XMOS Programming Guide

Document Number: XM004440A

Publication Date: 2015/9/18

SYNOPSIS

This document provides a consolidated guide on how to program XMOS devices.

The xCORE architecture delivers, in hardware, many of the elements that you would normally see in a real-time operating system (RTOS). This includes the task scheduler, timers, and channel communication as well as providing separate logical processor cores for the real-time tasks. The benefit of this is that real-time systems become much more predictable, much more scalable and respond much faster than conventional RTOS based sequential processor systems.

The software environment complements these hardware features by providing tools that makes it easy to define real-time tasks as a parallel system. The xTIMEcomposer tools come with fully standards compliant C and C++ compilers plus the standard language libraries, an IDE, simulator, symbolic debugger, runtime instrumentation and trace libraries and a static code timing analyzer (XTA). All of the components are aware of the real-time multicore nature of the programs, giving a fully integrated approach.

To help programmers access the real-time hardware features some easy to use, yet powerful, multicore language extensions for C have been added. These extensions form a programming language called xC which contains features for task based parallelism and communication, accurate timing and I/O, and safe memory management.

Table of Contents

Part A

Programming multicore applications

CONTENTS

1 Getting started

IN THIS CHAPTER

To program an XMOS device you can used C, C++ or xC (C with multicore extensions). In your software project you can mix all three types of source file. To use the xC, just use the `.xc` file extension for your source file instead of `.c`. The xTIMEcomposer compiler will automatically detect this file extension and enable the C extensions for that file.

The xTIMEcomposer tools provide fully standards compliant C and C++ compilation (for example, files with a `.c` will be compiled as standard C). Applications can contain code written in a mixture of xC and C - you can call functions written in xC from standard C and vice-versa.

1.1 Hello World

Let’s start with the traditional “hello world” program:

```
#include <stdio.h>

int main() {
  printf("Hello World\n");
  return 0;
}
```

This program is exactly the same in C and xC. Since XMOS devices are embedded devices, you will only see print when the debug adapter is connected. In that case prints will be directed through the debug adapter and appear on the xTIMEcomposer console when running a program.

When you compile projects, the tools keep track of what resources you have used. For example, if you compile this program with the `-report` option you get the following output:

```
Constraint check for "tile[0]" (node "0", tile 0):
  Cores available:            8,   used:          1 .  OKAY
  Timers available:          10,   used:          1 .  OKAY
  Chanends available:        32,   used:          0 .  OKAY
  Memory available:       65536,   used:       1176 .  OKAY
    (Stack: 336, Code: 720, Data: 120)
Constraints checks PASSED.
```

You can see that the compiler tells you *exactly* how much memory is used (including stack usage). The xC language extensions have been designed so that you can always get this information even when running tasks in parallel that use the same memory space.

1.2 Going parallel

One of the major features of the extensions to C provided by xC is the ability to run code in parallel. The following program runs three threads of execution, all of which print out a “Hello World” message:

```
#include <stdio.h>

void hw(unsigned n) {
  printf("Hello world from task number %u\n", n);
}

int main() {
  par {
    hw(0);
    hw(1);
    hw(3);
  }
  return 0;
}
```

The key is the `par` construct which runs several tasks in parallel. This is described in more detail later.

1.3 Accessing the I/O hardware: A flashing LED

This code implements a simple flashing led:

```
#include <platform.h>
#include <xs1.h>
#include <timer.h>

port p = XS1_PORT_1A;

int main() {
  while (1) {
     p <: 0;
     delay_milliseconds(200);
     p <: 1;
     delay_milliseconds(200);
  }
  return 0;
}
```

This example uses the xC `port` type to declare a port `p` set to the port `1A` (the details of ports are described later in §6). The `<:` operator outputs a value to a port. The example also uses the `delay_milliseconds` function which is part of the libraries supplied with the tools (in the header `timer.h`).

1.4 Integration with C and C++

Within an XMOS project you can have C, C++ and xC files. They will all compile together into the same binary. The build system compiles the file based on its extension. For example, if a project had the following directory structure:

```
app_my_project/
  Makefile
  src/
    foo.xc
    bar.c
```

The `foo.xc` would be compiled with the multicore extensions as xC and `bar.c` would be compiled as normal C. The two object files would then be linked together into the same binary.

xC provides extensions to C, but does not support some C features. The currently unsupported features are:

- goto statements
- bitfields
- function pointers
- C99-style designated initializers

Functions written in C can be called from xC and vice versa. Since xC is an extension to C, functions from C can be prototyped in xC (apart from prototypes that use from the unsupported language features above).

For example, if a file called `foo.c` contains the following function:

```
int f() {
  printf("This is a C function\n");
}
```

and a file called `bar.xc` contains this function:

```
extern "c" {
  extern int f();  // This function is defined in C
}

void g() {
  par {
    f();
    f();
  }
}
```

Then compiling and linking the files will work with the xC function `g` calling the C function `f`.

When calling from C to xC, some new types in xC are not available in C. However you can use C types that are converted into xC types at link time.

For example, a `chanend` can be passed to a C function that takes an `int` or `unsigned int` parameter. The `xccompat.h` header included with the xTIMEcomposer suite contains useful macros and typedefs that expand to the correct type in each language.

1.5 The multicore programming model

1.5.1 Parallel tasks of execution

On xCORE devices, programs are composed of multiple tasks running in parallel. The concurrent tasks manage their own state and resources and interact by performing transactions with each other. A sample task decomposition of a system is shown in Figure 1.

Tasks are defined in the same way as any C function e.g.:

```
void task1(int x, int a[20]) { ...  }
```

There are no special keywords and any function can be a task. Tasks can take any arguments but generally have a no return value. Usually tasks do not return at all and consist of a never ending loop:

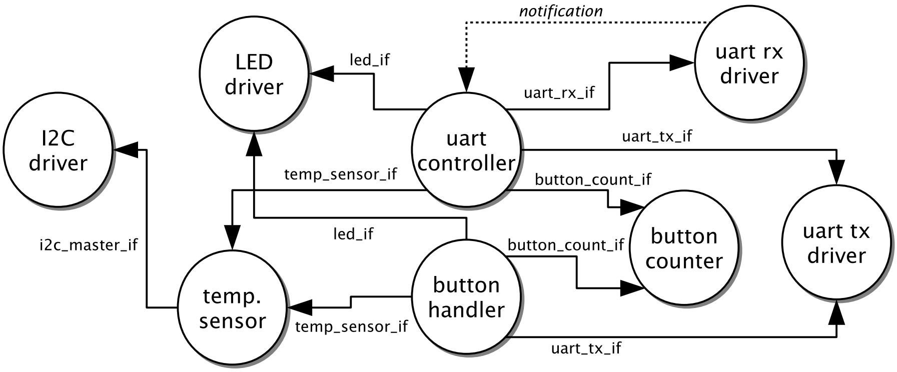


Figure 1: Example task communication diagram

```
void task1(...args...) {
  ... initialization ...
  while (1) {
    ... main loop ...
  }
}
```

Tasks are scheduled to run in the `main` function of the program using the `par` construct:

```
int main(void) {
  par {
   task1(...args...);
   task2(...args...);
   task3(...args...);
  }
}
```

Here, you can pass arguments into the tasks to configure them. Each task will run in parallel using the available hardware of the xCORE device.

The compiler automatically checks how many hardware resources are required and will error if too many are used. The compiler automatically allocates the required amount of stack and data memory to each task and reports back, telling you the total amount of memory used.

1.5.2 Explicit communication

Tasks always share information using explicit connections between the tasks. The connections allow synchronized transactions to occur between the tasks where data can be shared.

Each task runs independently on their own data. At some point one of the tasks will initiate a transaction between the tasks. This task will wait until the other task enters a state where it is ready to accept that transaction. At this point the two tasks will exchange some data and then both tasks will carry on. Figure 2 shows the progression of the transaction.

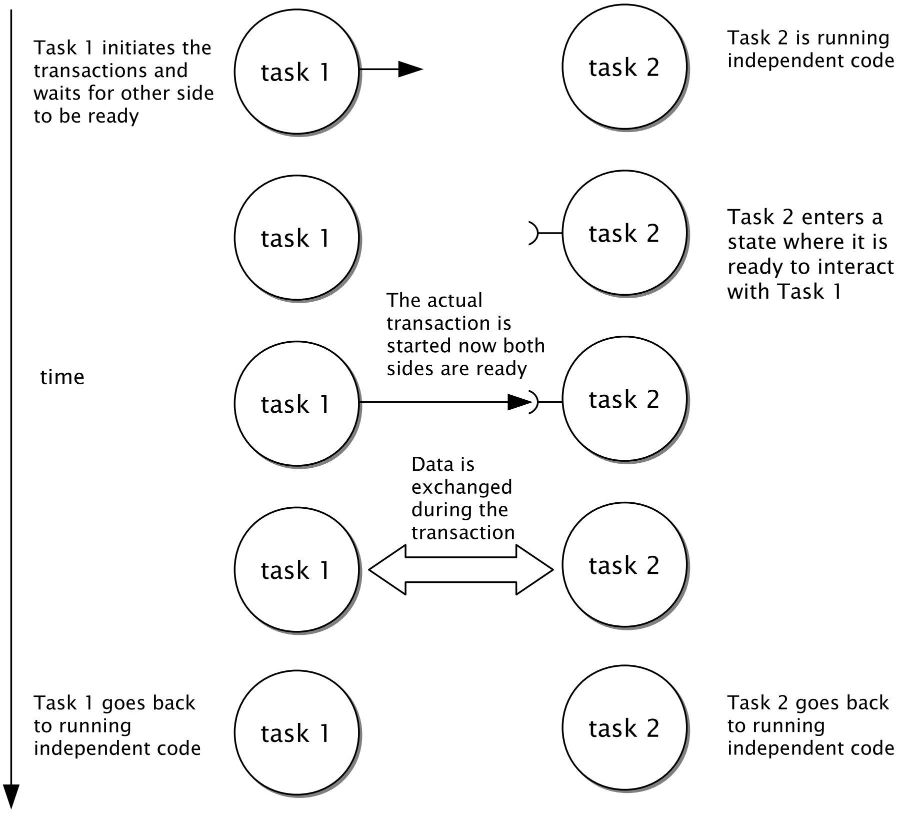

Task 1 initiates the transactions and waits for other side to be ready
task 1
task 2
Task 2 is running independent code
task 1
task 2
Task 2 enters a state where it is ready to interact with Task 1
The actual transaction is started now both sides are ready
time
task 1
task 2
Data is exchanged during the transaction
task 1
task 2
Task 1 goes back to running independent code
task 1
task 2
Task 2 goes back to running independent code

Figure 2: Two tasks performing a transaction

1.5.2.1 Shared memory access

Tasks never share data by accessing common data (e.g. global variables) so other synchronization methods (locks, mutexes, semaphores) are not used.

Two tasks can share common data by making a request to an intermediate task that owns the data. This keeps a clean separation of concerns. Figure 3 shows this method of sharing data.

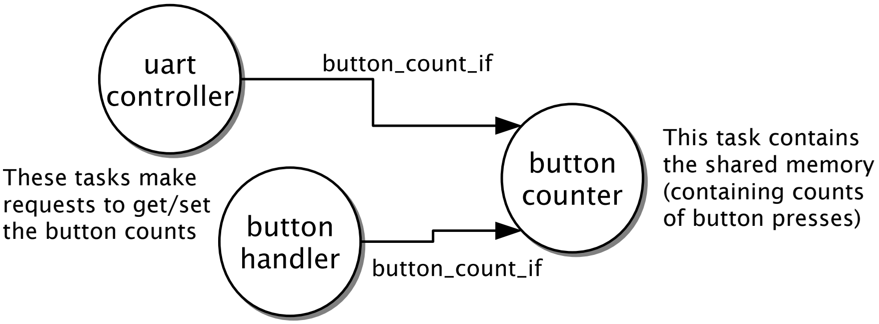

uart controller
button_count_if
button counter
This task contains the shared memory (containing counts of button presses)
These tasks make requests to get/set the button counts
button handler
button_count_if

Figure 3: Using an intermediate task to share memory

This approach makes any possible race conditions explicit in the task communication. In terms of efficiency, the compiler implements this method as efficiently as directly sharing memory.

1.5.2.2 Asynchronous communication

One aspect of the communication described here is that it is *synchronous* - one task waits for the other to be ready before the transaction can continue.

Sometimes asynchronous communication is required - a task wants to send some data to another without blocking. The task can then continue with other work. The sent data is buffered until the destination task is ready to capture the data.

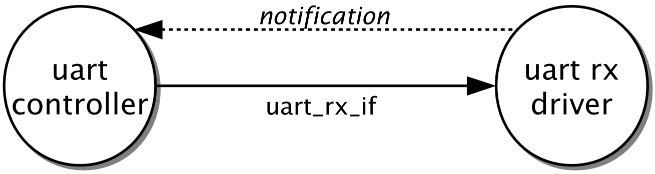


Figure 4: A notification between two tasks

There are two methods for asynchronous communication in xC. The first is to use a built-in communication method called *notifications* as shown in Figure 4. These allow a task to raise a flag between two tasks indicating a wish to communicate. Once the notification is raised the task can continue until the other tasks initiates a communication back. This is analogous to an hardware interrupt line back to the bus master in a hardware communication bus.

The second method to enable asynchronous communication is to insert a third task between two communicating tasks to act as a buffer as shown in Figure 5. The intermediate task is very responsive since it only handles buffering. One task can make a “push” transaction to put data in the buffer and then carry on. Asynchronously the other task can perform a “pull” transaction to extract the data from the buffer.

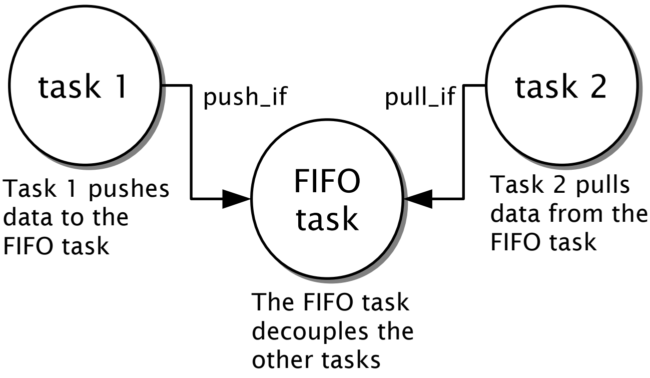

task 1
push_if
pull_if
task 2
FIFO task
Task 1 pushes data to the FIFO task
Task 2 pulls data from the FIFO task
The FIFO task decouples the other tasks

Figure 5: Two tasks with a shared memory FIFO task in between

This is the same as using a shared memory fifo between two tasks.

1.5.3 Event based programming

Tasks can react to events using the `select` construct, which pauses the tasks and waits for an event to occur. A `select` can wait for several events and handles the event that occurs first.

The syntax of a `select` statement is similar to a C `switch` statement:

```
select {
  case event1 :
    // handle the event
    ...
    break;
  case event2 :
    // handle the event
    ...
    break;
}
```

This statement will pause until either of the events occur and then execute the code within the relevant case. Although the `select` waits on several events, only one of the events is handled by the statement when an event occurs.

Often when programming other microcontrollers your program will react to external events via interrupts. A function will be registered against an interrupt that occurs when a certain event happens (e.g. a timeout or an external I/O event).

This function provides an *interrupt service routine* (ISR) that handles the event. Interrupts are not used to handle events in xC, the `select` construct provides all that is needed. The equivalent of an ISR is a separate task that executes a select. The advantages of the XMOS approach are:

- Response time to events can be drastically improved (in conjunction with the multi-core xCORE architecture)
- Reasoning about worst case execution time (WCET) is easier since code cannot be interrupted during its execution.

The full syntax for specifying cases depends on the type of event and is described later. Events can be caused by other tasks initiating a transaction (§2.2), timer events (§3) or external I/O events (§6).

1.6 The underlying hardware model

To understand programming in xC, it is useful to understand the underlying hardware that is being targeted. This section provides a description of how the hardware is organized. The description gives a high level overview; for specific details on a particular XMOS device consult the device datasheet.

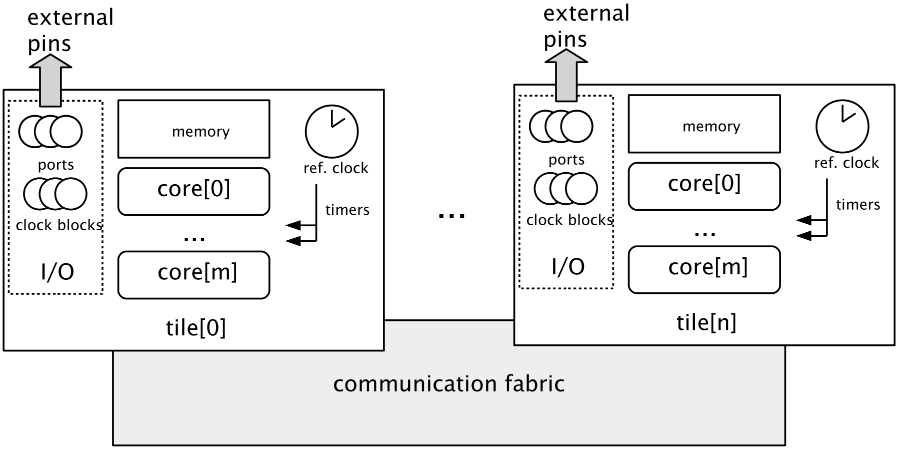


Figure 6: Underlying hardware system

Figure 6 shows the layout of the hardware that xC programs execute on. The system is composed of one or more *tiles* consisting of several *cores*.

1.6.1 Tiles

A system is split into *tiles*. A tile consists of a collection of hardware resources. Each tile has:

- A number of cores that can execute code

- A reference clock
- Some memory
- Access to an I/O sub-system

One of the key concepts of this model is that only code executing on a tile has direct access to the resources of that tile.

