C for xCore

Contents

Intended Audience

This presentation is intended to help current XC users transition to a new C-based approach to xCore development. Familiarity with the following is assumed:

- The XC and C programming languages;
- The hardware features of xCore devices

Why C?

- Less restrictive than XC - does not attempt to restrict usage patterns;
- Using C/C++ means there is no ambiguity in language semantics;
- Faster "onboarding" of new developers; there is less requirement for specialist skills;
- Leveraging of best-in-class compiler technology and tools;
- Google/StackOverflow support!

lib_xcore

lib_xcore

- Provides a low-level C API for all xCore features;
- XC-like constructs implemented as preprocessor macros to approximate `select` and `par`;
- Also exposes some features not available using XC (e.g. locks, interrupts);
- Additional headers for compatibility with legacy XC code (e.g. channels);

Ports

XC

```
#include <platform.h>

port my_port = XS1_PORT_1J;

void port_user(port p)
{
}

int main(void)

{
  port_user(my_port);
}
```

XC

```
#include <platform.h>

port my_port = XS1_PORT_1J;

void port_user(port p)
{
}

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

C

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

void port_user(port_t p)
{
}

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

XC

```
#include <platform.h>

port my_port = XS1_PORT_1J;

void port_user(port p)
{
}

int main(void)
{
```

C

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

void port_user(port_t p)
{
}

int main(void)
{
  port_t my_port = XS1_PORT_1J;
```

lib_xcore is a system library split into several headers; their names all begin with 'xcore/' and they broadly each correspond to a single resource or pattern

XC

```
#include <platform.h>

port my_port = XS1_PORT_1J;

void port_user(port p)
{
}

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

C

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

void port_user(port_t p)
{
}

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

In C, resources don't get any special treatment - instead we have resource handles which are just ordinary variables with no additional restrictions

```c
#include <platform.h>

port my_port = XS1_PORT_1J;

void port_user(port p)
{
}

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

In XC this line...

- Has special `port` type with special semantics;
- Can only appear in certain places;
- Automagically enables the physical Port resource for us

In C this line...

- Has type `port_t` which is a normal scalar type;
- Can appear anywhere any other variable can be declared;
- Doesn't have any side effects

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

void port_user(port_t p)
{
}

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

The user must...

- Enable the resource before first use;
- Disable it when it's no longer needed;
- Make sure any sharing between threads is safe

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

void port_user(port_t p)
{
}

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

XC

```
#include <platform.h>

port my_port = XS1_PORT_1J;

void port_user(port p)
{
}

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

C

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

void port_user(port_t p)
{
}

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

Of course, we haven't actually done anything useful with the port yet. Let's add some I/O...

XC

```
#include <platform.h>

port my_port = XS1_PORT_1J;

void port_user(port p)
{
  int a;
  p :> a;
}


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

C

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

void port_user(port_t p)
{
}

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

In C we no longer have XC's special input and output operators - instead we use functions provided by lib_xcore

XC

```
#include <platform.h>

port my_port = XS1_PORT_1J;

void port_user(port p)
{
  int a;
  p :> a;
}

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

C

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

void port_user(port_t p)
{
  int a = port_in(p);
}

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

The equivalent to ':>' for a port would be 'port_in'

XC

```
#include <platform.h>

port my_port = XS1_PORT_1J,
     my_output_port = XS1_PORT_32A;

void port_user(port p, port p_out)
{
  int a;
  p :> a;
  p_out <: a;
}

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

C

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

void port_user(port_t p, port_t p_out)
{
  int a = port_in(p);
  port_out(p_out, a);
}

int main(void)
{
  port_t my_port = XS1_PORT_1J,
         my_output_port = XS1_PORT_32A;
  port_enable(my_port);
  port_enable(my_output_port);
  port_user(my_port, my_output_port);
```

Output is similar, this time we use 'port_out'

```
#include <platform.h>

port my_port = XS1_PORT_1J,
     my_output_port = XS1_PORT_32A;

void port_user(port p, port p_out)
{
  int a;
  p :> a;

  p_out <: a;
}

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

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

void port_user(port_t p, port_t p_out)
{
  int a = port_in(p);
  port_out(p_out, a);
}

int main(void)
{
  port_t my_port = XS1_PORT_1J,
         my_output_port = XS1_PORT_32A;
  port_enable(my_port);
  port_enable(my_output_port);
  port_user(my_port, my_output_port);
  port_disable(my_port);
  port_disable(my_output_port);
}
```

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

void port_user(port_t p, port_t p_out)
{
  port_set_invert(p);
  port_set_trigger_in_not_equal(p, port_peek(p));

  int a = port_in(p); //Now blocking
  port_out(p_out, a);
}
```

All port functionality is exposed by lib_xcore - check the port.h documentation for the full API

With great power...

- C doesn't help with (or even know about) resources' internal states;
- Applications must ensure that resources are in a suitable state for use

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

void port_user(port_t p, port_t p_out)
{
  port_set_invert(p);
  port_set_trigger_in_not_equal(p, port_peek(p));

  int a = port_in(p); //Now blocking

  port_out(p_out, a);
}

int main(void)
{
  port_t my_port = XS1_PORT_1J,
         my_output_port = XS1_PORT_32A;
  port_enable(my_port);
  port_enable(my_output_port);
  port_user(my_port, my_output_port);
  port_disable(my_port);
```

In the previous example our function set a trigger condition on a port.

If we pass that port somewhere else, this trigger might be unexpected.

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

void port_user(port_t p, port_t p_out)
{
  port_set_invert(p);
  port_set_trigger_in_not_equal(p, port_peek(p));

  int a = port_in(p); //Now blocking

  port_out(p_out, a);
}

void port_user2(port_t p);

int main(void)
{
  port_t my_port = XS1_PORT_1J,
         my_output_port = XS1_PORT_32A;
  port_enable(my_port);
  port_enable(my_output_port);
  port_user(my_port, my_output_port);
  port_user2(my_port); //!!! May not expect port in to block
  port_disable(my_port);
  port_disable(my_output_port);
}
```

In this case, a call to `port_reset` might be suitable.

But perhaps our function should have cleared the trigger when it was done with it...

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

void port_user(port_t p, port_t p_out)
{
  port_set_invert(p);
  port_set_trigger_in_not_equal(p, port_peek(p));

  int a = port_in(p); //Now blocking

  port_out(p_out, a);
}

void port_user2(port_t p);

int main(void)
{
  port_t my_port = XS1_PORT_1J,
         my_output_port = XS1_PORT_32A;
  port_enable(my_port);
  port_enable(my_output_port);
  port_user(my_port, my_output_port);
  port_reset(my_port);
  port_user2(my_port); //!!! May not expect port in to block
  port_disable(my_port);
  port_disable(my_output_port);
}
```

Timers

xc

```
#include <platform.h>


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

XC

```
#include <platform.h>


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

C

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


int main(void)
{
  hwtimer_t t = hwtimer_alloc();
  hwtimer_free(t);
}
```

XC

```
#include <platform.h>

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

C

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

int main(void)
{
  hwtimer_t t = hwtimer_alloc();
  hwtimer_free(t);
}
```

Timer functions come from the 'hwtimer.h' system header under 'xcore'

XC

```
#include <platform.h>

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

C

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

int main(void)
{
  hwtimer_t t = hwtimer_alloc();
  hwtimer_free(t);
}
```

As with ports (and all resources) there's no special type in C, so we have to use lib_xcore to get a timer handle

XC

```
#include <platform.h>

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

C

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

int main(void)
{
  hwtimer_t t = hwtimer_alloc();
  hwtimer_free(t);
}
```

Unlike ports, timers are allocated from a pool by the xCore - so we don't specify a particular timer to use

XC

```
#include <platform.h>

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

C

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

int main(void)
{
  hwtimer_t t = hwtimer_alloc();
  hwtimer_free(t);
}
```

It's also our responsibility to free the timer once we're done with it so that it can be returned to the pool

XC

```
#include <platform.h>

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

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

C

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

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

  unsigned v = hwtimer_get_time(t);

  hwtimer_free(t);
}
```

We can read the value of the timer using 'hwtimer_get_time'

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

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

  unsigned v = hwtimer_get_time(t);

  hwtimer_free(t);
}
```

Again, lib_xcore gives you unrestricted control over the timer resource

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

  unsigned v = hwtimer_get_time(t);
  hwtimer_set_trigger_time(t, v + 100000000);
  v = hwtimer_get_time(t); // Now blocking

  hwtimer_change_trigger_time(t, hwtimer_get_trigger_time(t) + 100000000);
  v = hwtimer_get_time(t); // Now blocking

  hwtimer_wait_until(t, v + 100000000);

  hwtimer_delay(t, 100000000);

  hwtimer_free(t);
```

Check the timer.h documentation for the full list of available timer functions

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

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

  unsigned v = hwtimer_get_time(t);
  hwtimer_set_trigger_time(t, v + 100000000);
  v = hwtimer_get_time(t); // Now blocking

  hwtimer_change_trigger_time(t, hwtimer_get_trigger_time(t) + 100000000);
  v = hwtimer_get_time(t); // Now blocking

  hwtimer_wait_until(t, v + 100000000);
```

You can set a trigger so that reading the timer blocks; timer conditions can only be on reaching a certain time

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


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


  unsigned v = hwtimer_get_time(t);
  hwtimer_set_trigger_time(t, v + 100000000);
  v = hwtimer_get_time(t); // Now blocking


  hwtimer_change_trigger_time(t, hwtimer_get_trigger_time(t) + 100000000);
  v = hwtimer_get_time(t); // Now blocking


  hwtimer_wait_until(t, v + 100000000);


  hwtimer_delay(t, 100000000);


  hwtimer_free(t);
```

You can also read and modify the trigger time of a timer which already has a trigger set

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


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


  unsigned v = hwtimer_get_time(t);
  hwtimer_set_trigger_time(t, v + 100000000);
  v = hwtimer_get_time(t); // Now blocking


  hwtimer_change_trigger_time(t, hwtimer_get_trigger_time(t) + 100000000);
  v = hwtimer_get_time(t); // Now blocking


  hwtimer_wait_until(t, v + 100000000);


  hwtimer_delay(t, 100000000);
```

'hwtimer_change_trigger_time' sets the trigger time without having to re-set the trigger type

```
{
  hwtimer_t t = hwtimer_alloc();

  unsigned v = hwtimer_get_time(t);
  hwtimer_set_trigger_time(t, v + 100000000);
  v = hwtimer_get_time(t); // Now blocking

  hwtimer_change_trigger_time(t, hwtimer_get_trigger_time(t) + 100000000);
  v = hwtimer_get_time(t); // Now blocking

  hwtimer_wait_until(t, v + 100000000);

  hwtimer_delay(t, 100000000);

  hwtimer_free(t);
}
```

You can also wait until a given timestamp

```
    unsigned v = hwtimer_get_time(t);
    hwtimer_set_trigger_time(t, v + 100000000);
    v = hwtimer_get_time(t); // Now blocking

    hwtimer_change_trigger_time(t, hwtimer_get_trigger_time(t) + 100000000);
    v = hwtimer_get_time(t); // Now blocking

    hwtimer_wait_until(t, v + 100000000);

    hwtimer_delay(t, 100000000);

    hwtimer_free(t);
}
```

Or wait for a given number of ticks

```
#include <platform.h>


int main()

{

  [[hwtimer]]

  timer t;


  unsigned v;

  t :> v;


}
```

Thread-local Timers

- In XC using a `timer` doesn't guarantee you exclusive use of a hardware timer;
- Each thread has a 'thread local' timer which backs all normal `timers` in that thread;
- This introduces some overhead but makes it harder to run out of hardware timers

```
#include <platform.h>


int main()

{

  [[hwtimer]]

  timer t;



  unsigned v;

  t :> v;


}
```

Thread-local Timers

- In XC using a `timer` doesn't guarantee you exclusive use of a hardware timer;
- Each thread has a 'thread local' timer which backs all normal `timers` in that thread;
- This introduces some overhead but makes it harder to run out of hardware timers;
- The `[[hwtimer]]` attribute is used to access a dedicated hardware timer;
- In C, all `hwtimer_t`s are as if declared with `[[hwtimer]]`;
- If software-enabled sharing of timers is required, it can be implemented using lib_xcore

Locks

Locks

- Locks aren't available directly in XC;
- lib_xcore exposes them as they enable hardware-supported mutual exclusion

```c
#include <xcore/lock.h>


int main(void)
{
  lock_t l = lock_alloc();
  lock_free(l);
}
```

```c
#include <xcore/lock.h>

int main(void)
{
  lock_t l = lock_alloc();
  lock_free(l);
}
```

Lock functions are available from 'xcore/lock.h'

```c
#include <xcore/lock.h>


int main(void)
{
  lock_t l = lock_alloc();
  lock_free(l);
}
```

Like timers, they are allocated from a pool and must be freed when no longer needed

```c
#include <xcore/lock.h>

int main(void)
{
  lock_t l = lock_alloc();

  lock_acquire(l); // Blocks until available

  lock_release(l);

  lock_free(l);
}
```

Locks have a simple API - they support only 'acquire' and 'release'

```c
#include <xcore/lock.h>

int main(void)
{
  lock_t l = lock_alloc();

  lock_acquire(l); // Blocks until available

  lock_release(l);

  lock_free(l);
}
```

'lock_acquire' will block if the lock has already been acquired by another thread

```c
#include <xcore/lock.h>

int main(void)
{
  lock_t l = lock_alloc();

  lock_acquire(l); // Blocks until available

  lock_release(l);

  lock_free(l);
}
```

'lock_release' releases the lock (which will unblock a 'lock_acquire' if another thread is waiting for the lock)

```c
#include <xcore/lock.h>

int main(void)
{
  lock_t l = lock_alloc();

  lock_acquire(l); // Blocks until available

  lock_acquire(l);

  lock_release(l);

  lock_free(l);
}
```

It's fine to acquire a lock already held by the current thread - this will have no effect and won't block

```c
#include <xcore/lock.h>


int main(void)
{
  lock_t l = lock_alloc();


  lock_acquire(l); // Blocks until available

  lock_acquire(l);


  lock_release(l);

  lock_release(l); // Error: ILLEGAL_RESOURCE


  lock_free(l);
}
```

However, releasing a lock which isn't held by any thread will cause a trap

```c
#include <xcore/lock.h>

int main(void)
{
  lock_t l = lock_alloc();

  lock_acquire(l); // Blocks until available

  lock_acquire(l);

  lock_release(l);
  lock_release(l); // Error: ILLEGAL_RESOURCE

  lock_free(l);
}
```

Of course, locks aren't useful in a single threaded application

Par

Par

- Provided as a group of macros to allow two styles:
  - One which can call `void` functions with arbitrary argument packs;
    - But the functions must be declared using a macro
  - One which can call arbitrary functions with signature: `void (void *)`
- Implicit fork/join - all threads must complete before the Par block finishes;
- Participates in stack size calculation with thread stacks taken from calling stack;
- Implemented in terms of a publicly available thread API

xc

```
#include <stdio.h>

void print_int_sum(int a, int b)
{
  int result = a + b;
  printf("Int sum is: %d\n", result);
}

void print_float_sum(float a, float b)
{
  float result = a + b;
  printf("Float sum is: %.1f\n", result);
}

int main(void)
{
  par {
    print_int_sum(10, 5);
    print_float_sum(10.0, 5.0);
  }
```

XC

