Real-Time Systems and Programming Languages

WEB APPENDICES to 4th Edition

Occam 2 and Modula

Alan Burns and Andy Wellings

University of York

Introduction

In moving from the 3rd Edition to the 4th Edition we have removed the occam and Modula-1 languages from our discussions. However, both the occam and Modula models are very interesting and do illustrate some important concepts that we are trying to present in the book. Occam2 is also the nearest a general-purpose language has got to embodying the formalisms of CSP. Occam2 is also specifically designed for multi-computer execution, which is of increasing application and importance in the real-time domain. Historically Modula was the first programming language to enable the programming of device drivers in a high-level language. Hence, we provide discussion of these languages as appendices to the book available on the web.

APPENDIX A: Occam 2

A.1 An Overview of occam 2

Most imperative languages consists of a sequences of statements that can be combined into procedures or threads. The latter executing concurrently. What would be a sequence of statements in Ada, Java or C is a sequence of processes in occam2, all of which have the potential to be executed in parallel.

The form of a name can also be improved by the use of a separator. occam2 names can include a '.' (this is a somewhat unfortunate choice, as '.' is often used in languages to indicate a subcomponent, for example a field of a record). The following are example identifiers.

```
example.name.in.occam2
```

Occam2 is a fully block structured language. Any process can be preceded by the declaration of objects to be used in that process. To swap the two integers (`INT`s in occam2) requires a `SEQ` construct that specifies that the assignments that follow it must be executed in sequence:

```
INT temp: -- A declaration is terminated by a colon.
SEQ
  temp := A
  A := B
  B := temp
```

It should be noted that occam2 does not use a process separator (a semicolon in most languages). It requires each action (process) to be on a separate line. Moreover, the use of indentation, which merely (though importantly) improves readability in Ada, Java and C, is syntactically significant in occam2. The three assignments in the above code fragment have to start in the column under the `Q` of `SEQ`.

Data types

In common with all high-level languages, occam2 requires programs to manipulate objects that have been abstracted away from their actual hardware implementation. Programmers need not concern themselves about the representation or location of the entities that their programs manipulate. Moreover, by partitioning these entities into distinct types, the compiler can check for inconsistent usage, and thereby increase the security associated with using the languages.

By comparison with many high-level languages languages, occam2's type model is restrictive; in particular user-defined types are not allowed.
Table 1 compares the predefined discrete types supported in the Ada, Java, C and occam2 languages.

Occam2 is strongly typed (that is, assignments and expressions must involve objects of the same type), but explicit type conversions are supported. Enumeration types are not supported.

Many real-time applications (for example, signal processing, simulation and process control) require numerical computation facilities beyond those provided by integer arithmetic. There is a general need to be able to manipulate *real* numbers, although the sophistication of the arithmetic required varies widely between applications. In essence, there are two distinct ways of representing real values within a high-level language:

1. floating-point, and

2. scaled integer.

Floating-point numbers are a finite approximation to real numbers and are applicable to computations in which exact results are not needed. A floating-point number is represented by three values: a mantissa, M, an exponent, E, and a radix, R. It has a value of the form $M \times R^E$. The radix is (implicitly) implementation-defined and usually has the value 2. As the mantissa is limited in length, the representation has limited precision. The divergence between a floating-point number and its corresponding 'real' value is related to the size of the number (it is said to have **relative error**).

The use of scaled integers is intended for exact numeric computation. A scaled integer is a product of an integer and a scale. With the appropriate choice of scale, any value can be catered for. Scaled integers offer an alternative to floating-point numbers when non-integer calculations are required. The scale, however, must be known at compile time; if the scale of a value is not available until execution, a floating-point representation must be used. Although scaled integers provide exact values, not all numbers in the mathematical domain can be represented exactly. For example, 1/3 cannot be viewed as a finite scaled decimal integer. The difference between a scaled integer and its 'real' value is its **absolute error**.

Ada	**Java**	**C**	**Occam2**
`Integer`	`int`	`int`	`INT`
	`short`	`short`	`INT16`
	`long`	`long`	`INT32`
			`INT64`
	`byte`		`BYTE`
`Boolean`	`boolean`		`BOOL`
`Character`		`char`	
`Wide_Character`	`char`	`wchar_t`	

Table 1: Discrete types.

Scaled integers have the advantage (over floating-point) of dealing with exact numerical values and of making use of integer arithmetic. Floating-point operations require either special hardware (a floating-point unit) or complex software that will result in numerical operations being many times slower than the integer equivalent. Scaled integers are, however, more difficult to use, especially if expressions need to be evaluated that contain values with different scales.

Traditionally, languages have supported a single floating-point type (usually known as **real**) which has an implementation-dependent precision. Use of scaled integers has normally been left to the user (that is, the programmer had to implement scaled integer arithmetic using the system-defined integer type).

The designers of occam2 took the view that the need for an abstract `real` type is not as great as the need for the programmer to be aware of the precision of the operations being carried out. The occam2 'reals' are `REAL16`, `REAL32` and `REAL64`.

Structured data types

Occam2 only supports arrays, it does not support records.; Example include:

```
INT Max IS 10:   -- definition of a constant in occam2
[Max]REAL32 Reading:    -- Reading is an array with ten
                        -- elements Reading[0] .. Reading[9]
[Max][Max]BOOL Switches:    -- 2 dimensional array
```

All arrays in occam2 start at element zero. Note that occam2 uses the conventional square brackets.

Dynamic data types and pointers

The implementation of dynamic data types can represents a considerable overhead to the run-time support system for a language. For this reason, occam2 does not have any dynamic structures.

Control structures

There is now common agreement on the control abstractions needed in a sequential programming language. These abstractions can be grouped together into three categories: sequences, decisions and loops. Each will be considered in turn.

The sequential execution of statements is the normal activity of a (non-concurrent) programming language. In occam2, the normal execution of statements (called processes in occam2) could quite reasonably be concurrent, and it is therefore necessary to state explicitly that a collection of actions must follow a defined sequence. This is achieved by using the `SEQ` construct that was illustrated earlier. For example:

```
SEQ
  action 1
  action 2
  .
  .
  .
```

If a sequence is, in a particular circumstance, empty, occam2 uses a SKIP process to imply no action:

```
SEQ
  SKIP
```

or just:

```
SKIP
```

At the other extreme from a null action is one that causes the sequence to make no further progress. The occam2 STOP process has the effect of terminating the entire program.

Decision structures

A decision structure provides a choice as to the route that execution takes from some point in a program sequence to a later point in that sequence. The decision as to which route is taken will depend upon the current values of relevant data objects. The important property of a decision control structure is that all routes eventually come back together. With such abstract control structures there is no need to use a goto statement, which often leads to programs that are difficult to test, read and maintain. Occam2 does not possess a goto statement.

The most common form of decision structure is the if statement. Occam2 has clear unambiguous structures. To illustrate, consider a simple problem; find out if $B/A > 10$ – checking first to make sure that A is not equal to zero!

Consider first the general structure. In the following schema, let B1 .. Bn be boolean expressions and A1 .. An be actions :

```
IF
  B1
    A1
  B2
    A2
  .
  .
  Bn
    An
```

Firstly, it is important to remember that the layout of occam2 programs is syntactically significant. The boolean expressions are on separate lines (indented two spaces from the IF), as are the actions (indented a further two spaces). Like C, no 'then' token is used. On execution of this IF construct, the boolean expression B1 is evaluated. If it evaluates to TRUE then A1 is executed and that completes the action of the IF. However, if B1 is FALSE then B2 is evaluated. All the boolean expressions are evaluated until one is found TRUE and then the associated action is undertaken. If no boolean expression is TRUE then this is an error condition and the IF construct behaves like the STOP discussed in the last section.