On XMOS devices:

- Each tile has between 64KB and 256KB of memory (XS1 devices have 64KB and XCORE-200 deivces have up to 256KB).
- The memory on each tile has no cache.
- The memory on each tile has no data bus contention (all peripherals are implemented via the I/O sub-system which does not use the memory bus; there is no DMA for peripherals).

The last two properties ensure that a load or store from memory always takes one or two instruction cycles on each core. This makes worst case execution time analysis very accurate for code that accesses memory.

Tasks on different tiles do not share memory but can communicate via inter-core communication.

In xC, the underlying hardware platform being targeted provides a set of names that can be used to refer to tiles in the system. These are declared in the `platform.h` header file. The standard is to provide an array named `tile`. So the tiles of the system can be referred to as `tile[0]`, `tile[1]`, etc.

1.6.2 Cores

Within a tile, a *core* (or *logical core*) provides an independent unit to execute code on. All the cores on a tile run in parallel.

Cores may run a different speed depending on the configuration of the device, but are guaranteed a minimum MIPS.

In xC, you can refer to a core via the `core` array associated with a particular tile. For example, `tile[0].core[1]` or `tile[1].core[7]`.

1.6.3 Timers

Each tile has a reference clock running at a specified rate. On XMOS XS-1 devices the clock will run at 100MHz. Associated with this clock is a counter. This counter is 32-bits wide on XMOS XS-1 devices.

In xC, the `timer` type is used to reference this clock. You can declare variables of this type, and operations on these variables can read the current time and wait on a particular timeout to occur.

1.6.4 Communication fabric

Between the cores there is a communication fabric (the xCONNECT fabric on XMOS devices). This communication fabric allows any core on any tile to access any other core on any other tile. This means that any software task can perform a transaction with another task whether the other task is on the same core or not (or even on the same tile or not).

1.6.5 I/O

Each tile has its own I/O subsystem that consists of *ports* and *clock blocks*. Full details of these are given in §6. In xC, ports and clock blocks are represented by the xC types `port` and `clock` respectively.

2 Parallel tasks and communication

IN THIS CHAPTER

The most fundamental difference between xC programming and C is the integration of parallelism and task management into the language.

2.1 Parallelism and task placement

xC programs are built of tasks that run in parallel. The is no special syntax about a *task* - any xC function represents a task that can be run:

```
void task1(int x)
{
  printf("Hello world - %d\n", x);
}
```

Running tasks in parallel is done with the `par` construct (short for “run in `par`-allel”). Here is an example `par` statement:

```
par {
  task1(5);
  task2();
}
```

This statement will run `task1` and `task2` in parallel to completion. It will wait for both tasks to complete before carrying on.

Although any function represents a task (i.e. a block of code that can be run in parallel with other tasks), tasks often have a common form of a function that does not return at all and consists of a never ending loop:

```
void task1(args) {
     ... initialization ...
     while (1) {
        ... main loop ...
     }
}
```

Although code can be run in parallel anywhere in your program, the `main` function is special in that it can place tasks on different hardware entities.

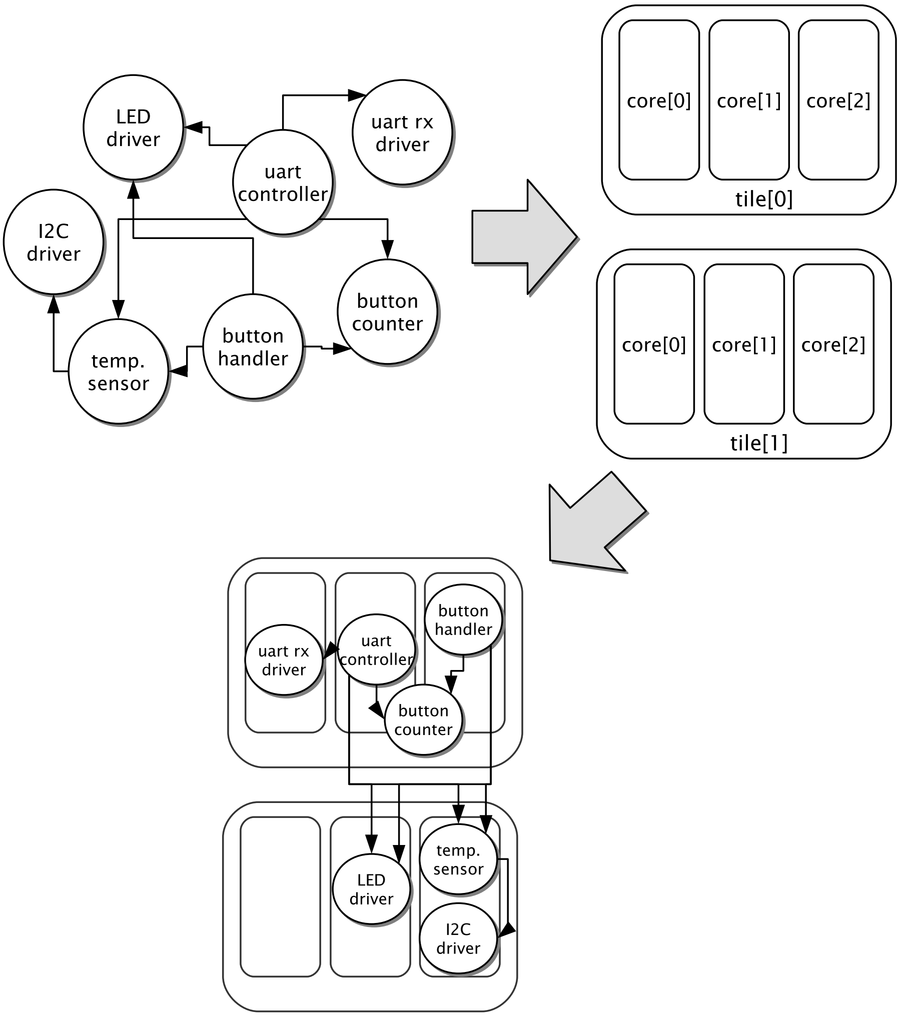


Figure 7: Task placement

Task placement involves assigning tasks to specific hardware elements (the tiles and cores of the system). Figure 7 show a possible placement of a group of tasks. Note that:

- Multiple tasks can run on the same logical core. This is possible via co-operative multitasking as described in §2.3.1.
- Some tasks run across multiple logical cores. These *distributable* tasks are described in §2.3.2.

As previously mentioned, task placement only occurs in the `main` function and is made by using the `on` construct within a `par`. Here is an example that places several tasks onto the hardware:

```
#include <platform.h>

...

int main() {
  par {
    on tile[0]: task1();
    on tile[1].core[0]: task2();
    on tile[1].core[0]: task3();
  }
}
```

In this example, `task2` and `task3` have been placed on the same core. This is only valid if these tasks can participate in cooperative multitasking (i.e are *combinable* functions - see § 2.3.1). If no core is specified in the placement, the task is automatically allocated a free logical core on the specified tile.

2.1.1 Replicated par statements

A replicated par statement can run several instances of the same task in parallel. The syntax is similar to a C `for` loop:

```
par(size_t i = 0; i < 4; i++)
  task(i);
```

This is the equivalent to the statement:

```
par {
  task(0);
  task(1);
  task(2);
  task(3);
}
```

The range of the iterator of the par (`i` in the example above) must step between compile-time constant bounds.

2.2 Communication

Tasks communicate via explicit transactions between them. Any task can communicate with any other, no matter which tiles and cores the tasks are running on. The compiler implements the transactions in the most efficient way possible using the underlying communication hardware.

All communication is done via point-to-point connections between tasks. These connections are explicit in the program. Figure 8 shows an example of some connected tasks.

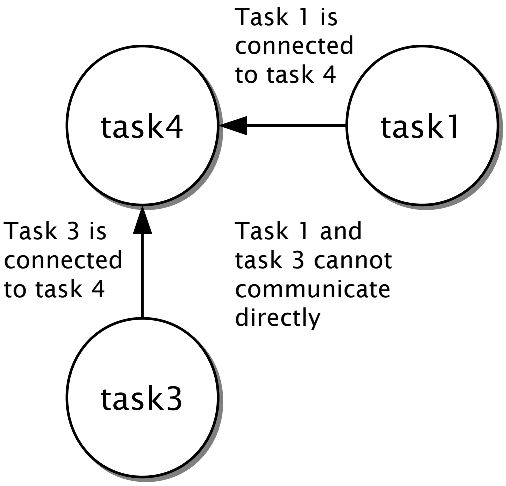

Task 1 is connected to task 4
task4
task1
Task 3 is connected to task 4
Task 1 and task 3 cannot communicate directly
task3

Figure 8: Example task connections

2.2.1 Channels and interfaces

xC provides three methods of inter-task communcation: interfaces, channels and streaming channels.

Channels provide a the simplest method of communication between tasks; they allow synchronous passing of untyped data between tasks. Streaming channels allow asynchronous communication between tasks; they exploit any buffering in the communication frabric of the hardware to provide a simple short-length FIFO between tasks. The amount of buffering is hardware dependent but is typically one or two words of data. Channels (both streaming and normal) are the lowest level of abstraction to the communication hardware available in xC and can be used to implement quite efficient inter-core communication but have no type-checking and *cannot be used between tasks on the same core*.

Interfaces provide a method to perform typed transactions between tasks. This allows the program to have multiple transaction functions (like remote function calls) between the tasks. Interfaces do allow communication between tasks running on the same logical core. In addition, interfaces also allow *notifications* to asynchronously signal between tasks during otherwise synchronous communications.

2.2.2 Interface connections

Interfaces provide the most structured and flexible method of inter-task connection. An interface defines the kind of transactions that can occur between the tasks and the data that is passed with them. For example, the following interface declaration defines two transaction types:

```
interface my_interface {
  void fA(int x, int y);
  void fB(float x);
};
```

Transaction types are defined like C functions. Interface functions can take the same arguments that any C function can. The arguments define what data is sent when the transaction between the tasks occurs. Since functions can have pass-by-reference parameters (see §5.1.1) or return values, data can flow both ways during a single transaction.

An interface connection between two tasks is made up of three parts: the connection itself, the *client* end and the *server* end. Figure 9 shows these parts and the xC types relating to each part. In the type system of the language:

- An interface connection is of type "`interface` *T*"
- The client end is of type "`client interface` *T*"
- The server end is of type "`server interface` *T*"

where *T* is the type of the interface.

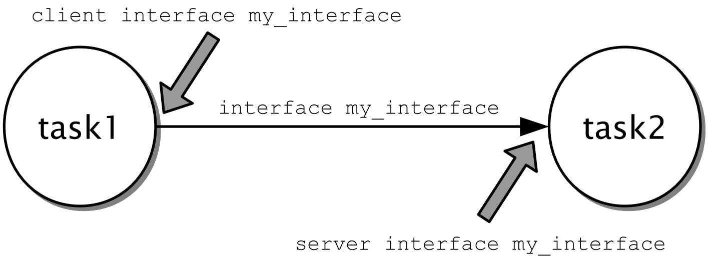


Figure 9: An interface connection.

A client end of a connection can be passed into a task as a parameter. The task that has access to the client end can initiate transactions using the following syntax similar to a function call:

```
void task1(client interface my_interface i)
{
  // 'i' is the client end of the connection,
  // let's communicate with the other end.
  i.fA(5, 10);
}
```

The server end can be passed into a task and that task can wait for transactions to occur using the select construct. A select waits until a transaction is initiated by the other side:

```
void task2(server interface my_interface i)
{
  // wait for either fA or fB over connection 'i'.
  select {
  case i.fA(int x, int y):
    printf("Received fA: %d, %d\n", x, y);
    break;
  case i.fB(float x):
    printf("Received fB: %f\n", x);
    break;
  }
}
```

Note how the select lets you handle several different types of transaction. Code can wait for many different types of transaction (using different interface types) from many different sources. Once one of the transactions has been initiated and the select has handled the event, the code will continue on. A select handles exactly one event.

When tasks are run, you can join them together by declaring an instance of an interface and passing it as an argument to both tasks:

```
int main(void)
{
  interface my_interface i;
  par {
    task1(i);
    task2(i);
  }
  return 0;
}
```

Only one task can use the server end of a connection and only one task can use the client end. If more than one task uses either end in a par, it will cause a compile-time error.

Tasks can be connected to multiple connections, for example:

```
int main(void)
{
  interface my_interface i1;
  interface my_interface i2;
  par {
    task1(i1);
    task3(i2);
    task4(i1, i2)
  }
  return 0;
}
```

This code corresponds to the connections show in Figure 8. A task can wait for events from multiple connections in one `select`:

```
void task4(interface my_interface server i1,
           interface my_interface server i2) {
   while (1) {
     // wait for either fA or fB over either connection.
     select {
     case i1.fA(int x, int y):
       printf("Received fA on interface end i1: %d, %d\n", x, y);
       break;
     case i1.fB(float x):
       printf("Received fB on interface end i1: %f\n", x);
       break;
     case i2.fA(int x, int y):
       printf("Received fA on interface end i2: %d, %d\n", x, y);
       break;
     case i2.fB(float x):
       printf("Received fB on interface end i2: %f\n", x);
       break;
     }
   }
}
```

With interface connections, the client end initiates communication. However, sometimes the server end needs to signal information to the client end independently. Notifications provide a way for the server to contact the client independently of the client making a call. They are asynchronous and non-blocking i.e. the server end can raise a signal and then carry on processing.

The following code declares an interface with a notification function in:

```
interface if1 {
  void f(int x);

 [[clears_notification]]
 int get_data();

 [[notification]] slave void data_ready(void);
};
```

This interface has two normal functions (f and get_data). However, it also has a notification function: data_ready. Within the interface declaration, a notification function can be declared with the [[notification]] attribute. This function must be declared as slave to indicate that the direction of communication is from the server end to the client end. In other words, the server will call the function and the client will respond. Notification functions *must* take no arguments and have a void return type.

It may seem that specifying both slave and [[notification]] on a function is redundant. The need for both is to future proof the language against further extensions where slave functions do not necessarily need to be notifications.

Once the server raises a notification, it triggers an event at the client end of the interface. However, repeatedly raising the notification has no effect until the client clears the notification. This can be done by marking one or more functions in the interface with the [[clears_notification]] attribute. The client will then clear the notification whenever it calls that function.

The server end of the interface can call the notification function to notify the client end i.e. it can execute the code:

```
void task(server interface if1 i) {
   ...
   i.data_ready();
```

As previously mentioned this task is non-blocking and raises a signal to the client. The signal can only be raised once - after calling data_ready, calling it again will have no effect.

The client end of the interface can make calls as normal, but can also select upon the notification from the server end of the interface. For example:

```
void task2(client interface if1 i)
{
   i.f(5);
   select {
   case i.data_ready():
     int x = i.get_data();
     printf("task2: Got data %d\n",x);
    break;
  }
}
```

Here the tasks calls data_ready after receiving the notification. As well as performing a transaction, this also clears the notification so the server can re-notify at a later time.

2.2.3 Channels

Channels provide a primitive method of communication between tasks. They connect tasks together and provide blocking communication but do not define any types of transaction. You connect two tasks together with a channel using a chan declaration:

```
chan c;
par {
    task1(c);
    task2(c);
}
```

With channels, the special operators <: and :> are used to send and receive data respectively. The operators send a value over the channel. For example, the following code sends the value 5 over the channel:

```
void task1(chanend c) {
  c <: 5;
}
```

The other end can receive the data in a select:

```
void task2(chanend c) {
  select {
  case c :> int i:
    printintln(i);
    break;
  }
}
```

You can also receive data by just using the input operator outside of a select:

```
void task1(chanend c) {
   int x;
   ...
   // Input a value from the channel into x
   c :> x;
```

By default, channel I/O is synchronous. This means that for every byte/word sent over the channel the task performing the output is blocked until the input task at the other end of the channel has received the data. The time taken to perform the synchronization along with any time spent blocked can result in reduced performance. Streaming channels provide a solution to this issue. They establish a permanent route between two tasks over which data can be efficiently communicated without synchronization.

You can then pass each end of the channel to each logical core, thus opening a permanent route between the two cores:

```
streaming chan c;
par {
    f1(c);
    f2(c);
}
```

2.2.4 Passing data via interface function calls

An interface function call passes data from the client end to the server end via its arguments. It is also possible to have return values. For example, the following interface declaration contains a function that returns an integer:

```
interface my_interface {
  int get_value(void);
};
```

The client end of the interface can use the result of that interface function call which has been passed back from the server:

```
void task1(client interface my_interface i) {
  int x;
  x = i.get_value();
  printintln(x);
}
```

When handling the function at the server end, you can declare a variable to hold the return value in the select case. This can be assigned in the body of the case and at the end of the case the value is returned back to the client:

```
void task2(server interface my_interface i) {
  int data = 33;
  select {
  case i.get_value() -> int return_val:
    // Set the return value
    return_val = data;
    break;
  }
}
```

Data can also pass both ways via pass-by-reference arguments (see ???) and array arguments:

```
interface my_interface {
  void f(int a[]);
};
```

The client end can pass an array into this function:

```
void task1(client interface my_interface i)
{
  int a[5] = {0,1,2,3,4};
  i.f(a);
}
```

When passing an array, it is a *reference* to the array that is passed. This is a handle that allows the server to access the elements of that array within the select case that handles that transaction:

```
...
select {
  case i.f(int a[]):
    x = a[2];
    a[3] = 7;
  break;
```

Note that the server can both read and write from the array. This works *even if the interface is connected across tiles* - in this case the array accesses are converted to efficient operations over the hardware's communication infrastructure.

The access to arrays also includes the use of `memcpy`. For example, an interface may contain a function to fill up a buffer:

```
interface my_interface {
  ...
  void fill_buffer(int buf[n], unsigned n);
};
```

At the server end of the interface, the memcpy in string.h can be used to copy local data to the remote array. This will be converted into an efficient inter-task copy:

```
int data[5];
...
select {
case i.fill_buffer(int a[n], unsigned n):
  // Copy data from the local array to the remote
  memcpy(a, data, n*sizeof(int));
  break;
}
```

2.2.5 Interface and channel arrays

It is useful to be able to connect one task to many others (this situation is shown in Figure 10).

