXCEPTION).

Function	**hwtimer_setup_select**
Description	Setup select events on a timer. Configures a timer to trigger select events when the timer has reached the specified time. It is used in combination with select_wait() et al functions, returning the enum_id when the event is triggered. Once the select event is setup you need to call hwtimer_enable_trigger() to enable it.

Continued on next page

Type	`xcore_c_error_t hwtimer_setup_select(hwtimer_t t, uint32_t time, uint32_t enum_id)`
Parameters	`t` The timer to setup the select event on `time` The time at which the timer will trigger an event. The default timer ticks are at a 10ns resolution. `enum_id` The value to be returned by select_wait() et al when the timer event is triggered. ET_ILLEGAL_RESOURCE not a valid timer. ET_RESOURCE_DEP another core is actively using the timer. ET_ECALL when xassert enabled, on XS1 bit 16 not set in enum_id.
Returns	error_none (or exception type if policy is XCORE_C_NO_EXCEPTION).

Function	**hwtimer_setup_select_callback**
Description	Setup select event on a timer where the events are handled by a function. Same as hwtimer_setup_select() except that a callback function is used rather than the event being passed back to the select_wait() et al functions. Once the event is setup you need to call hwtimer_enable_trigger() to enable it.
Type	`xcore_c_error_t hwtimer_setup_select_callback(hwtimer_t t, uint32_t time, void *data, select_callback_t func)`

Continued on next page

Parameters	`t` The timer to setup the select event on `time` The time at which the timer will trigger an event. The default timer ticks are at a 10ns resolution. `data` The value to be passed to the select_callback_t function `func` The select_callback_t function to handle the event `ET_ILLEGAL_RESOURCE` not a valid timer. `ET_RESOURCE_DEP` another core is actively using the timer. `ET_ECALL` when xassert enabled, on XS1 bit 16 not set in enum_id.
Returns	error_none (or exception type if policy is XCORE_C_NO_EXCEPTION).

Function	**hwtimer_setup_interrupt_callback**
Description	Setup interrupt event on a timer. Once the event is setup you need to call hwtimer_enable_trigger() to enable it.

Type	`xcore_c_error_t`
	`hwtimer_setup_interrupt_callback(hwtimer_t t, uint32_t time, void *data, interrupt_callback_t func)`

Parameters	`t` The timer to setup the events on `time` The time at which the timer will trigger an event. The default timer ticks are at a 10ns resolution. `data` The value to be passed to the interrupt_callback_t function `func` The interrupt_callback_t function to handle the events `ET_ILLEGAL_RESOURCE` not a valid timer. `ET_RESOURCE_DEP` another core is actively using the timer. `ET_ECALL` when xassert enabled, on XS1 bit 16 not set in enum_id.
Returns	error_none (or exception type if policy is XCORE_C_NO_EXCEPTION).

Function	hwtimer_enable_trigger
Description	Enable select & interrupt events on a timer. Prior to enabling, hwtimer_setup_select(), hwtimer_setup_select_callback() or hwtimer_setup_interrupt_callback() must have been called. Events can be temporarily disabled and re-enabled using hwtimer_disable_trigger() and hwtimer_enable_trigger(). When the event fires, hwtimer_get_time() must be called to clear the event. The time of the next event is set using hwtimer_change_trigger_time().
Type	`xcore_c_error_t hwtimer_enable_trigger(hwtimer_t t)`
Parameters	t The timer to enable events on ET_ILLEGAL_RESOURCE not a valid timer. ET_RESOURCE_DEP another core is actively using the timer.
Returns	error_none (or exception type if policy is XCORE_C_NO_EXCEPTION).

Function	hwtimer_disable_trigger
Description	Disable select & interrupt events for a given timer. This function prevents any further select or interrupt events being triggered by a given timer. **This does not clear the trigger setup**
Type	`xcore_c_error_t hwtimer_disable_trigger(hwtimer_t t)`
Parameters	t The timer to disable events on ET_ILLEGAL_RESOURCE not a valid timer. ET_RESOURCE_DEP another core is actively using the timer.
Returns	error_none (or exception type if policy is XCORE_C_NO_EXCEPTION).

2.11 Select events

Macro	ENUM_ID_BASE
Description	Starting value to use for the enum_id. The enum_id is passed to the *res*_setup_select() and returned by select_wait() et al On XS1 the environment vectors (EVs) are only 16-bit and bit 16 will be set to 1 as it is expected to be used as a memory address.

