Pascal-FC
Version 5
Language Reference Manual

G.L. Davies
University of Bradford, UK

1. INTRODUCTION

1.1. Purpose of Pascal-FC

Pascal-FC[5] is a dialect of Pascal which has been designed specifically as a teaching tool. Its purpose is to provide students with practical experience of concurrent programming. In courses in this subject, it is usual to consider a variety of facilities for inter-process communication. For example, Ben Ari[2], in a widely-used text, considers semaphores, monitors and the Ada rendezvous. For practical work, he provides a dialect of Pascal which includes an implementation of semaphores, but which lacks monitors and the Ada rendezvous. Pascal-FC, on the other hand, includes the following:

- semaphores;
- monitors;
- an occam/CSP style rendezvous;
- an Ada-style rendezvous;
- resources, which combine some of the features of Conditional Critical Regions[7] and monitors.

The aim of the system is to expose students to a variety of styles of inter-process communication without the distraction of having to learn the syntax of several different languages.

In addition to its concurrency facilities, the language also includes optional features which enable it to be used for teaching the programming of real-time *embedded systems* where suitable hardware is available. These optional features are concerned with the *timing* of events, and with low-level programming, including interrupt-handling.

1.2. Historical Background

Pascal-FC is a major extension of Ben Ari's concurrent Pascal-S. However, Ben Ari in turn based his system on a purely sequential Pascal-S developed by Wirth (see Berry[4]). Wirth's 'S' language was a subset of standard Pascal, from which had been omitted a number of features (for example, sets, files and dynamic data structures) found in standard Pascal. The reader experienced in standard Pascal will find, therefore, that these familiar features are not supported by Pascal-FC.

1.3. Scope of the Manual

Some of the features of the language are intended for real-time applications, and the inclusion and restrictions imposed on such features are necessarily implementation-dependent. This manual does not describe any specific implementation, so that only the general form of such features is given here: information specific to particular implementations will be provided in the User Guide for the implementation.

1.4. Syntax Notation

This Manual uses a notation similar to the one adopted in the Ada Language Reference Manual[1]. Specifically, the syntax is described in a variant of Backus-Naur Form (BNF), supplemented with ordinary English. Semantics are described in ordinary English.

The following conventions are adopted for the BNF notation.

- Each rule is introduced by the name of a syntactic category followed by "::=".
- Lower-case words, some of which contain underscore characters, are used to denote syntactic categories. For example:

```
identifier
select_statement
```

- Bold-face words are used to denote reserved words. For example:

```
begin
process
```

- A character enclosed in double quotes stands for itself, and is not a syntactic category or special symbol. For example, in the rule:

```
exponent_part  ::=

    ["e"|"E"] [+|-] unsigned_integer
```

the characters "e" and "E" are *not* the names of syntactic categories.

- Square brackets enclose optional items, except when enclosed in double quotes, when they stand for themselves. For example:

```
if_statement ::=

    if boolean_expression then
       statement
   [else
       statement]

array_index ::=

    "["ordinal_expression{,ordinal_expression}"]"
```

The optional part is either absent, or *one* such part is permitted.

- Braces enclose repeated items, except when enclosed in double quotes, when they stand for themselves. For example:

```
identifier  ::=

    letter{letter | digit}

comment ::=

    comment_start {character} comment_end
```

```
comment_start ::=

   "{" | (*

comment_end ::=

   "}" | *)
```

When braces enclose a repeated item, the item may be repeated zero or more times.

- Alternatives are separated by the "|" character, as in the above examples.
- Italics are used to convey some semantic information. For example:

  *boolean*_`expression`

  Such information is provided as a hint to the reader: a context-free grammar is, of course, not capable of representing the difference between, for example, a boolean expression and an integer expression.

2. PROGRAM STRUCTURE, DECLARATIONS AND STATEMENTS

The only compilation unit in Pascal-FC is the *program*. The next section describes the overall form of a program and some of its components: later sections describe declarations and statements.

2.1. Program

A program is defined a follows:

```
program  ::=

   program_header
      global_declaration_part
   begin
      main_statement_part
   end.


program_header  ::=

   program identifier;


identifier  ::=

   letter{letter | digit}
```

Here, `letter` signifies the normal upper- and lower-case alphabetical characters, and `digit` denotes the decimal digits. The case of alphabetical characters in identifiers is not significant.