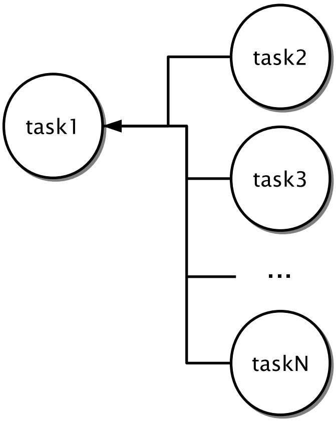


Figure 10: One task connecting to multiple other tasks

A task can connect to many others using an array of interfaces. One task can handle the ends of the entire array whilst the individual elements of the array can be passed to other tasks. For example the following code connects task3 to both task1 and task2:

```
int main() {
  interface if1 a[2];
  par {
    task1(a[0]);
    task2(a[1]);
    task3(a, 2);
  }
  return 0;
}
```

task1 and task2 are given an element of the array and can use the interface end as usual:

```
void task1(client interface if1 i)
{
  i.f(5);
}
```

task3 has the server ends of the entire array. The select construct can wait for a transaction over any of the connections. This is done using a *pattern variable* in the select case. The syntax is to declare the variable inside the array index of the array in the select case:

```
case a[int i].msg(int x):
   // handle the case
   ...
   break;
```

Here, the variable i is declared as a subscript to the array a, which means that the case will select over the entire array and wait for a transaction event from one of the elements.

When a transaction occurs, i is set to the index of the array element that the transaction occurs on. Here is a complete example of a task that handles an interface array:

```
void task3(server interface if1 a[n], unsigned n)
{
  while (1) {
    select {
    case a[int i].f(int x):
      printf("Received value %d from connection %d\n", x, i);
      break;
    }
  }
}
```

The exact same method will work for channel arrays e.g.:

```
int main() {
  chan c[2];
  par {
    task1(c[0]);
    task2(c[1]);
    task3(c, 2);
  }
  return 0;
}
```

and task3 can select over a chanend array:

```
void task3(chanend c[n], unsigned n)
{
  while (1) {
    select {
    case c[int i] :> int x:
      printf("Received value %d from connection %d\n", x, i);
      break;
    }
  }
}
```

2.2.6 Extending functionality on a client interface end

An interface can provide an API to a component of a system. *Client interface extensions* provide a way to extend this API with extra functionality that provides a layer on top of the basic interface. As an example, consider the following interface for a UART component:

```
interface uart_tx_if {
   void output_char(uint8_t data);
};
```

To extend a client interface a new function can be declared that acts like a new interface function. The syntax is:

`extends client interface` *T* { *function-declarations* }

The following example adds a new function to the uart_tx_if interface:

```
extends client interface uart_tx_if : {
   void output_string(client interface uart_tx_if self,
                      uint8_t data[n], unsigned n) {
     for (size_t i = 0; i < n; i++) {
       self.output_char(data[i]);
     }
   }
}
```

Here output_string extends the client interface uart_tx_if. Its first argument must be of that client interface type (in this example it uses the convention of being named self but can be any variable name). Within the function it can use this first argument to participate in transactions with the other end of the interface. The only restriction on the function definition is that it cannot access global variables.

The extension can be used in the same way as a interface function by the task that owns the client end of the interface:

```
void f(client interface uart_tx_if i) {
   uint8_t data[8];
   ...
   i.output_string(data, 8);
}
```

Here the `i` is implicitly passed as the first argument of the `output_string` function.

2.3 Creating tasks for flexible placement

xC programs are built up from several tasks running in parallel. These tasks can be of several different types that can be used in different ways. The following table shows the different types:

Task type	Usage
Normal	Tasks run on a logical core and run independently to other tasks. The tasks have predictable running time and can respond very efficiently to external events.
Combinable	Combinable tasks can be combined to have several tasks running on the same logical core. The core swaps context based on cooperative multitasking between the tasks driven by the compiler.
Distributable	Distributable tasks can run over several cores, running when required by the tasks connected to them.

Using these different tasks types you can maximize the resource usage of the device depending on the form and timing requirements of your tasks.

2.3.1 Combinable functions

If a tasks ends in an never-ending loop containing a select statement, it represents a task that continually reacts to events:

```
void task1(args) {
  .. initialization ...
  while (1) {
    select {
      case ... :
        break;
      case ... :
        break;
      ...
    }
  }
}
```

If a function complies to this format then it can be marked as *combinable* by adding the combinable attribute:

```
[[combinable]]
void counter_task(const char *taskId) {
  int count = 0;
  timer tmr;
  unsigned time;
  tmr :> time;
  // This task performs a timed count a certain number of times, then exits
  while (1) {
    select {
    case tmr when timerafter(time) :> int now:
      printf("Counter tick at time %x on task %s\n", now, taskId);
      count++;
      time += 1000;
      break;
    }
  }
}
```

This function uses timer events which are described later in §3.

A combinable function must obey the following restrictions:

- The function must have `void` return type.
- The last statement of the function must be a `while(1)` statement containing a single `select` statement.

Several combinable functions can run on one logical core. The effect of this is to “combine” the functions as shown in Figure 11.

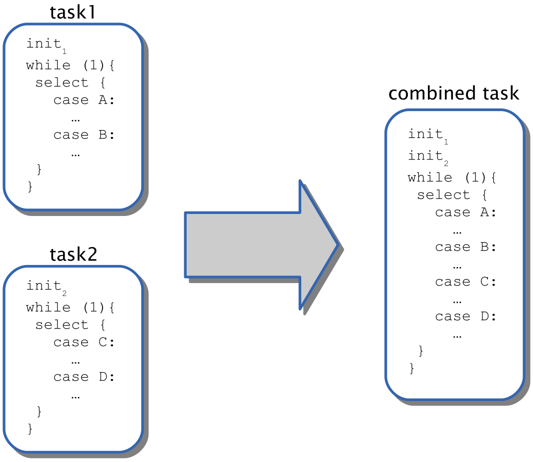


Figure 11: Combining several tasks

When tasks are combined, the compiler creates code that first runs the initial sequence from each function (in an unspecified order) and then enters a main loop. This loop enables the cases from the main selects of each task and waits for one of the events to occur. When the event occurs, a function is called to implement the body of that case from the task in question before returning to the main loop.

Within `main`, combinable functions can be run on the same logical core by using the `on` construct to place them:

```
int main() {
  par {
    on tile[0].core[0]: counter_task("task1");
    on tile[0].core[0]: counter_task("task2");
  }
  return 0;
}
```

The compiler will error if non-combinable functions are placed on the same core. Alternatively, a `par` statement can be marked to combine tasks anywhere in the program:

```
void f() {
  [[combine]]
  par {
    counter_task("task1");
    counter_task("task2");
  }
}
```

Tasks running on the same logical core can communicate with each other with one restriction: *channels cannot be used between combined tasks*. Interface connections must be used.

Combinable functions can be built up from smaller combinable functions. For example, the following code builds up the combinable function `combined_task` from the two smaller functions `task1` and `task2`:

```
[[combinable]]
void task1(server interface ping_if i);

[[combinable]]
void task2(server interface pong_if i_pong,
           client interface ping_if i_ping);

[[combinable]]
void combined_task(server interface pong_if i_pong)
{
  interface ping_if i_ping;
  [[combine]]
  par {
    task1(i_ping);
    task2(i_pong, i_ping);
  }
}
```

Note that `task1` and `task2` are connected to each other within `combined_task`.

2.3.2 Distributable functions

Sometime tasks contain state and provide services to other tasks, but do not need to react to any external events on their own. These kinds of tasks only run any code when communicating with other tasks. As such they do not need a core of their own but can share the logical cores of the tasks they communicate with (as shown in Figure 12).

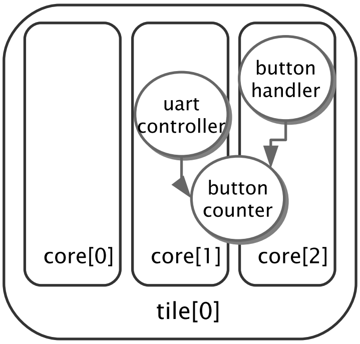


Figure 12: A task distributed between other tasks.

More formally, a task can be marked as *distributable* if:

- It satisfies the conditions to be *combinable* (i.e. ends in a never-ending loop containing a select)

- The cases within that select only respond to interface transactions

The following example shows a distributed tasks that responds to transactions over the interface connection `i` to access the port `p`:

```
[[distributable]]
void port_wiggler(server interface wiggle_if i, port p)
{
  // This task waits for a transaction on the interface i and
  // wiggles the port p the required number of times.
  while (1) {
    select {
    case i.wiggle(int n):
      printstrln("Wiggling port.");
      for (int j = 0; j < n;j++) {
        p <: 1;
        p <: 0;
      }
      break;
    case i.finish():
      return;
    }
  }
}
```

A distributable task can be implemented very efficiently if all the tasks it connects to are on the same tile. In this case the compiler will not allocate it a logical core of its own. For example, suppose the `port_wiggler` task was used in the following manner:

```
int main() {
  interface wiggle_if i;
  par {
    on tile[0]: task1(i);
    on tile[0]: port_wiggler(i, p);
  }
  return 0;
}
```

In this case `task1` would be allocated a core but `port_wiggler` would not. When `task1` creates a transaction with `port_wiggler`, the context on its core will be swapped to carry out the case in `port_wiggler`; after it is completed, context is swapped back to `task1`. Figure 13 shows the progression of such a transaction.

This implementation requires the core of the client task to have direct access to the state of the distributed task so only works when both are on the same tile. If the tasks are connected across tiles then the distributed task will act as a normal task (though it is still a combinable function so could share a core with other tasks).

If a distributed task is connected to several tasks, they cannot safely change its state concurrently. In this case the compiler implicitly uses a lock to protect the state of the task.

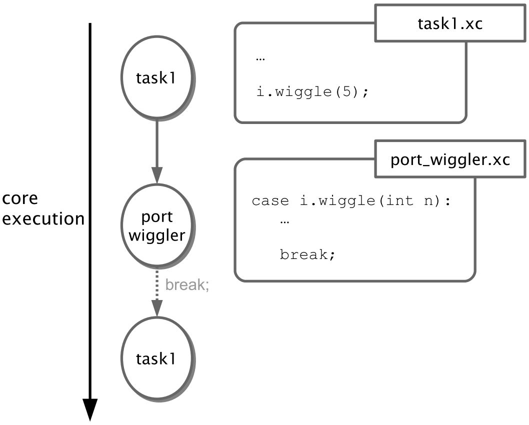


Figure 13: A transaction with a distributed task

3 Timing

Every tile in a system has a reference clock. On XMOS devices this is defined to always be a 100MHz clock. In xC, *timers* give the program access to this clock as shown in Figure 14.

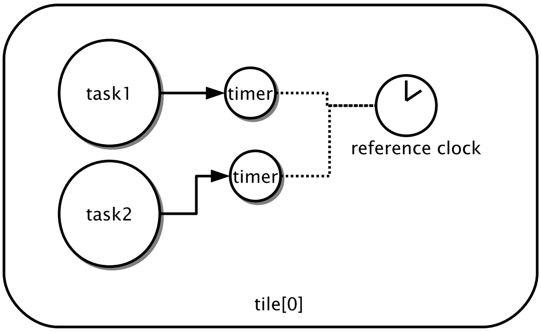


Figure 14: Timers

The reference clock has an associated 32-bit counter. Note that the current counter value of the clock is not the same on every tile, but if all tiles are on the same PCB the clocks are usually synchronized.

A timer variable can be declared at any point in an xC program:

```
timer t;
```

To get the current value of the timer counter the :> operator is used:

```
uint32_t time;

t :> time;  // this reads the current timer value into the variable 'time'
```

With this functionality a program can, for example, measure elapsed time:

```
uint32_t start_time, end_time;

t :> start_time;

// Do something here

t :> end_time;
printf("Number of timer ticks elapsed: %u", end_time - start_time);
```

Remember the range of the 32-bit counter. At 100MHz, you can only meaningfully measure up to 2^{32} - 1 ticks (about 42 seconds).

Timers can cause events when the counter reaches a certain value. These events can be reacted to in a `select` statement. The syntax for cases like this is:

> `case` *timer* `when timerafter (` *value* `) :>` [*var* | `void`] `:`

For example, the following select will event based on a timer:

```
uint32_t x;
timer t;
...
select {
  case t when timerafter(x) :> void:
     // handler the timer event
     ...
     break;
}
```

This event will trigger if the counter is greater than the value held in `x`. In this example `void` is after the `:>` symbol. This means that the actual counter value at the time of the event is ignored. Alternatively, the value could be read into a new variable:

```
case t when timerafter(x) :> int y:
```

The ability to react to timing events allows a program to perform actions periodically. For example, the following loop has a case that executes some code every 1ms:

```
timer t;
uint32_t time;
const uint32_t period = 100000; // 100000 timer ticks = 1ms

// get the initial timer value
t :> time;
while(1) {
  select {
  case t when timerafter(time) :> void:
    // perform periodic task
    ...
    time += period;
    break;
  }
}
```

This form of periodic event handling is particular useful when combined with other events handlers in the same `select` construct. It is also useful in defining *combinable* functions (see §2.3.1), which can run periodic tasks along with others on the same logical core.

4 Event handling

IN THIS CHAPTER

As already presented, the `select` statement allows code to wait for events to occur. It can wait for different events in different cases:

```
select {
  case A:
    ...
    break;
  case B:
    ...
    break;
  case C:
    ...
    break;
}
```

This section describes some advanced features of this event handling language construct.

4.1 Guards

When a select is waiting on several events, some events are conditionally enabled i.e. the code should only react to them if some guard expression evaluated to non-zero. The syntax for this is:

case *expr* => *...* :

The following example only reacts to the timer event if the variable `periodic_enabled` is non-zero:

```
int periodic_enabled;
timer tmr;
uint32_t t;

...

select {
  case periodic_enabled => tmr when timerafter(t) :> void:
    ..
    break;
  case ...

}
```

If the case being guarded is an interface transaction, the compiler needs extra information to be able to implement the guard. To enable the compiler to do this, if an interface function is guarded anywhere in the program, it must be marked as possibly guarded in the interface declaration using the `[[guarded]]` attribute. For example:

```
interface if1 {
   void f();
   [[guarded]] void g();  // this function may be guarded in the program
}

..
select {
  case i.f():
     ...
     break;
  case e => i.g():
     ...
     break;
}
```

4.2 Ordering

Generally there is no priority on the events in a select. If more than one event is ready when the select executes, the chosen event is unspecified.

Sometimes it is useful to force a priority by using the [[ordered]] attribute which says that a select is presented with events ordered in priority from highest to lowest. For example, if all three events in the allowing example are ready at the time of the select, case A will be chosen:

```
[[ordered]]
select {
  case A:
    ...
    break;
  case B:
    ...
    break;
  case C:
    ...
    break;
}
```

⚠ Ordered selects cannot be used in combinable or distributable functions.

4.3 Defaults

Usually a select waits until one of the events occurs. It is possible to add a default case to a select. This will fire if none of the other events are ready when the select is first executed:

```
select {
  case A:
    ...
    break;
  case B:
    ...
    break;
  default:
    ...
    break;
}
```

⚠ Defaults cannot be used in combinable or distributable functions.

4.4 Replicated cases

Replicated cases iterate the same case several times. This is useful if the program has an array of resources to react to. For example, the following code iterates over an array of timers and associated timeouts.

```
timer    tmr_array[5];
uint32_t timeout[5]
unit32_t period[5];

...

select {
   case(size_t i = 0; i < 5; i++)
     tmr_array[i] when timerafter(timeout[i]) :> void:
        ....
        timeout[i] += period[i];
        break;
}
```

4.5 Select functions

You can place several select cases together in a select function. These functions allow you to abstract parts of a select to reuse them. This allows you to create libraries of functions that encapsulate reactive behavior.

The following example defines a select function with two cases:

```
#include <print.h>
#include <xs1.h>

timer t;

select my_case(chanend c, unsigned timeout) {
  case c :> int x:
    printintln(x);
    break;
  case t when timerafter(timeout) :> void:
    printstrln("Timeout");
    break;
}
```

Note that the function definition has the `select` keyword before it. The body of a select function can only have a list of cases in it.

A select function can be called within a select in the following manner:

```
void f(chanend c, chanend d, chanend e) {
  unsigned timeout;
  t :> timeout;
  timeout += 5000;
  select {
    case my_case(c, timeout);
    case my_case(d, timeout);
    case e :> int y:
      e <: y + 2;
      break;
  }
```

Here the select function is re-used in the same select with different arguments.

5 Data handling and memory safety

IN THIS CHAPTER

5.1 Extra data handling features

5.1.1 References

xC, like C++, provides *references* as a method of indirectly refering to some data. For example, the following declarations create a reference `x` to the integer `i`:

```
int i = 5;
int &x = i;
```

Reading and writing a reference is the same as reading and writing to the original variable:

```
printf("The value of x is %d\n", x);
x = 7;
printf("x has been updated to %d\n", x);
printf("i has also been updated to %d\n", i);
```

References can also refer to array or structure elements:

```
int a[5] = {1,2,3,4,5};
int &y = a[0];

printf("y has value %d\n", y);
```

Function parameters can also be references. For example, the following function takes a reference and updates the value it refers to:

```
void f(int &x) {
  x = x + 1;
}
```

This function can be called with the value to refer to as an argument:

```
void pass_by_reference_example() {
  int i = 5;
  printf("Value of i is %d\n", i);
  f(i);
  printf("Value of i is %d\n", i);
}
```

References can be passed between tasks as interface function arguments. For example, the function `update` in the following interface can alter the variable provided as an argument:

```
interface if1 {
  void update(int &x);
};
```

This can be called as such:

```
void task(client interface if1 i) {

   ...
   i.update(y);  // this may change the value of y

}
```

Just as passing arrays over interface calls, updating a reference works even when the communicating tasks are on different tiles.