Function	select_disable_trigger_all
Description	Disable all select events on this logical core. This function is called before starting to configure select events for a new event loop. This will ensure that no events set up by other code will be triggered This affect events setup using *res*_setup_select() and *res*_setup_select_callback() but not *res*_setup_interrupt_callback()
Type	`xcore_c_error_t select_disable_trigger_all(void)`
Returns	error_none

Function	select_wait
Description	Wait for a select event to trigger. This function waits for an event to trigger and then returns the value the user has registered with the resource that triggered the event.
Type	`uint32_t select_wait(void)`
Returns	The enum_id registered with the resource when events were enabled

Function	select_no_wait
Description	Check whether any select events have triggered, otherwise return. This function tests for an event to being ready. If there is one ready then it returns the enum_id the user has registered with the resource that triggered the event. If no events are ready then returns the no_wait_id passed in by the user. **select_callback_t events are handled, but are not considered 'select events'**
Type	`uint32_t select_no_wait(uint32_t no_wait_id)`
Parameters	`no_wait_id` The enum_id to return if no 'select event' is triggered

Continued on next page

Returns	The enum_id registered with the resource which triggered an event or the no_wait_id passed in if no event fired

Function	**select_wait_ordered**
Description	Wait for an select event from a list of resources using an ordered enable sequence. • Starts by clearing all select events that have been configured for this core. This includes select_callback_t functions but not interrupt_callback_t functions. • Enables select events on each resource in turn so that there is a defined order in which pending events will be taken This function: **Enabled select_callback_t resources will be taken, but will not terminate the process. A user may wish to place these at the front of the list**

Type	`uint32_t`
	`select_wait_ordered(const resource_t ids[])`

Parameters	`ids` Null-terminated list of resources to enable events on ET_LOAD_STORE invalid *ids[]* argument.
Returns	The enum_id registered with the resource which triggers an event

Function	**select_no_wait_ordered**
Description	Wait for a select event from a list of resources using an ordered enable sequence. This function does the same as select_wait_ordered, but will return the no_wait_id if no select event fires by the end of the enabling sequence. **select_callback_t events are handled, but are not considered 'select events'**

Type	`uint32_t`
	`select_no_wait_ordered(uint32_t no_wait_id,`
	`const resource_t ids[])`

Parameters	`no_wait_id` The enum_id to return if no 'select event' is triggered `ids` Null-terminated list of resources to enable events on ET_LOAD_STORE invalid *ids[]* argument.
Returns	The enum_id registered with the resource which triggered an event or the no_wait_id passed in if no event fired

Macro	**DEFINE_SELECT_CALLBACK**

Description	Define a select callback handling function.
	• An ordinary function that may be called directly Its signature will be 'void *callback* (void* *data*)'
	• An select_callback_t function for passing to the res_setup_select_callback functions The select_callback_t function name is accessed using the SELECT_CALLBACK() macro
	This macro will define two functions for you:
	Example usage:
	`DEFINE_SELECT_CALLBACK(myfunc, arg)`
	`{`
	`  // This is the body of 'void myfunc(void* arg)'`
	`}`

Parameters	`callback` this is the name of the ordinary function `data` the name to use for the void* argument

Macro	**DECLARE_SELECT_CALLBACK**

Description	Declare a select callback handling function.
	Use this macro when you require a declaration of your select callback function types
	Example usage:
	`DECLARE_SELECT_CALLBACK(myfunc, arg);`
	`chanend_setup_select_callback(c, 0 , SELECT_CALLBACK(myfunc));`

Parameters	`callback` this is the name of the ordinary function `data` the name to use for the void* argument

Macro	**SELECT_CALLBACK**
Description	The name of the defined 'select_callback_t' function. Use this macro for retriving the name of the declared select callback function. This is the name that is passed to *res*`_setup_select_callback()` for registration.
Returns	the name of the defined select_callback_t function

2.12 Interrupt events

Macro	**XCORE_C_KSTACK_WORDS**
Description	Specify the minimum kernel stack size the interrupt permitting function should create. The user may specify a minimum kstack size by setting the XCORE_C_KSTACK_WORDS define in their Makefile. This should be done when the kstack is being used by more than interrupt_callback_t functions.

Macro	**DEFINE_INTERRUPT_PERMITTED**

Description	Define a function that allows interrpts to occur within its scope.
	• An ordinary function that may be called directly Its signature will be '*ret root_function* (*...*)'
	• A function that will also reserve space for and set up a stack for handling interrupts. The function name is accessed using the INTERRUPT_PERMITTED() macro
	This macro will define two functions for you:
	You would normally use this macro on the definition of the root function which will be called in a par statement. The interrupt stack (kernel stack) is created on the core's stack with the ksp and sp being modified as necessary. When the functions exits, neither the kernel stack nor ksp is valid.
	The kernel stack allocated has enough space for the interrupt_callback_t function (+callees) in the given 'group'. The use of the 'group' identifier allows a kernel stack to be no larger than that required by its greediest member.
	The kernel stack is not re-entrant so kernel mode must not be masked from within an interrupt_callback_t
	The user may specify a larger kernel stack by defining XCORE_C_KSTACK_WORDS.
	Example usage:
	`DEFINE_INTERRUPT_PERMITTED(groupA, int, rootfunc, chanend c, int i)`
	`{`
	`  // This is the body of 'int rootfunc(chanend c, int i)'`
	`}`