```c
#include <stdio.h>

void print_int_sum(int a, int b)
{
  int result = a + b;
  printf("Int sum is: %d\n", result);
}

void print_float_sum(float a, float b)
{
  float result = a + b;
  printf("Float sum is: %.1f\n", result);
}

int main(void)
{
  par {
    print_int_sum(10, 5);
    print_float_sum(10.0, 5.0);
  }
```

C (PJob style)

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

DECLARE_JOB(print_int_sum, (int, int));
void print_int_sum(int a, int b)
{
  int result = a + b;
  printf("Int sum is: %d\n", result);
}

DECLARE_JOB(print_float_sum, (float, float));
void print_float_sum(float a, float b)
{
  float result = a + b;
  printf("Float sum is: %.1f\n", result);
}

int main(void)
{
  PAR_JOBS(
    PJOB(print_int_sum, (10, 5)),
    PJOB(print_float_sum, (10.0, 5.0))
```

XC

```
#include <stdio.h>

void print_int_sum(int a, int b)
{
  int result = a + b;
  printf("Int sum is: %d\n", result);
}

void print_float_sum(float a, float b)
{
  float result = a + b;
```

C (PJob style)

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

DECLARE_JOB(print_int_sum, (int, int));
void print_int_sum(int a, int b)
{
  int result = a + b;
  printf("Int sum is: %d\n", result);
}

DECLARE_JOB(print_float_sum, (float, float));
```

Macros for 'par-style' threading come from 'xcore/parallel.h'

XC

```c
#include <stdio.h>

void print_int_sum(int a, int b)
{
  int result = a + b;
  printf("Int sum is: %d\n", result);
}

void print_float_sum(float a, float b)
{
```

C (PJob style)

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

DECLARE_JOB(print_int_sum, (int, int));
void print_int_sum(int a, int b)
{
  int result = a + b;
  printf("Int sum is: %d\n", result);
}

DECLARE_JOB(print_float_sum, (float, float));
void print_float_sum(float a, float b)
{
  float result = a + b;
  printf("Float sum is: %.1f\n", result);
```

For a function to be usable as a 'PJob' it must be declared using 'DECLARE_JOB' in the same translation unit as the 'PAR_JOBS' which uses it

XC

```c
#include <stdio.h>

void print_int_sum(int a, int b)
{
  int result = a + b;
  printf("Int sum is: %d\n", result);
}

void print_float_sum(float a, float b)
{
```

C (PJob style)

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

DECLARE_JOB(print_int_sum, (int, int));
void print_int_sum(int a, int b)
{
  int result = a + b;
  printf("Int sum is: %d\n", result);
}

DECLARE_JOB(print_float_sum, (float, float));
void print_float_sum(float a, float b)
{
  float result = a + b;
  printf("Float sum is: %.1f\n", result);
}
```

The argument pack must match the signature of the function, names must be omitted and this cannot contain arrays

XC

```
}

void print_float_sum(float a, float b)
{
  float result = a + b;
  printf("Float sum is: %.1f\n", result);
}

int main(void)
{
  par {
    print_int_sum(10, 5);
    print_float_sum(10.0, 5.0);
  }
}
```

C (PJob style)

```

DECLARE_JOB(print_float_sum, (float, float));
void print_float_sum(float a, float b)
{
  float result = a + b;
  printf("Float sum is: %.1f\n", result);
}

int main(void)
{
  PAR_JOBS(
    PJOB(print_int_sum, (10, 5)),
    PJOB(print_float_sum, (10.0, 5.0))
  );
}
```

'PAR_JOBS' runs two or more 'PJob's each in a different thread

XC

```
}

void print_float_sum(float a, float b)
{
  float result = a + b;
  printf("Float sum is: %.1f\n", result);
}

int main(void)
{
  par {
    print_int_sum(10, 5);
    print_float_sum(10.0, 5.0);
  }
}
```

C (PJob style)

```

DECLARE_JOB(print_float_sum, (float, float));
void print_float_sum(float a, float b)
{
  float result = a + b;
  printf("Float sum is: %.1f\n", result);
}

int main(void)
{
  PAR_JOBS(
    PJOB(print_int_sum, (10, 5)),
    PJOB(print_float_sum, (10.0, 5.0))
  );
}
```

Each argument to 'PAR_JOBS' must be an expansion of 'PJOB'; the arguments to 'PJOB' are a call to the function declared with 'DECLARE_JOB' but with the name and argument pack separated by a comma

XC

```
#include <stdio.h>

int print_int_sum(int a, int b)
{
  int result = a + b;
  printf("Int sum is: %d\n", result);
  return result;
}

float print_float_sum(float a, float b)
{
  float result = a + b;
  printf("Float sum is: %.1f\n", result);
  return result;
}

int main(void)
{
  int i;
  float f;
  par {
    i = print_int_sum(10, 5);
    f = print_float_sum(10.0, 5.0);
```

C (PJob style)

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

DECLARE_JOB(print_int_sum, (int, int, int *));
void print_int_sum(int a, int b, int *sum)
{
  int result = a + b;
  printf("Int sum is: %d\n", result);
  *sum = result;
}

DECLARE_JOB(print_float_sum, (float, float, float *));
void print_float_sum(float a, float b, float *sum)
{
  float result = a + b;
  printf("Float sum is: %.1f\n", result);
  *sum = result;
}

int main(void)
{
  int i;
  float f;
  PAR_JOBS(
    PJOB(print_int_sum, (10, 5, &i)),
    PJOB(print_float_sum, (10.0, 5.0, &f))
```

PJobs always have void return type - if a function needs to return a value it must do so through a pointer

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

DECLARE_JOB(my_void_void_job, (void));
void my_void_void_job(void)
{
  puts("Hello from my_void_void_job!");
}

int main(void)
{
  PAR_JOBS(
    PJOB(my_void_void_job, ()),
    PJOB(my_void_void_job, ()));
```

Functions with no arguments can be used as PJob functions like any others

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

DECLARE_JOB(my_void_void_job, (void));
void my_void_void_job(void)
{
  puts("Hello from my_void_void_job!");
}

int main(void)
```

Note that the argument signature pack must be exactly '(void)'

```
DECLARE_JOB(my_void_void_job, (void));

void my_void_void_job(void)

{

  puts("Hello from my_void_void_job!");

}


int main(void)

{

  PAR_JOBS(

    PJOB(my_void_void_job, ()),

    PJOB(my_void_void_job, ()));

}
```

And an empty argument pack must be passed to 'PJob'

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

void print_string(void *str)
{
  puts((const char *)str);
}

struct float_sum_args {
  float a;
  float b;
  float result;
};

void float_sum(void *args_)
{
  struct float_sum_args *args = args_;
  args->result = args->a + args->b;
}

int main(void)
{
  struct float_sum_args fs_args = {5.0, 10.0, 0};
  PAR_FUNCS(
    PFUNC(print_string, "Hello world!"),
    PFUNC(float_sum, &fs_args)
  );
```

As an alternative to PJobs, xcore/parallel.h also provides the 'PFunc' style of Par

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

void print_string(void *str)
{
  puts((const char *)str);
}

struct float_sum_args {
  float a;
  float b;
  float result;
};

void float_sum(void *args_)
{
  struct float_sum_args *args = args_;
  args->result = args->a + args->b;
}

int main(void)
{
  struct float_sum_args fs_args = {5.0, 10.0, 0};
  PAR_FUNCS(
    PFUNC(print_string, "Hello World!"),
    PFUNC(float_sum, &fs_args),
    PFUNC(print_string, "Hello World!")
  );
  printf("%f\n", fs_args.result);
  return 0;
}
```

This can parallelise two or more calls to functions with signature 'void(void*)'

```c
void float_sum(void *args_)
{
  struct float_sum_args *args = args_;
  args->result = args->a + args->b;
}


int main(void)
{
  struct float_sum_args fs_args = {5.0, 10.0, 0};
  PAR_FUNCS(
    PFUNC(print_string, "Hello world!"),
    PFUNC(float_sum, &fs_args)
  );
  printf("%.1f + %.1f = %.1f\n", fs_args.a, fs_args.b, fs_args.result);
}
```

The 'PAR_FUNCS' macro takes two or more arguments expanded from 'PFUNC'

```c
void float_sum(void *args_)
{
  struct float_sum_args *args = args_;
  args->result = args->a + args->b;
}


int main(void)
{
  struct float_sum_args fs_args = {5.0, 10.0, 0};
  PAR_FUNCS(
    PFUNC(print_string, "Hello world!"),
    PFUNC(float_sum, &fs_args)
  );
  printf("%.1f + %.1f = %.1f\n", fs_args.a, fs_args.b, fs_args.result);
}
```

PFUNC takes two arguments: the function to call, and a value for its parameter which must be implicitly convertible to 'void *'

```c
void print_string(void *str)
{
  puts((const char *)str);
}

struct float_sum_args {
  float a;
  float b;
  float result;
};

void float_sum(void *args_)
{
  struct float_sum_args *args = args_;
  args->result = args->a + args->b;
}

int main(void)
```

This style can be more flexible but usually requires additional code to deal with parameters

Channels

XC

```
void sends_first(chanend c)
{
}

void receives_first(chanend c)
{
}

int main(void)

{
  chan c;
  par {
    sends_first(c);
    receives_first(c);
  }
}
```

XC

```
void sends_first(chanend c)
{
}

void receives_first(chanend c)
{
}

int main(void)
{
  chan c;
  par {
    sends_first(c);
    receives_first(c);
  }
}
```

C

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

DECLARE_JOB(sends_first, (chanend_t));
void sends_first(chanend_t c)
{
}

DECLARE_JOB(receives_first, (chanend_t));
void receives_first(chanend_t c)
{
}

int main(void)
{
  channel_t c = chan_alloc();
  PAR_JOBS(
    PJOB(sends_first, (c.end_a)),
    PJOB(receives_first, (c.end_b))
  );
```

XC

```
void sends_first(chanend c)
{
}

void receives_first(chanend c)
{
}

int main(void)
```

C

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

DECLARE_JOB(sends_first, (chanend_t));
void sends_first(chanend_t c)
{
}

DECLARE_JOB(receives_first, (chanend_t));
```

Functions for using regular channels ('chan' in XC) are in 'xcore/channel.h'

XC

```
void sends_first(chanend c)
{
}

void receives_first(chanend c)
{
}

int main(void)
{
  chan c;
  par {
    sends_first(c);
    receives_first(c);
  }
}
```

C

```
DECLARE_JOB(sends_first, (chanend_t));
void sends_first(chanend_t c)
{
}

DECLARE_JOB(receives_first, (chanend_t));
void receives_first(chanend_t c)
{
}

int main(void)
{
  channel_t c = chan_alloc();
  PAR_JOBS(
    PJOB(sends_first, (c.end_a)),
    PJOB(receives_first, (c.end_b))
  );
  chan_free(c);
}
```

'chan_alloc' allocates and configures the chanends which form a channel

XC

```
{

}


void receives_first(chanend c)

{

}


int main(void)

{

  chan c;

  par {

    sends_first(c);

    receives_first(c);

  }

}
```

C

```
}


DECLARE_JOB(receives_first, (chanend_t));

void receives_first(chanend_t c)

{

}


int main(void)

{

  channel_t c = chan_alloc();

  PAR_JOBS(

    PJOB(sends_first, (c.end_a)),

    PJOB(receives_first, (c.end_b))

  );

  chan_free(c);

}
```

Unlike in XC, in C a channel won't 'decay' into chanends when passed to a function - 'end_a' or 'end_b' must be chosen and passed to users

XC

```
void sends_first(chanend c)
{
  c <: 1;
  c <: 5;
}

void receives_first(chanend c)
{
  int a;
  c :> a;
  c :> a;
}

int main(void)
{
  chan c;
```

C

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

DECLARE_JOB(sends_first, (chanend_t));
void sends_first(chanend_t c)
{
  chan_out_word(c, 1);
  chan_out_word(c, 5);
}

DECLARE_JOB(receives_first, (chanend_t));
void receives_first(chanend_t c)
{
  int a = chan_in_word(c);
  a = chan_in_word(c);
}

int main(void)
{
  channel_t c = chan_alloc();
```

Like with other resources, I/O operators are replaced by functions which operate on the channel ends

XC

```
void sends_first(chanend c)
{
  c <: 1;
  c <: 5;
}

void receives_first(chanend c)
{
  int a;

  c :> a;
  c :> a;
}

int main(void)
{
```

C

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

DECLARE_JOB(sends_first, (chanend_t));
void sends_first(chanend_t c)
{
  chan_out_word(c, 1);
  chan_out_byte(c, 5);
}

DECLARE_JOB(receives_first, (chanend_t));
void receives_first(chanend_t c)
{
  int a = chan_in_word(c);
  a = chan_in_byte(c);
}

int main(void)
{
```

It's possible to send bytes rather than whole words over a channel, but it's essential to make sure that the type sent is the type read

Channel Safety

- Since lib_xcore lets you choose the type used for input and output there's a risk of mismatch;
- Users must ensure that the type written to a chanend is the same as the type read from the other end

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

DECLARE_JOB(sends_first, (chanend_t));
void sends_first(chanend_t c)
{
  chan_out_word(c, 1);
  chan_out_byte(c, 5);
}

DECLARE_JOB(receives_first, (chanend_t));
void receives_first(chanend_t c)
{
  int a = chan_in_word(c);
  a = chan_in_byte(c);
}

int main(void)
{
  channel_t c = chan_alloc();
  PAR_JOBS(
    PJOB(sends_first, (c.end_a)),
    PJOB(receives_first, (c.end_b))
```

Arrays and Channels

- In addition to single bytes and words it's possible to send arrays of either;
- As before, it's the user's responsibility to ensure that the type read is the same one received;
    - This includes the length of the array

```
DECLARE_JOB(sends_first, (chanend_t));
void sends_first(chanend_t c)
{
  chan_out_word(c, 1);
  chan_out_byte(c, 5);

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

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

DECLARE_JOB(receives_first, (chanend_t));
void receives_first(chanend_t c)
{
  int a = chan_in_word(c);
  a = chan_in_byte(c);

  uint32_t words[5];
  chan_in_buf_word(c, words, 5);

  unsigned char bytes[4];
  chan_in_buf_byte(c, bytes, 4);
}
```

Streaming Channels

- Streaming channels (`steaming chan` in XC) are available in lib_xcore;
- These are similar to regular channels but do not synchronise or close the connection after each packet;
- Streaming channel functions are available from 'xcore/channel_streaming.h';
- It has the same API except that:
  - Function names are prefixed with `s_`;
  - The return type of `s_chan_alloc` is `streaming_channel_t`
- As with types, the user must ensure that the same type of channel method is used to read and to write;
  - e.g. a word written with `chan_out_word` **must not** be read with `s_chan_out_word`

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

DECLARE_JOB(sends_first, (chanend_t));
void sends_first(chanend_t c)
{
  s_chan_out_word(c, 1);
  s_chan_out_byte(c, 5);

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

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

DECLARE_JOB(receives_first, (chanend_t));
void receives_first(chanend_t c)
{
  int a = s_chan_in_word(c);
  a = s_chan_in_byte(c);

  uint32_t words[5];
  s_chan_in_buf_word(c, words, 5);

  unsigned char bytes[4];
  s_chan_in_buf_byte(c, bytes, 4);
}

int main(void)
{
  streaming_channel_t c = s_chan_alloc();
```