5.1.2 Nullable types

Resources in xC such as interfaces, chanends, ports and clocks, must always have a valid value. The nullable qualifier allows these types to be the special value `null` indicating no value. This is similar to optional types in some programming languages.

The nullable qualifier is a `?` symbol. So the following declaration is a nullable port:

```
port ?p;
```

Given a nullable typed variable, the program can check whether it is null using the `isnull` built-in function e.g.:

```
if (!isnull(p))  {
   // We know p is not null so can use it here
   ...
}
```

This facility is particularly useful for optional function parameter, for example:

```
// function that takes a port and optionally a second port
void f(port p, port ?q);
```

References can also be declared as nullable. Since the nullable qualifier applies to the reference it needs to appear to the right of the reference symbol, for example:

```
// Function that takes an optional integer 'y' to update
void f(int x, int &?y);
```

Finally, array can also be declared nullable. In this case the declaration needs to be explicit that the parameter is a reference to an array, for example:

```
// Function that takes an optional integer array 'a'
void f(int (&?a)[5]);
```

5.1.3 Variable length arrays

In xC, array declarations need to be a constant size. The exception to this is a local array that can be declared as a variable size based on a parameter, provided that parameter is marked both `static` and `const`:

```
void f(static const int n)
{
  printf("Array length = %d\n", n);

  int arr[n];
  for (int i = 0; i < n; i++) {
    arr[i] = i;
    for (int j = 0; j < i; j++) {
      arr[i] += arr[j];
    }
  }

  printf("-------\n");
  for (int i = 0; i < n; i++) {
    printf("Element %d of arr is %d\n", i, arr[i]);
  }
  printf("-------\n\n");
}
```

When calling functions with static parameters, the argument has to be either:

- a constant expression
- a static const parameter to the caller function

For example:

```
void g(static const int n)
{
  // static parameter can be called with a constant expression argument
  f(2);
  // or passing on a static const parameter
  f(n);
}
```

These restrictions mean that the compiler can still statically track stack usage despite the local array having variable size.

5.1.4 Multiple returns

In xC, functions can return multiple values. For example, the following function returns two values:

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

When calling the function, multiple values can be assigned at once:

```
int x = 5, y = 7;
{x, y} = swap(x, y);
```

5.2 Memory safety

In C and xC you have several ways of accessing memory (see Figure 15)

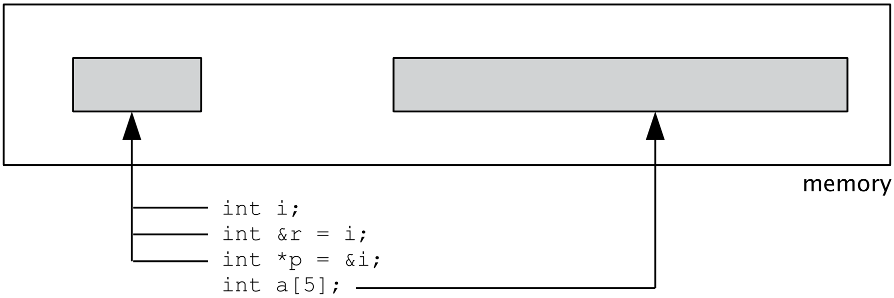
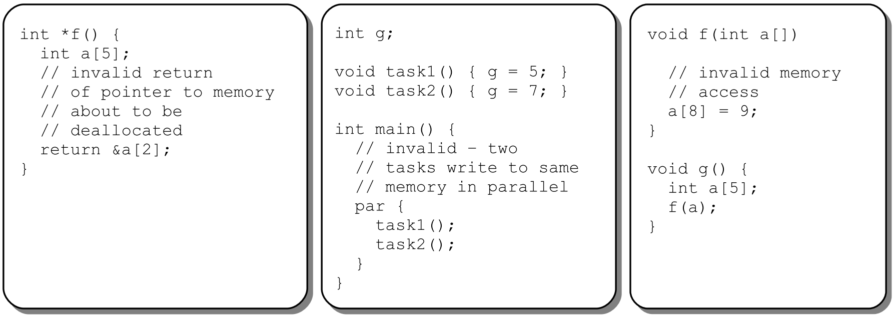


Figure 15: Different ways to access memory

In C memory might be allocated via variable declarations or `malloc`. Only access to these allocated regions is allowed, if the program tries to access outside these regions then it is invalid behavior and the results are undefined.

In xC, there is also the notion of parallel tasks and the language has an additional restriction: *no task is allowed to access memory that another task owns.* The task that owns memory has permission to write to it. Memory ownership can transfer between tasks at well defined points in the program.

If a task accesses memory it is not supposed to, it is a program error and can cause destructive, hard to trace bugs. xC helps by adding checks to catch invalid memory access early (e.g. at compile time or early on during program execution) to eliminate these bugs. For example, all the bugs in Figure 16 will be detected.



```
int *f() {
  int a[5];
  // invalid return
  // of pointer to memory
  // about to be
  // deallocated
  return &a[2];
}
```

Dangling pointer

```
int g;

void task1() { g = 5; }
void task2() { g = 7; }

int main() {
  // invalid - two
  // tasks write to same
  // memory in parallel
  par {
    task1();
    task2();
  }
}
```

Race condition

```
void f(int a[])

  // invalid memory
  // access
  a[8] = 9;
}

void g() {
  int a[5];
  f(a);
}
```

Out of bounds access

Figure 16: Common invalid memory operations

To be able to do all these checks and use C-style pointers, xC needs to have extra annotations on the pointer types. These restrictions help ensure memory safety - see §5.2.4.

5.2.1 Runtime exceptions

The xC compiler will try and spot memory errors at compile time and report the error during compilation. For example, given the following code:

```
void f()
{
  int a[5];
  a[7] = 10;
}
```

The compiler will fail to compile the code with the following error:

```
bad.xc: In function `f':
bad.xc:4: error: index of array exceeds its upper bound
```

However, sometimes it is unknown at compile time whether a memory error occurs, for example:

```
void f(int a[]) {
  a[7] = 10;
}
```

In this case the compiler will insert a runtime check that the memory operation is safe. If the operation is not safe then an *exception* is raised. The exception will cause the program to halt at the point of the error (rather than trashing memory and failing in some hard to trace way later on). If the debugger is connected then it will report that an exception has occurred and where it has occurred (if the program is compiled with debug information on).

For production release of an application, it is possible to install a general exception handler to the program that, for example, reboots the device on failure. Of course, by this time, no errors should actually occur in the program.

5.2.2 Bounds checking

The compiler keeps track of the bounds of all arrays and pointers (with the exception of *unsafe* pointers - see §5.2.4). Accessing outside of these bounds will cause a compiler error or runtime exception.

If the compiler cannot determine that usage is safe at compile time, it will insert runtime checks. However, these checks take time to execute. For arrays, the checks can often be eliminated by indicating to the compiler that the bound of an array is related to another variable using the following syntax:

```
// Function takes an array of size n
void f(int a[n], unsigned n) {
  for (int i = 0; i < n; i++)
    a[i] = i;
}
```

In this case the code needs no bounds checks since the compiler can infer that all access are within the bounds of the array (given by the variable `n`).

The compiler will still need to check that the bound is correct when the function is called. For example, the following would be cause an error:

```
void f(int a[n], unsigned n);

void g() {
  int a[5];
  f(a, 8);  // error - bound does not match
}
```

5.2.3 Parallel usage checks

At compile time, the tools check for parallel usage violations i.e. that no task accesses memory that another task owns. It can detect ownership by detecting which tasks write to a variable. For example, if you tried to compile the following program:

```
#include <stdio.h>

int g = 7;

void task1() { g = 7; }

void task2() { printf("%d",g);}

int main() {
  par {
    task1();
    task2();
  }
  return 0;
}
```

Then the compiler would return the following error:

```
par.xc:10: error: use of `g' violates parallel usage rules
par.xc:7: error: previously used here
par.xc:5: error: previously used here
```

If data is only read, two tasks can access it. So the following would be valid:

```
#include <stdio.h>

const int g = 7;

void task1() { printf("%d", g+2); }

void task2() { printf("%d", g);}

int main() {
  par {
    task1();
    task2();
  }
  return 0;
}
```

5.2.4 Pointers

Pointers are very powerful programming devices but it is also very easy to end up with a pointer performing an invalid memory access. The bounds checking stops pointers performing an invalid access via pointer arithmetic. However, pointers could still point to de-allocated memory and invalid parallel usage access could happen indirectly via pointers. xC detects these invalid uses and causes either a compile-time or run-time error.

To do this, every pointer needs to be allocated a *kind*. There are four kinds of pointer: *restricted*, *aliasing*, *movable* and *unsafe*. Any pointer declaration can describe the pointer kind with the following syntax:

> *pointee-type* * *pointer-kind pointer-variable*

For example, the following declaration is a movable pointer to `int` named `p`:

```
int * movable p;
```

If no pointer kind is described in the declaration, a default is assumed. The default depends on whether the declaration is a global variable, parameter or local variable. The following table shows the defaults.

Declaration location	Default
Global variable	Restricted
Parameter	Restricted
Local variable	Aliasing
Function returns	No default - must be explicitly declared

5.2.4.1 Aliasing

To keep track of pointers that point to de-allocated memory or that may cause an invalid parallel access of memory, the compiler must be able to track pointer *aliasing*. Aliasing happens when two program elements refer to the same region of memory. Figure 17 shows an example of aliasing.

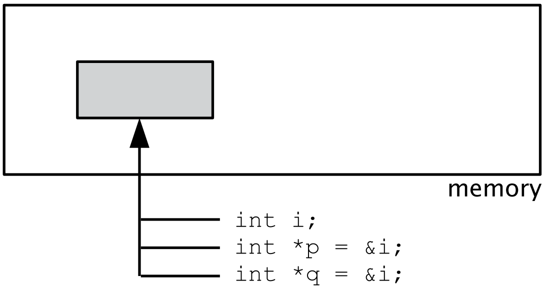


Figure 17: Three program elements referring to the same memory region

5.2.4.2 Restricted Pointers

In both C and xC there is the concept of a *restricted* pointer. This is a pointer that cannot alias i.e. the only way to access that memory location is via that pointer. In C, the compiler can assume that access via a restricted pointer is non-aliased but does not perform any checks to make sure this is the case. In xC, extra checks ensure that the non-aliasing restriction is true.

The first check in xC, is that given a restricted pointer to an object, the program cannot access the memory via the original variable:

```
int i = 5;
int * restrict p = &i;

printf("%d", i);  // this is an error
```

Function parameters default to restricted so the following would also be invalid:

```
int i = 5;

// The function argument defaults to a restricted pointer
void f(int *p) {
  i = 7; // this is an error due to the call below
}

void g() {
  f(&i);
}
```

The second check on restricted pointers that xC makes is that the pointer cannot be re-assigned or copied:

```
int i = 5, j = 7;
int * restrict p = &i;
int * restrict q;

p = &j;   // invalid - cannot reassign to a restricted pointer
q = p;    // invalid - cannot copy a restricted pointer
```

These checks ensure that a restricted pointer always points to a tracked non-aliased location. It also cannot point to de-allocated memory.

Since pointer function parameters are restricted by default, the compiler also checks that no aliases are created at the point of a function call:

```
// Function that takes two restricted pointers
void f(int *p, int *q);

void g() {
  int i;
  f(&i, &i);  // this is invalid since the arguments alias
}
```

Since restricted pointers cannot be copied, function return types cannot be of restricted type.

5.2.4.3 Pointers that are allowed to alias

Restricted pointers are quite limited in their use. It is often very convenient to have pointers that alias. The alias pointer kind allows this but with different usage rules to restricted pointers.

Local pointers default to aliasing so the following code is valid:

```
int f() {
   int i = 5, j = 7;
   int *p = &i;  // this is an aliasing pointer
   int *q = &j;  // so is this

   p = q;  // aliasing pointers can be reassigned and copied

   return i + *p + *q;
}
```

To keep track of the aliases made by aliasing pointers the following restrictions apply:

- You cannot pass alias pointers to different tasks in a par
- You cannot have indirect access to an alias pointer (e.g. a pointer to an alias pointer)

If a function takes pointer parameters or returns a pointer that may alias, it needs to be explicitly written into the type of the function. For example the following function's return value may alias its argument:

```
char * alias strchr(const char * alias haystack, int needle);
```

Global pointers can be accessed anywhere in the program so aliasing cannot be easily tracked. Accordingly, in xC, global pointers cannot be aliasing. They default to restricted but may be marked as unsafe or movable.

5.2.4.4 Function parameters

When passing pointers to functions, there are some special rules to allow conversion between pointer kinds. Firstly, a restricted pointer can be passed to a function taking an alias pointer arguments:

```
void f(int * alias x);

void g(int *y) {
   f(x);   // y is restricted but can be passed as an alias
}
```

In fact, within a function, a restricted pointer is treated like an aliasing pointer:

```
void g(int *y)  // 'y' is a restricted pointer
{
  // within the function 'y' acts like an aliasing pointer
  int *p = y;
  y = p;
  ...
}
```

An aliasing pointer can be passed to a function taking a restricted pointer argument if it does not alias any of the other arguments:

```
void f(int *x, int *y) { *x;}

int g() {
  int i = 5, j = 8;
  int *p = &i;
  int *q = &j;
  f(p, q);      // valid since p does not alias q
}
```

This only works if the source of the aliasing pointer is local to the function. If the aliasing pointer is assigned to a value that is from an incoming argument or global variable, an additional restriction is made on the function being called - it cannot access any global variables (since these may alias the pointer being passed). For example, the following is invalid:

```
int i = 5;

int f(int *q) {
  return *q + i; // compiler assumes 'q' does not alias 'i'
}

void g() {
  int *p = &i;
  f(p);         // invalid since f accesses a global and 'p' has
                // non-local scope
}
```

This code will fail at compile time with the error:

```
p.xc:10: error: passing non-local alias to function `f' which accesses a
  ↪ global variable
```

5.2.4.5 Transferring ownership (movable pointers)

It is useful to transfer the ownership of a pointer between parts of the program. For example:

- Transfer ownership to a global variable to be used at a later time
- Transfer ownership between tasks running in parallel

In these cases, the pointers still need to have no aliases to avoid race conditions and dangling pointers but *restricted* pointers cannot be reassigned or copied.

Movable pointers provide a solution. These pointers can be transferred but only in a way that means they retain non-aliasing properties and can never refer to de-allocated memory.

A movable pointer is declared using the `movable` type qualifier:

```
int i = 5;
int * movable p = &i;
```

Just like with restricted pointers, the program cannot use `i` in this case since that would break the non-aliasing property of the pointer.

Movable pointer values can be transferred using the `move` operator:

```
int * movable q;

q = move(p);
```

The `move` operator sets the source pointer to `null`. This ensures that only one variable has ownership of the memory location at a time.

The `move` operator also has to be used when passing a movable pointer into a function or returning a movable pointer:

```
int * movable global_p;

void f(int * movable p) { global_p = move(p); }

int * movable g(void) {
  return move(global_p); // need to use the move operator here
}

void h(void) {
  int i = 5;
  int * movable p = &i;
  f(move(p));  // need to use the move operator here
  p = g();
}
```

Movable pointers cannot refer to de-allocated memory. To ensure this the following restriction applies:

> A movable pointer must point to the same region it was initialized with when it goes out of scope.

A runtime check is inserted to ensure this (so an exception can happen when the pointer goes out of scope). For example, the following is invalid:

```
int* movable global_p;

void f() {
   int i = 5;
   int *p = &i;
   global_p = move(p);

 }  // <-- at this point an exception occurs since 'p'
    //     does not point to the region it was
    //     initialized to
```

This avoids `global_p` pointing to de-allocated memory.

5.2.4.6 Transferring pointers between parallel tasks

Pointers can be passed as interface function arguments, for example:

```
interface if1 {
  void f(int * alias p);
};
```

The tasks share the pointer for the duration of the transaction, for example:

```
void f(server interface if1 i) {
  select {
    case i.f(int * alias p):
      printf("%d", *(p+2));
      break;
  }
}
```

When an interface is declared containing functions with pointer arguments it cannot be used across tiles (since tiles have separate memory spaces).

Restricted and alias pointers can only be used for the duration of the transaction. For example, the following is invalid:

```
void f(server interface if1 i) {
  int * alias q;
  select {
    case i.f(int * alias p):
      q = p; // invalid, cannot move the alias to a larger scope
      break;
  }
  printf("%d", *q);
}
```

To transfer a pointer beyond the scope of the transaction, movable pointers should be used e.g:

```
interface if2 {
  void f(int * movable p);
};

void f(server interface if2 i) {
  int * movable q;
  select {
    case i.f(int * movable p):
      q = move(p); // ok, ownership is transferred
      break;
  }
  printf("%d", *q);
}
```

This way, one task must relinquish ownership of the memory region at a well defined point for the other task to use it (so no accidental race conditions can occur).

5.2.4.7 Unsafe pointers

An `unsafe` pointer type is provided for compatibility with C and to implement dynamic, aliasing data structures (for example linked lists). This is not the default pointer type and the onus is on the programmer to ensure memory safety for these types.

An unsafe pointer is opaque unless accessed in an `unsafe` region. A function can be marked as unsafe to show that its body is an unsafe region:

```
unsafe void f(int * unsafe x) {
  // We can dereference x in here,
  // but be careful - it may point to garbage
  printintln(*x);
}
```

Unsafe functions can only be called from unsafe regions. You can make a local unsafe region by marking a compound statement as unsafe:

```
void g(int * unsafe p) {
  int i = 99;
  unsafe {
    p = &i;
    f(p);
  }
  // Cannot dereference p or call f from here
}
```

These regions allow the programmer to manage the parts of their program that are safe by construction and the parts that require the programmer to ensure safety.

Within unsafe regions, unsafe pointers can be explicitly cast to safe pointers - providing a contract from the programmer that the pointer can be regarded as safe from then on.

