3 Local time

There are many applications of programmed devices where it is necessary for the program to be able to refer to, or to measure, the passage of time: for example, in long-distance communication, the participants are usually prepared to wait for replies for a limited time only, before taking action to recover from the loss of messages. To accommodate these needs, there are two primitive processes by which *occam* programs may refer to the changing state of a local clock. I mention them here to complete the presentation, but they will hardly be used in the programs which follow: you may want to pass by this section on a first reading.

The clock reading process

```
TIME ? variable
```

sets the value of the variable to the current reading on the clock. This is a word-sized bit pattern which changes at a uniform rate with the passage of time, the rate of change being dependent on the particular implementation. It counts up cyclically through a set of values distributed through the whole range of bit patterns, the most negative reading following after the most positive one. Notice that it is misleading for this process to look like an input process: the sequence of symbols `TIME ?` is semantically indivisible, `TIME` is not a channel, nor are clock reading processes governed by the rules that control the legal uses of channels: many concurrent processes may legally read the time from the same clock.

The clock delay process

```
TIME ? AFTER expression
```

is another process that does nothing, like `SKIP`, except that it may suspend execution. It does not terminate until the reading on the clock has satisfied the condition

```
reading AFTER expression
```

I have been careful in the wording of that last sentence: notice that there is no guarantee about the value of *variable* after the execution of

```
SEQ
  TIME ? AFTER expression
  TIME ? variable
```

As before, the sequence of symbols `TIME ? AFTER` is atomic, a delay is not an input process, but it is allowed to stand in the place of an input process as a guard of an alternative. Such a guard becomes ready as soon as the delay process may terminate.

The operator `AFTER` is intended for the comparison of readings taken from the clock. Provided that two times are separated by less than half the time that it takes for the clock to count around the complete cycle of its readings, one time is after the other if readings taken from the clock at those times are similarly related by `AFTER`. The cycle time of the clock depends on the word size, on the amount by which the reading is incremented at each tick, and the frequency of the clock ticks. Each of these depends on the particular implementation, and I will assume that you can supply a definition

```
DEF second = ...
```

in any program that needs it, indicating by how much the reading changes in one second. (This assumption will be unjustified if the clock cycle time is two seconds or less, as might be the case on a device with short words and a fast ticking clock.)

Any two readings being compared, either directly, or by the delay process, should be taken from the same clock: the language does not guarantee any relationship between readings taken in different branches of a `PAR` construct. Notionally, the clock is a register on a transputer, and no connection is to be expected between that register and the registers of any other transputers participating in the execution of a program. There is no mechanism in the language which maintains a global time, and it is the programmer's responsibility to implement one if it is needed. Similarly, if needed it is the programmer's task to provide a mechanism, using the clock, for timing long periods (those in excess of half a clock cycle time).

There are three idioms that, in combination, encompass almost all uses of the clock. First of all, to suspend execution for a fixed time, say ten seconds

```
VAR started :
SEQ
  TIME ? started
  TIME ? AFTER started + (10 * second)
```

This might happen 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, then

```
VAR next.dead.line :
SEQ
  TIME ? next.dead.line
  WHILE ...
    SEQ
      next.dead.line := next.dead.line + (10 * second)
      TIME ? AFTER next.dead.line
      ... perform action
```

will do this (provided that the 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.

```
VAR prompted, ch :
SEQ
  write.string(terminal.screen, "Yer wot? ")
  TIME ? prompted
  ALT
    terminal.keyboard ? ch
      to.program ! ch
    TIME ? AFTER prompted + (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.