Certain forms of declaration are only permitted in a global declaration part: these include monitor, resource and process declarations. The following syntax lists the possible forms of declaration:

```
global_declaration_part  ::=

     {
     constant_declaration
   | type_declaration
   | variable_declaration
   | monitor_declaration
   | resource_declaration
   | procedure_declaration
   | function_declaration
   | process_type_declaration
   | process_object_declaration
     }
```

The main statement part is the only place where a concurrent statement may be placed, and there may be at most one of these. The allowable forms of the main statement part are:

```
main_statement_part  ::=

     statement_sequence
    [;concurrent_statement
    [;statement_sequence]]
   | concurrent _statement
    [;statement_sequence]



statement_sequence ::=

     statement
    {;statement}
```

The concurrent statement and its use are described in Chapter 3.

2.2. Declarations

Monitor, resource, process and entry declarations are ignored here, as they are covered in later chapters. The remaining forms are, for the most part, like those of Pascal, but with some additions and restrictions. One such restriction is that there are no **`label`** declarations. One sense in which Pascal-FC is *less* restrictive than Pascal is that the *order* of different types of declaration is not fixed. Like Pascal, Pascal-FC is based on the principle of declaration-before-use.

2.2.1. Constant declarations

Constants are declared by means of a **const** declaration, which has the following form:

```
constant_declaration  ::=

   const
      identifier = constant;
     {identifier = constant;}
```

where:

```
constant  ::=

   constant_identifier
 | integer_literal
 | real_literal
 | character_literal
```

Note that, unlike Pascal, there are no string constants.

2.2.1.1. Character literals

As in Pascal, the syntax of a character literal is:

```
character_literal  ::=

   'character'
```

The set of permissible characters is not defined by the language.

2.2.1.2. Integer literals

There are two basic forms of integer literal: decimal and based. Based literals allow a number to be expressed in bases other than 10.

```
integer_literal  ::=

   decimal_integer
 | based_integer

decimal_integer  ::=

   [+|-] unsigned_integer

unsigned_integer  ::=

   decimal_digit{decimal_digit}
```

```
based_integer  ::=

   base#digit_character{digit_character}

base  ::=

   unsigned_integer
```

NOTES

1. The value for the base of a based integer may be restricted by the implementation.
2. The interpretation of a based integer (for example, unsigned integer or two's complement integer) is implementation-dependent.
3. The compiler will ensure that the digit characters used in a based integer are appropriate for the value of base selected.

2.2.1.3. Real literals

Real literals have the same form as in Pascal.

```
real_literal  ::=

   [+|-] unsigned_real

unsigned_real  ::=

   unsigned_integer exponent_part
 | unsigned_integer fractional_part [exponent_part]

exponent_part  ::=

   ["e"|"E"] [+|-] unsigned_integer

fractional_part  ::=

   .unsigned_integer
```

2.2.2. Type declarations

As in Pascal, a type declaration has the form:

```
type_declaration  ::=

   type
      identifier = type;
     {identifier = type;}
```

The allowable forms for a type are slightly different: Pascal-FC does not have subrange, set or pointer types, but it does introduce an additional channel type.

```
type  ::=

     type_identifier
   | enumeration_type
   | array_type
   | record_type
   | channel_type
```

Certain type identifiers are pre-defined (the so-called "standard types"). These, and the operations defined on the standard types, are given in Appendix C.

Types fall into two categories: scalar and structured. The scalar types are: `boolean`, `char`, `integer` and `real`. Of these, the first three are called "ordinal" types. Channel type declarations will not be considered here: they are described in Chapter 7.

2.2.2.1. Enumeration types

Enumeration types are like those of Pascal. The declaration of an enumeration type has the form:

```
enumeration_type  ::=

     (identifier_list)

identifier_list  ::=

     identifier{,identifier}
```

NOTE

Each identifier must be unique in the current scope.

2.2.2.2. Array types

These have the form:

```
array_type  ::=

    array index_type{index_type} of type

index_type  ::=

     "["ordinal_range{,ordinal_range}"]"

ordinal_range  ::=

     ordinal_constant..ordinal_constant
```

2.2.2.3. Record types

Pascal-FC differs from Pascal in two ways: variant records are not allowed, and an "offset indicator" may be given to fields.

```
record_type  ::=

   record
      field_list
   end

field_list  ::=

   field_declaration {;field_declaration}

field_declaration  ::=

   identifier[offset_indicator]
  {,identifier [offset_indicator]}
   : type

offset_indicator  ::=

   at offset integer_constant
```

The significance of the offset indicator is explained in Chapter 11.

