Application Note: AN10027

How to communicate between tasks with interfaces

This application note is a short how-to on programming/using the xTIMEcomposer tools. It shows how to communicate between tasks with interfaces.

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 communicate between tasks with interfaces

Interfaces provide the most structured and flexible method of inter-task communication. 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);
};
```

The types of transactions 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. If an array argument is given for an intra-tile communication, the array is not copied but the compiler will optimize it to just pass a reference to the array, allowing the recipient to access the array in memory.

Communication is done over asymmetric connections obeying an interface protocol. One end of the connection is declared as the client end and initiates transactions, and one end is the server end which responds to transactions. Once both sides enter the transaction, data can flow in both directions.

To initiate transactions, a task can take an argument which is the client end of an interface connection and use that variable to call a function over the interface:

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

Code can wait for transactions at the server end with the `select` construct. A `select` waits until a transaction is initiated by the other side:

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

Note how the `select` lets you handle several different types of message. The language extension also lets the compiler allocate local variables for the incoming data that the message handler can access.

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.

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 c;
  par {
    task1(c);
    task2(c);
  }
  return 0;
}
```

The types of the functions tell the tools which end is the server and which is the client. The compiler also checks that each connection gets exactly one server and one client end.