Select

- Macros are provided as a replacement for XC's `select`
- They support the basic features of XC's `select`:
  - Default cases;
  - Guarded cases (including defaults);
  - Ordered Select
- In C, Select blocks work on generic resources;
  - i.e. they don't do any resource-specific setup (e.g. reconfiguring timers)
- A lower-level API is also available to support unusual use cases

xc

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

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

  select
  {
  case t when timerafter(now + 10000000) :> now:
    printf("Timer handler at time: %lu\n", now);
    break;
  }
}
```

XC

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

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

  select
  {
  case t when timerafter(now + 10000000) :> now:
    printf("Timer handler at time: %lu\n", now);
    break;
  }
}
```

C

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

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

  hwtimer_set_trigger_time(t, now + 10000000);
  SELECT_RES(
    CASE_THEN(t, timer_handler))
  {
  timer_handler:
    now = hwtimer_get_time(t);
    printf("Timer handler at time: %lu\n", now);
    break;
  }

  hwtimer_free(t);
```

XC

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

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


  select
```

C

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

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


  hwtimer_set_trigger_time(t, now + 10000000);
  SELECT_RES(
```

Macros for Select are in the 'xcore/select.h' system header

XC

```xc
#include <platform.h>


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


  select
  {
  case t when timerafter(now + 10000000) :> now:
    printf("Timer handler at time: %lu\n", now);
    break;
  }
}
```

C

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


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


  hwtimer_set_trigger_time(t, now + 10000000);
  SELECT_RES(
    CASE_THEN(t, timer_handler))
  {
  timer_handler:
    now = hwtimer_get_time(t);
    printf("Timer handler at time: %lu\n", now);
    break;
  }


  hwtimer_free(t);
}
```

lib_xcore's select macros enable a style similar to XC

XC

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

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

  select
  {
  case t when timerafter(now + 10000000) :> now:
    printf("Timer handler at time: %lu\n", now);
    break;
  }
}
```

C

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

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

  hwtimer_set_trigger_time(t, now + 10000000);
  SELECT_RES(
    CASE_THEN(t, timer_handler))
  {
  timer_handler:
    now = hwtimer_get_time(t);
    printf("Timer handler at time: %lu\n", now);
    break;
  }

  hwtimer_free(t);
```

The 'SELECT_RES' macro begins a block like in XC

XC

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

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

  select
  {
  case t when timerafter(now + 10000000) :> now:
    printf("Timer handler at time: %lu\n", now);
    break;
  }
}
```

C

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

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

  hwtimer_set_trigger_time(t, now + 10000000);
  SELECT_RES(
    CASE_THEN(t, timer_handler))
  {
  timer_handler:
    now = hwtimer_get_time(t);
    printf("Timer handler at time: %lu\n", now);
    break;
  }

  hwtimer_free(t);
}
```

Unlike in XC, we have to provide all the cases at the top of the block, as arguments to 'SELECT_RES' expanded from macros like 'CASE_THEN'

XC

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

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

  select
  {
  case t when timerafter(now + 10000000) :> now:
    printf("Timer handler at time: %lu\n", now);
    break;
  }
}
```

C

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

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

  hwtimer_set_trigger_time(t, now + 10000000);
  SELECT_RES(
    CASE_THEN(t, timer_handler))
  {
  timer_handler:
    now = hwtimer_get_time(t);
    printf("Timer handler at time: %lu\n", now);
    break;
  }

  hwtimer_free(t);
}
```

This 'CASE_THEN' means that when an event occurs on resource 't', control will be transferred to the label 'timer_handler'

XC

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

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

  select
  {
  case t when timerafter(now + 10000000) :> now:
    printf("Timer handler at time: %lu\n", now);
    break;
  }
}
```

C

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

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

  hwtimer_set_trigger_time(t, now + 10000000);
  SELECT_RES(
    CASE_THEN(t, timer_handler))
  {
  timer_handler:
    now = hwtimer_get_time(t);
    printf("Timer handler at time: %lu\n", now);
    break;
  }

  hwtimer_free(t);
}
```

In C we also have to take care of trigger conditions and reading the resource manually

XC

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


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


  select
  {
  case t when timerafter(now + 10000000) :> now:
    printf("Timer handler at time: %lu\n", now);
    break;
  }
}
```

C

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


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


  hwtimer_set_trigger_time(t, now + 10000000);
  SELECT_RES(
    CASE_THEN(t, timer_handler))
  {
  timer_handler:
    now = hwtimer_get_time(t);
    printf("Timer handler at time: %lu\n", now);
    break;
  }
```

We set up the trigger outside the select

XC

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

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

  select
  {
  case t when timerafter(now + 10000000) :> now:
    printf("Timer handler at time: %lu\n", now);
    break;
  }
}
```

C

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

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

  hwtimer_set_trigger_time(t, now + 10000000);
  SELECT_RES(
    CASE_THEN(t, timer_handler))
  {
  timer_handler:
    now = hwtimer_get_time(t);
    printf("Timer handler at time: %lu\n", now);
    break;
  }

  hwtimer_free(t);
}
```

And read the timer's value in the event handler

XC

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

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

  select
  {
  case t when timerafter(now + 10000000) :> now:
    printf("Timer handler at time: %lu\n", now);
    break;
  }
}
```

C

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

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

  hwtimer_set_trigger_time(t, now + 10000000);
  SELECT_RES(
    CASE_THEN(t, timer_handler))
  {
  timer_handler:
    now = hwtimer_get_time(t);
    printf("Timer handler at time: %lu\n", now);
    break;
  }

  hwtimer_free(t);
```

This Select only runs once, but most Select blocks occur inside loops...

XC

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


  while (1)
  {
    select
    {
    case t when timerafter(now + 10000000) :> now:
      printf("Timer handler at time: %lu\n", now);
      break;
    }
  }
}
```

C

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


  hwtimer_set_trigger_time(t, now + 10000000);
  SELECT_RES(
    CASE_THEN(t, timer_handler))
  {
  timer_handler:
    now = hwtimer_get_time(t);
    hwtimer_change_trigger_time(t, now + 10000000);
    printf("Timer handler at time: %lu\n", now);
    continue;
  }


  hwtimer_free(t);
}
```

'SELECT_RES' is always a loop, so we can just 'continue' to go back and wait for another event

XC

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


  while (1)
  {
    select
    {
    case t when timerafter(now + 10000000) :> now:
      printf("Timer handler at time: %lu\n", now);
      break;
    }
  }
}
```

C

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


  hwtimer_set_trigger_time(t, now + 10000000);
  SELECT_RES(
    CASE_THEN(t, timer_handler))
  {
  timer_handler:
    now = hwtimer_get_time(t);
    hwtimer_change_trigger_time(t, now + 10000000);
    printf("Timer handler at time: %lu\n", now);
    continue;
  }

  hwtimer_free(t);
}
```

As the trigger is not part of the Select, we also have to explicitly change our trigger if it needs updating

XC

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

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

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

C

```
{
  hwtimer_t t = hwtimer_alloc();
  unsigned long now = hwtimer_get_time(t);

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

  hwtimer_free(t);
}
```

Default cases can be introduced using 'DEFAULT_THEN'

XC

```
  unsigned long now;
  t :> now;


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

C

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


  hwtimer_free(t);
}
```

Like in XC, this will run the default handler if no event is ready on entry to the Select

XC

```xc
#include <platform.h>

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

  int default_last = 0;
  while (1)
  {
    select
    {
    case default_last => t when timerafter(now + 10000000) :> now:
      printf("Timer handler at time: %lu\n", now);
      default_last = 0;
      break;
    default:
      puts("Nothing happened...");
      default_last = 1;
      break;
    }
  }
}
```

C

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

  int default_last = 0;
  hwtimer_set_trigger_time(t, now + 10000000);
  SELECT_RES(
    CASE_GUARD_THEN(t, default_last, timer_handler),
    DEFAULT_THEN(default_handler))
  {
  timer_handler:
    now = hwtimer_get_time(t);
    hwtimer_change_trigger_time(t, now + 10000000);
    printf("Timer handler at time: %lu\n", now);
    default_last = 0;
    continue;
  default_handler:
    puts("Nothing happened...");
    default_last = 1;
    continue;
```

We can put guards on cases by using the 'CASE_GUARD_THEN' and 'CASE_NGUARD_THEN' macros

XC

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


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


  int default_last = 0;
  while (1)
  {
    select
    {
    case default_last => t when timerafter(now + 10000000) :> now:
      printf("Timer handler at time: %lu\n", now);
      default_last = 0;
      break;
    default:
      puts("Nothing happened...");
      default_last = 1;
      break;
    }
  }
}
```

C

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


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


  int default_last = 0;
  hwtimer_set_trigger_time(t, now + 10000000);
  SELECT_RES(
    CASE_GUARD_THEN(t, default_last, timer_handler),
    DEFAULT_THEN(default_handler))
  {
  timer_handler:
    now = hwtimer_get_time(t);
    hwtimer_change_trigger_time(t, now + 10000000);
    printf("Timer handler at time: %lu\n", now);
    default_last = 0;
    continue;
  default_handler:
```

If an event is waiting on 't', we'll now only handle it if 'default_last' is true at the time we enter the Select (or continue from a handler)

XC

```
{
  [[hwtimer]]
  timer t;
  unsigned long now;
  t :> now;


  int default_last = 0;
  while (1)
  {
    select
    {
    case default_last => t when timerafter(now + 10000000) :> now:
      printf("Timer handler at time: %lu\n", now);
      default_last = 0;
      break;
    !default_last => default:
      puts("Nothing happened...");
      default_last = 1;
      break;
    }
  }
}
```

C

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


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


  int default_last = 0;
  hwtimer_set_trigger_time(t, now + 10000000);
  SELECT_RES(
    CASE_GUARD_THEN(t, default_last, timer_handler),
    DEFAULT_NGUARD_THEN(default_last, default_handler))
  {
  timer_handler:
    now = hwtimer_get_time(t);
    hwtimer_change_trigger_time(t, now + 10000000);
    printf("Timer handler at time: %lu\n", now);
    default_last = 0;
    continue;
  default_handler:
```

Guards can also be placed on default cases using the 'DEFAULT_GUARD_THEN' and 'DEFAULT_NGUARD_THEN' macros

XC

```
{
  [[hwtimer]]
  timer t;
  unsigned long now;
  t :> now;


  int default_last = 0;
  while (1)
  {
    select
    {
    case default_last => t when timerafter(now + 10000000) :> now:
      printf("Timer handler at time: %lu\n", now);
      default_last = 0;
      break;
    !default_last => default:
      puts("Nothing happened...");
      default_last = 1;
      break;
    }
  }
}
```

C

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


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


  int default_last = 0;
  hwtimer_set_trigger_time(t, now + 10000000);
  SELECT_RES(
    CASE_GUARD_THEN(t, default_last, timer_handler),
    DEFAULT_NGUARD_THEN(default_last, default_handler))
  {
  timer_handler:
    now = hwtimer_get_time(t);
    hwtimer_change_trigger_time(t, now + 10000000);
    printf("Timer handler at time: %lu\n", now);
    default_last = 0;
    continue;
  default_handler:
```

Note that the 'NGUARD' variant inverts the condition argument

XC

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

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

  int default_last = 0;
  while (1)
  {
    [[ordered]]
    select
    {
    case default_last => t when timerafter(now + 10000000) :> now:
      printf("Timer handler at time: %lu\n", now);
      default_last = 0;
      break;
    !default_last => default:
      puts("Nothing happened...");
      default_last = 1;
      break;
    }
```

C

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

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

  int default_last = 0;
  hwtimer_set_trigger_time(t, now + 10000000);
  SELECT_RES_ORDERED(
    CASE_GUARD_THEN(t, default_last, timer_handler),
    DEFAULT_NGUARD_THEN(default_last, default_handler))
  {
  timer_handler:
    now = hwtimer_get_time(t);
    hwtimer_change_trigger_time(t, now + 10000000);
    printf("Timer handler at time: %lu\n", now);
    default_last = 0;
```

Finally, we can use 'SELECT_RES_ORDERED' to get an ordered Select

XC

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

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

  int default_last = 0;
  while (1)
  {
    [[ordered]]
    select
    {
    case default_last => t when timerafter(now + 10000000) :> now:
      printf("Timer handler at time: %lu\n", now);
      default_last = 0;
      break;
    !default_last => default:
      puts("Nothing happened...");
      default_last = 1;
      break;
    }
```

C

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

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

  int default_last = 0;
  hwtimer_set_trigger_time(t, now + 10000000);
  SELECT_RES_ORDERED(
    CASE_GUARD_THEN(t, default_last, timer_handler),
    DEFAULT_NGUARD_THEN(default_last, default_handler))
  {
  timer_handler:
    now = hwtimer_get_time(t);
    hwtimer_change_trigger_time(t, now + 10000000);
    printf("Timer handler at time: %lu\n", now);
    default_last = 0;
    continue;
  default_handler:
    puts("Nothing happened...");
    default_last = 1;
    continue;
  }
```

This gives cases which appear earlier in the arguments list higher priority (in this trivial Select it has no effect)

Nested Selects

- Select blocks modify the thread's global state and don't restore it on exit;
- This means that, when used recursively, an outer Select will have to re-run its global setup when an inner Select completes;
- This happens automatically when you `continue` in a Select handler, but uses a runtime check to determine if it's necessary;

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

void my_function_which_selects()
{
  hwtimer_t t = hwtimer_alloc();
  SELECT_RES(CASE_THEN(t, timer_handler))
  {
  timer_handler:
    puts("Inner handler...");
    break;
  }
  hwtimer_free(t);
}

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

  int default_last = 0;
  hwtimer_set_trigger_time(t, now + 10000000);
  SELECT_RES_ORDERED(
    CASE_GUARD_THEN(t, default_last, timer_handler),
    DEFAULT_NGUARD_THEN(default_last, default_handler))
  {
  timer_handler:
    now = hwtimer_get_time(t);
    hwtimer_change_trigger_time(t, now + 10000000);
    printf("Timer handler at time: %lu\n", now);
    my_function_which_selects();
    default_last = 0;
    SELECT_CONTINUE_RESET;
  default_handler:
    puts("Nothing happened...");
```

Nested Selects

- Select blocks modify the thread's global state and don't restore it on exit;
- This means that, when used recursively, an outer Select will have to re-run its global setup when an inner Select completes;
- This happens automatically when you `continue` in a Select handler, but uses a runtime check to determine if it's necessary;
- The `SELECT_CONTINUE_RESET` macro will unconditionally jump back to the start of the innermost Select and re-run any one-off setup;
    - This may be used instead of `continue` where a Select handler is likely to have used a Select;
    - It's not necessary if the outer handler uses `break` to exit the Select

```
  }
  hwtimer_free(t);
}

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

  int default_last = 0;
  hwtimer_set_trigger_time(t, now + 10000000);
  SELECT_RES_ORDERED(
    CASE_GUARD_THEN(t, default_last, timer_handler),
    DEFAULT_NGUARD_THEN(default_last, default_handler))
  {
  timer_handler:
    now = hwtimer_get_time(t);
    hwtimer_change_trigger_time(t, now + 10000000);
    printf("Timer handler at time: %lu\n", now);
    my_function_which_selects();
    default_last = 0;
    SELECT_CONTINUE_RESET;
  default_handler:
    puts("Nothing happened...");
    default_last = 1;
    continue;
  }

  hwtimer_free(t);
}
```