2.2.3. Variable declarations

Variable declarations are similar to Pascal, but with the addition of an optional "mapping indicator".

```
variable_declaration  ::=

   var
      variable_list : type;
     {variable_list : type;}

variable_list  ::=

   identifier [mapping_indicator]
  {,identifier [mapping_indicator]}

mapping_indicator  ::=

   at integer_constant
```

The significance of the mapping indicator is described in Chapter 11.

2.2.4. Procedure and function declarations

In Pascal-FC, there are three classes of subprogram: procedures, functions and processes. The first two of these are known as "sequential subprograms". Processes are considered in Chapter 3.

Procedure and function declarations are largely as in Pascal. Forward declarations are supported. However, there are no conformant array parameters, or parameters that are themselves subprograms.

```
sequential_subprogram_declaration ::=

   full_sequential_subprogram_declaration
 | deferred_sequential_subprogram_declaration

full_sequential_subprogram_declaration  ::=

   sequential_subprogram_header
     [declaration_part]
   begin
      statement_sequence
   end;

sequential_subprogram_header  ::=

   procedure_header
 | function_header

procedure_header  ::=

   procedure identifier [formal_part];

function_header  ::=

   function identifier [formal_part]
   : type_identifier;

formal_part ::=

   ([var] identifier_list : type_identifier
   {;[var] identifier_list : type_identifier})
```

```
declaration_part  ::=

   {
   constant_declaration
 | type_declaration
 | variable_declaration
 | procedure_declaration
 | function_declaration
   }

deferred_sequential_subprogram_declaration  ::=

   procedure_header forward; procedure_stub
 | function_header forward; function_stub

procedure_stub  ::=

   procedure identifier;
     [declaration_part]
   begin
      statement_sequence
   end;

function_stub  ::=

   function identifier;
     [declaration_part]
   begin
      statement_sequence
   end;
```

NOTES

1. When a deferred declaration is used, other declarations may separate the header from the stub.
2. The type identifier in a function header declares the type of the returned result, and it must be of a scalar type.
3. The result of a function is returned by an assignment statement in the enclosed sequence of statements, in which the function name appears on the left of the assignment operator.
4. If a function is exited without the execution of the above type of assignment, no error need be signalled, but the returned result is undefined.
5. **var** parameters ("variable parameters") are passed by reference. Other parameters are called "value parameters", and are passed by value.

2.3. Statements

The allowable statements in Pascal-FC are described by the following syntax.

```
statement ::=

    assignment_statement
  | procedure_call
  | for_statement
  | repeat_statement
  | while_statement
  | if_statement
  | case_statement
  | compound_statement
  | empty_statement
  | concurrent_statement
  | process_activation
  | monitor_call
  | channel_operation
  | select_statement
  | entry_call
  | accept_statement
  | resource_call
  | requeue_statement
  | null_statement
```

Assignment statements, procedure calls, the empty statement, the compound statement and **`if`** and **`case`** statements have the same form as in Pascal. The loops are also similar, except that the **`for`** statement has no **`downto`** variant, and the **`repeat`** statement is slightly extended.

Many of the above statements are concerned with the concurrency features of Pascal-FC. They will not be described here, but in the appropriate later chapters.

2.3.1. The assignment statement

This has the same form as in Pascal:

```
assignment_statement  ::=

    variable := expression


variable  ::=

    variable_identifier{selector}
```

selector ::=

  array_subscript
| field_selector

array_subscript ::=

  "["*ordinal*_expression{,*ordinal*_expression}"]"

expression ::=

  simple_expression {rel_op simple_expression}

rel_op ::=

  < | <= | > | >= | = | <> | **in**

simple_expression ::=

  [+|-] term {add_op term}

add_op ::=

  + | - | **or**

term ::=

  factor {mul_op factor}

mul_op ::=

  * | / | **div** | **mod** | **and**

factor ::=

  unsigned_integer
| based_integer
| unsigned_real
| *constant*_identifier
! variable
! function_identifier [(actual_parameters)]
| **not** factor
! bitset_literal
| (expression)

```
actual_parameters  ::=

   expression {,expression}

bitset_literal  ::=

   "[""]"
 | "["integer_expression{,integer_expression}"]"

field_selector  ::=

   .record_field_identifier
```

Pascal-FC is strongly typed, so that the types of the two operands of the assignment operator (":=") must be equivalent.