With this form of IF statement, no distinct ELSE part is required; the boolean expression TRUE as the final test is bound to be taken if all other choices have failed. The b/a > 10 example in occam2, therefore, takes the form:

```
IF
  a /= 0
    IF
      b/a > 10
        high := TRUE
      TRUE
        high := FALSE
  TRUE       -- These last two lines are needed so
    SKIP     -- that the IF does not become STOP when
             -- a has the value 0
```

To give another illustration of the important ‘if’ construct, consider a multiway branch. In occam2, the code is quite clear:

```
IF
  number < 10
    digits := 1
  number < 100
    digits := 2
  number < 1000
    digits := 3
  number < 10000
    digits := 4
  TRUE
    digits := 5
```

The above is an example of a multiway branch constructed from a series of binary choices. In general, a multiway decision can be more explicitly stated and efficiently implemented using a `case` (or `switch`) structure. The occam2 version of this is somewhat more restricted. To illustrate, consider a character (byte) value ‘command’ which is used to decide upon four possible actions:

In occam2, alternative values and ranges are not supported; the code is, therefore, protracted:

```
CASE command
  'A'
    action1
  'a'
    action1
  't'
    action2
  'e'
    action3
  'x'
    action4
  'y'
    action4
  'z'
    action4
  ELSE
    SKIP
```

Loop structures

A loop structure allows the programmer to specify that a statement, or collection of statements, is to be executed more than once. There are two distinct forms for con-

structing such loops:

1. iteration, and

2. recursion.

The distinctive characteristic of iteration is that each execution of the loop is completed before the next is begun. With a recursive control structure, the first loop is interrupted to begin a second loop, which may be interrupted to begin a third loop and so on. At some point, loop n will be allowed to complete, this will then allow loop $n-1$ to complete, then loop $n-2$ and so on, until the first loop has also terminated. Recursion is usually implemented via recursive procedure calls. Attention here is focused on iteration.

Iteration comes in two forms: a loop in which the number of iterations is usually fixed prior to the execution of the loop construct; and a loop in which a test for completion is made during each iteration. The former is known generally as the `for` statement, the latter as the `while` statement. Most languages' `for` constructs also provide a counter that can be used to indicate which iteration is currently being executed.

The following example code illustrates the `for` construct in occam2; the code assigns into the first ten elements of array `A` the value of their position in the array:

```
SEQ i = 0 FOR 10          -- i is defined by the construct
  A[i]:= i                -- i is read only in the loop
                          -- i is out of scope after the loop
-- note that the range of i is 0 to 9,
-- as in the Ada and C example
```

Note that occam2 has restricted the use of the loop variable. The free use of this variable can be the cause of many errors.

The main variation with `while` statements concerns the point at which the test for exit from the loop is made. The most common form involves a test upon entry to the loop and subsequently before each iteration is made. This is support by occam2.

```
WHILE <boolean expression>
  SEQ
    <statements>
```

Subprograms

Even in the construction of a component or module, further decomposition is usually desirable. This is achieved by the use of procedures and functions; known collectively as *subprograms*.

Subprograms not only aid decomposition but also represent an important form of abstraction. They allow arbitrary complex computations to be defined and then invoked by the use of a simple identifier. This enables such components to be reused both within a program and between programs. The generality, and therefore usefulness, of subprograms is, of course, increased by the use of parameters.

Parameter-passing modes and mechanisms

A parameter is a form of communication; it is a data object being transferred between the subprogram user and the subprogram itself. There are a number of ways of describing the mechanisms used for this transfer of data. Firstly, one can consider the way parameters are transferred. From the invoker's point of view, there are three distinct modes of transfer.

1. Data is passed into the subprogram.
2. Data is passed out from the subprogram.
3. Data is passed into the subprogram, is changed and is then passed out of the subprogram.

These three modes are often called: **in**, **out** and **in out**.

The second mechanism of describing the transfer is to consider the binding of the formal parameter of the subprogram and the actual parameter of the call. There are two general methods of interest here: a parameter may be bound by value or by reference. A parameter that is bound by value only has the value of the parameter communicated to the subprogram (often by copying into the subprogram's memory space); no information can return to the caller via such a parameter. When a parameter is bound by reference, any updates to that parameter from within the subprogram is defined to have an effect on the memory location of the actual parameter.

A final way of considering the parameter-passing mechanism is to examine the methods used by the implementation. The compiler must satisfy the semantics of the language, be these expressed in terms of modes or binding, but is otherwise free to implement a subprogram call as efficiently as possible. For example, a large array parameter that is 'pass by value' need not be copied if no assignments are made to elements of the array in the subprogram. A single pointer to the actual array will be more efficient but behaviourally equivalent. Similarly, a call by reference parameter may be implemented by a copy in and copy out algorithm.

In occam2, parameters by default are passed by reference; a `VAL` tag is used to imply pass by value. Significantly, within the procedure (called a `PROC` in occam2) a VAL parameter acts as a constant and, therefore, the errors that can occur in other languages by missing out the constant tag are caught by the compiler.

```
PROC quadratic(VAL REAL32 A, B, C,
               REAL32 R1, R2, BOOL OK)
  -- note, as with C and Java, the parameter separator
  -- is not a semicolon
```

Procedures

Procedure bodies are illustrated by completing the 'quadratic' definitions given above. All procedures assume that a `sqrt` function is in scope.

```
PROC quadratic(VAL REAL32 A, B, C,
               REAL32 R1, R2, BOOL OK)
  REAL32 Z:
```

```
  SEQ
    Z:= (B*B) - (4.0*(A*C))   -- brackets are needed
                              -- to fully specify expression
    IF
      (Z < 0) OR (A = 0.0)
        SEQ
          OK:= FALSE
          R1:= 0.0            -- arbitrary values
          R2:= 0.0
      TRUE                    -- no return statement in occam2
        SEQ
          OK:= TRUE
          R1:= (-B + SQRT(Z)) / (2.0*A)
          R2:= (-B - SQRT(Z)) / (2.0*A)
:   -- colon needed to show end of PROC declaration
```

The invoking of these procedures merely involves naming the procedure and giving the appropriately typed parameters in parenthesis.

Recursive (and mutually recursive) procedure calls are not supported in occam2 because of the dynamic overhead they create at run-time. Procedures (and functions) can also not be nested.

Functions

Occam2 defines the semantics of a function so that no side-effects are possible.

The parameters to an occam2 function are passed by value. In addition, the body of a function is defined to be a VALOF. A VALOF is the sequence of statements necessary to compute the value of an object (which will be returned from the function). The important property of the VALOF is that the only other variables whose values can be changed are those defined locally within the VALOF. This prohibits side effects. The simple minimum function defined earlier would therefore have the form:

```
INT FUNCTION minimum (VAL INT X, Y)
  INT Z:    -- Z will be the value returned
  VALOF
    IF
      X > Y
        Z:= Y
      TRUE
        Z:= X
    RESULT Z
:
```

In concurrent languages, another form of side effect is hidden concurrency; this can have a number of unfortunate consequences. The VALOF of occam2 is further restricted to disallow any concurrency within it.

Missing features

Occam2 does not provided any language support for modular decomposition such as modules, packages or classes that are found in modern languages. It also doesn't provide any exception handling facility.