It is undefined behavior for an unsafe pointer to be written from one task and read from another.

6 I/O

IN THIS CHAPTER

As well as allowing users to program real-time, parallel applications; XMOS microcontrollers allow complex I/O protocols to be implemented. This is done via the Hardware-Response ports within the microcontroller which can be programmed via the multicore extensions to C.

On XMOS devices, pins are used to interface with external components. Current XMOS devices have 64 digital I/O pins each of which can serve as either an input or output pin. The pins have an operating voltage of 3.3v. Not all products have all 64 pins accessible externally on the package - check the product datasheet to determine how many pins are available.

When referring to pins the following naming convention is use: X*n*D*pq* where *n* is the tile number within the device and *pq* is the number of the pin (*e.g.* X0D05).

6.1 Ports

The pins on the device are accessed via the hardware-response *ports*. The port is responsible for driving output on the pins or sampling input data.

Different ports have different *widths*: there are 1-bit, 4-bit, 8-bit, 16-bit and 32-bit ports. An *n*-bit port will simultaneously drive or sample *n* bits at once.

In current devices, each tile has 29 ports.

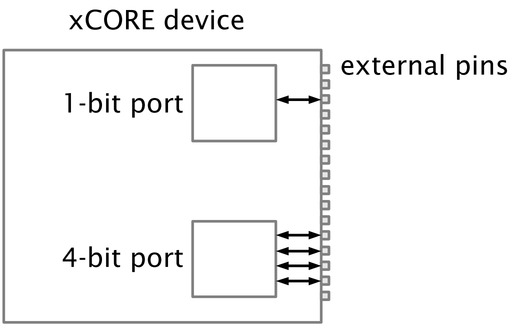


Figure 18: Ports of different widths

Port width	Number of ports	Port names
1	16	1A, 1B, 1C, 1D, 1E, 1F, 1G, 1H, 1I, 1J, 1K, 1L, 1M, 1N, 1O ,1P
4	6	4A, 4B, 4C, 4D, 4E ,4F
8	4	8A, 8B, 8C ,8D
16	2	16A, 16B
32	1	32A

In your code, you can access ports by declaring variables of `port` type. The header file `xs1.h` defines macros to initialize a port variable to access a specific port. For example, the following declaration creates a port variable `p` to access port 4A:

```
#include <xs1.h>

port p = XS1_PORT_4A;
```

Since the ports input and output all the bits at once, they should be used for inputs and outputs that work logically together. For example, a 4-bit port is not designed to drive 4 independent signals (e.g. the clock and data lines of a serial bus) - it is better to use independent 1-bit ports. However, a 4-bit port is very efficient when used to input or output from for a 4-bit wide data bus.

There is a fixed mapping between the external pins and the ports. Some pins map to multiple ports and, in general, overlapping ports should not be used together. The mapping between ports and pins can be found in the relevant devices datasheet.

6.2 Clock blocks

All ports are *clocked* - they are attached to a clock block in the device to control reading and writing from the port. A clock block provides a regular clock signal to the port.

Each port has a register called the *shift register* within it that holds either data to output or data just input depending on whether the port is in input or output mode. At each clock tick, the port samples the external pins into the shift register or drives the external pins based on the contents of the shift register. When a program “inputs” or “outputs” to a port it is actually reads or writes the shift register.

There are six clock blocks per tile. Any port can be connected to any of these six clock blocks. Each port can be set to one of two modes:

Mode	Description
Divide	The clock runs at a rate which is an integer divide of the core clock rate of the chip (*e.g.* a divide of 500MHz for a 500MHz part).
Externally driven	The clock runs at a rate governed by an port input.

The second mode is used to synchronize I/O to an external clock. For example, if the device was connected to an Ethernet PHY using the MII protocol, a clock block could be attached to a port connected to the RXCLK signal, which could then be used to drive the port which samples the data of the RXD signal.

By default, all ports are connected to clock block 0 which is designated the *reference clock block* and always runs at 100MHz.

You can access other clocks by declaring a variable of type `clock`. This type and initializers representing the clock on the device are declared in the `xs1.h` header file. For example, the following code declares a variable that allows you to access clock block 2:

```
#include <xs1.h>

clock clk = XS1_CLKBLK_2;
```

You can connect ports and clock blocks together using configuration library functions defined in `xs1.h`. Details of these functions are shown in the following sections.

6.3 Outputting data

A simple program that toggles a pin high and low is shown below:

```
#include <xs1.h>

out port p = XS1_PORT_1A;

int main(void) {
  p <: 1;
  p <: 0;
}
```

The declaration

```
out port p = XS1_PORT_1A;
```

declares an output port named p, which refers to the 1-bit port identifier 1A.

The statement

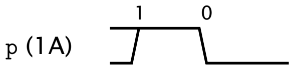
```
p <: 1;
```

outputs the value 1 to the port p, causing the port to drive its corresponding pin high. The port continues to drive its pin high until execution of the next statement

```
p <: 0;
```

which outputs the value 0 to the port, causing the port to drive its pin low. Figure 19 shows the output generated by this program.

Figure 19: Output waveform diagram



The pin is initially not driven; after the first output is executed it is driven high; and after the second output is executed it is driven low. In general, when outputting to an *n*-bit port, the least significant *n* bits of the output value are driven on the pins and the rest are ignored.

All ports must be declared as global variables, and no two ports may be initialized with the same port identifier. After initialization, a port may not be assigned to. Passing a port to a function is allowed as long as the port does not appear in more than one of a function's arguments, which would create an illegal alias.

6.4 Inputting data

The program below continuously samples the 4 pins of an input port, driving an output port high whenever the sampled value exceeds 9:

```
#include <xs1.h>

in  port p_in = XS1_PORT_4A;
out port p_out = XS1_PORT_1A;

int main(void) {
  int x;
  while (1) {
    p_in :> x;
    if (x > 9)
      p_out <: 1;
    else
      p_out <: 0;
  }
}
```

The declaration

```
in port p_in = XS1_PORT_4A;
```

declares an input port named `p_in`, which refers to the 4-bit port identifier 4A.

The statement

```
p_in :> x;
```

inputs the value sampled by the port `p_in` into the variable `x`. Figure 20 shows example input stimuli and expected output for this program.

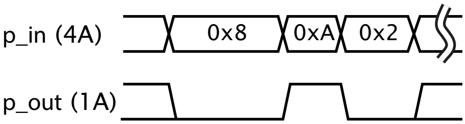


Figure 20: Input waveform diagram

The program continuously inputs from the port `p_in`: when 0x8 is sampled the output is driven low, when 0xA is sampled the output is driven high and when 0x2 is sampled the output is again driven low. Each input value may be sampled many times.

6.5 Waiting for a condition on an input pin

A port can trigger an event on one of two conditions on a pin: equal to or not equal to some value. The program below uses a `select` to count the transitions on a pin up to a certain value:

```
#include <xs1.h>

void wait_for_transitions(in port p, unsigned n) {
  unsigned i = 0;

  p :> x;
  while (i < n) {
    select {
      case p when pinsneq(x) :> x:
        i++;
        break;
    }
  }
}
```

The statement

```
p when pinsneq(x) :> x;
```

instructs the port `p` to wait until the value on its pins is not equal to `x` before sampling and providing an event to the task to react to. When the case is taken the current value is stored back in `x`.

As another example, a task could wait for an Ethernet preamble on a 4-bit port with the following condition:

```
p_eth_data when pinseq(0xD) :> void:
```

Here, `void` is used after the `:>` to show that the input value is not stored anywhere.

Selecting on conditional inputs is more power efficient than polling the port in software, because it allows the processor to idle, consuming less power, while the port remains active monitoring its pins.

6.6 Generating a clock signal

The program below configures a port to be clocked at a rate of 12.5MHz, outputting the corresponding clock signal with its output data:

```
#include <xs1.h>

out port p_out       = XS1_PORT_8A;
out port p_clock_out = XS1_PORT_1A;
clock    clk         = XS1_CLKBLK_1;

int main(void) {
  configure_clock_rate(clk, 100, 8);
  configure_out_port(p_out, clk, 0);
  configure_port_clock_output(p_clock_out, clk);
  start_clock(clk);

  for (int i=0; i<5; i++)
    p_out <: i;
}
```

The program configures the ports p_out and p_clock_out as illustrated in Figure 21.

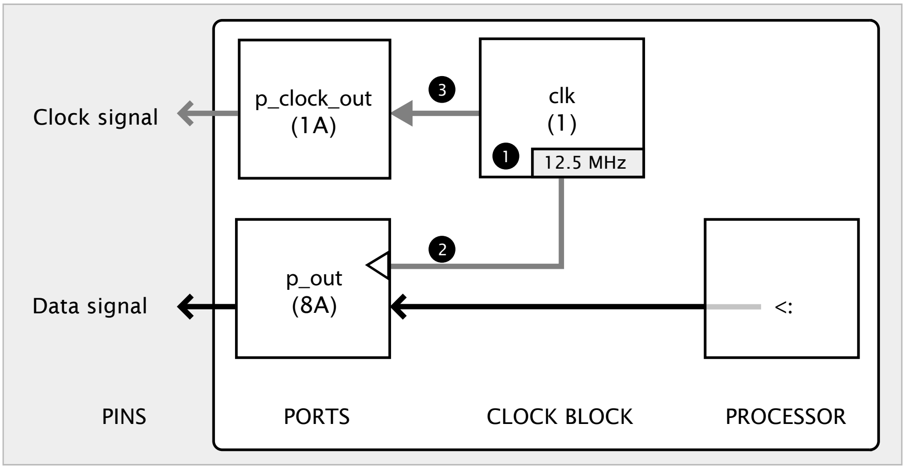


Figure 21: Port configuration diagram

The declaration

```
clock clk = XS1_CLKBLK_1;
```

declares a clock named clk, which refers to the clock block identifier XS1_CLKBLK_1. Clocks are declared as global variables, with each declaration initialized with a unique resource identifier.

The statement:

```
    configure_clock_rate(clk, 100, 8);
```

configures the clock `clk` to have a rate of 12.5MHz. The rate is specified as a fraction (100/8) because xC only supports integer arithmetic types.

The statement:

```
configure_out_port(p_out, clk, 0);
```

configures the output port `p_out` to be clocked by the clock `clk`, with an initial value of 0 driven on its pins.

The statement:

```
    configure_port_clock_output(p_clock_out, clk)
```

causes the clock signal `clk` to be driven on the pin connected to the port `p_clock_out`, which a receiver can use to sample the data driven by the port `p_out`.

The statement:

```
    start_clock(clk);
```

causes the clock block to start producing edges.

A port has an internal 16-bit counter, which is incremented on each falling edge of its clock. Figure 22 shows the port counter, clock signal and data driven by the port.

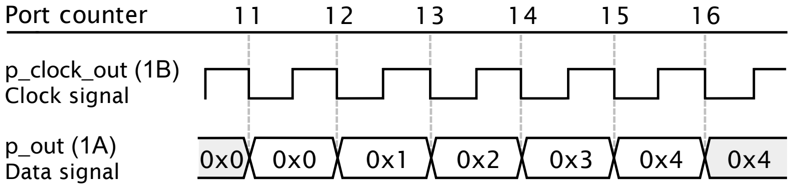


Figure 22: Waveform diagram

An output by the processor causes the port to drive output data on the next falling edge of its clock; the data is held by the port until another output is performed.

6.7 Using an external clock

The following program configures a port to synchronize the sampling of data to an external clock:

```
#include <xs1.h>

in port p_in       = XS1_PORT_8A;
in port p_clock_in = XS1_PORT_1A;
clock   clk        = XS1_CLKBLK_1;

int main(void) {
  configure_clock_src(clk, p_clock_in);
  configure_in_port(p_in, clk);

  start_clock(clk);
  for (int i=0; i<5; i++)
    p_in :> int x;
}
```

The program configures the ports `p_in` and `p_clock_in` as illustrated in Figure 23.

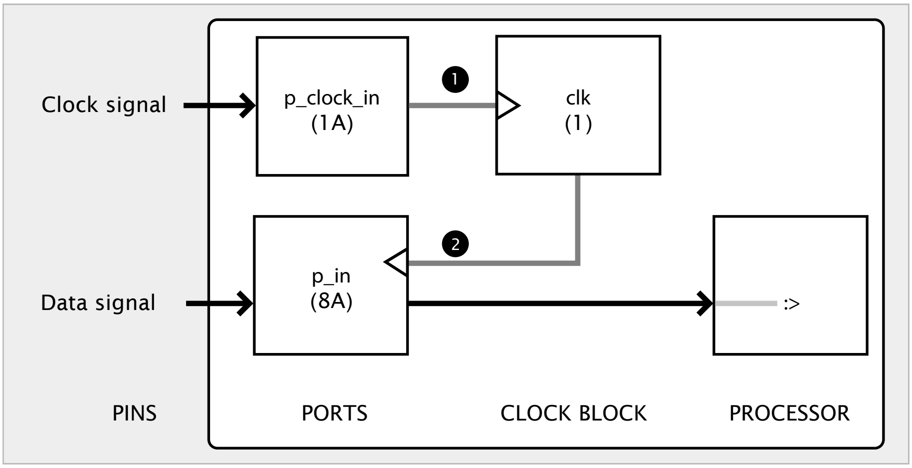


Figure 23: Port configuration diagram

The statement:

```
configure_clock_src(clk, p_clock_in);
```

configures the 1-bit input port `p_clock_in` to provide edges for the clock `clk`. An edge occurs every time the value sampled by the port changes.

The statement :

```
configure_in_port(p_in, clk);
```

configures the input port p_in to be clocked by the clock clk.

Figure 24 shows the port counter, clock signal, and example input stimuli.

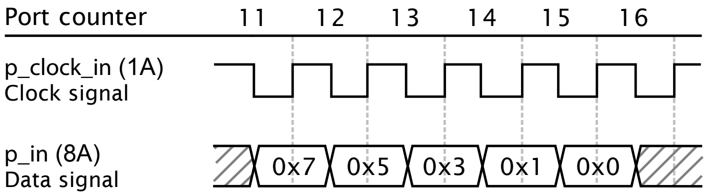


Figure 24: Waveform diagram

An input by the processor causes the port to sample data on the next rising edge of its clock. The values input are 0x7, 0x5, 0x3, 0x1 and 0x0.

6.8 Performing I/O on specific clock edges

It is often necessary to perform an I/O operation on a port at a specific time with respect to its clock. The program below drives a pin high on the third clock period and low on the fifth:

```
void do_toggle(out port p) {
  int count;
  p <: 0 @ count;    // timestamped output
  while (1) {
    count += 3;
    p @ count <: 1;  // timed output
    count += 2;
    p @ count <: 0;  // timed output
  }
}
```

The statement

```
p <: 0 @ count;
```

performs a *timestamped output*, outputting the value 0 to the port p and reading into the variable count the value of the port counter when the output data is driven on the pins. The program then increments count by a value of 3 and performs a *timed output* statement

```
p @ count <: 1;
```

This statement causes the port to wait until its counter equals the value count+3 (advancing three clock periods) and to then drive its pin high. The last two statements delay the next output by two clock periods. Figure 25 shows the port counter, clock signal and data driven by the port.

The port counter is incremented on the falling edge of the clock. On intermediate edges for which no value is provided, the port continues to drive its pins with the data previously output.

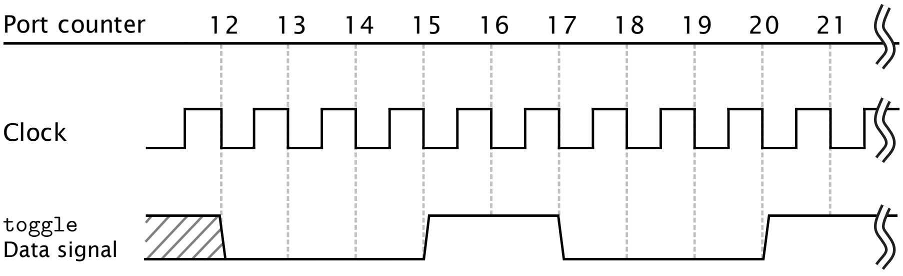


Figure 25: Waveform diagram

6.9 Using a buffered port

XMOS devices provide buffers that can improve the performance of programs that perform I/O on clocked ports. A buffer can hold data output by the processor until the next falling edge of the port's clock, allowing the processor to execute other instructions during this time. It can also store data sampled by a port until the processor is ready to input it. Using buffers, a single thread can perform I/O on multiple ports in parallel.

The following program uses a buffered port to decouple the sampling and driving of data on ports from a computation:

```
#include <xs1.h>

in  buffered port:8 p_in        = XS1_PORT_8A;
out buffered port:8 p_out       = XS1_PORT_8B;
in port             p_clock_in = XS1_PORT_1A;
clock               clk         = XS1_CLKBLK_1;

int main(void) {
  configure_clock_src(clk, p_clock_in);
  configure_in_port(p_in, clk);
  configure_out_port(p_out, clk, 0);
  start_clock(clk);
  while (1) {
    int x;
    p_in  :> x;
    p_out <: x + 1;
    f();
  }
}
```

The program configures the ports `p_in`, `p_out` and `p_clock_in` as illustrated in Figure 26.

