Application Note: AN10016

How to implement a fair select

This application note is a short how-to on programming/using the xTIMEcomposer tools. It shows how to implement a fair select.

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 How to implement a fair select

A select statement waits for one of a set of inputs to become ready. If more than one input is ready the select statement arbitrarily chooses one to service. Since the choice of which input to service is made arbitrarily, no guarantee of fairness is provided.

Consider the following code which handles incoming messages from multiple clients using a select statement in a loop:

```
unsigned received = 0;
while (1) {
  select {
  case (unsigned i = 0; i < NUM_CLIENTS; i++) c[i] :> int x:
    printstr("Received message from client ");
    printintln(i);
    received++;
    if (received == NUM_MESSAGES)
      return;
    break;
  }
}
```

If a client sends a message while the server is handling a message for another client, the output statement in that client task will block. If the other clients are repeatedly sending messages to the server then, since the select is not fair, the server may never get round to handling the message from this client.

To fix this problem we can use the combination of boolean guards on select cases and the default case to implement a fair server that gives equal priority to all its clients.

```
int disabled[NUM_CLIENTS] = {0};
int default_enabled = 0;
unsigned received = 0;
while (1) {
  select {
  case (unsigned i = 0; i < NUM_CLIENTS; i++) !disabled[i] => c[i] :> int x:
    printstr("Received message from client ");
    printintln(i);
    disabled[i] = 1;
    default_enabled = 1;
    received++;
    if (received == NUM_MESSAGES)
      return;
    break;
  default_enabled => default:
    for (unsigned i = 0; i < NUM_CLIENTS; i++) {
      disabled[i] = 0;
    }
    default_enabled = 0;
    break;
  }
}
```

After an event on a channel end has been received, events on that channel end are disabled. This ensures that the server doesn't accept two events from the same channel end without handling events on other channel ends that are ready to event first.

Each time an event is serviced, the number of channel ends that have events enabled is reduced by one. Eventually we will run out of channel ends that are both ready and have events enabled. At this point the default case in the select will be taken and events will be re-enabled for all of the channel ends.

The fair server satisfies the property that if an client sends a message to the server, the server will handle at most one message from every other client before handling the message from this client. This provides a minimum guarantee of responsiveness to each client regardless of what the other clients are doing.