A.2 Concurrent execution in occam2

Occam2 uses a cobegin structure called PAR. For example, consider the concurrent execution of two simple assignments (A:=1 and B:=1):

```
PAR
  A := 1
  B := 1
```

This PAR structure, which can be nested, can be compared to the sequential form:

```
SEQ
  A := 1
  B := 1
```

Note that a collection of actions must be explicitly defined to be either executing in sequence or parallel; there is no default. Indeed, it can be argued that PAR is the more general form and should be the natural structure to use unless the actual code in question requires a SEQuence.

If the process designated by a PAR is an instance of a parameterized PROC (procedure) then data can be passed to the process upon creation. For example, the following code creates three processes from a single PROC and passes an element of an array to each:

```
PAR
  ExampleProcess(A[1])
  ExampleProcess(A[2])
  ExampleProcess(A[3])
```

A greater collection of processes can be created from the same PROC by the use of a 'replicator' to increase the power of the PAR:

```
PAR i=1 FOR N
  ExampleProcess(A[i])
```

Earlier, a replicator was used with a SEQ to introduce a standard 'for' loop. The only distinction between a replicated SEQ and a replicated PAR is that the number of replications (N in the above example) must be a constant in the PAR; that is, known at compile time. It follows that occam2 has a static process structure.

The following program fragment in occam2 is the robot arm example. Note that occam2 does not support enumeration types and so the dimensions are represented by the integers 1, 2 and 3.

```
PROC control(VAL INT dim)
  INT  position,              -- absolute position
       setting:               -- relative movement
  SEQ
    position := 0             -- rest position
    WHILE TRUE
      SEQ
        MoveArm(dim,position)
        NewSetting(dim,setting)
        position := position + setting
:
```

```
PAR
  control(1)
  control(2)
  control(3)
```

Process termination is quite straightforward in occam2. There is no abort facility nor are there any exceptions. A process must either terminate normally or not terminate at all (as in the above example).

In general, occam2 has a fine grain view of concurrency. Most concurrent programming languages have the notion of process added to an essentially sequential framework. This is not the case with occam2; the concept of process is basic to the language. All activities, including the assignment operations and procedure calls, are considered to be processes. Indeed, the notion of statement is missing from occam2. A program is a single process that is built from a hierarchy of other processes. At the lowest level, all primitive actions are considered to be processes and the constructors (`IF`, `WHILE`, `CASE` and so on) are themselves constructor processes.

A.3 Inter-process communication

Occam2 only allows communication and synchronization to be based on message passing. It incorporates indirect symmetric synchronous message passing.

The occam2 model

Occam2 processes are not named, and therefore it is necessary during communication to use indirect naming via a **channel**. Each channel can only be used by a single writer and a single reader process. Both processes name the channel; the syntax is somewhat terse:

```
ch ! X   -- write value of expression X
         -- onto channel ch

ch ? Y   -- read from channel ch
         -- into variable Y
```

In the above, the variable `Y` and the expression `X` will be of the same type. The communication is synchronous; therefore whichever process accesses the channel first will be suspended. When the other process arrives, data will pass from `X` to `Y` (this can be viewed as the distributed assignment `Y := X`). The two processes will then continue their executions concurrently and independently. To illustrate this communication, consider two processes that are passing 1000 integers between them:

```
CHAN OF INT ch:
PAR
  INT V:
  SEQ i = 0 FOR 1000          --- process 1
    SEQ
      -- generate value V
      ch ! V
```

```
INT C:
SEQ i = 0 FOR 1000         --- process 2
  SEQ
    ch ? C
    -- use C
```

With each iteration of the two loops, a rendezvous between the two processes occurs.

Channels in occam2 are typed and can be defined to pass objects of any valid type including structured types. Arrays of channels can also be defined.

It is important to appreciate that the input and output operations on channels are considered to be fundamental language primitives. They constitute two of the five primitive processes in occam2. The others being SKIP, STOP and assignment.

The occam2 ALT

Consider a process that reads integers down three channels (`ch1`, `ch2` and `ch3`) and then outputs whatever integers it receives down a further channel (`chout`). If the integers arrived in sequence down the three channels then a simple loop construct would suffice.

```
WHILE TRUE
  SEQ
    ch1   ?  I        -- for some local integer I
    chout !  I
    ch2   ?  I
    chout !  I
    ch3   ?  I
    chout !  I
```

However, if the order of arrival is unknown then each time the process loops a choice must be made between the three alternatives:

```
WHILE TRUE
  ALT
    ch1   ?  I
      chout    !  I
    ch2   ?  I
      chout    !  I
    ch3   ?  I
      chout    !  I
```

If there is an integer on `ch1`, `ch2` or `ch3`, it will be read and the specified action taken, which in this case is always to output the newly acquired integer down the output channel `chout`. In a situation where more than one of the input channels is ready for communication, an arbitrary choice is made as to which one is read. Before considering the behaviour of the `ALT` statement when none of the channels are ready, the general structure of an `ALT` statement will be outlined. It consists of a collection of guarded processes:

```
ALT
  G1
    P1
  G2
```

```
    P2
  G3
    P3
    :
  Gn
    Pn
```

The processes themselves are not restricted – they are any occam2 process. The guards (which are also processes) can have one of three forms.

```
<boolean_expression> & channel_input_operation

channel_input_operation

<boolean_expression> & SKIP
```

The most general form is therefore a boolean expression and a channel read, for example:

```
NOT BufferFull & ch ? BUFFER[TOP]
```

If the boolean expression is simply TRUE then it can be omitted altogether (as in the earlier example). The SKIP form of guard is used to specify some alternative action to be taken when other alternatives are precluded; for example:

```
ALT
  NOT BufferFull & ch ?  BUFFER[TOP]
    SEQ
      TOP := ...
  BufferFull & SKIP
    SEQ
      -- swap buffers
```

On execution of the ALT statement, the boolean expressions are evaluated. If none evaluate to TRUE (and there are no default TRUE alternatives) then the ALT process cannot proceed and it becomes equivalent to the STOP (error) process. Assuming a correct execution of the ALT, the channels are examined to see if there are processes waiting to write to them. One of the following possibilities could then ensue:

1. There is only one ready alternative, that is one boolean expression evaluates to true (with a process waiting to write or a SKIP guard) – this alternative is chosen, the rendezvous takes place (if it is not a SKIP) and the associated subprocess is executed.

2. There is more than one ready alternative – one is chosen arbitrarily, this could be the SKIP alternative if present and ready.

3. There are no ready alternatives – the ALT is suspended until some other process writes to one of the open channels of the ALT.

The ALT will, therefore, become a STOP process if all the boolean expressions evaluate to FALSE, but will merely be suspended if there are no outstanding calls. Because of the non-shared variable model of occam2, it is not possible for any other process to change the value of any component of the boolean expression.