2.3.2. The case statement

The **`case`** statement is the same as in Pascal. Its syntax is:

```
case_statement  ::=

   case ordinal_expression of
      case_alternative
     {;case_alternative}
   end

case_alternative  ::=

   case_label{,case_label}:statement

case_label  ::=

   ordinal_constant
```

NOTES

1. The expression following the reserved word, **`case`** (known as the "selector expression"), and the case labels must have equivalent types.
2. It is a compile-time error to have the same value of case label appearing more than once in a single **`case`** statement.
3. It is a run-time error if the value of the selector expression does not equal *any* of the case labels.
4. At run time the statement which has a label equal to the selector expression will be executed (if there is one) and the **`case`** statement will then be exited.

2.3.3. The compound statement

As in Pascal, the compound statement has the form:

```
compound_statement  ::=

    begin
    statement_sequence
    end
```

2.3.4. The empty statement

This is included for consistency with Pascal, but it is recommended that the **null** statement be used as a more readable alternative.

```
empty_statement  ::=

    {white_space_character}
```

2.3.5. The for statement

This is a slightly restricted version of the **for** statement found in Pascal, in that there is no **downto** variant. The syntax is:

```
for_statement  ::=

   for variable := expression to expression do
       statement
```

NOTES

1. The variable (known as "the loop control variable") and the two expressions (known respectively as "the initial value" and "the terminal value") must be equivalent ordinal types.
2. The value of the loop control variable is undefined on exit from the loop.
3. If the initial value is greater than the terminal value, no iterations are performed and the loop is immediately exited.
4. If the initial value is less than or equal to the terminal value, the number of iterations will be `1 + (terminal value - initial value)`.
5. The number of iterations cannot be modified by making an assignment to the loop control variable in the nested statement.

2.3.6. The if statement

As in Pascal, the **if** statement has the form:

```
if_statement ::=

   if boolean_expression then
      statement
  [else
      statement]
```

2.3.7. Procedure call

As in Pascal, this has the form:

```
procedure_call  ::=

   procedure_identifier [(actual_parameters)]
```

The expressions in the actual parameters must agree in number, type and mode with the formal part of the procedure declaration.

2.3.8. The null statement

The null statement has the form:

```
null_statement ::=

   null
```

The execution of a **null** statement has no effect, and it may be used instead of the empty statement for enhanced readability.

2.3.9. The repeat statement

Compared with Pascal, the **repeat** statement is augmented to be:

```
repeat_statement ::=

   repeat
      statement_sequence
   repeat_limit



repeat_limit ::=

   until boolean_expression
 | forever
```

2.3.10. The while statement

As in Pascal, this has the form:

```
while_statement  ::=

   while boolean_expression do
      statement
```

2.4. Comments

Comments in a Pascal-FC progrm have the form:

```
comment ::=

   comment_start {character} comment_end

comment_start ::=

   "{" | (*

comment_end ::=

   "}" | *)
```

NOTE

Comments must not be nested.

3. PROCESSES

Pascal-FC has a "flat" process structure in that processes may only be declared at the outermost lexical level. The declaration of a process is not sufficient to bring it into execution: process *activation* is required to achieve this.

3.1. Process States

The process state model of Pascal-FC is comparatively simple. It is illustrated in Figure 3.1.