Nested Selects

- Select blocks modify the thread's global state and don't restore it on exit;
- This means that, when used recursively, an outer Select will have to re-run its global setup when an inner Select completes;
- This happens automatically when you `continue` in a Select handler, but uses a runtime check to determine if it's necessary;
- The `SELECT_CONTINUE_RESET` macro will unconditionally jump back to the start of the innermost Select and re-run any one-off setup;
  - This may be used instead of `continue` where a Select handler is likely to have used a Select;
  - It's not necessary if the outer handler uses `break` to exit the Select
- The `SELECT_CONTINUE_NO_RESET` macro unconditionally jumps back to the start of the innermost Select **without** re-running global setup;
  - This may be used instead of `continue` where a Select handler has **definitely not** used a Select;
  - Again, where the outer Select handler exits with a `break`, there's no need to re-run global setup

```
  puts("Hello World!");
}


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


  int default_last = 0;
  hwtimer_set_trigger_time(t, now + 10000000);
  SELECT_RES_ORDERED(
    CASE_GUARD_THEN(t, default_last, timer_handler),
    DEFAULT_NGUARD_THEN(default_last, default_handler))
  {
  timer_handler:
    now = hwtimer_get_time(t);
    hwtimer_change_trigger_time(t, now + 10000000);
    printf("Timer handler at time: %lu\n", now);
    my_function_which_does_not_select();
    default_last = 0;
    SELECT_CONTINUE_NO_RESET;
  default_handler:
    puts("Nothing happened...");
    default_last = 1;
    continue;
  }


  hwtimer_free(t);
}
```

Other Features

Assertions

```c
#include <xcore/assert.h>

int main(void)
{
  xassert(1);
  xassert_not(0);

  xassert(0);
}
```

'xcore/assert.h' contains assertion macros which (by default) utilise the xCore's assertion instructions

```c
#include <xcore/assert.h>

int main(void)
{
  xassert(1);
  xassert_not(0);

  xassert(0);
}
```

'xassert' is equivalent to the standard library 'assert'

Assertions: Configuration

```
$ xcc assert.c -target=XCORE-200-EXPLORER -o assert.xe

$ xsim assert.xe

Uhandled exception: ECALL, data: 0x00000000

$ xcc -DLIBXCORE_XASSERT_IS_ASSERT assert.c -target=XCORE-200-EXPLORER -o assert.xe

$ xsim assert.xe

assertion "(0)" failed: file "assert.c", line 8, function: main

Unhandled exception: ECALL, data: 0x00000000


$ xcc -NDEBUG assert.c -target=XCORE-200-EXPLORER -o assert.xe

$ xsim assert.xe

$
```

The effect of using these assertions can be altered using macro definitions

```
$ xcc assert.c -target=XCORE-200-EXPLORER -o assert.xe

$ xsim assert.xe

Uhandled exception: ECALL, data: 0x00000000

$ xcc -DLIBXCORE_XASSERT_IS_ASSERT assert.c -target=XCORE-200-EXPLORER -o assert.xe

$ xsim assert.xe

assertion "(0)" failed: file "assert.c", line 8, function: main

Unhandled exception: ECALL, data: 0x00000000


$ xcc -NDEBUG assert.c -target=XCORE-200-EXPLORER -o assert.xe

$ xsim assert.xe
```

By default the assertions are effective and use the assertion instructions

```
$ xcc assert.c -target=XCORE-200-EXPLORER -o assert.xe

$ xsim assert.xe

Uhandled exception: ECALL, data: 0x00000000

$ xcc -DLIBXCORE_XASSERT_IS_ASSERT assert.c -target=XCORE-200-EXPLORER -o assert.xe

$ xsim assert.xe

assertion "(0)" failed: file "assert.c", line 8, function: main

Unhandled exception: ECALL, data: 0x00000000


$ xcc -NDEBUG assert.c -target=XCORE-200-EXPLORER -o assert.xe

$ xsim assert.xe

$
```

Defining 'LIBXCORE_XASSERT_IS_ASSERT' will cause the assertions to expand to standard library assertions

```
$ xcc assert.c -target=XCORE-200-EXPLORER -o assert.xe

$ xsim assert.xe

Uhandled exception: ECALL, data: 0x00000000

$ xcc -DLIBXCORE_XASSERT_IS_ASSERT assert.c -target=XCORE-200-EXPLORER -o assert.xe

$ xsim assert.xe

assertion "(0)" failed: file "assert.c", line 8, function: main

Unhandled exception: ECALL, data: 0x00000000

$ xcc -NDEBUG assert.c -target=XCORE-200-EXPLORER -o assert.xe

$ xsim assert.xe

$
```

The assertions can be made ineffective by defining 'NDEBUG'

Low-level threading API

```c
#include <xcore/thread.h>
#include <stdio.h>

void say(void *str)
{
  puts((const char *)str);
}

static const unsigned stack_words = 512;
static const unsigned stack_bytes = stack_words * sizeof(long);
#define STACK_ALIGN 4

void say_hello_using_xthreads(void)
{
  char stack_0[stack_bytes] __attribute__ ((aligned (STACK_ALIGN)));
  char stack_1[stack_bytes] __attribute__ ((aligned (STACK_ALIGN)));

  xthread_t thread_0 = xthread_alloc_and_start(say, "(XT1) Hello world!", stack_base(stack_0, stack_words)),
            thread_1 = xthread_alloc_and_start(say, "(XT2) Hello world!", stack_base(stack_1, stack_words));

  say("(XT Master) Hello world!");

  xthread_wait_and_free(thread_0);
  xthread_wait_and_free(thread_1);
}

void say_hello_using_thread_group(void)
{
  char stack_0[stack_bytes] __attribute__ ((aligned (STACK_ALIGN)));
  char stack_1[stack_bytes] __attribute__ ((aligned (STACK_ALIGN)));

  threadgroup_t group = thread_group_alloc();

  thread_group_add(group, say, "(TG1) Hello world!", stack_base(stack_0, stack_words)),
  thread_group_add(group, say, "(TG2) Hello world!", stack_base(stack_1, stack_words));

  thread_group_start(group);

  say("(TG Master) Hello world!");

  thread_group_wait_and_free(group);
}

static char stack_async[stack_bytes] __attribute__ ((aligned (STACK_ALIGN)));

void say_hello_asynchronously(void)
{
  run_async(say, "(Async) Hello world!", stack_base(stack_async, stack_words));
}
```

'xcore/thread.h' contains a lower level API which can be used where Par isn't sufficiently flexible

```c
#include <xcore/thread.h>
#include <stdio.h>

void say(void *str)
{
  puts((const char *)str);
}

static const unsigned stack_words = 512;
static const unsigned stack_bytes = stack_words * sizeof(long);
#define STACK_ALIGN 4

void say_hello_using_xthreads(void)
{
  char stack_0[stack_bytes] __attribute__((aligned (STACK_ALIGN)));
```

Thread functions must have signature 'void(void*)'

```
  threadgroup_t group = thread_group_alloc();

  thread_group_add(group, say, "(TG1) Hello world!", stack_base(stack_0, stack_words)),
  thread_group_add(group, say, "(TG2) Hello world!", stack_base(stack_1, stack_words));

  thread_group_start(group);

  say("(TG Master) Hello world!");

  thread_group_wait_and_free(group);
}

static char stack_async[stack_bytes] __attribute__ ((aligned (STACK_ALIGN)));

void say_hello_asynchronously(void)
{
  run_async(say, "(Async) Hello world!", stack_base(stack_async, stack_words));
}

int main(void)
```

It is the user's responsibility to provide a stack pointer with suitable alignment and a large enough region of memory

```
  thread_group_add(group, say, "(TG2) Hello world!", stack_base(stack_1, stack_words));

  thread_group_start(group);

  say("(TG Master) Hello world!");

  thread_group_wait_and_free(group);
}

static char stack_async[stack_bytes] __attribute__ ((aligned (STACK_ALIGN)));

void say_hello_asynchronously(void)
{
  run_async(say, "(Async) Hello world!", stack_base(stack_async, stack_words));
}

int main(void)
{
  say_hello_asynchronously();
  say_hello_using_xthreads();
  say_hello_using_thread_group();
```

The 'stack_base' function computes the correct stack pointer given an aligned base pointer and a size in words

```
  thread_group_add(group, say, "(TG1) Hello world!", stack_base(stack_0, stack_words)),
  thread_group_add(group, say, "(TG2) Hello world!", stack_base(stack_1, stack_words));

  thread_group_start(group);

  say("(TG Master) Hello world!");

  thread_group_wait_and_free(group);
}

static char stack_async[stack_bytes] __attribute__ ((aligned (STACK_ALIGN)));

void say_hello_asynchronously(void)
{
  run_async(say, "(Async) Hello world!", stack_base(stack_async, stack_words));
}

int main(void)
{
  say_hello_asynchronously();
```

The simplest way of launching another thread is with 'run_async'

```
  thread_group_add(group, say, "(TG2) Hello world!", stack_base(stack_1, stack_words));

  thread_group_start(group);

  say("(TG Master) Hello world!");

  thread_group_wait_and_free(group);
}

static char stack_async[stack_bytes] __attribute__ ((aligned (STACK_ALIGN)));

void say_hello_asynchronously(void)
{
  run_async(say, "(Async) Hello world!", stack_base(stack_async, stack_words));
}

int main(void)
{
  say_hello_asynchronously();
  say_hello_using_xthreads();
  say_hello_using_thread_group();
```

This launches a thread but returns no handle - so it's not trivially possible to know it's finished

```
  puts((const char *)str);
}


static const unsigned stack_words = 512;
static const unsigned stack_bytes = stack_words * sizeof(long);
#define STACK_ALIGN 4


void say_hello_using_xthreads(void)
{
  char stack_0[stack_bytes] __attribute__ ((aligned (STACK_ALIGN)));
  char stack_1[stack_bytes] __attribute__ ((aligned (STACK_ALIGN)));

  xthread_t thread_0 = xthread_alloc_and_start(say, "(XT1) Hello world!", stack_base(stack_0, stack_words)),
            thread_1 = xthread_alloc_and_start(say, "(XT2) Hello world!", stack_base(stack_1, stack_words));

  say("(XT Master) Hello world!");

  xthread_wait_and_free(thread_0);
  xthread_wait_and_free(thread_1);
}


void say_hello_using_thread
```

Individual threads can also be launched using the 'xthread' style

```
{
  puts((const char *)str);
}


static const unsigned stack_words = 512;
static const unsigned stack_bytes = stack_words * sizeof(long);
#define STACK_ALIGN 4


void say_hello_using_xthreads(void)
{
  char stack_0[stack_bytes] __attribute__ ((aligned (STACK_ALIGN)));
  char stack_1[stack_bytes] __attribute__ ((aligned (STACK_ALIGN)));

  xthread_t thread_0 = xthread_alloc_and_start(say, "(XT1) Hello world!", stack_base(stack_0, stack_words)),
            thread_1 = xthread_alloc_and_start(say, "(XT2) Hello world!", stack_base(stack_1, stack_words));

  say("(XT Master) Hello world!");

  xthread_wait_and_free(thread_0);
  xthread_wait_and_free(thread_1);
}
```

'xthread_alloc_and_start' starts a function call in another thread and returns a handle to it

```
static const unsigned stack_bytes = stack_words * sizeof(long);

#define STACK_ALIGN 4


void say_hello_using_xthreads(void)

{

  char stack_0[stack_bytes] __attribute__ ((aligned (STACK_ALIGN)));

  char stack_1[stack_bytes] __attribute__ ((aligned (STACK_ALIGN)));


  xthread_t thread_0 = xthread_alloc_and_start(say, "(XT1) Hello world!", stack_base(stack_0, stack_words)),

            thread_1 = xthread_alloc_and_start(say, "(XT2) Hello world!", stack_base(stack_1, stack_words));


  say("(XT Master) Hello world!");


  xthread_wait_and_free(thread_0);

  xthread_wait_and_free(thread_1);

}


void say_hello_using_thread_group(void)

{

  char stack_0[stack_bytes] __attribute__ ((aligned (STACK_ALIGN)));

  char stack_1[stack_bytes] __attribute__ ((aligned (STACK_ALIGN)));
```

'xthread_wait_and_free' waits for an xthread to finish given its handle; it also frees the hardware thread

```
  say("(XT Master) Hello world!");

  xthread_wait_and_free(thread_0);
  xthread_wait_and_free(thread_1);
}

void say_hello_using_thread_group(void)
{
  char stack_0[stack_bytes] __attribute__ ((aligned (STACK_ALIGN)));
  char stack_1[stack_bytes] __attribute__ ((aligned (STACK_ALIGN)));

  threadgroup_t group = thread_group_alloc();

  thread_group_add(group, say, "(TG1) Hello world!", stack_base(stack_0, stack_words)),
  thread_group_add(group, say, "(TG2) Hello world!", stack_base(stack_1, stack_words));

  thread_group_start(group);

  say("(TG Master) Hello world!");

  thread_group_wait_and_free(group);
}
```

Multiple related threads can be launched using thread groups

```
            thread_1 = xthread_alloc_and_start(say, "(XT2) Hello world!", stack_base(stack_1, stack_words));

  say("(XT Master) Hello world!");

  xthread_wait_and_free(thread_0);
  xthread_wait_and_free(thread_1);
}

void say_hello_using_thread_group(void)
{
  char stack_0[stack_bytes] __attribute__ ((aligned (STACK_ALIGN)));
  char stack_1[stack_bytes] __attribute__ ((aligned (STACK_ALIGN)));

  threadgroup_t group = thread_group_alloc();

  thread_group_add(group, say, "(TG1) Hello world!", stack_base(stack_0, stack_words)),
  thread_group_add(group, say, "(TG2) Hello world!", stack_base(stack_1, stack_words));

  thread_group_start(group);

  say("(TG Master) Hello world!");
```

A thread group is allocated using 'thread_group_alloc'

```
  say("(XT Master) Hello world!");

  xthread_wait_and_free(thread_0);
  xthread_wait_and_free(thread_1);
}

void say_hello_using_thread_group(void)
{
  char stack_0[stack_bytes] __attribute__ ((aligned (STACK_ALIGN)));
  char stack_1[stack_bytes] __attribute__ ((aligned (STACK_ALIGN)));

  threadgroup_t group = thread_group_alloc();

  thread_group_add(group, say, "(TG1) Hello world!", stack_base(stack_0, stack_words)),
  thread_group_add(group, say, "(TG2) Hello world!", stack_base(stack_1, stack_words));

  thread_group_start(group);

  say("(TG Master) Hello world!");

  thread_group_wait_and_free(group);
}
```