The ALT when combined with the SEQ, IF, WHILE, CASE and PAR furnishes the complete set of occam2 program constructs. A replicator can be attached to an ALT in the same way that it has been used with other constructs. For example, consider a concentrator process that can read from 20 processes (rather than three as before); however, rather than use 20 distinct channels, the server process uses an array of channels as follows:

```
WHILE TRUE
  ALT j = 0 FOR 20
    ch[j]  ?  I
      chout  !  I
```

Finally, it should be noted that occam2 provides a variant of the ALT construct which is not arbitrary in its selection of ready alternatives. If the programmer wishes to give preference to a particular channel then it should be placed as the first component of a PRI ALT. The semantics of PRI ALT dictate that the textually first ready alternative is chosen. The following are examples of PRI ALT statements.

```
PRI ALT
  VeryImportantChannel ? message
    -- action
  ImportantChannel ? message
    -- action
  LessImportantChannel ? message
    -- action

WHILE TRUE
  PRI ALT j = 0 FOR 20  -- ch[0] is given highest preference
    ch[j]  ?  I
      chout  !  I
```

The bounded buffer

Occam2 provides no shared variable communication primitives and, therefore, resource controllers such as buffers have to be implemented as server processes. To implement a single reader and single writer buffer, requires the use of two channels that link the buffer process to the client processes (to cater for more readers and writers would require arrays of channels):

```
CHAN OF Data Take, Append:
```

Unfortunately, the natural form for this buffer would be:

```
VAL INT Size IS 32:
INT Top, Base, NumberInBuffer:
[Size]Data Buffer:
SEQ
  NumberInBuffer := 0
  Top := 0
  Base := 0
  WHILE TRUE
    ALT
      NumberInBuffer < Size & Append ? Buffer[Top]
```

```
        SEQ
          NumberInBuffer := NumberInBuffer + 1
          Top := (Top + 1) REM Size
      NumberInBuffer > 0 & Take ! Buffer[Base]  -- not legal occam
        SEQ
          NumberInBuffer := NumberInBuffer - 1
          Base := (Base + 1) REM Size
```

Output operations in this context are not allowed in occam2. Only input operations can form part of an ALT guard. The reason for this restriction is implementational efficiency on a distributed system. The essence of the problem is that the provision of symmetric guards could lead to a channel being accessed by an ALT at both ends. The arbitrary decision of one ALT would therefore be dependent on the decision of the other (and vice versa). If the ALTs are on different processors then the agreement on a collective decision would involve the passing of a number of low-level protocol messages.

To circumvent the restriction on guards, occam2 forces the Take operation to be programmed as a double interaction. First the client process must indicate that it wishes to TAKE, and then it must Take; a third channel is thus needed:

```
CHAN OF Data Take, Append, Request:
```

The client must make the following calls

```
SEQ
  Request ! ANY      -- ANY is an arbitrary token
  Take ? D           -- D is of type DATA
```

The buffer process itself has the form:

```
VAL INT size IS 32:
INT Top, Base, NumberInBuffer:
[Size]Data Buffer:
SEQ
  NumberInBuffer := 0
  Top := 0
  Base := 0
  Data ANY:
  WHILE TRUE
    ALT
      NumberInBuffer < Size & Append ? Buffer[Top]
        SEQ
          NumberInBuffer := NumberInBuffer + 1
          Top := (Top + 1) REM Size
      NumberInBuffer > 0 & Request ? ANY
        SEQ
          Take ! Buffer[Base]
          NumberInBuffer := NumberInBuffer - 1
          Base := (Base + 1) REM Size
```

The correct functioning of the buffer is thus dependent on correct usage by the client processes. This dependence is a reflection of poor modularity. Although the Ada select is also asymmetric (that is, you cannot select between accepts and entry calls), the fact that data can pass in the opposite direction to the call removes the difficulty that manifests itself in occam2.

A.4 Real-Time facilities

TIMERs in occam2

Any occam2 process can obtain a value of the 'local' clock by reading from a TIMER (there is no facility for accessing calendar time). To be consistent with the occam2 model of communication (which is one-to-one), each process must use a distinct TIMER. Reading from a TIMER follows the syntax of channel read, but the semantics are different in that a TIMER read cannot lead to suspension, that is, the clock is always ready to output.

```
TIMER clock:
INT Time:
SEQ
  clock ? Time    -- read time
```

The value produced by a TIMER is of type INT, but of implementation-dependent meaning: it gives a relative, not absolute, clock value. A single reading of a TIMER is, therefore, meaningless, but the subtraction of two readings will give a value for the passage of time between the two readings:

```
TIMER clock:
INT old, new, interval:
SEQ
  clock ? old
  -- other computations
  clock ? new
  interval := new MINUS old
```

The operator MINUS is used rather than '-' to take account of wrap-around. This occurs because the integer given by a TIMER is incremented by one for each unit of time; eventually the maximum integer is reached, so for the subsequent 'tick' the integer becomes the most negative one and then continues to be incremented. Users can be unaware of this action as long as they use the appropriate (language-defined) arithmetic operators, which are MINUS, PLUS, MULT and DIVIDE. In effect, each TIMER undertakes a 'PLUS 1' operation for each increment of the clock.

As the above illustrates, the facilities provided by occam2 are primitive (though arguably quite adequate). As only one integer is allocated for the clock there is clearly a trade-off between the granularity of the clock (that is, what interval of time each tick of the clock represents) and the range of times that can be accommodated. With a 32 bit integer, Table 2 gives typical values.

Absolute delays

Occam2 only supports absolute delays. To emphasize its open-ended semantics, the keyword AFTER is used. If a process wishes to wait for 10 seconds, it must first read the TIMER clock, add 10 seconds and then delay until this time. The value 10 seconds is obtained via the constant G which is introduced (here) to give a measure of the granularity of the implementation (G is the number of TIMER updates per second):

Granularity	Range (approximately)
1 microsecond	71.6 minutes
100 microsecond	119 hours
1 millisecond	50 days
1 second	136 years

Table 2: TIMER granularities for a 32-bit integer.

```
SEQ
  clock ? now
  clock ? AFTER now PLUS (10 * G)
```

In occam2, the code for avoiding cumulative drift is:

```
INT next, now:
VAL interval IS 7*G:
SEQ
  clock ? now
  next := now PLUS interval
  WHILE TRUE
    SEQ
      ACTION
      clock ? AFTER next
      next := next PLUS interval
```

Programming timeouts

A timeout facility is common in message-based concurrent programming languages. Occam2 uses the ‘delay’ primitive as part of a selective wait construct to indicate a timeout on a message receive:

```
WHILE TRUE
  SEQ
    ALT
      call ? new_temp
        -- other actions
      clock ? AFTER (10 * G)
        -- action for timeout
```

where clock is a TIMER.

Priority

Occam has only a rudimentary priority model. It has a variation of the PAR construct that indicates that static priorities should be assigned to the designated processes:

```
PRI PAR
  P1
  P2
  PAR
   P3
```

```
    P4
  P5
```

Here, relative priorities are used, with the textual order of the processes in the PRI PAR being significant. Hence P1 has the highest priority, P2 the second highest; P3 and P4 share the next priority level and P5 has the lowest priority. No minimum range of priorities need be supported by an implementation. There is no support for priority inheritance.

A.5 Distributed Systems

Occam2 has been specifically designed so that programs can be executed in a distributed environment, that of a multi-transputer network. In general, occam2's processes do not share variables so the unit of partitioning is the process itself. Configuration is achieved by the PLACED PAR construct. A program constructed as a top-level PAR, such as:

```
PAR
  p1
  p2
  p3
  p4
  p5
```

can be distributed, for example as follows:

```
PLACED PAR
  PROCESSOR 1
    p1
  PROCESSOR 2
    PAR
      p2
      p3
  PROCESSOR 3
    PAR
      p4
      p5
```

It is important to note that the transformation of the program from one that has a simple PAR to one that uses a PLACED PAR will not invalidate the program. However, occam2 does allow variables to be read by more than one process on the same processor. Therefore, a transformation may not be possible if the programmer has used this facility.