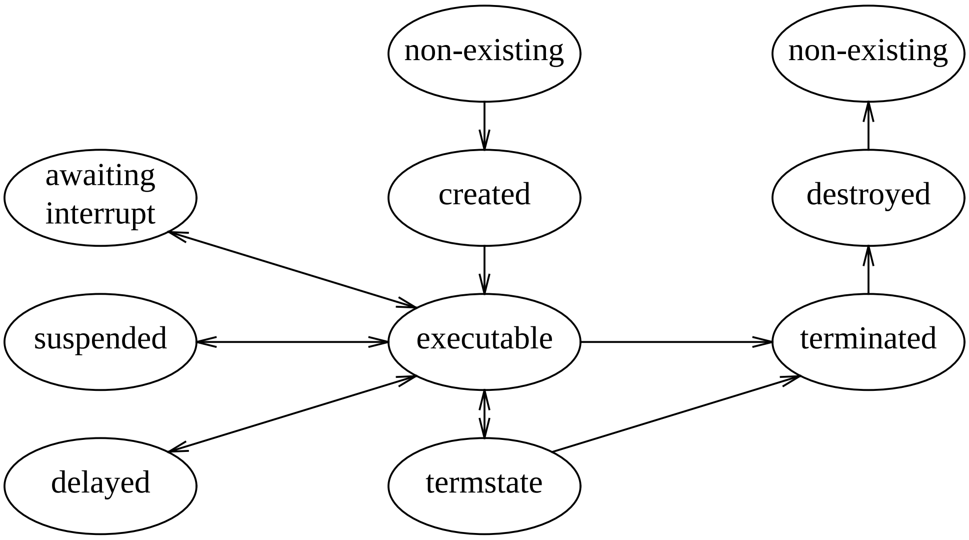

Figure 3.1: Process States and Transitions

NOTES

1. A process is "created" by a process object declaration.
2. A process is made "executable" by a process activation.
3. A process becomes "terminated" when it has finished its execution.
4. When *all* processes have become "terminated", they are all "destroyed" simultaneously.
5. A process that is "delayed" is one that is non-executable for a bounded time (see Chapter 10).
6. A "suspended" process is one whose non-executable period is not necessarily bounded. It is suspended on some inter-process communication primitive and is dependent on some other process to make it executable again (see Chapters 4 - 9).
7. A process that is "awaiting interrupt" is blocked on a semaphore, channel or entry that has been mapped to a source of interrupts (see Chapter 11).

8. For an explanation of "termstate", see Chapter 9.
9. The term "blocked" will sometimes be used to refer to a process in any of the states "awaiting interrupt", "suspended", "delayed" or "termstate".

3.2. Process Declarations

Process declarations are either process object declarations or process type declarations.

3.2.1. Process Type Declarations

```
process_type_declaration ::=

  [process_type_provides_declaration]
   process_type_body_declaration

process_type_provides_declaration ::=

  process type identifier[formal_part] provides
      entry_declaration
     {entry_declaration}
  end;

process_type_body_declaration ::=

   process type identifier[formal_part];
       {entry_declaration}
       [declaration_part]
   begin
      statement_sequence
   end;
```

Entry declarations will be described in Chapter 8.

Where the optional "provides" declaration is used, the following points should be noted:

1. There must be a corresponding body declared later in the same declaration part.
2. These two components may be separated by other declarations.
3. The two parts must correspond exactly in their formal parts.
4. The number of entries, their identifiers and the formal parts of the entries, must match exactly.
5. The order of declaration of the entries is not constrained to be the same.

The declaration of a process type does not bring into existence any objects of that type, but instead introduces a type identifier which may be used in type and variable declarations. Process types can be elements of arrays, but not of records. The formal parameters of a process have the same form as the formal parameters of procedures and functions.

```
formal_part ::=

   ([var] identifier_list : type_identifier
   {;[var] identifier_list : type_identifier})
```

There are certain restrictions on the types of formal parameters. Process type identifiers are not permitted, and identifiers of types containing semaphores, conditions or channels must be **var** parameters.

3.2.2. Process Object Declarations

A process object declaration brings into existence an executable instance of a process type (anonymous or named). It introduces an identifier which may then be used in a process activation. In terms of Figure 3.1, declaration of a process object causes the process to make the transition from "non-existing" to "created".

```
process_object_declaration ::=

   anonymous_process_type_declaration
 | process_variable_declaration

anonymous_process_type_declaration ::=

  [provides_declaration]
   process_body_declaration
```

The forms of the "provides" and "body" declarations are the same as those described in the previous section, except that **type** is omitted. Restrictions are also the same. A process variable declaration may only appear in a **var** declaration in a global declaration part, and has the following form:

```
process_variable_declaration ::=

   identifier_list : process_type;


process_type  ::=

   process_identifier
  | process_array_type
```

3.3. Process Activation

A process object declaration brings into existence an instance of a process type, but it does not automatically make it executable: a process activation is required. In terms of Figure 3.1, the activation of a process causes it to make the transition from "created" to "executable".

```
process_activation ::=

    process_object_identifier[array_index][(actual_parameters)]
```

A process activation can only be placed in a concurrent statement.

3.3.1. The concurrent statement

A concurrent statement has the form:

```
concurrent_statement  ::=

    cobegin
    statement_sequence
    coend
```

NOTES

1. Although arbitrary statements may be placed in a concurrent statement, only *processes* are executed concurrently.
2. Within the concurrent statement, any particular process may only be activated once.
3. The language prescribes no particular scheduling policy for processes.
4. No process will begin its execution before all processes activated in the concurrent statement have been made executable.
5. The concurrent statement cannot terminate until all activated processes have terminated.
6. If any process encounters a fatal error during its execution, the entire program is aborted.