Threads can then be added using 'thread_group_add'

```
  xthread_wait_and_free(thread_1);
}

void say_hello_using_thread_group(void)
{
  char stack_0[stack_bytes] __attribute__ ((aligned (STACK_ALIGN)));
  char stack_1[stack_bytes] __attribute__ ((aligned (STACK_ALIGN)));

  threadgroup_t group = thread_group_alloc();

  thread_group_add(group, say, "(TG1) Hello world!", stack_base(stack_0, stack_words)),
  thread_group_add(group, say, "(TG2) Hello world!", stack_base(stack_1, stack_words));

  thread_group_start(group);

  say("(TG Master) Hello world!");

  thread_group_wait_and_free(group);
}

static char stack_async[stack_bytes] __attribute__ ((aligned (STACK_ALIGN)));
```

The thread functions won't begin until 'thread_group_start' is called on the group

```
{
  char stack_0[stack_bytes] __attribute__ ((aligned (STACK_ALIGN)));
  char stack_1[stack_bytes] __attribute__ ((aligned (STACK_ALIGN)));

  threadgroup_t group = thread_group_alloc();

  thread_group_add(group, say, "(TG1) Hello world!", stack_base(stack_0, stack_words)),
  thread_group_add(group, say, "(TG2) Hello world!", stack_base(stack_1, stack_words));

  thread_group_start(group);

  say("(TG Master) Hello world!");

  thread_group_wait_and_free(group);
}

static char stack_async[stack_bytes] __attribute__ ((aligned (STACK_ALIGN)));

void say_hello_asynchronously(void)
{
  run_async(say, "(Async) Hello world!", stack_base(stack_async, stack_words));
```

'thread_group_wait_and_free' waits until all threads associated with the given group have finished, it then frees the thread group

Low-level Chanend API

```c
#include <stdio.h>
#include <xcore/chanend.h>
#include <xcore/parallel.h>
#include <xcore/assert.h>

DECLARE_JOB(thread0, (chanend_t));
void thread0(chanend_t c)
{
  chanend_out_control_token(c, XS1_CT_END);
  chanend_check_control_token(c, XS1_CT_END);
  puts("0: Handshake done");
  chanend_out_byte(c, 'a');
  chanend_out_control_token(c, 12);
  chanend_out_control_token(c, 11);
  chanend_out_word(c, 0xdeadbeef);
  chanend_out_control_token(c, XS1_CT_END);
  chanend_check_control_token(c, XS1_CT_END);
  puts("0: Done!");
}

DECLARE_JOB(thread1, (chanend_t));
void thread1(chanend_t c)
{
  chanend_check_control_token(c, XS1_CT_END);
  chanend_out_control_token(c, XS1_CT_END);
  puts("1: Handshake done");
  unsigned t = chanend_test_control_token_next_byte(c);
  xassert_not(t);
  t = chanend_test_control_token_next_word(c);
  xassert(t == 2);
  char b = chanend_in_byte(c);
  printf("1: 0 sends char: %c\n", b);
  t = chanend_test_control_token_next_byte(c);
  xassert(t);
  b = chanend_in_control_token(c);
  printf("1: 0 sends control token: %d\n", (int)b);
  chanend_check_control_token(c, 11);
  uint32_t w = chanend_in_word(c);
  printf("1: 0 sends word: 0x%lx\n", w);
  chanend_check_control_token(c, XS1_CT_END);
  chanend_out_control_token(c, XS1_CT_END);
  t = chanend_test_dest_local(c);
  xassert(t);
  puts("1: Done!");
}

int main(void)
{
  chanend_t c0 = chanend_alloc(),
            c1 = chanend_alloc();

  chanend_set_dest(c0, c1);
  chanend_set_dest(chanend_get_dest(c0), c0);
```

'xcore/chanend.h' contains a low-level chanend API which can be used for implementing new channel protocols

```
  printf("1: 0 sends word: 0x%lx\n", w);
  chanend_check_control_token(c, XS1_CT_END);
  chanend_out_control_token(c, XS1_CT_END);
  t = chanend_test_dest_local(c);
  xassert(t);
  puts("1: Done!");
}

int main(void)
{
  chanend_t c0 = chanend_alloc(),
            c1 = chanend_alloc();

  chanend_set_dest(c0, c1);
  chanend_set_dest(chanend_get_dest(c0), c0);

  PAR_JOBS(
```

Individual chanends can be allocated using 'chanend_alloc'

```
  t = chanend_test_dest_local(c);
  xassert(t);
  puts("1: Done!");
}

int main(void)
{
  chanend_t c0 = chanend_alloc(),
            c1 = chanend_alloc();

  chanend_set_dest(c0, c1);
  chanend_set_dest(chanend_get_dest(c0), c0);

  PAR_JOBS(
    PJOB(thread0, (c0)),
    PJOB(thread1, (c1)));
```

Destinations can be set and read

```c
#include <stdio.h>
#include <xcore/chanend.h>
#include <xcore/parallel.h>
#include <xcore/assert.h>

DECLARE_JOB(thread0, (chanend_t));
void thread0(chanend_t c)
{
  chanend_out_control_token(c, XS1_CT_END);
  chanend_check_control_token(c, XS1_CT_END);
  puts("0: Handshake done");
  chanend_out_byte(c, 'a');
  chanend_out_control_token(c, 12);
  chanend_out_control_token(c, 11);
  chanend_out_word(c, 0xdeadbeef);
  chanend_out_control_token(c, XS1_CT_END);
  chanend_check_control_token(c, XS1_CT_END);
  puts("0: Done!");
}

DECLARE_JOB(thread1, (chanend_t));
void thread1(chanend_t c)
{
  chanend_check_control_token(c, XS1_CT_END);
  chanend_out_control_token(c, XS1_CT_END);
  puts("1: Handshake done");
  unsigned t = chanend_test_control_token_next_byte(c);
  xassert_not(t);
  t = chanend_test_control_token_next_word(c);
  xassert(t == 2);
  char b = chanend_in_byte(c);
  printf("1: 0 sends char: %c\n", b);
  t = chanend_test_control_token_next_byte(c);
  xassert(t);
  b = chanend_in_control_token(c);
  printf("1: 0 sends control token: %d\n", (int)b);
  chanend_check_control_token(c, 11);
  uint32_t w = chanend_in_word(c);
  printf("1: 0 sends word: 0x%lx\n", w);
  chanend_check_control_token(c, XS1_CT_END);
  chanend_out_control_token(c, XS1_CT_END);
  t = chanend_test_dest_local(c);
```

Functions are provided to access all chanend functionality

```
void thread0(chanend_t c)

{
  chanend_out_control_token(c, XS1_CT_END);
  chanend_check_control_token(c, XS1_CT_END);
  puts("0: Handshake done");
  chanend_out_byte(c, 'a');
  chanend_out_control_token(c, 12);
  chanend_out_control_token(c, 11);
  chanend_out_word(c, 0xdeadbeef);
  chanend_out_control_token(c, XS1_CT_END);
  chanend_check_control_token(c, XS1_CT_END);
  puts("0: Done!");
}


DECLARE_JOB(thread1, (chanend_t));
void thread1(chanend_t c)
{
  chanend_check_control_token(c, XS1_CT_END);
```

Such as sending and checking control tokens

```
{
  chanend_out_control_token(c, XS1_CT_END);
  chanend_check_control_token(c, XS1_CT_END);
  puts("0: Handshake done");
  chanend_out_byte(c, 'a');
  chanend_out_control_token(c, 12);
  chanend_out_control_token(c, 11);
  chanend_out_word(c, 0xdeadbeef);
  chanend_out_control_token(c, XS1_CT_END);
  chanend_check_control_token(c, XS1_CT_END);
  puts("0: Done!");
}

DECLARE_JOB(thread1, (chanend_t));
void thread1(chanend_t c)
{
  chanend_check_control_token(c, XS1_CT_END);
  chanend_out_control_token(c, XS1_CT_END);
  puts("1: Handshake done");
  unsigned t = chanend_test_control_token_next_byte(c);
  xassert_not(t);
  t = chanend_test_control_token_next_word(c);
  xassert(t == 2);
  char b = chanend_in_byte(c);
  printf("1: 0 sends char: %c\n", b);
  t = chanend_test_control_token_next_byte(c);
  xassert(t);
  b = chanend_in_control_token(c);
```

Sending and receiving individual bytes and words

```
DECLARE_JOB(thread1, (chanend_t));

void thread1(chanend_t c)

{

  chanend_check_control_token(c, XS1_CT_END);

  chanend_out_control_token(c, XS1_CT_END);

  puts("1: Handshake done");

  unsigned t = chanend_test_control_token_next_byte(c);

  xassert_not(t);

  t = chanend_test_control_token_next_word(c);

  xassert(t == 2);

  char b = chanend_in_byte(c);

  printf("1: 0 sends char: %c\n", b);

  t = chanend_test_control_token_next_byte(c);

  xassert(t);

  b = chanend_in_control_token(c);

  printf("1: 0 sends control token: %d\n", (int)b);

  chanend_check_control_token(c, 11);
```

Testing for buffered control tokens

```
  char b = chanend_in_byte(c);

  printf("1: 0 sends char: %c\n", b);

  t = chanend_test_control_token_next_byte(c);

  xassert(t);

  b = chanend_in_control_token(c);

  printf("1: 0 sends control token: %d\n", (int)b);

  chanend_check_control_token(c, 11);

  uint32_t w = chanend_in_word(c);

  printf("1: 0 sends word: 0x%lx\n", w);

  chanend_check_control_token(c, XS1_CT_END);

  chanend_out_control_token(c, XS1_CT_END);

  t = chanend_test_dest_local(c);

  xassert(t);

  puts("1: Done!");

}


int main(void)

{
```

And checking a chanend's destination is on the local tile

Interrupts

```c
#include <stdio.h>
#include <xcore/hwtimer.h>
#include <xcore/triggerable.h>
#include <xcore/port.h>
#include <xcore/interrupt.h>
#include <xcore/interrupt_wrappers.h>

DEFINE_INTERRUPT_PERMITTED(interrupt_handlers, void, interruptable_task, void)
{
  hwtimer_t timer = hwtimer_alloc();

  interrupt_unmask_all();
  for (;;)
  {
    puts("I'm still running.");
    hwtimer_delay(timer, 100000000);
  }
  interrupt_mask_all();

  hwtimer_free(timer);
}

DEFINE_INTERRUPT_CALLBACK(interrupt_handlers, interrupt_task, button)
{
  port_set_trigger_in_not_equal(*(port_t *)button, 1);
  printf("(%x) caused an interrupt\n", *(port_t *)button);
}

int main(void)
{
  port_t button1 = XS1_PORT_1J,
         button2 = XS1_PORT_1K;

  port_enable(button1);
  port_enable(button2);

  triggerable_setup_interrupt_callback(button1, &button1, INTERRUPT_CALLBACK(interrupt_task));
  triggerable_setup_interrupt_callback(button2, &button2, INTERRUPT_CALLBACK(interrupt_task));

  port_set_trigger_in_not_equal(button1, 1);
  port_set_trigger_in_not_equal(button2, 1);

  triggerable_enable_trigger(button1);
  triggerable_enable_trigger(button2);

  INTERRUPT_PERMITTED(interruptable_task)();

  port_disable(button1);
```

lib_xcore also supports interrupts

```c
#include <stdio.h>
#include <xcore/hwtimer.h>
#include <xcore/triggerable.h>
#include <xcore/port.h>
#include <xcore/interrupt.h>
#include <xcore/interrupt_wrappers.h>


DEFINE_INTERRUPT_PERMITTED(interrupt_handlers, void, interruptable_task, void)
{
  hwtimer_t timer = hwtimer_alloc();

  interrupt_unmask_all();
```

Functions for controlling the thread's global interrupt state are found in 'xcore/interrupt.h'

```c
#include <stdio.h>
#include <xcore/hwtimer.h>
#include <xcore/triggerable.h>
#include <xcore/port.h>
#include <xcore/interrupt.h>
#include <xcore/interrupt_wrappers.h>


DEFINE_INTERRUPT_PERMITTED(interrupt_handlers, void, interruptable_task, void)
{
  hwtimer_t timer = hwtimer_alloc();


  interrupt_unmask_all();
  for (;;)
```

'xcore/interrupt_wrappers.h' provides macros for creating interruptible functions and interrupt handlers

```
  {
    puts("I'm still running.");
    hwtimer_delay(timer, 100000000);
  }
  interrupt_mask_all();

  hwtimer_free(timer);
}

DEFINE_INTERRUPT_CALLBACK(interrupt_handlers, interrupt_task, button)
{
  port_set_trigger_in_not_equal(*(port_t *)button, 1);
  printf("(%x) caused an interrupt\n", *(port_t *)button);
}

int main(void)
{
  port_t button1 = XS1_PORT_1J,
```

'DEFINE_INTERRUPT_CALLBACK' allows creation of an interrupt handler

```
  interrupt_unmask_all();
  for (;;)
  {
    puts("I'm still running.");
    hwtimer_delay(timer, 100000000);
  }
  interrupt_mask_all();

  hwtimer_free(timer);
}

DEFINE_INTERRUPT_CALLBACK(interrupt_handlers, interrupt_task, button)
{
  port_set_trigger_in_not_equal(*(port_t *)button, 1);
  printf("(%x) caused an interrupt\n", *(port_t *)button);
}

int main(void)
{
```

Its first argument sets the group the handler belongs to for stack-size calculation purposes

```
  interrupt_unmask_all();
  for (;;)
  {
    puts("I'm still running.");
    hwtimer_delay(timer, 100000000);
  }
  interrupt_mask_all();

  hwtimer_free(timer);
}

DEFINE_INTERRUPT_CALLBACK(interrupt_handlers, interrupt_task, button)
{
  port_set_trigger_in_not_equal(*(port_t *)button, 1);
  printf("(%x) caused an interrupt\n", *(port_t *)button);
}

int main(void)
{
```

The second argument sets a base name used to retrieve the handler later

```
  for (;;)
  {
    puts("I'm still running.");
    hwtimer_delay(timer, 100000000);
  }
  interrupt_mask_all();

  hwtimer_free(timer);
}

DEFINE_INTERRUPT_CALLBACK(interrupt_handlers, interrupt_task, button)
{
  port_set_trigger_in_not_equal(*(port_t *)button, 1);
  printf("(%x) caused an interrupt\n", *(port_t *)button);
}

int main(void)
{
```

The final argument sets the name of the 'void*' argument which is passed to the interrupt handler; this can be used to implement generic handlers

```
#include <xcore/port.h>
#include <xcore/interrupt.h>
#include <xcore/interrupt_wrappers.h>


DEFINE_INTERRUPT_PERMITTED(interrupt_handlers, void, interruptable_task, void)
{
  hwtimer_t timer = hwtimer_alloc();

  interrupt_unmask_all();
  for (;;)
  {
    puts("I'm still running.");
    hwtimer_delay(timer, 100000000);
  }
  interrupt_mask_all();

  hwtimer_free(timer);
}
```

'DEFINE_INTERRUPT_PERMITTED' creates an interruptible function; the purpose of this ensures that the stack allocator sets aside enough stack space for any interrupt handler which may be invoked

```c
#include <stdio.h>
#include <xcore/hwtimer.h>
#include <xcore/triggerable.h>
#include <xcore/port.h>
#include <xcore/interrupt.h>
#include <xcore/interrupt_wrappers.h>


DEFINE_INTERRUPT_PERMITTED(interrupt_handlers, void, interruptable_task, void)
{
  hwtimer_t timer = hwtimer_alloc();


  interrupt_unmask_all();
  for (;;)
  {
    puts("I'm still running...");
```