For the transputers, it is also necessary to associate each external channel with an appropriate transputer link. This is achieved by using the PLACE AT construct. For example, consider the above example with the following integer channels shown in Figure 1.

The program for execution on a single transputer is:

```
CHAN OF INT c1, c2, c3, c4, c5:
PAR
  p1
  p2
```

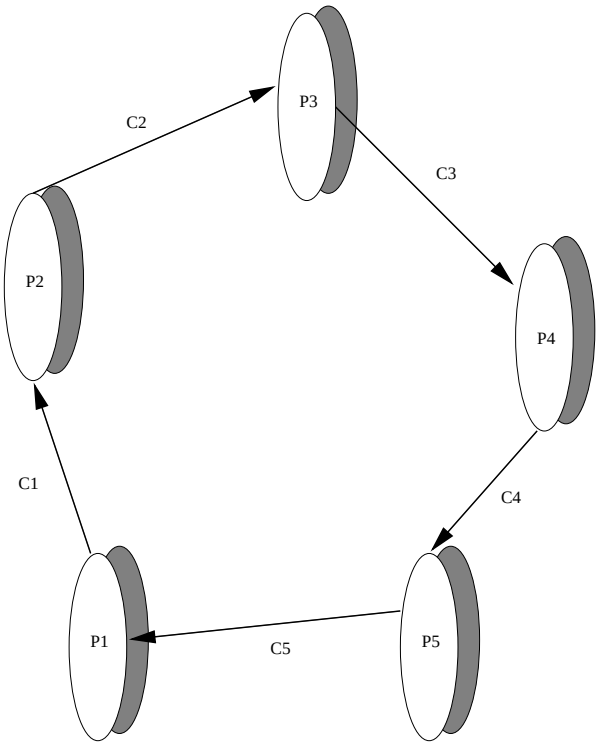

Figure 1: Five occam2 processes connected by five channels.

```
  p3
  p4
  p5
```

If the program is configured to three transputers, as illustrated in Figure 2, the occam2 program becomes:

```
CHAN OF INT c1, c3, c5:
PLACED PAR
  PROCESSOR 1
    PLACE c1 at 0:
    PLACE c5 at 1:
    p1
  PROCESSOR 2
    PLACE c1 at 2:
    PLACE c3 at 1:
    CHAN OF INT c2:
    PAR
      p2
      p3
  PROCESSOR 3
    PLACE c3 at 0:
    PLACE c5 at 2:
```

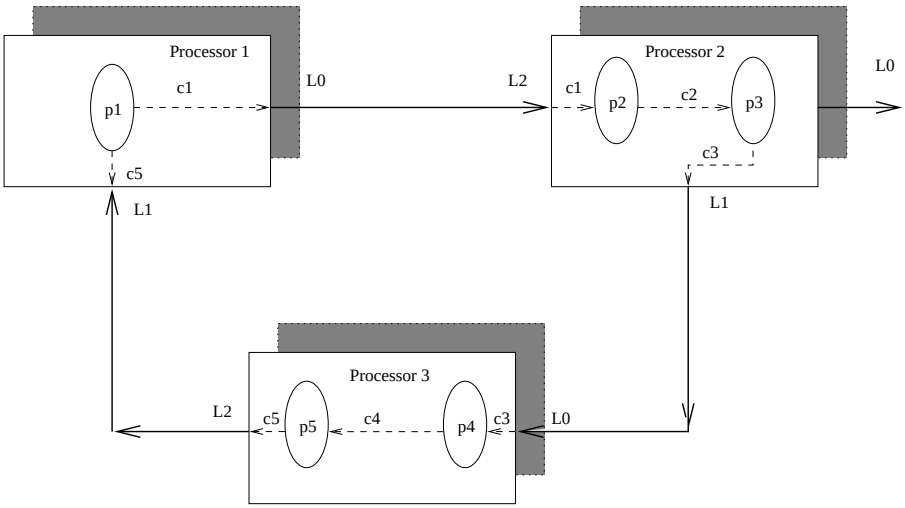

Figure 2: Five occam2 processes configured for three transputers.

```
CHAN OF INT c4:
PAR
  p4
  p5
```

The ease with which occam2 programs can be configured for execution on a distributed system is one of the main attractions of occam2.

Allocation is not defined by the occam2 language nor are there any facilities for reconfiguration. Further, access to resources is not transparent.

A.6 Low-level programming

Occam2 is a language that supports a message-based model to control devices.

Although occam2 was designed for the transputer, in the following discussion it is considered as a machine-independent language. The model is presented first, and then consideration is given to its implementation on memory-mapped and special-instruction machines. The three issues of device encapsulation, register manipulation and interrupt handling must be considered.

Modularity and encapsulation facility

The only encapsulation facility provided by occam2 is the procedure, and it is this that must, therefore, be used to encapsulate device drivers.

Addressing and manipulating device registers

Device registers are mapped onto PORTs, which are conceptually similar to occam2 channels. For instance, if a 16-bit register is at address X then a PORT P is defined by:

```
PORT OF INT16 P:
PLACE P AT X:
```

Note that this address can be interpreted as either a memory address or a device address depending on the implementation. Interaction with the device register is obtained by reading or writing to this port:

```
P ! A  -- write value of A to the port

P ? B  -- read value of port into B
```

A port cannot be defined as read or write only.

The distinction between ports and channels in occam2, which is a significant one, is that there is no synchronization associated with the port interaction. Neither reads nor writes can lead to the executing process being suspended; a value is always written to the address specified and, similarly, a value is always read. A port is thus a channel in which the partner is always ready to communicate.

Occam2 provides facilities for manipulating device registers using shift operations and bitwise logical expressions. There is, however, no equivalent to Modula-1's bit type or Ada representation specifications.

Interrupt handling

An interrupt is handled in occam2 as a rendezvous with the hardware process. Associated with the interrupt, there must be an implementation-dependent address which, in the simple input/output system described in this chapter, is the address of the interrupt vector; a channel is then mapped onto this address (ADDR):

```
CHAN OF ANY Interrupt:
PLACE Interrupt AT ADDR:
```

Note that this is a channel and not a port. This is because there is synchronization associated with an interrupt where there is none associated with access to a device register. The data protocol for this channel will also be implementation-dependent.

The interrupt handler can wait for an input from the designated channel thus:

```
INT ANY:  -- define ANY to be of the protocol type
SEQ
  -- using ports enable interrupt
  Interrupt ? ANY
  -- actions necessary when interrupt has occurred.
```

The run-time support system must, therefore, synchronize with the designated channel when an external interrupt occurs. To obtain responsiveness, the process handling the interrupt will usually be given a high priority. Therefore not only will it be made

executable by the interrupt event, but it will, within a short period of time, actually be executing (assuming that no other high-priority process is running).

To cater for interrupts which are lost if not handled within a specified period, it is necessary to view the hardware as issuing a timeout on the communication. The hardware must therefore conceptually issue:

```
ALT
  Interrupt ? ANY
    SKIP
  CLOCK ? AFTER Time PLUS Timeout
    SKIP
```

and the handler must execute:

```
Interrupt ! ANY
```

This is because only an input request can have a timeout associated with it.

Implementation on memory-mapped and special-instruction machines

To map the occam2 model of device driving to memory-mapped machines simply requires that input and output requests on ports be mapped to read and write operations on the device registers. To map the model to special-instruction machines requires the following:

- an occam2 PORT to be associated with an I/O port using the PLACE statement;
- the data which is sent to an occam2 PORT to be placed in an appropriate accumulator for use with the output machine instruction;
- the data which is received from an occam2 PORT to become available, via an appropriate accumulator, after the execution of the input instruction.