The declaration

```
in buffered port:8 p_in = XS1_PORT_8A;
```

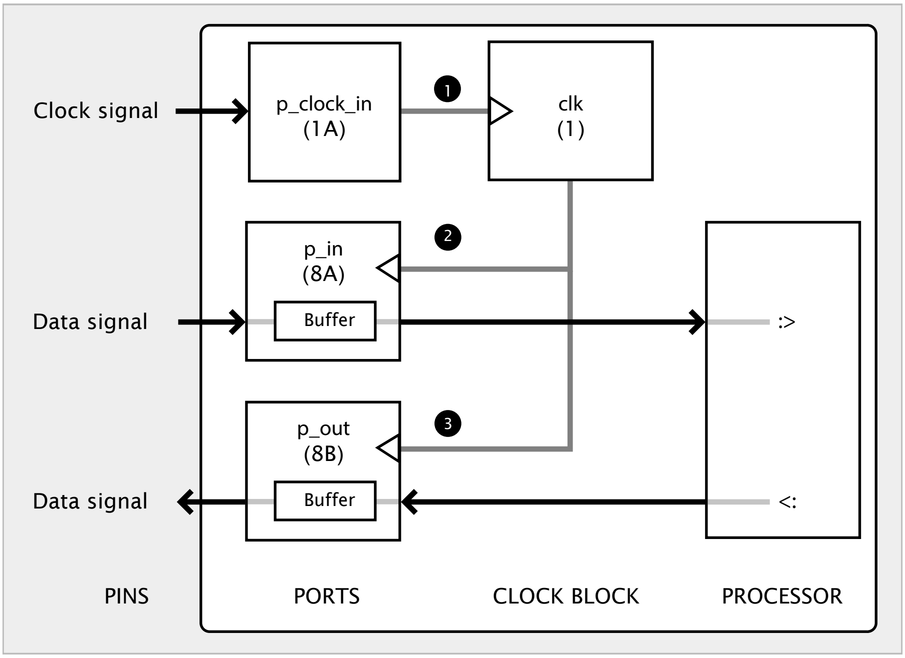


Figure 26: Port configuration diagram

declares a buffered input port named `p_in`, which refers to the 8-bit port identifier 8A.

The statement:

```
configure_clock_src(clk, p_clock_in);
```

configures the 1-bit input port `p_clock_in` to provide edges for the clock `clk`.

The statement:

```
configure_in_port(p_in, clk);
```

configures the input port `p_in` to be clocked by the clock `clk`.

The statement:

```
configure_out_port(p_out, clk, 0);
```

configures the output port `p_out` to be clocked by the clock `clk`, with an initial value of 0 driven on its pins.

Figure 27 shows example input stimuli and expected output for this program. It also shows the relative waveform of the statements executed in the `while` loop by the processor.

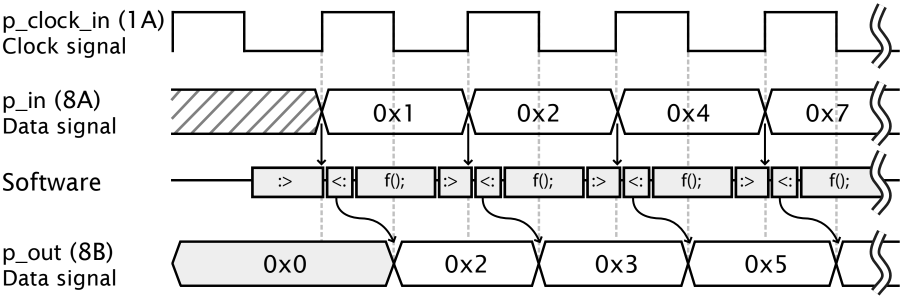


Figure 27: Waveform diagram relative to processor execution

The first three values input are 0x1, 0x2 and 0x4, and in response the values output are 0x2, 0x3 and 0x5.

Figure 28 illustrates the buffering operation in the hardware. It shows the processor executing the `while` loop that outputs data to the port. The port buffers this data so that the processor can continue executing subsequent instructions while the port drives the data previously output for a complete period. On each falling edge of the clock, the port takes the next byte of data from its buffer and drives it on its pins. As long as the instructions in the loop execute in less time than the port's clock period, a new value is driven on the pins on every clock period.

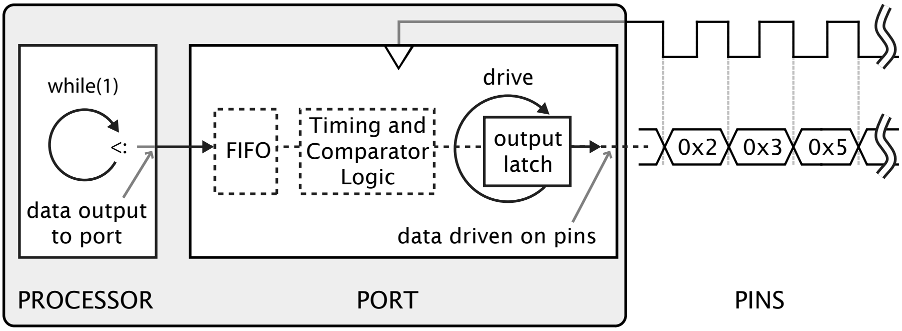


Figure 28: Port hardware logic

The fact that the first input statement is executed before a rising edge means that the input buffer is not used. The processor is always ready to input the next data before it is sampled, which causes the processor to block, effectively slowing itself down to the rate of the port. If the first input occurs after the first value is sampled, however, the input buffer holds the data until the processor is ready to accept it and each output blocks until the previously output value is driven.

⚠ Timed operations represent time in the future. The waveform and comparator logic allows timed outputs to be buffered, but for timed and conditional inputs the buffer is emptied before the input is performed.

6.10 Synchronising clocked I/O on multiple ports

By configuring more than one buffered port to be clocked from the same source, a single thread can cause data to be sampled and driven in parallel on these ports. The program below first synchronizes itself to the start of a clock period, ensuring the maximum amount of time before the next falling edge, and then outputs a sequence of 8-bit character values to two 4-bit ports that are driven in parallel.

```
#include <xs1.h>

out buffered port p:4      = XS1_PORT_4A;
out buffered port q:4      = XS1_PORT_4B;
in           port p_clock_in = XS1_PORT_1A;
clock        clk           = XS1_CLKBLK_1;

int main(void) {

  configure_clock_src(clk, p_clock_in);
  configure_out_port(p, clk, 0);
  configure_out_port(q, clk, 0);
  start_clock(clk);

  p <: 0;  // start an output
  sync(p); // synchronize to falling edge

  for (char c='A'; c<='Z'; c++) {
    p <: (c & 0xF0) >> 4;
    q <: (c & 0x0F);
  }
}
```

The statement

```
sync(p);
```

causes the processor to wait until the next falling edge on which the last data in the buffer has been driven for a full period, ensuring that the next instruction is executed just after a falling edge. This ensures that the subsequent two output statements in the loop are both executed in the same clock period. Figure 29 shows the data output by the processor and driven by the two ports.

The recommended way to synchronize to a rising edge is to clear the buffer using the standard library function `clearbuf` and then make an input.

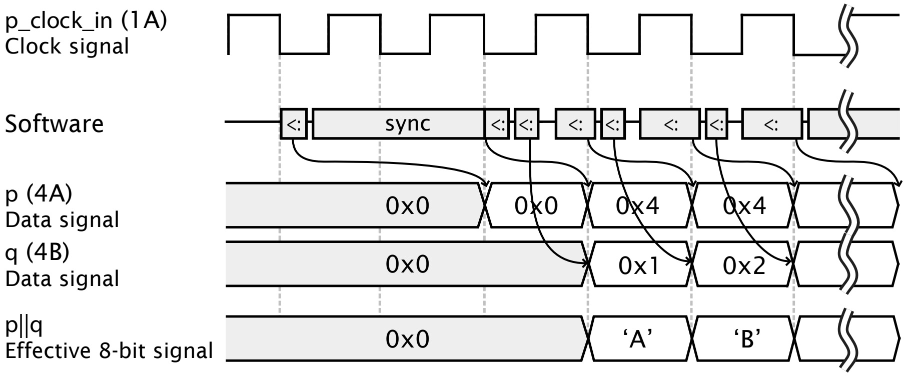


Figure 29: Processor synchronizing data on two output ports

6.11 Serializing output data using a port

XMOS devices provide hardware support for operations that frequently arise in communication protocols. A port can be configured to perform *serialization*, useful if data must be communicated over ports that are only a few bits wide, and *strobing*, useful if data is accompanied by a separate data valid signal. Offloading these tasks to the ports frees up more processor time for executing computations.

A clocked port can serialize data, reducing the number of instructions required to perform an output. The program below outputs a 32-bit value onto 8 pins, using a clock to determine for how long each 8-bit value is driven.

```
#include <xs1.h>

out buffered port:32 p_out    = XS1_PORT_8A;
in port p_clock_in            = XS1_PORT_1A;
clock clk                     = XS1_CLKBLK_1;

int main(void) {
  int x = 0xAA00FFFF;
  configure_clock_src(clk, p_clock_in);
  configure_out_port(p_out, clk, 0);
  start_clock(clk);

  while (1) {
    p_out <: x;
    x = f(x);
  }
}
```

The declaration

```
out buffered port:32 p_out = XS1_PORT_8A;
```

declares the port `p_out` to drive 8 pins from a 32-bit *shift register*. The type `port:32` specifies the number of bits that are transferred in each output operation (the *transfer width*). The initialization `XS1_PORT_8A` specifies the number of physical pins connected to the port (the *port width*). Figure 30 shows the data driven by this program.

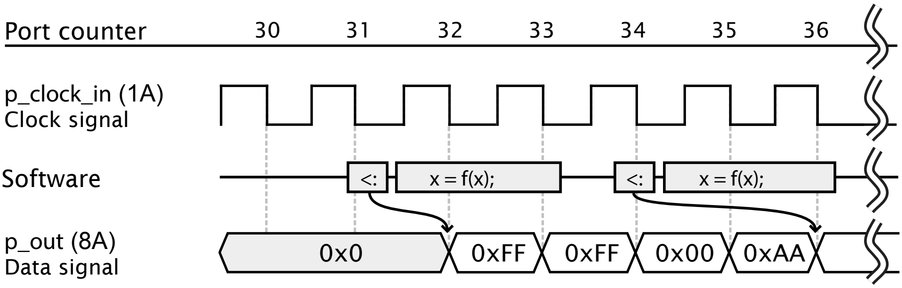


Figure 30: Serialized output waveform diagram

By offloading the serialization to the port, the processor has only to output once every 4 clock periods. On each falling edge of the clock, the least significant 8 bits of the shift register are driven on the pins; the shift register is then right-shifted by 8 bits.

Ports used for serialization must be qualified with the keyword `buffered`; see XM-000971-PC for further explanation.

6.12 Deserializing input data using a port

A port can deserialize data, reducing the number of instructions required to input data. The program below performs a 4-to-8 bit conversion on an input port, controlled by a 25MHz clock.

```
#include <xs1.h>
in  buffered port:8 p_in = XS1_PORT_4A;
out port p_clock_out     = XS1_PORT_1A;
clock clk                = XS1_CLKBLK_1;

int main(void) {
  configure_clock_rate(clk, 100, 4);
  configure_in_port(p_in, clk);
  configure_port_clock_output(p_clock_out, clk);
  start_clock(clk);
  while (1) {
    int x;
    p_in :> x;
    f(x);
} }
```

The program declares `p_in` to be a 4-bit wide port with an 8-bit transfer width, meaning that two 4-bit values can be sampled by the port before they must be input by the processor. As with output, the deserializer reduces the number of instructions required to obtain the data. Figure 31 shows example input stimuli and the period during which the data is available in the port's buffer for input.

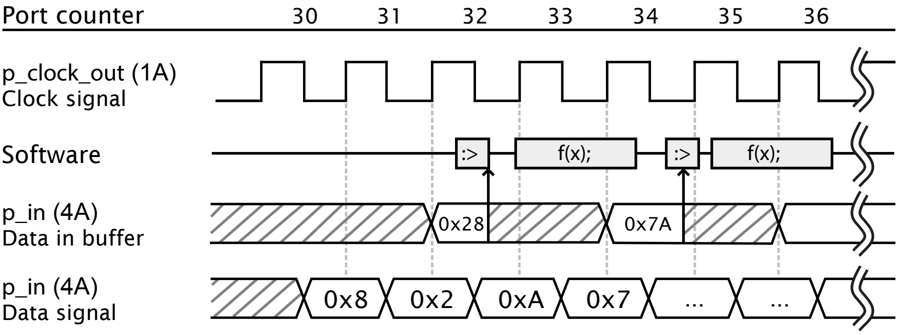


Figure 31: Deserialized input waveform diagram

Data is sampled on the rising edges of the clock and, when shifting, the least significant nibble is read first. The sampled data is available in the port's buffer for input for two clock periods. The first two values input are 0x28 and 0x7A.

6.13 Inputting data accompanied by a data valid signal

A clocked port can interpret a *ready-in* strobe signal that determines the validity of the accompanying data. The program below inputs data from a clocked port only when a ready-in signal is high.

```
#include <xs1.h>

in buffered port:8 p_in = XS1_PORT_4A;
in port p_ready_in      = XS1_PORT_1A;
in port p_clock_in      = XS1_PORT_1B;
clock clk               = XS1_CLKBLK_1;

int main(void) {
  configure_clock_src(clk, p_clock_in);
  configure_in_port_strobed_slave(p_in, p_ready_in, clk);
  start_clock(clk);

  p_in :> void;
}
```

The statement

```
configure_in_port_strobed_slave(p_in, p_ready_in, clk);
```

configures the input port p_in to be sampled only when the value sampled on the port p_ready_in equals 1. The ready-in port must be 1-bit wide. Figure 32 shows example input stimuli and the data input by this program.

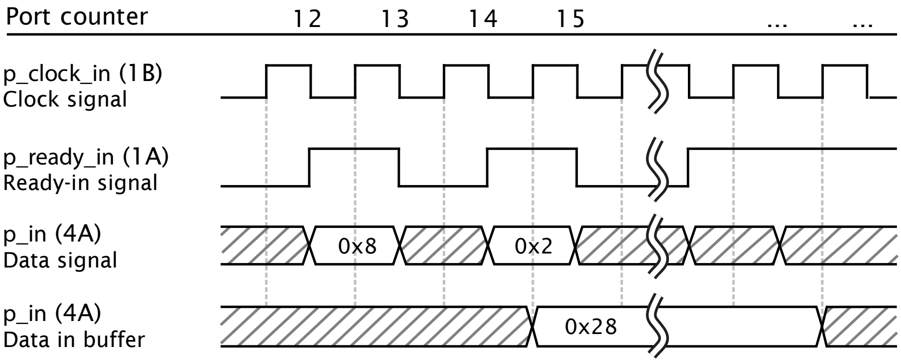


Figure 32: Input data with data valid signal

Data is sampled on the rising edge of the clock whenever the ready-in signal is high. The port samples two 4-bit values and combines them to produce a single 8-bit value for input by the processor; the data input is 0x28. The ports have a single-entry buffer, which means that data is available for input until the ready-in signal is high for the next two rising edges of the clock. Note that the port counter is incremented on every clock period, regardless of whether the strobe signal is high.

6.14 Outputting data and a data valid signal

A clocked port can generate a *ready-out* strobe signal whenever data is output. The program below causes an output port to drive a data valid signal whenever data is driven on a 4-bit port.

```
#include <xs1.h>

out buffered port:8 p_out = XS1_PORT_4B;
out port p_ready_out      = XS1_PORT_1A;
in port p_clock_in        = XS1_PORT_1B;
clock clk                 = XS1_CLKBLK_1;

int main(void) {
  configure_clock_src(clk, p_clock_in);
  configure_out_port_strobed_master(p_out, p_ready_out, clk, 0);
  start_clock(clk);

  p_out <: 0x85;
}
```

The statement

```
configure_out_port_strobed_master(p_out, p_ready_out, clk, 0);
```

configures the output port p_out to drive the port p_ready_out high whenever data is output. The ready-out port must be 1-bit wide. Figure 33 shows the data and strobe signals driven by this program.

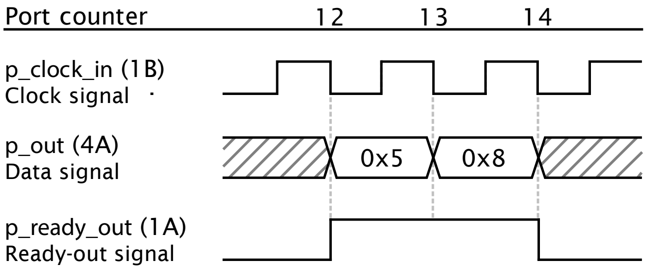


Figure 33: Output data with data valid signal

The port drives two 4-bit values over two clock periods, raising the ready-out signal during this time.

It is also possible to implement control flow algorithms that output data using a ready-in strobe signal and that input data using a ready-out strobe signal; when both signals are configured, the port implements a symmetric strobe protocol that uses a clock to handshake the communication of the data (see XM-000969-PC).

Ports used for strobing must be qualified with the keyword buffered; see XM-000971-PC for further explanation.

Part B

Programming examples

CONTENTS

7 Flashing LEDs

The simplest flashing LED program loops forever alternating between driving a port high and driving it low. The <: operator drives a value on a port. In between outputs you can make the task pause with the delay_milliseconds function. There are several delay_ functions defined in timer.h.

```
void flashing_led_task1(port p, int delay_in_ms) {
  while (1) {
    p <: 0;
    delay_milliseconds(delay_in_ms);
    p <: 1;
    delay_milliseconds(delay_in_ms);
  }
}
```

The simple example above will block the logical core while waiting between port outputs. This is quite inefficient. To allow other computation to occur in between outputs you need to use timer events. This requires declaring a variable of timer type and then using a select statement which reacts to events on this timer.

```
[[combinable]]
void flashing_led_task2(port p, int delay_in_ms) {
  timer tmr;
  unsigned t;
  // Convert delay from ms to 100Mhz timer ticks
  const int delay_ticks = delay_in_ms * 100000;
  // The value we are going to output
  unsigned val = 0;

  // read the initial timer value
  tmr :> t;
  while (1) {
    select {
    // This case will event when the timer moves past (t + delay_ticks) i.e
    // delay_ticks after when we took the timestamp t
    case tmr when timerafter(t + delay_ticks) :> void:
      p <: val;
      val = ~val;
      // set up the next event
      t += delay_ticks;
      break;
    }
  }
}
```

Note that this function has been marked as [[combinable]]. This means it can share a logical core with other combinable functions that will handle other events in between the port outputs of this flashing LED task.