Its first argument is the group name for the interrupt handlers which may be invoked during the execution of the interruptible function

```c
#include <stdio.h>
#include <xcore/hwtimer.h>
#include <xcore/triggerable.h>
#include <xcore/port.h>
#include <xcore/interrupt.h>
#include <xcore/interrupt_wrappers.h>


DEFINE_INTERRUPT_PERMITTED(interrupt_handlers, void, interruptable_task, void)
{
  hwtimer_t timer = hwtimer_alloc();


  interrupt_unmask_all();
  for (;;)
  {
```

The second and third arguments set the return type and base name of the function, all remaining arguments are used unaltered as an argument signature

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

#include <xcore/interrupt.h>

#include <xcore/interrupt_wrappers.h>



DEFINE_INTERRUPT_PERMITTED(interrupt_handlers, void, interruptable_task, void)

{

  hwtimer_t timer = hwtimer_alloc();


  interrupt_unmask_all();

  for (;;)

  {

    puts("I'm still running.");

    hwtimer_delay(timer, 100000000);

  }

  interrupt_mask_all();


  hwtimer_free(timer);

}
```

Within the interrupt handler, 'interrupt_unmask_all' and 'interrupt_mask_all' can be used to enable and disable interrupts respectively

```
  printf("(%x) caused an interrupt\n", *(port_t *)button);
}

int main(void)
{
  port_t button1 = XS1_PORT_1J,
         button2 = XS1_PORT_1K;

  port_enable(button1);
  port_enable(button2);

  triggerable_setup_interrupt_callback(button1, &button1, INTERRUPT_CALLBACK(interrupt_task));
  triggerable_setup_interrupt_callback(button2, &button2, INTERRUPT_CALLBACK(interrupt_task));

  port_set_trigger_in_not_equal(button1, 1);
  port_set_trigger_in_not_equal(button2, 1);

  triggerable_enable_trigger(button1);
```

We can use 'triggerable_setup_interrupt_callback' to set the interrupt handler for a resource

```
  printf("(%x) caused an interrupt\n", *(port_t *)button);
}

int main(void)
{
  port_t button1 = XS1_PORT_1J,
         button2 = XS1_PORT_1K;

  port_enable(button1);
  port_enable(button2);

  triggerable_setup_interrupt_callback(button1, &button1, INTERRUPT_CALLBACK(interrupt_task));
  triggerable_setup_interrupt_callback(button2, &button2, INTERRUPT_CALLBACK(interrupt_task));

  port_set_trigger_in_not_equal(button1, 1);
  port_set_trigger_in_not_equal(button2, 1);

  triggerable_enable_trigger(button1);
```

'INTERRUPT_CALLBACK' is used to get a pointer to an interrupt handler created with 'DEFINE_INTERRUPT_CALLBACK'

```
  printf("(%x) caused an interrupt\n", *(port_t *)button);
}

int main(void)
{
  port_t button1 = XS1_PORT_1J,
         button2 = XS1_PORT_1K;

  port_enable(button1);
  port_enable(button2);

  triggerable_setup_interrupt_callback(button1, &button1, INTERRUPT_CALLBACK(interrupt_task));
  triggerable_setup_interrupt_callback(button2, &button2, INTERRUPT_CALLBACK(interrupt_task));

  port_set_trigger_in_not_equal(button1, 1);
  port_set_trigger_in_not_equal(button2, 1);

  triggerable_enable_trigger(button1);
```

The second argument is passed to the handler's 'void*' argument when the interrupt occurs

```
int main(void)

{

  port_t button1 = XS1_PORT_1J,

         button2 = XS1_PORT_1K;


  port_enable(button1);

  port_enable(button2);


  triggerable_setup_interrupt_callback(button1, &button1, INTERRUPT_CALLBACK(interrupt_task));

  triggerable_setup_interrupt_callback(button2, &button2, INTERRUPT_CALLBACK(interrupt_task));


  port_set_trigger_in_not_equal(button1, 1);

  port_set_trigger_in_not_equal(button2, 1);


  triggerable_enable_trigger(button1);

  triggerable_enable_trigger(button2);


  INTERRUPT_PERMITTED(interruptable_task)();
```

For ports and timers, a trigger must be configured (for chanends the trigger is when data becomes available)

```
        button2 = XS1_PORT_1K;


    port_enable(button1);

    port_enable(button2);


    triggerable_setup_interrupt_callback(button1, &button1, INTERRUPT_CALLBACK(interrupt_task));

    triggerable_setup_interrupt_callback(button2, &button2, INTERRUPT_CALLBACK(interrupt_task));


    port_set_trigger_in_not_equal(button1, 1);

    port_set_trigger_in_not_equal(button2, 1);


    triggerable_enable_trigger(button1);

    triggerable_enable_trigger(button2);


    INTERRUPT_PERMITTED(interruptable_task)();


    port_disable(button1);

    port_disable(button2);
```

The trigger must be enabled for all resource types before it can generate interrupts, this is done using 'triggerable_enable_trigger'

```
    port_enable(button2);

    triggerable_setup_interrupt_callback(button1, &button1, INTERRUPT_CALLBACK(interrupt_task));
    triggerable_setup_interrupt_callback(button2, &button2, INTERRUPT_CALLBACK(interrupt_task));

    port_set_trigger_in_not_equal(button1, 1);
    port_set_trigger_in_not_equal(button2, 1);

    triggerable_enable_trigger(button1);
    triggerable_enable_trigger(button2);

    INTERRUPT_PERMITTED(interruptable_task)();

    port_disable(button1);
    port_disable(button2);
}
```

Finally, we can call the interruptible function - we retrieve its name using 'INTERRUPT_PERMITTED'

Consideration: Resource Allocation Failure

Resource Allocation Failure

- In all these examples we've assumed that resource allocation is successful;
- Any lib_xcore allocation function can fail if there aren't enough resources available;
- In this case, they'll return `0` (or both ends will be `0` in the case of channels);
- Usually if an allocation fails a subsequent operation on that resource will trap

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

#include <xcore/lock.h>


void my_function(lock_t, channel_t);


int main(void)

{

  lock_t l = lock_alloc();

  channel_t c = chan_alloc();



  //Resources might be invalid!

  my_function(l, c);



  lock_free(l);

  chan_free(c);

}
```

Often it is appropriate to check that resource allocation was successful;

You can simply check that the result of the allocation is nonzero...

```c
#include <xcore/channel.h>
#include <xcore/lock.h>

void my_function(lock_t, channel_t);

int main(void)
{
  lock_t l = lock_alloc();
  channel_t c = chan_alloc();

  if (l == 0 || c.end_a == 0) { return 1; }

  my_function(l, c);

  lock_free(l);
  chan_free(c);
}
```

... Or just assert on the resource handle

```c
#include <xcore/assert.h>
#include <xcore/channel.h>
#include <xcore/lock.h>

void my_function(lock_t, channel_t);

int main(void)
{
  lock_t l = lock_alloc();
  channel_t c = chan_alloc();

  xassert(l);
  xassert(c.end_a);

  my_function(l, c);

  lock_free(l);
  chan_free(c);
}
```

Consideration: Code Optimisation

```
void main_pipe(chanend_t led_chanend, chanend_t button_chanend)
{
  int button_up = 1;
  int button_event_count = 0;

  SELECT_RES_ORDERED(
    CASE_GUARD_THEN(button_chanend, button_event_count < 21, event_button_chanend),
    CASE_THEN(led_chanend, event_led_chanend),
    DEFAULT_NGUARD_THEN(button_up, default_label))

  {
  default_label:
    puts("Button is still down!");
    SELECT_RESET;

  event_button_chanend:
    {
      uint32_t tmp = chan_in_word(button_chanend);
      chan_out_word(led_chanend, tmp);
      button_up = tmp;
    }
    button_event_count += 1;
    continue;

  event_led_chanend:
    {
      uint32_t tmp = chan_in_word(led_chanend);
      chan_out_word(button_chanend, tmp);
    }
    continue;
  }
```

```
<main_pipe>:
        0x000102b0: 4b 77:      entsp (u6)      0xb
        0x000102b2: 0a 54:      stw (ru6)       r0, sp[0xa]
        0x000102b4: 49 54:      stw (ru6)       r1, sp[0x9]
        0x000102b6: d1 a6:      mkmsk (ru6)     r0, 0x1
        0x000102b8: 08 54:      stw (ru6)       r0, sp[0x8]
        0x000102ba: 40 68:      ldc (ru6)       r1, 0x0
        0x000102bc: 47 54:      stw (ru6)       r1, sp[0x7]
        0x000102be: 33 70:      bt (ru6)        r0, 0x33 <.label163>
        0x000102c0: 33 73:      bu (u6)         0x33 <.label166>
.label168   0x000102c2: 40 68:      ldc (ru6)       r0, 0x0
        0x000102c4: 2c 70:      bt (ru6)        r0, 0x2c <.label161>
        0x000102c6: 00 73:      bu (u6)         0x0 <.label166>
.label166   0x000102c8: 00 f0 be d1: bl (lu10)      0x1be <__xcore_select_disable_trigger_all>
        0x000102cc: 00 73:      bu (u6)         0x0 <.label167>
.label167   0x000102ce: 07 5c:      ldw (ru6)       r0, sp[0x7]
        0x000102d0: 55 68:      ldc (ru6)       r1, 0x15
        0x000102d2: 01 c0:      lsx (3r)        r0, r0, r1
        0x000102d4: 05 54:      stw (ru6)       r0, sp[0x5]
        0x000102d6: 09 5c:      ldw (ru6)       r0, sp[0x9]
        0x000102d8: 45 5c:      ldw (ru6)       r1, sp[0x5]
        0x000102da: 00 f0 0f d1: bl (lu10)      0x10f <__xcore_resource_event_enable_if_true>
        0x000102de: 41 7b:      setsr (u6)      0x1
        0x000102e0: 01 7b:      clrsr (u6)      0x1
        0x000102e2: 00 73:      bu (u6)         0x0 <.label168>
.label168   0x000102e4: 00 73:      bu (u6)         0x0 <.label169>
.label169   0x000102e6: 0a 5c:      ldw (ru6)       r0, sp[0xa]
        0x000102e8: 00 f0 f4 d0: bl (lu10)      0xf4 <__xcore_resource_event_enable_unconditional>
        0x000102ec: 41 7b:      setsr (u6)      0x1
        0x000102ee: 01 7b:      clrsr (u6)      0x1
        0x000102f0: 00 73:      bu (u6)         0x0 <.label170>
.label170   0x000102f2: 00 73:      bu (u6)         0x0 <.label171>
.label171   0x000102f4: 08 5c:      ldw (ru6)       r0, sp[0x8]
        0x000102f6: 04 54:      stw (ru6)       r0, sp[0x4]
        0x000102f8: d1 a6:      mkmsk (ru6)     r0, 0x1
        0x000102fa: 0a 70:      bt (ru6)        r0, 0xa <.label172>
        0x000102fc: 00 73:      bu (u6)         0x0 <.label173>
.label173   0x000102fe: 04 5c:      ldw (ru6)       r0, sp[0x4]
        0x00010300: 03 70:      bf (ru6)        r0, 0x3 <.label174>
        0x00010302: 00 73:      bu (u6)         0x0 <.label175>
.label175   0x00010304: ec 07:      waiteu (0r)
        0x00010306: 00 73:      bu (u6)         0x0 <.label174>
.label174   0x00010308: 41 7b:      setsr (u6)      0x1
        0x0001030a: 01 7b:      clrsr (u6)      0x1
        0x0001030c: 00 73:      bu (u6)         0x0 <.label176>
.label176   0x0001030e: 1b 73:      bu (u6)         0x1b <.label177>
.label172   0x00010310: 41 7b:      setsr (u6)      0x1
        0x00010312: 01 7b:      clrsr (u6)      0x1
        0x00010314: 00 73:      bu (u6)         0x0 <.label178>
.label178   0x00010316: 04 5c:      ldw (ru6)       r0, sp[0x4]
        0x00010318: e0 0f:      waitet (1r)     r0
        0x0001031a: 00 73:      bu (u6)         0x0 <.label179>
.label179   0x0001031c: 14 73:      bu (u6)         0x14 <.label177>
.label165   0x0001031e: d1 a6:      mkmsk (ru6)     r0, 0x1
        0x00010320: 11 70:      bt (ru6)        r0, 0x11 <.label180>
        0x00010322: 00 73:      bu (u6)         0x0 <.label181>
.label181   0x00010324: 00 73:      bu (u6)         0x0 <.label163>
.label163   0x00010326: 00 73:      bu (u6)         0x0 <.label164>
.label164   0x00010328: 02 d8:      ldap (u10)      r11, 0x2 <.label182>
        0x0001032a: c6 54:      stw (ru6)       r11, sp[0x6]
        0x0001032c: 00 73:      bu (u6)         0x0 <.label182>
.label182   0x0001032e: 09 5c:      ldw (ru6)       r0, sp[0x9]
        0x00010330: 13 d8:      ldap (u10)      r11, 0x13 <.label183>
        0x00010332: 9c 91:      add (2ru6)      r1, r11, 0x0
        0x00010334: 00 f0 fa d0: bl (lu10)      0xfa <__xcore_resource_register_event_vector>
        0x00010338: 0a 5c:      ldw (ru6)       r0, sp[0xa]
        0x0001033a: 1d d8:      ldap (u10)      r11, 0x1d <.label184>
        0x0001033c: 9c 91:      add (2ru6)      r1, r11, 0x0
        0x0001033e: 00 f0 f5 d0: bl (lu10)      0xf5 <__xcore_resource_register_event_vector>
        0x00010342: 23 73:      bu (u6)         0x23 <.label185>
.label180   0x00010344: 00 73:      bu (u6)         0x0 <.label177>
.label177   0x00010346: 00 f0 4b 7f: ldaw (lu6)     r11, sp[0xb]
        0x0001034a: 8c 91:      add (2ru6)      r0, r11, 0x0
        0x0001034c: 00 f0 7e d1: bl (lu10)      0x17e <puts>
        0x00010350: 00 73:      bu (u6)         0x0 <.label186>
.label186   0x00010352: 06 5c:      ldw (ru6)       r0, sp[0x6]
        0x00010354: 01 54:      stw (ru6)       r0, sp[0x1]
        0x00010356: 1c 73:      bu (u6)         0x1c <.label187>
.label183   0x00010358: 09 5c:      ldw (ru6)       r0, sp[0x9]
        0x0001035a: 00 f0 21 d1: bl (lu10)      0x121 <chan_in_word>
        0x0001035e: 03 54:      stw (ru6)       r0, sp[0x3]
        0x00010360: 0a 5c:      ldw (ru6)       r0, sp[0xa]
        0x00010362: 43 5c:      ldw (ru6)       r1, sp[0x3]
        0x00010364: 00 f0 88 d1: bl (lu10)      0x188 <chan_out_word>
        0x00010368: 03 5c:      ldw (ru6)       r0, sp[0x3]
        0x0001036a: 08 54:      stw (ru6)       r0, sp[0x8]
        0x0001036c: 07 5c:      ldw (ru6)       r0, sp[0x7]
        0x0001036e: 01 90:      add (2ru6)      r0, r0, 0x1
        0x00010370: 07 54:      stw (ru6)       r0, sp[0x7]
```