An example device driver

To illustrate the use of the low-level input/output facilities that occam2 provides, a process will be developed that controls an analogue to digital converter (ADC) for a memory-mapped machine. The converter is the same as the one described in the main book and implemented in Ada and Java. In order to read a particular analogue input, a channel address (not to be confused with an occam2 channel) is given in bits 8 to 13 and then bit 0 is set to start the converter. When a value has been loaded into the results register, the device will interrupt the processor. The error flag will then be checked before the results register is read. During this interaction it may be desirable to disable the interrupt.

The device driver will loop round receiving requests and providing results; it is programmed as a PROC with a two-channel interface. When an address (for one of the eight analogue input channels) is passed down `input`, a 16-bit result will be returned via channel `output`.

```
CHAN OF INT16 request:
CHAN OF INT16 return:
PROC ADC(CHAN OF INT16 input, output)
  -- body of PROC, see below
PRI PAR
  ADC(request, return)
  PAR
    -- rest of program
```

A PRI PAR is desirable as the ADC must handle an interrupt each time it is used, and therefore should run at the highest priority.

Within the body of the PROC, the interrupt channel and the two PORTs must first be declared:

```
PORT OF INT16 Control.Register:
PLACE Control.Register AT #AA12#:
PORT OF INT16 Buffer.Register:
PLACE Buffer.Register AT #AA14#:
CHAN OF ANY Interrupt:
PLACE Interrupt AT #40#:
INT16 Control.R:  -- variable representing control register
```

Where #AA12# and #AA14# are the defined hexadecimal addresses for the two registers and #40# is the interrupt vector address.

To instruct the hardware to undertake an operation requires bits 0 and 6 to be set on the control register; at the same time all other bits apart from those between 8 and 13 (inclusive) must be set to zero. This is achieved by using the following constants;

```
VAL INT16 zero IS 0:
VAL INT16 Go IS 65:
```

Having received an address from channel ‘input’, its value must be assigned to bits 8 through 10 in the control register. This is accomplished by using a shift operation. The actions that must be taken in order to start a conversion are, therefore:

```
INT16 Address:
SEQ
  input ? Address
  IF
    (Address < 0) OR (Address > 63)
      output ! MOSTNEG INT16  -- error condition
    TRUE
      SEQ
        Control.R := zero
        Control.R := Address << 8
        Control.R := Control.R BITOR Go
        Control.Register ! Control.R
```

Once an interrupt has arrived, the control register is read and the error flag and ‘Done’ checked. To do this, the control register must be masked against appropriate constants:

```
VAL INT16 Done IS 128:
VAL INT16 Error IS MOSTNEG INT16:
```

MOSTNEG has the representation 1 000 000 000 000 000.

The checks are thus:

```
SEQ
  Control.Register ? Control.R
  IF
    ((Done BITAND Control.R) = 0) OR
    ((Error BITAND Control.R) <> zero)
      -- error
    TRUE
      -- appropriate value is in buffer register
```

Although the device driver will be run at a high priority, the client process in general will not, and hence the driver would be delayed if it attempted to call the client directly and the client was not ready. With input devices that generate data asynchronously, this delay could lead to the driver missing an interrupt. To overcome this, the input data must be buffered. A suitable circular buffer is given below. Note that because the client wishes to read from the buffer and because the ALT in the buffer cannot have output guards, another single buffer item is needed. To ensure that the device driver is not delayed by the scheduling algorithm, the two buffer processes (as well as the driver) must execute at high priority.

```
PROC buffer(CHAN OF INT put, get)
  CHAN OF INT Request, Reply:
  PAR
    VAL INT Buf.Size IS 32:
    INT top, base, contents:
    [Buf.Size]buffer:
    SEQ
      contents := 0
      top := 0
      base := 0
      INT ANY:
      WHILE TRUE
        ALT
          contents < Buf.Size & put ? buffer [top]
            SEQ
              contents := contents + 1
              top := (top + 1) REM Buf.Size
          contents > 0 & Request ? ANY
            SEQ
              Reply ! buffer[base]
              contents := contents - 1
              base := (base + 1) REM Buf.Size
    INT Temp:   -- single buffer process
    VAL INT ANY IS 0:   -- dummy value
    WHILE TRUE
      SEQ
        Request ! ANY
        Reply ? Temp
        get ! Temp
:
```

The full code for the PROC can now be given. The device driver is again structured so that three attempts are made to get a correct reading.

```
PROC ADC(CHAN OF INT16 input, output)
  PORT OF INT16 Control.Register:
  PLACE Control.Register AT #AA12#:
```

```
PORT OF INT16 Buffer.Register:
PLACE Buffer.Register AT #AA14#:

CHAN OF ANY Interrupt:
PLACE Interrupt AT #40#:
TIMER CLOCK:

INT16 Control.R:  -- variable representing control buffer
INT16 Buffer.R:   -- variable representing results buffer
INT Time:

VAL INT16 zero IS 0:
VAL INT16 Go IS 65:
VAL INT16 Done IS 128:
VAL INT16 Error IS MOSTNEG INT16:
VAL INT Timeout IS 600000:  -- or some other appropriate value
INT ANY:
INT16 Address:
BOOL Found, Error:
CHAN OF INT16 Buff.In:

PAR
  buffer(Buff.In, output)
  INT16  Try:
  WHILE TRUE
    SEQ
      input ? Address
      IF
        (Address < 0) OR (Address > 63)
          Buff.In ! MOSTNEG INT16
                    -- error condition
        TRUE
          SEQ
            Try := 0
            Error := FALSE
            Found := FALSE
            WHILE (Try < 3) AND ((NOT Found) AND (NOT Error))
              -- Three attempts are made to get a reading from
              -- the ADC. This reading may be either correct or
              -- is flagged as being an error.
              SEQ
                Control.R := zero
                Control.R := Address << 8
                Control.R := Control.R BITOR Go
                Control.Register ! Control.R
                CLOCK ? Time
                ALT
                  Interrupt ? ANY
                    SEQ
                      Control.Register ? Control.R
                      IF
                        ((Done BITAND Control.R) = 0) OR
                             ((Error BITAND Control.R) <> zero)
                          SEQ
                            Error := TRUE
                            Buff.In ! MOSTNEG INT16 -- error condition
                        TRUE
```

```
                        SEQ
                          Found := TRUE
                          Buffer.Register ? Buffer.R
                          Buff.In ! Buffer.R
                    CLOCK ? AFTER Time PLUS Timeout
                      -- The device is not responding
                      Try := Try + 1
            IF
              (NOT Found) AND (NOT Error)
                Buff.In ! MOSTNEG INT16
              TRUE
                SKIP
:
```

Difficulties with device driving in occam2

The above example illustrates some of the difficulties in writing device drivers and interrupt handlers in occam2. In particular, there is no direct relationship between the hardware priority of the device and the priority assigned to the driver process. To ensure that high-priority devices are given preference, it is necessary to order all the device drivers appropriately at the outer level of the program in a `PRI PAR` construct.

The other main difficulty stems from the lack of data structures for representing device registers. This results in the programmer having to use low-level bit manipulation techniques, which can be error-prone.

APPENDIX B: Modula-1

Modula-1 (as it must now be known) is the forerunner to Modula-2 and Modula-3. It has a Pascal like syntax.

B.1 Concurrency model

Modula employs explicit process declaration and monitors which are termed **interface modules**.

