Application Note: AN10011

A double buffering example

This application note is a short how-to on programming/using the xTIMEcomposer tools. It shows a double buffering example.

Required tools and libraries

This application note is based on the following components:

- xTIMEcomposer Tools - Version 14.0.0

Required hardware

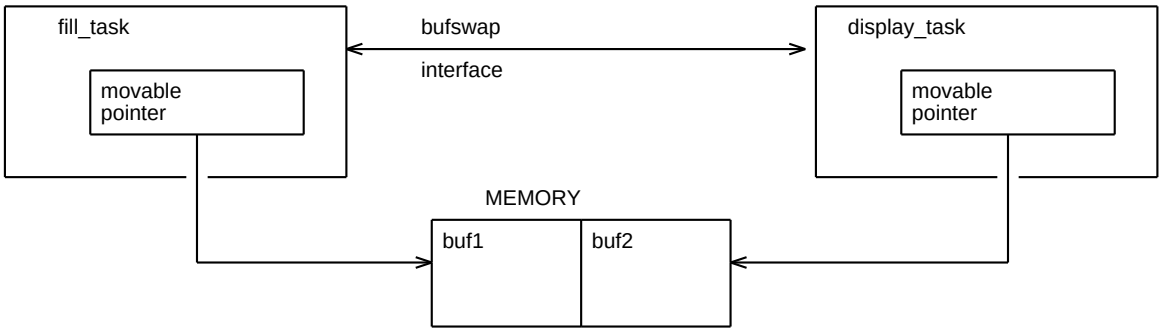
Programming how-tos are generally not specific to any particular hardware and can usually run on all XMOS devices. See the contents of the note for full details.

1 A 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;
}
```

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.