As C for xCore makes function calls where before there was language support, the compiler often produces slower code than for XC at low optimisation levels

-O1

```
void main_pipe(chanend_t led_chanend, chanend_t button_chanend)
{
  int button_up = 1;
  int button_event_count = 0;

  SELECT_RES_ORDERED(
    CASE_GUARD_THEN(button_chanend, button_event_count < 21, event_button_chanend),
    CASE_THEN(led_chanend, event_led_chanend),
    DEFAULT_NGUARD_THEN(button_up, default_label))

  {
  default_label:
    puts("Button is still down!");
    SELECT_RESET;

  event_button_chanend:
    {
      uint32_t tmp = chan_in_word(button_chanend);
      chan_out_word(led_chanend, tmp);
      button_up = tmp;
    }
    button_event_count += 1;
    continue;

  event_led_chanend:
    {
      uint32_t tmp = chan_in_word(led_chanend);
      chan_out_word(button_chanend, tmp);
    }
    continue;
  }
```

```
<main_pipe>:
           0x00010280: 48 77:      entsp (u6)      0x8
           0x00010282: 07 55:      stw (ru6)       r4, sp[0x7]
           0x00010284: 46 55:      stw (ru6)       r5, sp[0x6]
           0x00010286: 85 55:      stw (ru6)       r6, sp[0x5]
           0x00010288: c4 55:      stw (ru6)       r7, sp[0x4]
           0x0001028a: 03 56:      stw (ru6)       r8, sp[0x3]
           0x0001028c: 42 56:      stw (ru6)       r9, sp[0x2]
           0x0001028e: 81 56:      stw (ru6)       r10, sp[0x1]

           0x00010290: 44 90:      add (2rus)      r4, r1, 0x0
           0x00010292: 50 90:      add (2rus)      r5, r0, 0x0
           0x00010294: 19 a7:      mkmsk (rus)     r6, 0x1
           0x00010296: 00 6a:      ldc (ru6)       r8, 0x0
           0x00010298: 55 6a:      ldc (ru6)       r9, 0x15
           0x0001029a: 20 92:      add (2rus)      r10, r8, 0x0
.label63   0x0001029c: 0a d8:      ldap (u10)      r11, 0xa <.label55>
           0x0001029e: 9c 91:      add (2rus)      r1, r11, 0x0
           0x000102a0: c0 90:      add (2rus)      r0, r4, 0x0
           0x000102a2: 00 f0 cb d0: bl (lu10)      0xcb <__xcore_resource_register_event_vector>
           0x000102a6: 10 d8:      ldap (u10)      r11, 0x10 <.label56>
           0x000102a8: 9c 91:      add (2rus)      r1, r11, 0x0
           0x000102aa: c4 90:      add (2rus)      r0, r5, 0x0
           0x000102ac: 00 f0 c6 d0: bl (lu10)      0xc6 <__xcore_resource_register_event_vector>
           0x000102b0: 09 73:      bu (u6)         0x9 <.label57>
.label55   0x000102b2: c0 90:      add (2rus)      r0, r4, 0x0
           0x000102b4: 00 f0 e6 d0: bl (lu10)      0xe6 <chan_in_word>
           0x000102b8: 60 90:      add (2rus)      r6, r0, 0x0
           0x000102ba: c4 90:      add (2rus)      r0, r5, 0x0
           0x000102bc: d8 90:      add (2rus)      r1, r6, 0x0
           0x000102be: 00 f0 c9 d0: bl (lu10)      0xc9 <chan_out_word>
           0x000102c2: 29 92:      add (2rus)      r10, r10, 0x1
.label57   0x000102c4: 79 c6:      lss (3r)        r7, r10, r9
           0x000102c6: 07 73:      bu (u6)         0x7 <.label58>
.label56   0x000102c8: c4 90:      add (2rus)      r0, r5, 0x0
           0x000102ca: 00 f0 db d0: bl (lu10)      0xdb <chan_in_word>
           0x000102ce: 10 90:      add (2rus)      r1, r0, 0x0
           0x000102d0: c0 90:      add (2rus)      r0, r4, 0x0
           0x000102d2: 00 f0 bf d0: bl (lu10)      0xbf <chan_out_word>
.label58   0x000102d6: 00 f0 09 d1: bl (lu10)      0x109 <__xcore_select_disable_trigger_all>
           0x000102da: c0 90:      add (2rus)      r0, r4, 0x0
           0x000102dc: dc 90:      add (2rus)      r1, r7, 0x0
           0x000102de: 00 f0 a9 d0: bl (lu10)      0xa9 <__xcore_resource_event_enable_if_true>
           0x000102e2: 41 7b:      setsr (u6)      0x1
           0x000102e4: 01 7b:      clrsr (u6)      0x1
           0x000102e6: 00 73:      bu (u6)         0x0 <.label59>
.label59   0x000102e8: c4 90:      add (2rus)      r0, r5, 0x0
           0x000102ea: 00 f0 9f d0: bl (lu10)      0x9f <__xcore_resource_event_enable_unconditional>
           0x000102ee: 41 7b:      setsr (u6)      0x1
           0x000102f0: 01 7b:      clrsr (u6)      0x1
           0x000102f2: 00 73:      bu (u6)         0x0 <.label60>
.label60   0x000102f4: 41 7b:      setsr (u6)      0x1
           0x000102f6: 01 7b:      clrsr (u6)      0x1
           0x000102f8: 00 73:      bu (u6)         0x0 <.label61>
.label61   0x000102fa: e6 bf:      waiteu (r)      r6
           0x000102fc: 00 73:      bu (u6)         0x0 <.label62>
```

However, lib_xcore is highly optimisable and almost all functions are inlinable

-O2

```
void main_pipe(chanend_t led_chanend, chanend_t button_chanend)
{
  int button_up = 1;
  int button_event_count = 0;

  SELECT_RES_ORDERED(
    CASE_GUARD_THEN(button_chanend, button_event_count < 21, event_button_chanend),
    CASE_THEN(led_chanend, event_led_chanend),
    DEFAULT_NGUARD_THEN(button_up, default_label))

  {
  default_label:
    puts("Button is still down!");
    SELECT_RESET;

  event_button_chanend:
    {
      uint32_t tmp = chan_in_word(button_chanend);
      chan_out_word(led_chanend, tmp);
      button_up = tmp;
    }
    button_event_count += 1;
    continue;

  event_led_chanend:
    {
      uint32_t tmp = chan_in_word(led_chanend);
      chan_out_word(button_chanend, tmp);
    }
    continue;
  }
```

```
<main_pipe>:
           0x0001026c: 46 77:      entsp (u6)      0x6
           0x0001026e: 05 55:      stw (ru6)       r4, sp[0x5]
           0x00010270: 44 55:      stw (ru6)       r5, sp[0x4]
           0x00010272: 83 55:      stw (ru6)       r6, sp[0x3]
           0x00010274: c2 55:      stw (ru6)       r7, sp[0x2]
           0x00010276: 01 56:      stw (ru6)       r8, sp[0x1]
           0x00010278: 44 90:      add (2rus)      r4, r1, 0x0
           0x0001027a: 50 90:      add (2rus)      r5, r0, 0x0

           0x0001027c: d1 a6:      mkmsk (rus)     r0, 0x1
           0x0001027e: 80 69:      ldc (ru6)       r6, 0x0
           0x00010280: d5 69:      ldc (ru6)       r7, 0x15
           0x00010282: 48 91:      add (2rus)      r8, r6, 0x0
.label62   0x00010284: 04 d8:      ldap (u10)      r11, 0x4 <.label154>
           0x00010286: f4 47:      setv (1r)       res[r4], r11
           0x00010288: 0f d8:      ldap (u10)      r11, 0xf <.label155>
           0x0001028a: f5 47:      setv (1r)       res[r5], r11
           0x0001028c: 0b 73:      bu (u6)         0xb <.label156>
.label154  0x0001028e: 11 cf:      chkct (rus)     res[r4], 0x1
           0x00010290: 11 4f:      outct (rus)     res[r4], 0x1
           0x00010292: 80 b7:      in (2r)         r0, res[r4]
           0x00010294: 11 cf:      chkct (rus)     res[r4], 0x1
           0x00010296: 11 4f:      outct (rus)     res[r4], 0x1
           0x00010298: 15 4f:      outct (rus)     res[r5], 0x1
           0x0001029a: 15 cf:      chkct (rus)     res[r5], 0x1
           0x0001029c: 81 af:      out (r2r)       res[r5], r0
           0x0001029e: 15 4f:      outct (rus)     res[r5], 0x1
           0x000102a0: 15 cf:      chkct (rus)     res[r5], 0x1
           0x000102a2: 01 92:      add (2rus)      r8, r8, 0x1
.label156  0x000102a4: d3 c3:      lss (3r)        r1, r8, r7
           0x000102a6: 0a 73:      bu (u6)         0xa <.label157>
.label155  0x000102a8: 15 cf:      chkct (rus)     res[r5], 0x1
           0x000102aa: 15 4f:      outct (rus)     res[r5], 0x1
           0x000102ac: 89 b7:      in (2r)         r2, res[r5]
           0x000102ae: 15 cf:      chkct (rus)     res[r5], 0x1
           0x000102b0: 15 4f:      outct (rus)     res[r5], 0x1
           0x000102b2: 11 4f:      outct (rus)     res[r4], 0x1
           0x000102b4: 11 cf:      chkct (rus)     res[r4], 0x1
           0x000102b6: 88 af:      out (r2r)       res[r4], r2
           0x000102b8: 11 4f:      outct (rus)     res[r4], 0x1
           0x000102ba: 11 cf:      chkct (rus)     res[r4], 0x1
.label157  0x000102bc: ed 07:      clre (0r)
           0x000102be: 94 27:      eet (2r)        r1, res[r4]
           0x000102c0: 41 7b:      setsr (u6)      0x1
           0x000102c2: 01 7b:      clrsr (u6)      0x1
           0x000102c4: 00 73:      bu (u6)         0x0 <.label158>
.label158  0x000102c6: f5 07:      eeu (1r)        res[r5]
           0x000102c8: 41 7b:      setsr (u6)      0x1
           0x000102ca: 01 7b:      clrsr (u6)      0x1
           0x000102cc: 00 73:      bu (u6)         0x0 <.label159>
.label159  0x000102ce: 41 7b:      setsr (u6)      0x1
           0x000102d0: 01 7b:      clrsr (u6)      0x1
           0x000102d2: 00 73:      bu (u6)         0x0 <.label60>
.label60   0x000102d4: e0 0f:      waiteт (1r)     r0
           0x000102d6: 00 73:      bu (u6)         0x0 <.label61>
.label61   0x000102d8: 80 f0 4b 7f: ldaw (lu6)      r11, cp[0x4b]
```

Under modern Clang, code using lib_xcore often outperforms equivalent XC code compiled with XCC

-O3

```
void main_pipe(chanend_t led_chanend, chanend_t button_chanend)
{
  int button_up = 1;
  int button_event_count = 0;

  SELECT_RES_ORDERED(
    CASE_GUARD_THEN(button_chanend, button_event_count < 21, event_button_chanend),
    CASE_THEN(led_chanend, event_led_chanend),
    DEFAULT_NGUARD_THEN(button_up, default_label))

  {
  default_label:
    puts("Button is still down!");
    SELECT_RESET;

  event_button_chanend:
    {
      uint32_t tmp = chan_in_word(button_chanend);
      chan_out_word(led_chanend, tmp);
      button_up = tmp;
    }
    button_event_count += 1;
    continue;

  event_led_chanend:
    {
      uint32_t tmp = chan_in_word(led_chanend);
      chan_out_word(button_chanend, tmp);
    }
    continue;
  }
```

```
<main_pipe>:
            0x0001026c: 46 77:      entsp (u6)      0x6
            0x0001026e: 05 55:      stw (ru6)       r4, sp[0x5]
            0x00010270: 44 55:      stw (ru6)       r5, sp[0x4]
            0x00010272: 83 55:      stw (ru6)       r6, sp[0x3]
            0x00010274: c2 55:      stw (ru6)       r7, sp[0x2]
            0x00010276: 01 56:      stw (ru6)       r8, sp[0x1]
            0x00010278: 44 90:      add (2rus)      r4, r1, 0x0
            0x0001027a: 50 90:      add (2rus)      r5, r0, 0x0

            0x0001027c: d1 a6:      mkmsk (rus)     r0, 0x1
            0x0001027e: 80 69:      ldc (ru6)       r6, 0x0
            0x00010280: d5 69:      ldc (ru6)       r7, 0x15
            0x00010282: 48 91:      add (2rus)      r8, r6, 0x0
.label161   0x00010284: 04 d8:      ldap (u10)      r11, 0x4 <.label154>
            0x00010286: f4 47:      setv (1r)       res[r4], r11
            0x00010288: 13 d8:      ldap (u10)      r11, 0x13 <.label155>
            0x0001028a: f5 47:      setv (1r)       res[r5], r11
            0x0001028c: 0b 73:      bu (u6)         0xb <.label156>
.label154   0x0001028e: 11 cf:      chkct (rus)     res[r4], 0x1
            0x00010290: 11 4f:      outct (rus)     res[r4], 0x1
            0x00010292: 80 b7:      in (2r)         r0, res[r4]
            0x00010294: 11 cf:      chkct (rus)     res[r4], 0x1
            0x00010296: 11 4f:      outct (rus)     res[r4], 0x1
            0x00010298: 15 4f:      outct (rus)     res[r5], 0x1
            0x0001029a: 15 cf:      chkct (rus)     res[r5], 0x1
            0x0001029c: 81 af:      out (r2r)       res[r5], r0
            0x0001029e: 15 4f:      outct (rus)     res[r5], 0x1
            0x000102a0: 15 cf:      chkct (rus)     res[r5], 0x1
            0x000102a2: 01 92:      add (2rus)      r8, r8, 0x1
.label156   0x000102a4: d3 c3:      lss (3r)        r1, r8, r7
            0x000102a6: ed 07:      clre (0r)
            0x000102a8: 94 27:      eet (2r)        r1, res[r4]
            0x000102aa: 41 7b:      setsr (u6)      0x1
            0x000102ac: 01 7b:      clrsr (u6)      0x1
            0x000102ae: 0f 73:      bu (u6)         0xf <.label157>
.label155   0x000102b0: 15 cf:      chkct (rus)     res[r5], 0x1
            0x000102b2: 15 4f:      outct (rus)     res[r5], 0x1
            0x000102b4: 89 b7:      in (2r)         r2, res[r5]
            0x000102b6: 15 cf:      chkct (rus)     res[r5], 0x1
            0x000102b8: 15 4f:      outct (rus)     res[r5], 0x1
            0x000102ba: 11 4f:      outct (rus)     res[r4], 0x1
            0x000102bc: 11 cf:      chkct (rus)     res[r4], 0x1
            0x000102be: 88 af:      out (r2r)       res[r4], r2
            0x000102c0: 11 4f:      outct (rus)     res[r4], 0x1
            0x000102c2: 11 cf:      chkct (rus)     res[r4], 0x1
            0x000102c4: ed 07:      clre (0r)
            0x000102c6: 94 27:      eet (2r)        r1, res[r4]
            0x000102c8: 41 7b:      setsr (u6)      0x1
            0x000102ca: 01 7b:      clrsr (u6)      0x1
            0x000102cc: 00 73:      bu (u6)         0x0 <.label157>
.label157   0x000102ce: f5 07:      eeu (1r)        res[r5]
            0x000102d0: 41 7b:      setsr (u6)      0x1
            0x000102d2: 01 7b:      clrsr (u6)      0x1
            0x000102d4: 00 73:      bu (u6)         0x0 <.label158>
.label158   0x000102d6: 41 7b:      setsr (u6)      0x1
            0x000102d8: 01 7b:      clrsr (u6)      0x1
            0x000102da: 00 73:      bu (u6)         0x0 <.label159>
.label159   0x000102dc: e0 0f:      waitet (1r)     r0
            0x000102de: 00 73:      bu (u6)         0x0 <.label160>
.label160   0x000102e0: 00 f0 4b 7f: ldaw (lu6)     r11, cp[0xb]
```

Getting lib_xcore

Available in the latest tools! lib_xcore is a system library and linked by default

API is being continuously expanded - please feed back any issues or suggestions

There are some limitations...