3.3.2. Activating elements of an array of processes

Given the following declarations:

```
const
  max = 5;

process type proc;
  (* declarations *)
begin
  (* statements *)
end;

var
  p: array[1..max]of  proc;
```

One method of activating the elements of the array would be:

```
cobegin
  p[1];
  p[2];
  p[3];
  p[4];
  p[5]
coend
```

This quickly becomes tedious, however, and Pascal-FC allows a **`for`** loop to be used as a shorthand notation. Given a suitable declaration for the loop control variable, the above could more succinctly be expressed as:

```
cobegin
  for i := 1 to max do
    p[i]
coend
```

The **`repeat`** and **`while`** loops can also be used for the same purpose.

3.4. Phases of Execution of a Pascal-FC Program

In general, there will be four distinct phases in the execution of a Pascal-FC program:

1. A preliminary sequential phase, which will usually be used to prepare the execution environment for concurrent processes (for example, by the initialisation of global variables). This is the set of statements before the **`cobegin`**.
2. A *process activation* phase, during which the concurrent processes themselves are prepared for execution (that is, they make the transition from "created" to "executable").
3. Concurrent execution of processes.
4. A sequential *completion phase* , which begins only when the last concurrent process has terminated. This corresponds with the statements following the **`coend`**.

3.5. Process Scheduling and Priority

The order of activation of processes within the concurrent statement is not significant, and the language does not specify any particular scheduling policy. However, a standard procedure, `priority`, is provided. A call has the form:

```
priority(p)
```

where `p` is an integer expression. The procedure sets the priority of the current process to the indicated value. The procedure may be called from any part of a program.

NOTES

1. An implementation is free to treat this procedure as a **`null`** statement.
2. The range of the argument may be constrained by an implementation.

3. The interpretation of the argument is implementation-dependent.
4. An implementation may assign default priorities to processes and the main statement part.

3.6. An Example: Multiple Update of a Shared Variable

The following program illustrates the declaration and activation of concurrent processes and the use of the initial and completion phases of the main program statement part. It also illustrates the problem of concurrent update of a shared variable: depending on the process scheduler in use, the output value may display variation.

```
program gardens1;

(* Multiple Update *)

var
     count: integer;

process turnstile1;

var
     loop: integer;
begin
     for loop := 1 to 20 do
          count := count + 1
end;  (* turnstile1 *)

process turnstile2;

var
     loop: integer;
begin
     for loop := 1 to 20 do
          count := count + 1
end;  (* turnstile2 *)

begin
     count := 0;
     cobegin
          turnstile1;
          turnstile2
     coend;
     writeln('Total admitted: ',count)
end.
```

3.7. Deadlock

The definition of "deadlock" in Pascal-FC is:

> Deadlock is a state in which no process is executable, and at least one process is "suspended".

MOTES

1. An implementation must abort the program if a state of deadlock occurs during its execution.
2. A program cannot deadlock as long as at least one process is "awaiting interrupt".
3. A program cannot deadlock as long as at least one process is "delayed".

4. SEMAPHORES

In this chapter, it is assumed that the reader is familiar with the concept of semaphores[6]. See, for example, Ben Ari[2, 3] for an introduction.

4.1. Declaration

Semaphore objects are introduced in **`var`** declarations. A standard type, `semaphore`, is included. Semaphores may be declared singly, or as components in arrays or records (types or variables).

NOTES

1. Semaphores, or objects containing them, may only be declared in a global declaration part.
2. Semaphore objects may be passed as parameters to subprograms, but the corresponding formal parameters must be **`var`** parameters.
3. Semaphores are guaranteed to have no processes blocked on them initially, but the value is undefined until the semaphore has been passed to the `initial` procedure.

4.2. Operations on Semaphores

The allowable operations on semaphores are restricted to:

- The `wait` and `signal` procedures;
- the `initial` procedure;
- the `write(ln)` procedure.

Each of `wait` and `signal` takes a single parameter, which must be a semaphore.

4.2.1. The `initial` procedure

A call to the `initial` procedure has the form:

```
initial(s,v)
```

where `s` is a semaphore and `v` is an integer expression.