Parameters	`group` — this is the group of interrupt_callback_t function that may be safely enabled - see DEFINE_INTERRUPT_CALLBACK() `ret` — the return type of the ordinary function `root_function` — the name of the ordinary function `...` — the arguments of the ordinary function

Macro	**DECLARE_INTERRUPT_PERMITTED**

Continued on next page

Description	Declare an interrupt permitting function.
	Use this macro when you require a declaration of your interrupt permitting function types
	Example usage:
	`// In another file:`
	`//   DEFINE_INTERRUPT_PERMITTED(groupA, int, rootfunc, chanend c, int i)`
	`//   DEFINE_INTERRUPT_PERMITTED(groupB, void, anotherfunc, void)`
	`DECLARE_INTERRUPT_PERMITTED(int, rootfunc, chanend c, int i);`
	`DECLARE_INTERRUPT_PERMITTED(void, anotherfunc, void);`
	`...`
	`  par {`
	`      int ret = INTERRUPT_PERMITTED(rootfunc)(c,i);  // kstack for groupA.`
	`      INTERRUPT_PERMITTED(anotherfunc)();  // kstack for groupB.`

Parameters	`ret` the return type of the ordinary function `root_function` the name of the ordinary function `...` the arguments of the ordinary function

Macro	**INTERRUPT_PERMITTED**
Description	The name of the defined interrupt permitting function. Use this macro for retriving the name of the declared interrupt function. This is the name used to invoke the function.
Returns	the name of the defined interrupt permitting function

Function	**interrupt_mask_all**
Description	Mask all interrupts on this logical core. Prevent any enabled *res*_`setup_interrupt_callback()` functions from triggering. This has no effect on *res*_`setup_select_callback()` triggering. They can be restored by using interrupt_unmask_all().
Type	`xcore_c_error_t interrupt_mask_all(void)`
Returns	error_none

Function	**interrupt_unmask_all**
Description	Unmask all interrupts on this logical core. Allow any *res*_`setup_interrupt_callback()` functions to trigger. They can be suppressed by using interrupt_mask_all().

Continued on next page

Type	`xcore_c_error_t interrupt_unmask_all(void)`
Returns	error_none

Macro	**DEFINE_INTERRUPT_CALLBACK**

Description	Define an interrupt handling function.
	• An ordinary function that may be called directly Its signature will be 'void *intrpt* (void* *data*)'
	• An interrupt_callback_t function for passing to a res_setup_interrupt_callback function. The interrupt_callback_t function name is accessed using the INTERRUPT_CALLBACK() macro
	This macro will define two functions for you:
	The kernel stack is not re-entrant so kernel mode must not be masked from within an interrupt_callback_t
	Example usage:
	`DEFINE_INTERRUPT_CALLBACK(groupA, myfunc, arg)`
	`{`
	`  // This is the body of 'void myfunc(void* arg)'`
	`}`

Parameters	`group` — the group of interrupt_callback_t function we belong to see DEFINE_INTERRUPT_PERMITTED() `intrpt` — this is the name of the ordinary function `data` — the name to use for the void* argument

Macro	**DECLARE_INTERRUPT_CALLBACK**

Description	Declare an interrupt handling function.
	Use this macro when you require a declaration of your interrupt function types
	Example usage:
	`DECLARE_INTERRUPT_CALLBACK(myfunc, arg);`
	`chanend_setup_interrupt_callback(c, 0 , INTERRUPT_CALLBACK(myfunc));`

Parameters	`intrpt` — this is the name of the ordinary function `data` — the name to use for the void* argument

Macro	**INTERRUPT_CALLBACK**
Description	The name of the defined 'interrupt_callback_t' function. Use this macro for retriving the name of the declared interrupt callback function. This is the name that is passed to *res*_`setup_interrupt_callback()` for registration.

Continued on next page

Returns	the name of the defined interrupt_callback_t function

APPENDIX A - Known Issues

No known issues.

APPENDIX B - lib_xcore_c change log

B.1 2.0.0

- Alteration & extension of the API
- Changes to dependencies:
  - lib_trycatch: Added dependency 1.0.0
  - lib_xassert: Added dependency 2.0.1

B.2 1.0.1

- EVENT_DEFAULT change to EVENT_NONE

B.3 1.0.0

- Initial release