8 Handling button presses

To handle buttons a task needs to event when a pin changes value. This can be done using the `select` construct and the `pinsneq` predicate on the select case:

```
// This function is combinable - it can run on a logical core with other
   ↪ tasks.
[[combinable]]
void task1(port p_button)
{
  // The last read value off the port.
  int current_val = 0;
  while (1) {
   select {
   // event when the button changes value
   case p_button when pinsneq(current_val) :> int new_val:
     if (new_val == 1) {
       printf("Button up\n");
     } else {
       printf("Button down\n");
     }
     current_val = new_val;
     break;
   }
  }
}
```

This code will react when the I/O pins change value. However, due to the button bouncing up and down, after a button is pressed the I/O pin will change value many times, very quickly. To avoid reacting to each of these changes you can add a debouncing period.

To do this, add a guard to the select case. This guard says do not react to the button unless the variable `is_stable` evaluates to true (i.e. non-zero). When a button is pressed `is_stable` is set to 0 and a timeout is setup. A separate case handles this timeout expiring (using a `timer`) at which point `is_stable` is set back to 1.

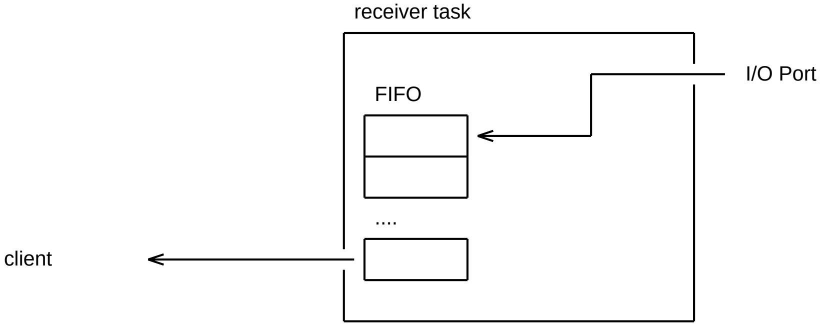
```
[[combinable]]
void task1a(port p_button)
{
  int current_val = 0;
  int is_stable = 1;
  timer tmr;
  const unsigned debounce_delay_ms = 50;
  unsigned debounce_timeout;
  while (1) {
   select {
   // If the button is "stable", react when the I/O pin changes value
   case is_stable => p_button when pinsneq(current_val) :> current_val:
     if (current_val == 1) {
       printf("Button up\n");
     } else {
       printf("Button down\n");
     }
     is_stable = 0;
     int current_time;
     tmr :> current_time;
     // Calculate time to event after debounce period
     // note that XS1_TIMER_HZ is defined in timer.h
     debounce_timeout = current_time + (debounce_delay_ms * XS1_TIMER_HZ);
     break;

   // If the button is not stable (i.e. bouncing around) then select
   // when we the timer reaches the timeout to renter a stable period
   case !is_stable => tmr when timerafter(debounce_timeout) :> void:
     is_stable = 1;
     break;
   }
  }
}
```

9 A buffered receiver

IN THIS CHAPTER

- Full example

This example shows a receiver task that reads data off external I/O pins, and buffers that data. A client task can read the data out of the buffer.



Buffering data in this way decouples the client task from the receiver so that the client is not bound by the real time constraints of the I/O. Note that *you do not always have to buffer between input and the client*. In real-time streaming applications, it better to directly output to the client and design the client to keep up with the data rate. However, buffering is sometimes needed in the following cases:

- When he input rate is bursty, the max data rate is higher than the client can deal with but the average data rate is not
- When the behaviour of the client is bursty (this can be in the case where the client is dealing with several events on one logical core) so it will consume at irregular intervals.

In both cases, it is important to determine whether overflow of the buffer is possible and, if so, what the application will do in this case.

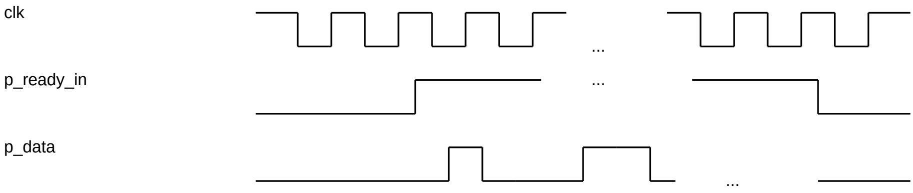
The receiver receives data on a simple clocked port, 32 bits of data at a time. When it receives data on the port, it will place the data in a FIFO buffer and notify the client. The client can then pull 32-bit words out of this FIFO.

The first thing to define in the program is the interface between the receiver task and its client.

```
interface receiver_if {

  // This *notification* function signals to the client when there is
  // data in the buffer. Since it is a notification function it is
  // special - instead of being called by the client it is used
  // by the client to event when data is ready.
  [[notification]] slave void data_ready();

  // This function can be used by the client to pull data out of the fifo.
  // It clears the notification raised by the data_ready() function.
  [[clears_notification]] unsigned get_data();

  // This function can be called to check if the receiver has any data in
  // the buffer.
  // Generally, you do not need to poll the receiver with this function
  // since you can use the data_ready() notification instead
  unsigned has_data();
};
```

The receiver task takes arguments to set it up. It takes the server end of an interface connection using the `receiver_if` interface - this will be connected to the client. It also takes the buffer size required and the ports and clock block variable for using the external I/O pins. The I/O interface requires a data pin, a clock block and a pin to provide a readyIn signal.

The I/O protocol is a simple protocol directly supported by the XMOS Hardware-Response port blocks. The clock block provides a clock for the data. When the externally driven *p_ready_in* signal is driven high, it signals the start of the data. After that data is clocked into the *p_data* port on the rising edge of the clock. Since the data port is a `buffered` port, the data is deserialize into 32-bit chunks, so the program will receive a 32-bit word at a time from the port.



This is the prototype of the receiver task:

```
void receiver(server interface receiver_if i,
              static const unsigned bufsize,
              in buffered port:32 p_data,
              clock clk,
              in port p_ready_in)
{
```

Within definition of the task the first thing required is to define the local state. The `buffer` array provides the memory space for the FIFO. To implement a FIFO, the `fifo_first_elem` and `fifo_last_elem` hold the indices to the first and last element in the FIFO. All the array elements between these indices hold the data (the FIFO may wrap around the end of the array back to the beginning).

```
unsigned buffer[bufsize];
unsigned fifo_first_elem = 0, fifo_last_elem = 0;
```

The initial part of the task sets up the port block. The protocol on the pins is one supported by the Hardware-Response port blocks, so you can configure the port using a library function to set it to *Strobed Slave* mode (i.e. data input is governed by the readyIn signal). The port configuration functions are found in `xs1.h`.

```
configure_in_port_strobed_slave(p_data, p_ready_in, clk);
```

The main body of the task is a loop with a `select` inside. This select will either react to the port providing input, or to a request from the client over the interface connection:

```
while (1) {
  select {
  case p :> unsigned data:
    // handle port input
    ..
  case i.get_data() -> unsigned result:
    // request from client to get data, pop element off fifo
    // and place it in the return value 'result'
    ..
  case i.has_data() -> unsigned result:
   // request form client to determine if there is data in buffer
   // put 1 or 0 in the retrun value 'result'
   ..
  }
}
```

When the port signals that it has data, the tasks reads it into the `data` variable.

```
case p_data :> unsigned data:
```

To handle the port input the program works out where it needs to be added to the FIFO by adding one to the last element index (and wrapping round in the buffer if needed).

```
unsigned new_last_elem = fifo_last_elem + 1;
if (new_last_elem == bufsize)
  new_last_elem = 0;
```

If the last element index wraps all the way around to the beginning of the FIFO, there is buffer overflow. In this case the task just drops the data but different overflow handling code could be added here.

```
if (new_last_elem == fifo_first_elem) {
  //handle buffer overflow
  break;
}
```

If there is room in the buffer, the data is inserted into the array and the last element index is updated.

```
buffer[fifo_last_elem] = data;
fifo_last_elem = new_last_elem;
```

Finally, the server calls the `data_ready` notification. This signals to the client that there is some data in the buffer.

```
i.data_ready();
```

The following case responds to a client request for data. The return value back to the client is declared as a variable `result`. The body of this case can set this variable to pass a return value back to the client.

```
case i.get_data() -> unsigned result:
```

The data to extract from the FIFO is in the array at the position marked by the first element index variable. However, if this is the same as the last element then the buffer is empty. In this case the task returns the value 0 to the client, but different buffer underflow handling code could be added here.

```
if (fifo_first_elem == fifo_last_elem) {
  // handle buffer underflow
  result = 0;
  break;
}
```

To pop an element from the FIFO, the `result` variable needs to be set and the first element index variable needs to be incremented (possibly wrapping around the buffer array).

```
result = buffer[fifo_first_elem];
fifo_first_elem++;
if (fifo_first_elem == bufsize)
  fifo_first_elem = 0;
```

Finally, if the FIFO is not empty, the task re-notifies the client that data is available.

```
if (fifo_first_elem != fifo_last_elem)
  i.data_ready();
```

The final request the receiver task handles is from the client requesting whether data is available. This case is quite simple, just needing to return the current state based on the index variables for the FIFO.

```
case i.has_data() -> unsigned result:
  // request form client to determine if there is data in buffer
  // put 1 or 0 in the retrun value 'result'
  result = (fifo_first_elem != fifo_last_elem);
  break;
```

Tasks can attach to this receiver task and access data via the inteface connection. For example, the following `consumer` task takes a client end of interface connection to the receiver:

```
void consumer(client interface receiver_if i) {
  // This consumer task can wait for the data from the receiver task
  while (1) {
    select {
    case i.data_ready():
      unsigned x = i.get_data();
      // handle the data here
      break;
    }
  }
}
```

The tasks can be run in parallel using a `par` statement and connected together via an interface:

```
int main()
{
  interface receiver_if i;
  par {
    consumer(i);
    receiver(i, 1024, p_data, clk, p_ready_in);
  }
  return 0;
}
```

9.1 Full example

```
#include <xs1.h>


interface receiver_if {

  // This *notification* function signals to the client when there is
  // data in the buffer. Since it is a notification function it is
  // special - instead of being called by the client it is used
  // by the client to event when data is ready.
  [[notification]] slave void data_ready();

  // This function can be used by the client to pull data out of the fifo.
  // It clears the notification raised by the data_ready() function.
  [[clears_notification]] unsigned get_data();

  // This function can be called to check if the receiver has any data in
  // the buffer.
  // Generally, you do not need to poll the receiver with this function
  // since you can use the data_ready() notification instead
  unsigned has_data();
};
```

```
void receiver(server interface receiver_if i,
              static const unsigned bufsize,
              in buffered port:32 p_data,
              clock clk,
              in port p_ready_in)
{

  unsigned buffer[bufsize];
  unsigned fifo_first_elem = 0, fifo_last_elem = 0;


  configure_in_port_strobed_slave(p_data, p_ready_in, clk);

  while (1) {
    select {
    case p_data :> unsigned data:
      unsigned new_last_elem = fifo_last_elem + 1;
      if (new_last_elem == bufsize)
        new_last_elem = 0;
      if (new_last_elem == fifo_first_elem) {
        //handle buffer overflow
        break;
      }
      buffer[fifo_last_elem] = data;
      fifo_last_elem = new_last_elem;
      i.data_ready();
      break;

    case i.get_data() -> unsigned result:
      if (fifo_first_elem == fifo_last_elem) {
        // handle buffer underflow
        result = 0;
        break;
      }
      result = buffer[fifo_first_elem];
      fifo_first_elem++;
      if (fifo_first_elem == bufsize)
        fifo_first_elem = 0;

      if (fifo_first_elem != fifo_last_elem)
        i.data_ready();
      break;

    case i.has_data() -> unsigned result:
      // request form client to determine if there is data in buffer
      // put 1 or 0 in the retrun value 'result'
      result = (fifo_first_elem != fifo_last_elem);
      break;
    }
  }

}
```

```
void consumer(client interface receiver_if i) {
  // This consumer task can wait for the data from the receiver task
  while (1) {
    select {
    case i.data_ready():
      unsigned x = i.get_data();
      // handle the data here
      break;
    }
  }
}


in buffered port:32 p_data = XS1_PORT_1A;
clock clk = XS1_CLKBLK_1;
in port p_ready_in = XS1_PORT_1B;

int main()
{
  interface receiver_if i;
  par {
    consumer(i);
    receiver(i, 1024, p_data, clk, p_ready_in);
  }
  return 0;
}
```

10A startKIT tic-tac-toe game

IN THIS CHAPTER

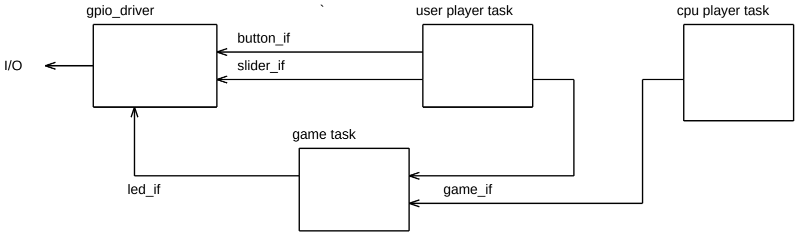
The tic-tac-toe demo is a program that plays tic-tac-toe (also known as noughts and crosses) on a XMOS startKIT development board. It is provided as a demonstation program of how to program the device. The 3x3 display of LEDs shows the board status:

- Full LEDs: user player (marking a O)
- Dimmed LEDs: computer player (marking a 1)

When it is the user's move, one of the LEDs flashes - this is a cursor and it can be moved by swiping the sliders. Pressing the button makes a move, and the computer player will make the next move.

The application consists of four tasks:

- The `startkit_gpio_driver` task drives the LEDs on the device (using PWM to make the lights glow at different levels of intensity), the capacitive sensors on the sliders and the button. It has three interface connections connected to it - one for the button, one for the LEDs and one for the slider.
- The `game` task which controls the game state. It is connected to the two player tasks and to the gpio task to drive the LEDs to display the game state.
- The `user_player` task which receives notifications from and sends commands to the game task. It also connects to the gpio task to read the sliders and buttons when the user player makes a move.
- The `cpu_player` task which receives notifications from and send commands to the game task. It uses an internal AI algorithm to determine what move to make.



The four tasks are spread across two logical cores. One logical core runs the gpio driver which needs to be responsive to the I/O pins. The other core runs the other three tasks which do not have real-time constraints and share the core via co-operative multitasking.

The main program consists of a par statement to run all the tasks in parallel with three tasks placed on the same core. The declarations are typedefs of interface types to connect the tasks together.

```
// The port structure required for the GPIO task
startkit_gpio_ports gpio_ports =
  {XS1_PORT_32A, XS1_PORT_4A, XS1_PORT_4B, XS1_CLKBLK_1};

int main() {
  startkit_button_if i_button;
  startkit_led_if i_led;
  slider_if i_slider_x, i_slider_y;
  player_if i_game[2];
  par {
    on tile[0].core[0]: game(i_game, i_led);
    on tile[0].core[0]: user_player(i_game[0],
                                    i_slider_x, i_slider_y, i_button);
    on tile[0].core[0]: computer_player(i_game[1]);
    on tile[0]: startkit_gpio_driver(i_led, i_button,
                                     i_slider_x, i_slider_y,
                                     gpio_ports);

  }
  return 0;
}
```

10.1 The game task and player interface

The game task controls the board state and blinking cursor, displaying them via the LED connection to the GPIO driver task. The key interface is between the game task and the two player tasks. This includes functions for getting and updating the current game state.

The game task uses notifications to inform the player tasks that a move is required.

```
typedef enum board_val_t {
  BOARD_EMPTY,
  BOARD_X,
  BOARD_O,
} board_val_t;

typedef interface player_if {
  // This function will fill in the supplied board array with the
  // current game state.
  void get_board(char board[3][3]);

  // Set the user cursor to the specified position.
  void set_cursor(unsigned row, unsigned col);

  // Clear the user cursor from the board.
  void clear_cursor();

  // This function can be called by players to determine whether they
  // are the X piece or the O piece.
  board_val_t get_my_val();

  // This notification will be signalled by the game to the player when
  // a move is required from the player.
  [[notification]] slave void move_required();

  // This function is called by the player to make a move in the specified
  // position.
  [[clears_notification]] void play(unsigned row, unsigned col);
} player_if;

// This task controls the game state providing two connections to the
// two players of the game.
[[combinable]]
void game(server player_if players[2], client startkit_led_if i_led);
```

Note that game task is *combinable* allowing it to share processing time with other combinable tasks in between reacting to events.

Full details of the implementation of the game task can be found in code base delivered via the Community Code browser in the xTIMEcomposer Studio.

10.2 The user player task

The user player task connects to the game task and the gpio task. It is either in a playing state or idle state. When it gets a move request notification from the game task, it moves into the playing state and sets up the cursor in the game task. Whilst in the playing state it reacts to slider and button events to move the cursor and complete the game move.

```
[[combinable]]
void user_player(client player_if i_game,
                 client slider_if i_slider_x,
                 client slider_if i_slider_y,
                 client startkit_button_if i_button)
{
```

The task has some local state - a variable to determine whether it is in a playing state or not, x and y variables to store the current position of the cursor and a local copy of the board state.

```
int playing = 0;
int x = 0, y = 0;
char board[3][3];
```

The main body of the task consists of a `while (1) select` loop.

```
while (1) {
  select {
```

The first case in the select reacts when the game tasks requests a move is played. This causes the player task to enter the playing state. At this point the task takes a copy of the board state and sets up the cursor by interacting with the game task.

```
case i_game.move_required():
  // Get a local copy of the board state
  i_game.get_board(board);
  // Find an empty place to place the cursor
  int found = 0;
  for (int i = 0; i < 3 && !found; i++) {
    for (int j = 0 ; j < 3 && !found; j++) {
      if (board[i][j] == BOARD_EMPTY) {
        x = i;
        y = j;
        found = 1;
      }
    }
  }
  i_game.set_cursor(x, y);
  playing = 1;
  break;
```

If the button is pressed it causes an event on the connection to the gpio driver. The following case reacts to this event and if the task is in the playing state and the cursor is at an empty space on the board, it calls the `play` function over the connection to the game task to play the move, and then leave the playing state.

```
case i_button.changed():
  button_val_t val = i_button.get_value();
  if (playing && val == BUTTON_DOWN) {
    if (board[x][y] == BOARD_EMPTY) {
      // Make the move
      i_game.clear_cursor();
      i_game.play(x, y);
      playing = 0;
    }
  }
  break;
```

The task also reacts to changes in the slider. In this case it moves the cursor if the slider notifies the task of a LEFTING or RIGHTING event (indicating that the user has swiped left or right).

```
case i_slider_x.changed_state():
  sliderstate state = i_slider_x.get_slider_state();
  if (!playing)
    break;
  if (state != LEFTING && state != RIGHTING)
    break;
  int dir = state == LEFTING ? 1 : -1;
  int new_x = x + dir;
  if (new_x >= 0 && new_x < 3) {
    x = new_x;
    i_game.set_cursor(x, y);
  }
  break;
```

The case to handle the vertical slider is similar. Handling move requests, slider swipes and button presses completes the player task.