NOTES

1. An implementation must not permit the execution of `initial` when `v` is less than zero.
2. Only the main program thread must be allowed to execute this procedure. Such a call can appear in the main statement part, or in the statement part of a subprogram called by the main program. (The implementation must not permit a *process* to execute `initial` by calling such a subprogram).

4.2.2. The `wait` procedure

A call to this procedure has the form:

```
wait(s)
```

where `s` is a semaphore.

NOTE

If the semaphore value is 0 at the time of execution, the calling process becomes "suspended" if the semaphore is not mapped to a source of interrupts (see Chapter 11) or enters the state "awaiting interrupt" if it is so mapped.

4.2.3. The `signal` procedure

A call to this procedure has the form:

```
signal(s)
```

where `s` is a semaphore.

NOTE

The language does not prescribe any particular queuing discipline on semaphores: if a `signal` is carried out on a semaphore on which several processes are currently "suspended", *one* of them will be allowed to proceed, but the decision as to which is arbitrary.

4.3. An Example: Multiple Update

Chapter 3 introduced a program that displayed the problem of concurrent update of a shared variable. The following program uses one semaphore to enforce mutually exclusive access to the variable. The final value in this case is always 40, regardless of the scheduler in use.

```
program gardens2;

(* Semaphore solution to multiple update *)

var
     count: integer;
     mutex: semaphore;
```

```
process turnstile1;

var
     loop: integer;
begin
     for loop := 1 to 20 do
          begin
          wait(mutex);
          count := count + 1;
          signal(mutex)
          end
end;  (* turnstile1 *)

process turnstile2;

var
     loop: integer;
begin
     for loop := 1 to 20 do
          begin
          wait(mutex);
          count := count + 1;
          signal(mutex)
          end
end;  (* turnstile2 *)

begin
     count := 0;
     initial(mutex,1);
     cobegin
          turnstile1;
          turnstile2
     coend;
     writeln('Total admitted: ',count)
end.
```

4.4. Process States and Transitions

This section summarises the effects on process state of the features described in this chapter.

1. A process executing a `wait` at a time when the named semaphore is zero becomes "suspended" if the semaphore is not mapped to a source of interrupts, or "awaiting interrupt" if it is so mapped.

2. A process suspended on a semaphore can be made executable by a `signal` operation on the semaphore or (in the case of a semaphore mapped to a source of

interupts) by the occurrence of an appropriate interrupt.

5. MONITORS

Pascal-FC's implementation of monitors follows closely the suggestions made by Hoare[8]. The only operations that are exported from monitors are procedures. thec ompiler guarantees mutually exclusive access to monitors, and condition synchronisation is done by means of `condition` variables, which are also described in this chapter.

5.1. Declaration

A monitor is one of the forms of declaration that are permitted only in a global declaration part. It has the following form:

```
monitor_declaration ::=

   monitor identifier;
      export_list
     [declaration_part]
  [monitor_body]
   end;

export_list ::=

   export procedure_identifier_list;
  {export procedure_identifier_list;}

monitor_body ::=

      begin
         statement_sequence
```

NOTES

1. Certain instances of type and variable declaration are not permitted in a monitor: specifically, those involving processes, semaphores and channels (types or variables).
2. The only declarations in a monitor that are visible from outside that monitor are procedures whose identifiers appear in the export list (these are called "exported procedures").
3. It is a compile-time error for such an identifier not to have a corresponding procedure declared in the monitor.
4. Exported procedures may not be nested within subprogram declarations.
5. The body of a monitor, if present, is executed once, before the first statement of the main statement part.
6. If there are several monitors declared in a program, the order in which their bodies are executed is not defined by the language.

7. Code within a monitor is guaranteed to be executed under mutual exclusion. A *boundary queue* is used to block processes wishing to gain access to a monitor already occupied by a process.
8. The boundary queue is defined to be a *priority queue*: within a given priority value, a FIFO discipline is used. The queuing scheme hence degenerates to plain FIFO in an implementation in which process priorities are not discriminated.

5.2. Calls to monitors

A monitor call is a call to an exported procedure of a monitor. In general, it takes the form:

`monitor_call ::=`

*monitor*_`identifier.`*exported_procedure*_`identifier`
`[(actual_parameters)]`

If the called procedure is declared within the same monitor as the call, a shorter form may be used as an alternative:

*exported_procedure*_`identifier[(actual_parameters)]`

This is semantically equivalent to the longer form. In particular, there is no attempt in either case to gain mutually exclusive access, since the calling process must already have such access.

