How to define and use a combinable function

`version` 1.0.0

`scope` Example. This code is provided as example code for a user to base their code on.

`description` How to define and use a combinable function

`boards` Unless otherwise specified, this example runs on the SliceKIT Core Board, but can easily be run on any XMOS device by using a different XN file.

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 wait 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.

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.