Application Note: AN10020

Generating several controllable pulse signals

This application note is a short how-to on programming/using the xTIMEcomposer tools. It shows generating several controllable pulse signals.

Required tools and libraries

This application note is based on the following components:

- xTIMEcomposer Tools - Version 14.0.0

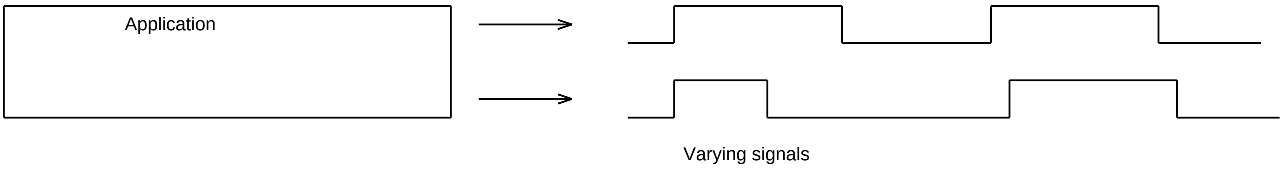
Required hardware

Programming how-tos are generally not specific to any particular hardware and can usually run on all XMOS devices. See the contents of the note for full details.

Copyright 2016 XMOS Ltd.

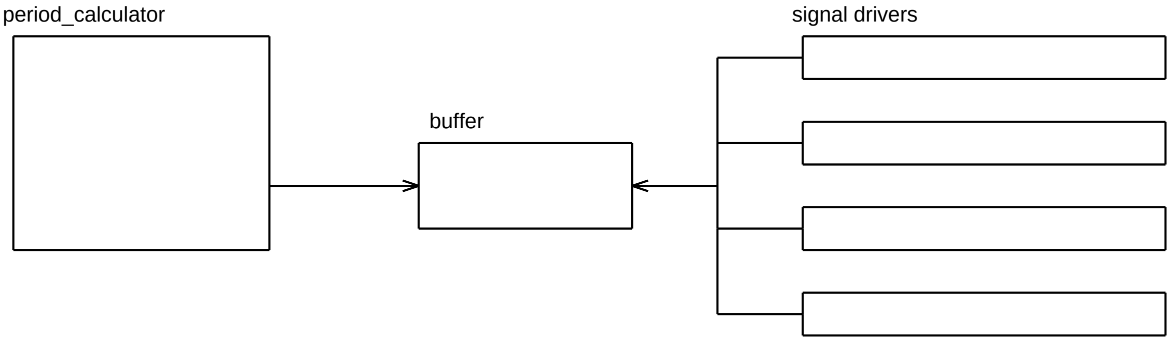
1 Generating several controllable pulse signals

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:



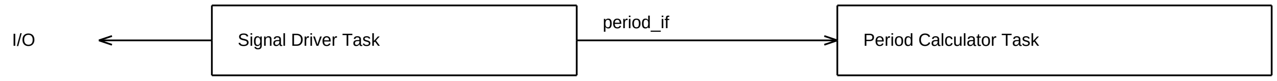
This type of application is similar to those used to control external hardware such as stepper motors.

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.



2 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:

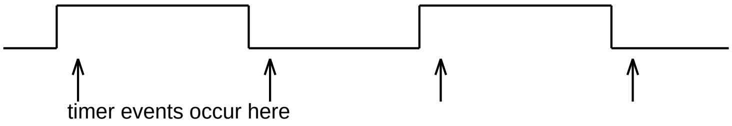


The interface `period_if` is defined as such:

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

3 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)
{
```

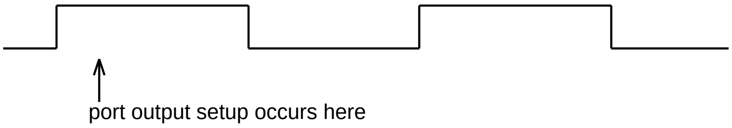
The task intializes its internal state and then goes into a while(1) select loop that uses a timer to select a periodic update. The timer events occur in the main loop via a select case:

```
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:

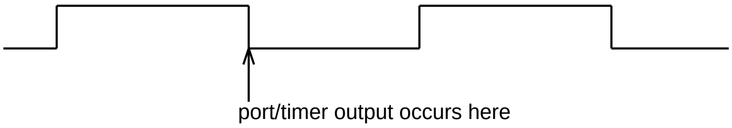


To do this you keep track of when the next port counter time is and do a *timed output* to set up the future output:

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

p <: val @ next_port_time;
```

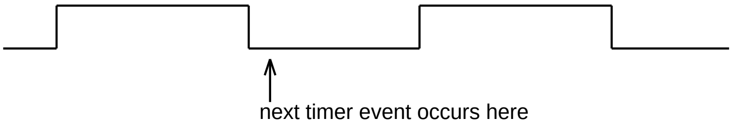
This will make p output at port time `next_port_time`:



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.

4 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).

5 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.

6 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;
    }
  }
}
```

Copyright © 2016, All Rights Reserved.

Xmos Ltd. is the owner or licensee of this design, code, or Information (collectively, the “Information”) and is providing it to you “AS IS” with no warranty of any kind, express or implied and shall have no liability in relation to its use. Xmos Ltd. makes no representation that the Information, or any particular implementation thereof, is or will be free from any claims of infringement and again, shall have no liability in relation to any such claims.