Application Note: AN10007

How to define and use a combinable function

This application note is a short how-to on programming/using the xTIMEcomposer tools. It shows how to define and use a combinable function.

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 define and use a combinable function

Combinable functions represent tasks that can be combined to run on a single logical core.

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;
      ...
    }
  }
}
```

These kind of tasks can be marked as *combinable* by adding a special attribute:

```
[[combinable]]
void counter_task(char *taskId, int n) {
  int count = 0;
  timer tmr;
  unsigned time;
  tmr :> time;
  // This task perfoms 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++;
      if (count > n)
        return;
      time += 1000;
      break;
    }
  }
}
```

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)-select` statement.

Several combinable functions can be run in parallel with a *combined* `par`. This will run them on the same logical core using co-operative multitasking:

```
int main() {
  [[combine]]
  par {
    counter_task("task1", 5);
    counter_task("task2", 2);
  }
  return 0;
}
```

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.

You cannot use the [[combine]] attribute directly in a par with tile placements but can nest par statements:

```
int main(void) {
  par {
    on tile[0]: task1( ... );
    on tile[1]: task2( ... );
    on tile[1]:
      [[combine]]
      par {
        task3( ... );
        task4( ... );
      }
  }
  return 0;
}
```

The above program will run task1 on a logical core on tile[0] and task2 on its own logical core on tile[1]. A further logical core on tile[1] will run both task3 and task4 by using co-operative multitasking.