The following example is in Modula-1. It presents a simple structure for a robot arm controller. A distinct process is used to control each dimension of movement. These processes loop around, each reading a new setting for its dimension and then calling a low-level procedure `move_arm` to cause the arm to move.

```
MODULE main;
  TYPE dimension = (xplane, yplane, zplane);

  PROCESS control(dim : dimension);
    VAR position : integer;     (* absolute position *)
        setting : integer;      (* relative movement *)
  BEGIN
    position := 0;              (* rest position *)
    LOOP
       move_arm(dim, position);
       new_setting(dim, setting);
       position := position + setting
    END
  END control;

BEGIN
  control(xplane);
  control(yplane);
  control(zplane)
END main.
```

In the above, the process `control` is declared with a parameter to be passed on creation. The example then creates three instances of this process, passing each a distinct parameter.

Modula supports Hoare's monitors via interface modules. Somewhat confusingly, condition variables are called signals and are acted upon by three procedures.

1. The procedure `wait(s,r)` delays the calling process until it receives the signal `s`. The process, when delayed, is given a priority (or delay rank `r`) where `r` must be a positive valued integer expression whose default is 1.

2. The procedure `send(s)` sends the signal `s` to that process with the highest priority which has been waiting for `s`. If several waiting processes all have the same priority then the one which has been waiting the longest receives the signal. The process executing the send is suspended. If no process is waiting the call has no effect.

3. The boolean function `awaited(s)` yields the value true if there is at least one process blocked on `s`; false otherwise.

The following is a simple resource controller monitor:

```
INTERFACE MODULE resource_control;

  DEFINE allocate, deallocate;  (* export list *)

  VAR busy : BOOLEAN;
      free : SIGNAL;

  PROCEDURE allocate;
  BEGIN
    IF busy THEN WAIT(free) END;
    busy := TRUE;
  END;

  PROCEDURE deallocate;
  BEGIN
    busy := FALSE;
    SEND(free)
  END;

BEGIN (* initialization of module *)
  busy := FALSE
END.
```

Note that in `deallocate`:

```
if AWAITED(free) then SEND(free)
```

could have been inserted, but as the effect of `SEND(free)` is null, when `AWAITED(free)` is false, there is nothing to be gained by doing the test.

B.2 Modula-1 device driving

Modula-1 was one of the first high-level programming languages which had facilities for programming device drivers.

In Modula-1, the unit of modularity and encapsulation is the module. A special type of module, called an **interface module**, which has the properties of a monitor, is used to control access to shared resources. Processes interact via signals (condition

variables) using the operators WAIT, SEND and AWAITED (see Chapter 8). A third type of module, called a **device module** is a special type of interface module used to encapsulate the interaction with a device. It is only from within a device module that the facilities for handling interrupts can be used.

Addressing and manipulating device registers

Associating a variable with a register is fairly straightforward. In Modula-1, this is expressed by an octal address following the name in a declaration. For example, a data buffer register for the simple I/O architecture described in the main Book would be defined as:

```
var rdbr[177562B] char;
```

where 177562B denotes an octal address which is the location of the register in memory.

The mapping of a character into a character buffer register is also a straightforward activity, since the type has no internal structure. A control and status register is more interesting. In Modula-1, only scalar data types can be mapped onto a device register; consequently registers which have internal structures are considered to be of the predefined type *bits* whose definition is:

```
TYPE BITS = ARRAY 0:no_of_bits_in_word OF BOOLEAN;
```

Variables of this type are packed into a single word. A control and status register at octal address 177560B can, therefore, be defined by the following Modula-1 code:

```
VAR rcsr[177560B] : BITS;
```

To access the various fields in the register, an index into the array is supplied by the programmer. For example, the following code will enable the device:

```
rcsr[0] := TRUE;
```

and the following turns off interrupts:

```
rcsr[6] := FALSE;
```

In general, these facilities are not powerful enough to handle all types of register conveniently. The general structure of the control and status register was given earlier:

```
bits
 15 - 12  : Errors
 11       : Busy
 10 -  8  : Unit select
  7       : Done/ready
  6       : Interrupt enable
  5 -  3  : reserved
  2 -  1  : Device function
  0       : Device enable
```

To set the selected unit (bits 8–10) using boolean values is very clumsy. For example, the following statements set the device unit to the value 5.

```
rcsr[10] := TRUE;
rcsr[9]  := FALSE;
rcsr[8]  := TRUE;
```

It is worth noting that on many machines more than one device register can be mapped to the same physical address. Consequently, several variables may be mapped to the same location in memory. Furthermore, these registers are often read or write only. Care, therefore, must be taken when manipulating device registers. In the above example, if the control and status register was a pair of registers mapped to the same location, the code presented will probably not have the desired effect. This is because to set a particular bit may require code to be generated which reads the current value into the machine accumulator. As the control register is write-only, this would produce the value of the status register. It is advisable, therefore, to have other variables in a program which represent device registers. These can be manipulated in the normal way. When the required register format has been constructed, it may then be assigned to the actual device register. Such variables are often called **shadow device registers**.

Interrupt handling

The facilities for handling interrupts in Modula-1 are based around the concept of an ideal hardware device. This device has the following properties:

- For each device operation, it is known how many interrupts are produced.
- After an interrupt has occurred, the device status indicates whether or not another associated interrupt will occur.
- No interrupt arrives unexpectedly.
- Each device has a unique interrupt location.

The facilities provided by Modula-1 may be summarized by the following points.

- Each device has an associated device module.
- Each device module has a hardware priority specified in its header following the module name.
- *All code within the module executes at the specified hardware priority.*
- Each interrupt to be handled within a device module requires a process called a **device process**.
- When the device process is executing, it has sole access to the module (that is, it holds the monitor lock using the ceiling priority specified in the device module header).
- A device process is not allowed to call any non-local procedures and cannot send signals to other device processes. This is to ensure that device processes will not be inadvertently blocked.

- When a device process sends a signal, the semantics of the send operation are different from those for ordinary Modula-1 processes; in this case the receiving process is not resumed, but the signalling process continues. Again this is to ensure that the process is not blocked.
- `WAIT` statements within device processes may only be of rank 1 (highest level).
- An interrupt is considered to be a form of signal. The device process, however, instead of issuing a `WAIT` request issues a `DOIO` request.
- The address of the vector through which the device interrupts is specified in the header of the process.
- *Only device processes can contain `DOIO` statements.*
- `DOIO` and `WAIT` calls lower the processor priority and, therefore, release the monitor lock.
- Only one instance of a device process may be activated.

For example, consider a device module which handles a real-time clock for the simple machine architecture outlined in the main Book. On receipt of an interrupt, the handler sends a signal to a process which is waiting for the clock to tick.

```
DEVICE MODULE rtc[6]; (* hardware priority 6 *)

    DEFINE tick;
    VAR tick : SIGNAL;

    PROCESS clock[100B];
        VAR csr[177546B] : BITS;
    BEGIN
        csr[0] := TRUE;  (* enable device *)
        csr[6] := TRUE;  (* enable interrupts *)
        LOOP
            DOIO;
            WHILE AWAITED(tick) DO
                SEND(tick);
            END
        END
    END;
BEGIN
    clock; (* create one instance of the clock process *)
END rtc;
```

The heading of the device module specifies an interrupt priority of `6`, at which all code within the module will be executed. The value `100B` on the process header indicates that the device will interrupt through the vector at address (octal) `100`. After enabling interrupts, the device process enters a simple loop of waiting for an interrupt (the `DOIO`) and then sending sufficient signals (that is, one per waiting process). Note that the device process does not give up its mutually exclusive access to the module when it sends a signal, but continues until it executes a `WAIT` or a `DOIO` statement.