C/lib_xcore limitations

- There is no `[[combinable]]` - Select blocks must be created manually;
- Non-memory resource calculation is not available - this will be omitted from the resource report
- XC-style interfaces are not available - explicit channels should be used instead

Documentation

- API is fully Doxygen annotated;
- These slides

Automatic Stack Size Calculation

Automatic Stack Size Calculation

A standard approach to stack allocation is to assign a block of memory at link-time.

xCore applications usually have several threads; often many transitory threads will be launched throughout execution.

This means that manually assigning stacks to threads can be difficult.

To simplify this, the xCore compilers attempt to calculate stack requirements of applications.

There are some cases where this isn't possible for normal C code. By far the most common such case is indirect calls.

For this reason, function pointers must be annotated so that the compiler can calculate the stack footprint of calls through them.

Example

```c
#include <stdio.h>

void func1(void)
{
  char cs[64]; cs[0] = 0;
  puts("Hello from func1!");
}


void func2(void)
{
  char cs[256]; cs[0] = 0;
  puts("Hello from func2");
}

int main(void)
{
  void(*fp)(void) = func1;
  fp();
  fp = func2;
  fp();
```

```
$ xcc -target=XCORE-200-EXPLORER fpointers/fpointer0.c -report

fpointers/fpointer0.c:18:3: warning: Dereferencing a function pointer whose declaration has no fptrgroup. The stack size will not be calculable. [-Wxcore-fptrgroup]

  fp();

  ^

fpointers/fpointer0.c:20:3: warning: Dereferencing a function pointer whose declaration has no fptrgroup. The stack size will not be calculable. [-Wxcore-fptrgroup]

  fp();

  ^


2 warnings generated.

Constraint check for tile[0]:

  Memory available:       262144,   used:       4520+.  MAYBE

    (Stack: 348+, Code: 3656, Data: 516)

Constraints checks PASSED WITH CAVEATS.
```

```
$ xcc -target=XCORE-200-EXPLORER fpointers/fpointer0.c -report
fpointers/fpointer0.c:18:3: warning: Dereferencing a function pointer whose declaration has no fptrgroup. The stack size will not be calculable. [-Wxcore-fptrgroup]
  fp();
  ^
fpointers/fpointer0.c:20:3: warning: Dereferencing a function pointer whose declaration has no fptrgroup. The stack size will not be calculable. [-Wxcore-fptrgroup]
  fp();
  ^

2 warnings generated.
Constraint check for tile[0]:
  Memory available:       262144,   used:       4520+.  MAYBE
    (Stack: 348+, Code: 3656, Data: 516)
Constraints checks PASSED WITH CAVEATS.
```

XCC can't calculate resource requirements for unannotated function pointers

```
$ xcc -target=XCORE-200-EXPLORER fpointers/fpointer0.c -report
fpointers/fpointer0.c:18:3: warning: Dereferencing a function pointer whose declaration has no fptrgroup. The stack size will not be calculable. [-Wxcore-fptrgroup]
  fp();
  ^
fpointers/fpointer0.c:20:3: warning: Dereferencing a function pointer whose declaration has no fptrgroup. The stack size will not be calculable. [-Wxcore-fptrgroup]
  fp();
  ^

2 warnings generated.
Constraint check for tile[0]:
  Memory available:       262144,   used:       4520+.  MAYBE
    (Stack: 348+, Code: 3656, Data: 516)
Constraints checks PASSED WITH CAVEATS.
```

As a consequence, xMap doesn't know how much stack to allocate

```c
#include <stdio.h>

void func1(void)
{
  char cs[64]; cs[0] = 0;
  puts("Hello from func1!");
}

void func2(void)
{
  char cs[256]; cs[0] = 0;
  puts("Hello from func2");
}

int main(void)
{
  void(*fp)(void) = func1;
  fp();
  fp = func2;
  fp();
```

In order for the tools to determine stack requirements, we need to tell the compiler which functions our function pointer might point to

```c
#include <stdio.h>

__attribute__(( fptrgroup("my_functions") ))
void func1(void)
{
  char cs[64]; cs[0] = 0;
  puts("Hello from func1!");

}
__attribute__(( fptrgroup("my_functions") ))
void func2(void)
{
  char cs[256]; cs[0] = 0;
  puts("Hello from func2");
}

int main(void)
{
```

We do this by annotating the C code

```
 8 }
 9 __attribute__(( fptrgroup("my_functions") ))
10 void func2(void)
11 {
12   char cs[256]; cs[0] = 0;
13   puts("Hello from func2");
14 }
15
16 int main(void)
17 {
18   __attribute__(( fptrgroup("my_functions") )) void(*fp)(void) = func1;
19   fp();
20   fp = func2;
21   fp();
22 }
```

Each function pointer is annotated with a 'function pointer group', using the 'fptrgroup' attribute

```
 8 }
 9 __attribute__(( fptrgroup("my_functions") ))
10 void func2(void)
11 {
12   char cs[256]; cs[0] = 0;
13   puts("Hello from func2");
14 }
15
16 int main(void)
17 {
18   __attribute__(( fptrgroup("my_functions") )) void(*fp)(void) = func1;
19   fp();
20   fp = func2;
21   fp();
22 }
```

This tells the compiler that 'fp' will point to a function in a function pointer group named 'my_functions', but we also need to tell it which functions belong to that group

```c
#include <stdio.h>

__attribute__(( fptrgroup("my_functions") ))
void func1(void)
{
  char cs[64]; cs[0] = 0;
  puts("Hello from func1!");

}
__attribute__(( fptrgroup("my_functions") ))
void func2(void)
{
  char cs[256]; cs[0] = 0;
```

Each function needs to be annotated with the group (or groups) it belongs to, again using the 'fptrgroup' attribute

```c
#include <stdio.h>

__attribute__(( fptrgroup("my_functions") ))
void func1(void)
{
  char cs[64]; cs[0] = 0;
  puts("Hello from func1!");

}
__attribute__(( fptrgroup("my_functions") ))
void func2(void)
{
  char cs[256]; cs[0] = 0;
  puts("Hello from func2");
}

int main(void)
{
```

The tools can now determine that the worst-case stack requirement for a call through 'fp' is greatest stack requirement of 'func1' and 'func2'

```c
#include <stdio.h>

__attribute__(( fptrgroup("my_functions") ))
void func1(void)
{
  char cs[64]; cs[0] = 0;
  puts("Hello from func1!");

}
__attribute__(( fptrgroup("my_functions") ))
void func2(void)
{
  char cs[256]; cs[0] = 0;
  puts("Hello from func2");
}

int main(void)
{
```

Care must be taken to ensure that all possible callees are added to a pointer's group

```c
#include <stdio.h>

__attribute__(( fptrgroup("my_functions") ))
void func1(void)
{
  char cs[64]; cs[0] = 0;
  puts("Hello from func1!");

}
__attribute__(( fptrgroup("my_typo") ))
void func2(void)
{
  char cs[256]; cs[0] = 0;
  puts("Hello from func2");
}

```

This not-so-subtle typo will prevent the compiler considering the function a possible pointee of our pointer

```
$ xcc -target=XCORE-200-EXPLORER fpointers/fpointer2.c -report

Constraint check for tile[0]:

  Memory available:       262144,   used:       4576 .  OKAY

    (Stack: 404, Code: 3656, Data: 516)

Constraints checks PASSED.
```

This is likely to result in an insufficient stack allocation (note that nothing warned us that our errant group was never used)

```c
#include <stdio.h>

__attribute__(( fptrgroup("my_functions") ))
void func1(void)
{
  char cs[64]; cs[0] = 0;
  puts("Hello from func1!");
}
__attribute__(( fptrgroup("my_functions") ))
void func2(void)
{
  char cs[256]; cs[0] = 0;
  puts("Hello from func2");
}

int main(void)
{
  __attribute__(( fptrgroup("my_functions") )) void(*fp)(void) = func1;
  fp();
  fp = func2;
  fp();
}
```

In case of mistakes, we can enable runtime checking of function pointer group membership

```
 8 }
 9 __attribute__(( fptrgroup("my_functions") ))
10 void func2(void)
11 {
12   char cs[256]; cs[0] = 0;
13   puts("Hello from func2");
14 }
15
16 int main(void)
17 {
18   __attribute__(( fptrgroup("my_functions", 1) )) void(*fp)(void) = func1;
19   fp();
20   fp = func2;
21   fp();
22 }
```

This causes the compiler to emit additional code which makes sure that the value of 'fp' is known to be a member of its group

```c
#include <stdio.h>

__attribute__(( fptrgroup("my_functions") ))
void func1(void)
{
  char cs[64]; cs[0] = 0;
  puts("Hello from func1!");
}
__attribute__(( fptrgroup("my_typo") ))
void func2(void)
{
  char cs[256]; cs[0] = 0;
  puts("Hello from func2");
}

int main(void)
```

So this sort of mistake would be a runtime error

```
$ xsim fpointers/fpointer5.xe

Hello from func1!

Unhandled exception: ECALL, data: 0x00000000
```

With runtime checking enabled we'll get a trap if an unknown function is called through our pointer

```c
#include <stdio.h>

void func1(void)
{
  char cs[64]; cs[0] = 0;
  puts("Hello from func1!");
}


void func2(void)
{
  char cs[256]; cs[0] = 0;
  puts("Hello from func2");
}

int main(void)
{
  void(*fp)(void) = func1;
  fp();
  fp = func2;
  fp();
```

We can set a function group on our pointer

```
 7 }

 8

 9 void func2(void)
10 {
11   char cs[256]; cs[0] = 0;
12   puts("Hello from func2");
13 }
14
15 int main(void)
16 {
17   __attribute__(( fptrgroup("my_functions") )) void(*fp)(void) = func1;
18   fp();
19   fp = func2;
20   fp();
21 }
```

This tells the compiler which functions the pointer might refer to so it can calculate worst-case resource requirements

Setting Stack Size Manually

```
.globl main.nstackwords

.set main.nstackwords,1024
```

You can manually provide the stack requirement for a function by setting its 'nstackwords'; note that this must be enough for all callees and child threads

```
.globl main.nstackwords

.set main.nstackwords,1024
```

In this case we're setting the requirement for the whole entrypoint, but this works for any nonstatic function

```
.globl main.nstackwords

.set main.nstackwords, 1024 + my_function0.nstackwords
```

It's possible to reference other symbols - this is useful to express stack size in terms of other functions' requirements

```
.globl main.nstackwords

.set main.nstackwords, 1024 + (my_function0.nstackwords $M my_function1.nstackwords)
```

The $M operator finds the worst-case of its two arguments - this is useful for allocating enough stack for the hungriest callee

```
.globl main.nstackwords

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

The stack size must be a multiple of the stack size alignment in words; the $A operator will round up its LHS argument to a multiple of the RHS

```
$ xcc -target=PHOENIX main.c stack3.s -report

main.c:17:3: warning: Dereferencing a function pointer whose declaration has no fptrgroup. The stack size will not be calculable. [-Wxcore-fptrgroup]

  fp();

  ^

1 warning generated.

Constraint check for tile[0]:

  Memory available:       524288,   used:       5024 .  OKAY


    (Stack: 4428, Code: 516, Data: 80)

Constraints checks PASSED WITH CAVEATS.
```

You can provide this at build time, but you'll still get a warning about function pointers if you've used them

```
$ xcc -target=PHOENIX main.c stack3.s -report -Wno-xcore-fptrgroup

Constraint check for tile[0]:

  Memory available:       524288,   used:       5024 .  OKAY

    (Stack: 4428, Code: 516, Data: 80)

Constraints checks PASSED WITH CAVEATS.
```

This can be suppressed for the whole build with the '-Wno-xcore-fptrgroup' option

Multi-tile applications

- So far we've only looked at targeting a single xCore tile;
- In fact, by its very nature, C can only target a single tile;
- In order to utilise multiple tiles we need something to orchestrate our tile-level applications;
- For now this can be done using XC;
- In the future, this top-level XC will be replaced by a declarative format and a code generator;
- Applications can be 'future proofed' by constraining XC use to the bare minimum required to start the tiles

```
#include <platform.h>

typedef chanend chanend_t;

extern "C"
{
  void tile0_main(chanend_t);

  void tile1_main(chanend_t);
}

int main(void)
{
  chan tile_link;
  par
  {
    on tile[0]: tile0_main(tile_link);
    on tile[1]: tile1_main(tile_link);
```

That platform definition can be written in XC like this

```
#include <platform.h>


typedef chanend chanend_t;


extern "C"
{
  void tile0_main(chanend_t);

  void tile1_main(chanend_t);
}


int main(void)
{
  chan tile_link;
  par
  {
    on tile[0]: tile0_main(tile_link);
    on tile[1]: tile1_main(tile_link);
```

This example is runs two 'tile-level' entrypoint functions on their own tiles

```
#include <platform.h>

typedef chanend chanend_t;

extern "C"
{
  void tile0_main(chanend_t);
  void tile1_main(chanend_t);
}

int main(void)
{
  chan tile_link;
  par
```

First, the tile-level entrypoint functions are forward-declared

```c
#include <platform.h>

typedef chanend chanend_t;

extern "C"
{
  void tile0_main(chanend_t);
  void tile1_main(chanend_t);
}

int main(void)
```

Resource arguments use their XC counterpart types

```
typedef chanend chanend_t;


extern "C"
{
  void tile0_main(chanend_t);

  void tile1_main(chanend_t);
}


int main(void)
{
  chan tile_link;
  par
  {
    on tile[0]: tile0_main(tile_link);
    on tile[1]: tile1_main(tile_link);
  }
}
```

A single channel is declared for communication between the two tiles; this can be used to bootstrap additional channels if required

```
{
  void tile0_main(chanend_t);

  void tile1_main(chanend_t);
}


int main(void)
{
  chan tile_link;
  par
  {
    on tile[0]: tile0_main(tile_link);
    on tile[1]: tile1_main(tile_link);
  }
}
```

Finally, a 'par' block launches the two entrypoints on their respective tiles

```
#include <platform.h>

typedef chanend chanend_t;

extern "C"
{
  void tile0_main(chanend_t);

  void tile1_main(chanend_t);
}

int main(void)
{
  chan tile_link;
  par
  {
    on tile[0]: tile0_main(tile_link);
    on tile[1]: tile1_main(tile_link);
```

Notice that this code is essentially declarative - no application logic is written in XC