10.3 The computer player task

The computer player uses an auxiliary function to find a move for the computer to play. It fills the best_i and best_j reference parameters with the position of the best move based on an AI algorithm that searches possible future move combinations. The parameter board is the current board state and the parameter me indicates which type of piece the computer is playing.

```
static void find_best_move(char board[3][3],
                           int &best_i,
                           int &best_j,
                           board_val_t me);
```

With this function, the computer player task is quite simple. It just waits for the game tasks to request a move, gets a copy of the board state, determines the best move to play and then communicates back with the game state playing the move.

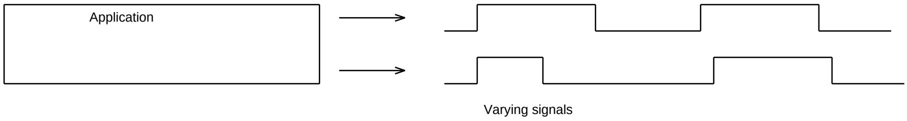
```
[[combinable]]
void computer_player(client player_if game)
{
  while (1) {
    select {
    case game.move_required():
      char board[3][3];
      int i, j;
      game.get_board(board);
      find_best_move(board, i, j, game.get_my_val());
      game.play(i, j);
      break;
    }
  }
}
```

Full details of the implementation of `find_best_move` function can be found in code base delivered via the Community Code browser in the xTIMEcomposer Studio.

11 Generating several controllable pulse signals

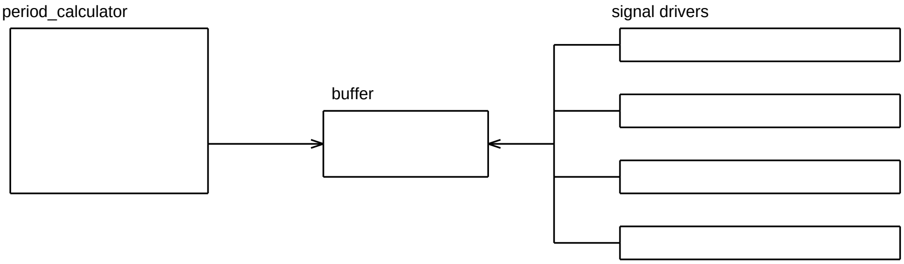
IN THIS CHAPTER

This case study covers how to write an application in C with XMOS multicore extensions (xC) that implements a controllable signal generator *i.e.* an application that outputs several signals whose period varies based on the application:



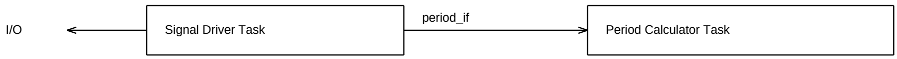
This type of application is similar to those used to control external hardware such as stepper motors.

The application is made up of a period_calculator task that works out what period the signals should be. This will vary depending on the purpose of the generator. It connects to several signal driving tasks via a buffer task. The buffer task decouples the calculation of the period from the driving tasks.



11.1 Obtaining the period

The signal driver task needs to know the period to output. To do this it obtains the period from a separate task running in parallel. Tasks can talk to each other over defined *interfaces*. So, between the port driving task and the calculator you can define an interface that allows the port driver to ask the calculator for the next period:

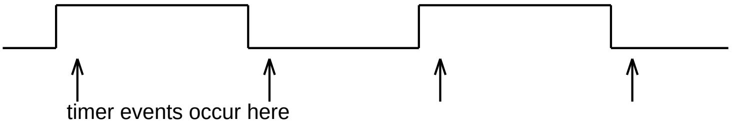


The interface `period_if` is defined as such:

```
interface period_if {
  int get_period();
};
```

11.2 The signal driving task

The signal driving task repeatedly wakes up based on a timer event. At each event it sets up the next port output and then waits for the next event. The timeline of this task looks like this:



The top level loop of this code is as follows:

```
while (1) {
   select {
     case tmr when timerafter(tmr_timeout) :> void:
        .... [set up port output] ...
        tmr_timeout += period;
        break;
   }
}
```

The `select` statement waits for an event and is saying “wait for the hardware timer named `tmr` to go past the value `tmr_timeout`”. When this event occurs the code sets up the port output and resets the timeout so it can wait until the next event. How the task gets the `period` value and how it sets up the port are covered later.

While the task is waiting for the timer event it is paused and not doing anything. This time can be used for something else (in this case running the other signal generating tasks).

The signal driver task takes arguments of the port to output to, and the interface connection that it can get the required period from.

```
[[combinable]]
void signal_driver(port p, client interface period_if updater)
{
```

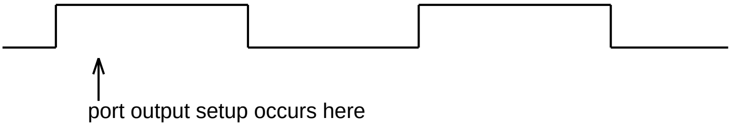
The task intializes its internal state and then goes into a while(1) select loop that uses a timer to select a periodic update. The timer events occur in the main loop via a select case:

```
select {
case tmr when timerafter(tmr_timeout) :> void:
```

The first thing to do in this case is obtain the current period over the interface connection to the buffer containing the period values.

```
period = updater.get_period();
```

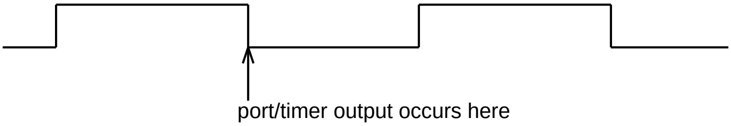
After this the port output needs to be setup:



To do this you keep track of when the next port counter time is and do a *timed output* to set up the future output:

```
next_port_time += period;
val = ~val;

p <: val @ next_port_time;
```

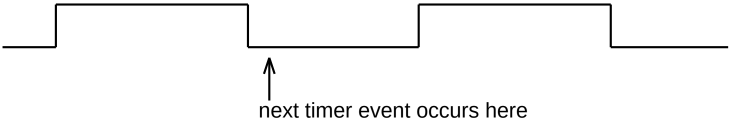
This will make `p` output at port time `next_port_time`:



Before returning to the main loop, update the timer output to happen after the next port output (note that the timer and port are running of the same underlying clock):

```
tmr_timeout += period;
```

So now the loop will continue and event at the next timer point:



Since the `port_driver` task consists of a main loop that has the form `while(1) { select ... }` it can be marked as *combinable*. Functions of this type can be combined together on the same logical core. The top level loops of the functions run together so at any point the core could handle an event from one of the combined tasks.

11.3 Decoupling the update from the driver

If the signal drivers were directly connected to the application calculating the period there would be a problem. The driver tasks rely on the period calculator since it calls the synchronous `get_period` interface function. At this point it pauses to get a period from that task every time it sets up an output. This will be a problem if the period calculator is not ready yet. You can decouple this timing dependence by placing a buffer task in between the driver tasks and the period calculator.

The buffer task maintains a list of the most recent periods. It can update the port driving tasks and also accept updates from the period calculator. Note that there is a new type of interface between the period calculator and the buffer:

```
interface supply_period_if  {
  // Set the period value in the buffer for signal driver ``n`` to ``period
    ↪ ``
  void set_period(int n, int period);

  // Get what period has been requested to update from the buffer.
  [[clears_notification]] int get_next_required_period_index();

  // This notification is signalled when one of the signal generators has
    ↪ used
  // the previous buffer value.
  [[notification]] slave void demand_next_period();
};
```

The intermediate buffer task acts as a server to both the period calculator and the signal generator. The implementation of the buffer is straightfoward.

```
[[distributable]]
void buffer(server interface supply_period_if c_supplier,
            server interface period_if c_driver[n],
            unsigned n)
{
  int period[MAX_SIGNALS];
  int next_index = 0;
  for (int i = 0; i < n; i++)
    period[i] = INITIAL_PERIOD;

  while (1) {
    select {
    case c_driver[int i].get_period() -> int x:
      x = period[i];
      c_supplier.demand_next_period();
      next_index = i;
      break;
    case c_supplier.get_next_required_period_index() -> int index:
      index = next_index;
      break;
    case c_supplier.set_period(int i, int val):
      period[i] = val;
      break;
    }
  }
}
```

This top-level loop is selecting on interface calls. A couple of pieces of syntax are worth explaining:

- `-> int x` - specifies that the variable `x` contains the return value of the interface call
- `c_driver[int i]` selects over all elements of the interface array `c_driver` (if one is selected the variable `i` will hold the index of the selected interface).

Note that this task has been marked as *distributable*. This is possible since it only selects on requests over interfaces from other task. If all the other tasks are on the same tile, then the buffer task will not take up a logical core but will use the cores of the clients it is connected to. The resources of the tasks are shared between the clients (so the buffer becomes a shared memory buffer between tasks).

11.4 The top level

The top level of the complete application now looks like:

```
// This function calculates periods and fills the intermediate buffer.
[[combinable]] void calc_periods(client interface supply_period_if c);

port ps[4] = {XS1_PORT_1A, XS1_PORT_1B, XS1_PORT_1C, XS1_PORT_1D};

int main()
{
  interface period_if c_update[4];
  interface supply_period_if c_supplier;
  par {
    on tile[0].core[0]: signal_driver(ps[0], c_update[0]);
    on tile[0].core[0]: signal_driver(ps[1], c_update[1]);
    on tile[0].core[0]: signal_driver(ps[2], c_update[2]);
    on tile[0].core[0]: signal_driver(ps[3], c_update[3]);
    on tile[0]:         buffer(c_supplier, c_update, 4);
    on tile[0]:         calc_periods(c_supplier);
  }
  return 0;
}
```

The main function consists of a par statement running the tasks in parallel. It also uses interface variable declarations to connect the tasks up. The four signal driving tasks run on the same logical core with the calc_periods task on a different (unnamed) logical core. The buffer task is distributable so does not takes up a logical core of its own, meaning that the whole application takes up two logical cores.

11.5 The application (calculating the period lengths)

The calc_periods task is the client to the signal drivers, setting the signal period lengths. The example here is a simple one that sets a fixed period for each signal. It reacts to the demand_next_period event and when this occurs finds out what period is requested and sets that value in the buffer:

```
[[combinable]]
void calc_periods(client interface supply_period_if c);
{
  while (1) {
    select {
    case c.demand_next_period():
      int i = c.get_next_required_period_index();
      // Calculate period for i
      c.set_period(i, (INITIAL_PERIOD * (i+1) * 2) / 3);
      break;
    }
  }
}
```

12 Using safe pointers for string processing

IN THIS CHAPTER

When using the multicore extensions to C, pointers are safe. This means that extra checks are inserted to avoid common memory access errors. It also means that pointers sometimes need to be annotated to indicate their use.

A couple of standard string processing functions are presented here to show the use of safe pointers in xC.

12.1 strlen

The following function implements a version of the standard `strlen` function that gets the length of a zero-terminated string:

```
int my_strlen_1(const char *str)
{
  int n = 0;
  while (*str != 0) {
    str++;
    n++;
  }
  return n;
}
```

Here there is no difference to standard C. However, the implementation implements bounds checking. This means that if a string is passed in that is not zero terminated, the program will trap instead of reading invalid memory and returing a nonsense value:

```
char str[3] = "abc";  // oh-oh, not zero-terminated
int len = my_strlen_1(str); // this will trap since the loop will
                            // read past the three bytes allocated to str
```

Since the trap occurs here at the point of error it is much easier to debug than a later follow-on subtle program error.

Although the above definition of `strlen` is workable, it could be better. In xC, arrays and pointers are not the same - although they can be implictly converted. In particular:

- Array parameters can only access the elements in front of them (you can subtract from pointers to access behind).
- Arrays cannot be null.

If your function satisfies the properties of being an array, then it is more efficient and safer to use an array argument. So the function becomes:

```
int my_strlen_2(const char str[])
{
  int n = 0;
  const char *p = str;
  while (*p != 0) {
    p++;
    n++;
  }
  return n;
}
```

In this case you get a more efficient implementation since the bounds checking does not need to worry about elements earlier in memory than the pointer str. You also get extra safety checks. The argument cannot be null so the following code will trap:

```
char *str = null;
int len = my_strlen_2(str);  // this will trap at the point of the call
```

This will trap at the point of the function call (not within the my_strlen_2 function). Having the trap as early as possible in execution greatly helps you debug where the cause of the error is.

12.2 strchr

The strchr function returns a pointer to the first occurence of a character within a string. The C prototype for the function is:

```
char * strchr(char * str, int c);
```

However, if you try and prototype a function like this in xC you will get the error:

```
error: pointer return type must be marked movable, alias or unsafe
```

To understand this error, you need to understand that safe pointers have three types in xC: *restricted*, *aliasing* and *movable*. By default, local pointers are aliasing - so you can have more than one pointer pointing to the same object. Function parameters default to *restricted* - so they cannot alias each other. Return values to functions cannot be restricted so they must be explicitly marked as aliasing or movable.

If a pointer return value is marked as aliasing, it can alias any of the aliasing pointer parameters to the function (or any global objects). In this case both the incoming pointer and the return value need to be marked as aliasing. Then the function can be written in the obvious way:

```
char * alias my_strchr(char * alias str, int c) {
  char *p = str;
  while (*p != 0 && *p != c) {
    p++;
  }
  if (*p == 0)
    return null;
  else
    return p;
}
```

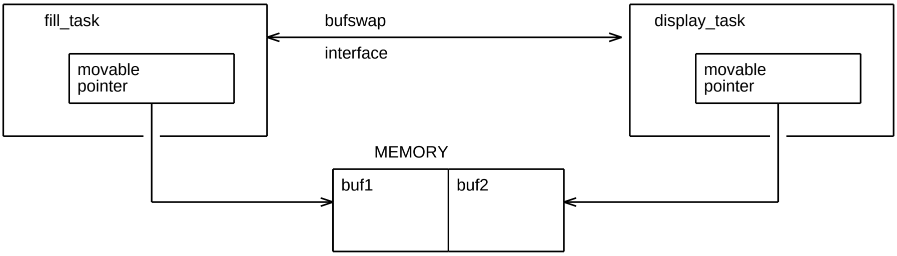
By keeping track of aliasing pointers, the compiler can check for program errors involving parallel race conditions and dangling pointers.

13 Double buffering example

This example shows how two tasks can implement a double buffering mechanism accessing a shared memory area.

One task fills a buffer whilst the other task displays a buffer. Both tasks access these buffers via *movable* pointers. These pointers can be safely transferred between tasks without any race conditions between the tasks using them.

When the tasks have finished filling and displaying the data, they connect via an interface connection and swap their pointers allowing them to work on different buffers.



The interface between the two tasks is a single transaction that swaps the movable pointers. It has an argument that is a reference to a movable pointer. Since it is a reference the server side of the connection can update the argument.

```
interface bufswap {
  void swap(int * movable &x);
};
```

The filling task takes as arguments the server end of an interface connection and the initial buffer it is going to fill. It is initialized by creating a movable pointer to this buffer and then filling it. Filling is done via a function call to a fill function which is application dependent and not defined here.

```
void fill_task(server interface bufswap display,
               int initial_buffer[])
{
  int * movable buffer = &initial_buffer[0];
  // fill the initial buffer
  fill(buffer);
```

The main loop of the filling task waits for a swap transaction with the other task and implements the swap of pointers. After that it fills the new buffer it has been given:

```
  while (1) {
    // swap buffers
    select {
    case display.swap(int * movable &display_buffer):
      // Swapping uses the 'move' operator. This operator transfers the
      // pointer to a new variable, setting the original variable to null.
      // The 'display_buffer' variable is a reference, so updating it will
      // update the pointer passed in by the other task.
      int * movable tmp;
      tmp = move(display_buffer);
      display_buffer = move(buffer);
      buffer = move(tmp);
      break;
    }
    // fill the buffer with data
    fill(buffer);
  }
}
```

The displaying task takes the other end of the interface connection and its initial buffer as arguments. It also creates a movable pointer to that buffer.

```
void display_task(client interface bufswap filler,
                  int initial_buffer[]) {
  int * movable buffer = &initial_buffer[0];
```

The main loop of the display task first calls the swap transaction, which synchronizes with the fill task and updates the buffer pointer to the new swapped memory location. After that it calls an auxiliary display function to do the actual displaying. This function is application dependent and not defined here.

```
  while (1) {
    // swap buffers
    filler.swap(buffer);
    // display the buffer
    display(buffer);
  }
}
```

The application runs both of these tasks in parallel using a par statement. The two buffers are declared at this level and passed into the two tasks:

```
int main() {
  int buffer0[200];
  int buffer1[200];
  interface bufswap c;
  par {
    fill_task(c, buffer0);
    display_task(c, buffer1);
  }
  return 0;
}
```