The following illustrates how Modula-1 deals with the general characteristics of an interrupt driven device which were outlined in Sections 15.1.2 and 15.1.3.

- **Device control** – I/O registers are represented by variables.
- **Context switching** – The interrupt causes an immediate context switch to the interrupt-handling process, which waits using the `DOIO`.
- **Interrupt device identification** – The address of the interrupt vector is given with the device process's header.
- **Interrupt identification** – In the above example, only one interrupt was possible. In other cases, however, the device status register should be checked to identify the cause of the interrupt.
- **Interrupt control** – The interrupt control is status driven and provided by a flag in the device register.
- **Priority control** – The priority of the device is given in the device module header. *All* code in the module runs at this priority (that is, the device module has a hardware ceiling priority and executes with the Immediate Priority Ceiling Protocol.

An example terminal driver

To illustrate further the Modula-1 approach to device driving, a simple terminal device module is presented. The terminal has two components: a display and a keyboard. Each component has an associated control and status register, a buffer register and an interrupt.

Two procedures are provided to allow other processes in the program to read and write characters. These procedures access a bounded buffer to allow characters to be typed ahead for input and buffered for output. These buffers must be included in the device module because device processes *cannot* call non-local procedures. Although separate modules for the display and keyboard could have been used, they have been combined to illustrate that a device module can handle more than one interrupt.

```
DEVICE MODULE terminal[4];

  DEFINE readch, writech;

  CONST n=64;           (* buffer size *)

  VAR KBS[177560B]: BITS; (* keyboard status *)
      KBB[177562B]: CHAR; (* keyboard buffer *)
      DPS[177564B]: BITS; (* display status  *)
      DPB[177566B]: CHAR; (* display buffer  *)
      in1, in2, out1, out2 : INTEGER;
      n1, n2 : INTEGER;
      nonfull1, nonfull2,
      nonempty1, nonempty2 : SIGNAL;
      buf1, buf2 : ARRAY 1:n OF CHAR;
```

```
  PROCEDURE readch(VAR ch : CHAR);
  BEGIN
    IF n1 = 0 THEN WAIT(nonempty1) END;
    ch := buf1[out1];
    out1 := (out1 MOD n)+1;
    DEC(n1);
    SEND(nonfull1)
  END readch;

  PROCEDURE writech(ch : CHAR);
  BEGIN
    IF n2 = n THEN WAIT(nonfull2) END;
    buf2[in2] := ch;
    in2 := (in2 MOD n)+1;
    INC(n2);
    SEND(nonempty2)
  END writech;



  PROCESS keyboarddriver[60B];
  BEGIN
    KBS[0] := TRUE;  (* enable device *)
    LOOP
      IF n1 = n THEN WAIT(nonfull1) END;
      KBS[6] := TRUE;
      DOIO;
      KBS[6] := FALSE;
      buf1[in1] := KBB;
      in1 := (in1 MOD n)+1;
      INC(n1);
      SEND(nonempty1)
    END
  END keyboarddriver;

  PROCESS displaydriver[64B];
  BEGIN
    DPS[0] := TRUE;  (* enable device *)
    LOOP
      IF n2 = 0 THEN WAIT(nonempty2) END;
      DPB := buf2[out2];
      out2 := (out2 MOD n)+1;
      DPS[6] := TRUE;
      DOIO;
      DPS[6] := FALSE;
      DEC(n2);
      SEND(nonfull2)
    END
  END displaydriver;

BEGIN
  in1 :=1; in2 := 1;
  out1 := 1; out2 :=1;
  n1 :=0; n2 := 0;
  keyboarddriver;
  displaydriver
END terminal;
```

Timing facilities

Modula-1 provides no direct facilities for manipulating time; these have to be provided by the application. This requires a device module which handles the clock interrupt and then issues a regular signal, say, every second. This module is now presented; it is a modified version of the one previously defined. The hardware clock is assumed to tick every fiftieth of a second.

```
DEVICE MODULE hardwareclock[6];
    DEFINE tick;
    VAR tick : SIGNAL;

    PROCESS handler[100B];
        VAR count : INTEGER;
            statusreg[177546B] : BITS;
    BEGIN
        count := 0;
        statusreg[0] := TRUE;
        statusreg[6] := TRUE;
        LOOP
            DOIO;
            count := (count+1) MOD 50;
            IF count = 0 THEN
                WHILE AWAITED(tick) DO
                    SEND(tick)
                END
            END
        END
    END handler;
BEGIN
    handler
END hardwareclock;
```

An interface module which maintains the time of day can now be easily provided.

```
INTERFACE MODULE SystemClock;
    (* defines procedures for getting and setting the time of day *)
    DEFINE GetTime, SetTime;

    (* import the abstract data type time, and the tick signal *)
    USE time, initialise, add, tick;

    VAR TimeOfDay, onesec : time;

    PROCEDURE SetTime(t: time);
    BEGIN
        TimeOfDay := t
    END SetTime;

    PROCEDURE GetTime(VAR t: time);
    BEGIN
        t := TimeOfDay
    END GetTime;

    PROCESS clock;
    BEGIN
        LOOP
```

```
            WAIT(tick);
            addtime(TimeOfDay, onesec)
        END
    END clock;
BEGIN
    inittime(TimeOfDay, 0, 0, 0);
    inittime(onesec, 0, 0, 1);
    clock;
END SystemClock;
```

Note that the clock process is logically redundant. The device process could increment `systemtime` directly, thereby saving a context switch. However, it is not allowed in Modula-1 for a device process to call a non-local procedure.

Delaying a process

In real-time systems, it is often necessary to delay a process for a period. Although Modula-1 has no direct facilities for achieving this, they can be programmed.

Problems with the Modula-1 approach to device driving

Modula-1 was designed to attack the stronghold of assembly language programming – that of interfacing to devices. In general, it has been considered a success; however, there are a few criticisms that have been levelled at its facilities.

- Modula-1 does not allow a device process to call a non-local procedure because device processes must be kept as small as possible and must run at the hardware priority of the device. To call procedures defined in other modules, whose implementation is hidden from the process, might lead to unacceptable delays. Furthermore, it would require these procedures to execute at the device's priority. Unfortunately, as a result of this restriction, programmers either have to incorporate extra functionality into a device module which is *not* directly associated with driving the device (as in the terminal driver example, where a bounded buffer was included in the device module), or they have to introduce extra processes to wait for a signal sent by a device process. In the former case, this can lead to very large device modules and in the latter, unnecessary inefficiency.

- Modula-1 only allows a single instance of a device process because the process header contains the information necessary to associate the process with the interrupt. This makes the sharing of code between similar devices more difficult; the problem is compounded by not being able to call non-local procedures.

- Modula-1 was designed for memory-mapped machines and consequently it is difficult to use its facilities for programming devices which are controlled by special instructions. However, it is easy to imagine a simple extension to solve this problem. One suggestion is the possibility of using the following notation:

  ```
  VAR x AT PORT 46B : INTEGER;
  ```

The compiler is then able to recognize when a port is being addressed and can generate the correct instructions.

- It has already been pointed out that many device registers are read- or write-only. It is not possible to define variables that are read- or write-only in Modula-1. Furthermore, there is an implicit assumption that a compiler will not optimize access to device registers and cache them in local registers.