6

Local time

One of the principal applications of occam is to be programming what are commonly called *embedded* applications, that is programs to be executed on a computer that is only a part of some machine or system. Programs which interact in this way with parts of the world outside the computer on which they are executed often have a need to measure and record the passage of time. For this reason, facilities for dealing with real-time clocks are incorporated in the language.

This chapter is here to complete the presentation, but timers will hardly be used in the rest of the book so you may want to pass over this chapter on a first reading.

Timers

A *timer* is an object in a program rather like a channel; two timers are declared by

```
TIMER gmt, bst :
```

As with variables, arrays of timers can be declared, timers can be renamed by abbreviation,

```
[24]TIMER time.zone :
TIMER universal IS time.zone[12] :
```

and they can be passed as parameters to procedures, but not to functions.

Each timer is associated with a *clock*, which is rather like another process operating in parallel with the program. This process behaves as though it had a hidden INT variable; the value of this variable is called the *reading* of the clock. Each clock has a characteristic period, and increments its reading with the passage of each period. The expected implementation of a timer would be a hardware counter driven by a constant frequency clock signal.

The reading on a timer can be read by a process that looks like an input process

```
universal ? now
```

which immediately stores in an `INT` variable, *now*, the current reading of the specified timer. It is a trifle misleading for this to look like an input process: there are no restrictions on the use of timers in different branches of a parallel command as there are for channels; many concurrent processes may read the same timer without interference.

Because the reading on a clock passes through the whole range of `INT`s, with t being followed by t `PLUS` 1, the reading will be the same again after 2^n periods, where n is the number of bits in an `INT`. The right way of comparing two times is with `AFTER`. If two readings are taken from a clock less than `MOSTPOS INT` periods apart, then the second will be `AFTER` the first. As a guide to what this means, the clock visible to low priority processes on an IMST800 transputer has a period of $64\,\mu$s, and counts around in 2^{32} periods – about seventy-six and a half hours. This means that `AFTER` gives the expected answer if used to compare two readings taken less than a day and a half apart.

Later in the book, we assume that you can provide a definition

```
VAL INT second IS ... :
```

indicating by how much the reading on a timer changes in a second; this would only be possible for a clock which did not complete its cycle in less than two seconds. It is not unreasonable to assume that you can compare readings taken a minute apart, although the cycle time of a sixteen-bit 1μs clock would only be a sixteenth of a second or so.

Delay

The *delay* process

```
universal ? AFTER midnight
```

is another process, like `SKIP`, which does nothing in the sense that it eventually terminates, having neither communicated on any channel, nor changed any variable. It may, however, suspend execution, and does not terminate until the reading of *universal* has satisfied *reading* `AFTER` *midnight*. That is a carefully worded sentence: notice that there is no guarantee about the value of *variable* after the execution of

```
SEQ
  universal ? AFTER midnight
  universal ? variable
```

nor is there any guaranteed upper limit on how long it takes to execute a delay process. As before, a delay is not really an input process, but it can be used either as an ordinary process or as a guard in an alternative. A guard which is a delay is ready as soon as the delay could have terminated.

Notice that there is no way of setting the reading of a timer, nor is there any guaranteed relationship between the reading on two different clocks. That means that the normal way of setting a deadline for a delay is by taking a reading from the same timer, and calculating what the reading will be at the deadline.

Using timers

There are a very few idioms that encompass almost all uses of timers.

To suspend execution for (at least) ten seconds, it suffices to take a reading from any timer, and to wait until that timer has moved on by ten seconds.

```
TIMER t :
INT start :
SEQ
  t ? start
  t ? AFTER start PLUS (10 * second)
```

This might be done as a once-only action in a program while starting or stopping some mechanical peripheral device.

If an action is to be performed at regular intervals, say once every ten seconds, the program must wait for the passage of a succession of deadlines, each ten seconds later than the previous one. The process

```
TIMER t :
INT dead.line :
SEQ
  t ? dead.line
  WHILE ...
    SEQ
      dead.line := dead.line PLUS (10 * second)
      t ? AFTER dead.line
      ... perform the regular action
```

will do this, provided that the regular action can be completed in under ten seconds. Notice that each deadline is set relative to the previous deadline, so as to avoid slippage.

Finally, using delay guards allows a process to limit the time for which it is prepared to wait for input.

```
TIMER t :
INT prompted :
SEQ
  write.string(terminal.screen, "Yer wot, mate? ")
  t ? prompted
  ALT
    BYTE ch :
    terminal.keyboard ? ch
      to.program ! operator.typed.character; ch
    t ? AFTER prompted PLUS (30 * second)
      to.program ! operator.asleep.or.dead
```

Provided that the input from the *terminal.keyboard* arrives within thirty seconds of the clock being read, the alternative will select its first guard. After that time, the other guard is ready and the process is no longer obliged to wait for its input.