Nested monitor calls are permitted: in this case, mutual exclusion on the monitor from which the call is made is retained.

5.3. Condition Variables

5.3.1. Declaration

Condition objects are introduced by **`var`** declarations. The standard type, `condition`, is provided. Condition variables may be declared, but there are no constants of this type. Conditions may be declared as simple variables, or as components of arrays or records (types or variables).

NOTES

1. Type declarations involving this type may be made in a global declaration part or in the declaration part of a monitor.
2. Declarations of *variables* involving this type may only be made in the declaration part of a monitor.
3. Conditions declared as formal parameters to subprograms must be **`var`** parameters.
4. Conditions are defined to be priority queues. Within a given priority value, a FIFO discipline is used. Hence, conditions degenerate to FIFO queues in implementations which do not discriminate process priority.

5. Conditions are guaranteed to be initialised to the empty queue on declaration.

5.3.2. Operations on conditions

The operations on conditions are restricted to:

- the `delay` and `resume` procedures;
- the `empty` function.

5.3.2.1. The `delay` procedure

A call to this procedure has the form:

```
delay(c)
```

where `c` is a condition. The calling process becomes "suspended" and releases mutual exclusion on the monitor.

5.3.2.2. The `resume` procedure

A call to this procedure has the form:

```
resume(c)
```

where `c` is a condition.

NOTES

1. If there are no processes currently suspended on `c`, the call has the same effect as a **`null`** statement.
2. If a `resume` operation unsuspends a process, the unsuspended process inherits mutual exclusion on the monitor. The process that called `resume` joins the *chivalry queue* associated with that monitor (there is one such queue per monitor). The queuing discipline on such a queue is the same as for the boundary queue.
3. Processes suspended on a chivalry queue have preference over any waiting on the same monitor's boundary queue when mutual exclusion of that monitor is released by another process.

5.3.2.3. The `empty` function

This is a `boolean` function. The single parameter must be a `condition`. The function returns `true` if there are no processes currently suspended on the named condition.

5.4. An Example: the Bounded Buffer

The following program illustrates the declaration and use of monitors and condition variables. It shows the communication of a producer and a consumer via a bounded buffer.

```
program PCON4;

(* producer-consumer problem  - monitor solution *)

monitor BUFFER;

export
     PUT, TAKE;

const
     BUFFMAX = 4;

var
     STORE: array[0..BUFFMAX] of char;
     COUNT: integer;
     NOTFULL, NOTEMPTY: condition;
     NEXTIN, NEXTOUT: integer;

procedure PUT(CH: char);

begin
     if COUNT > BUFFMAX then
          delay(NOTFULL);
     STORE[NEXTIN] := CH;
     COUNT := COUNT + 1;
     NEXTIN := (NEXTIN + 1) mod (BUFFMAX + 1);
     resume(NOTEMPTY)
end;  (* PUT *)

procedure TAKE(var CH: char);

begin
     if COUNT = 0 then
          delay(NOTEMPTY);
     CH := STORE[NEXTOUT];
     COUNT := COUNT - 1;
     NEXTOUT := (NEXTOUT + 1) mod (BUFFMAX + 1);
     resume(NOTFULL)
end;  (* TAKE *)

begin  (* body of BUFFER *)
     COUNT := 0;
     NEXTIN := 0;
     NEXTOUT := 0
end;  (* BUFFER *)
```

```
process PRODUCER;

var
     LOCAL: char;

begin
     for LOCAL := ’a’ to ’z’ do
          BUFFER.PUT(LOCAL);
end;  (* PRODUCER *)

process CONSUMER;

var
     CH: char;

begin
     repeat
          BUFFER.TAKE(CH);
          write(CH)
     until CH = ’z’;
     writeln
end;  (* CONSUMER *)

begin  (* main *)
     cobegin
          PRODUCER;
          CONSUMER
     coend
end.
```

5.5. Process States and Monitors

This section summarises the effects on process state of the features described in this chapter.

1. A process that attempts to enter a monitor that is already occupied becomes "suspended" on the monitor boundary queue.
2. A process that executes a `delay` becomes "suspended" on the named condition.
3. A process that executes a `resume` *that has the effect of unsuspending a process* becomes "suspended" on the monitor chivalry queue.
4. A process suspended on a condition can be made "executable" by the `resume` operation.
5. A process suspended on a monitor boundary queue can be made executable by a process leaving the monitor.

6. A process suspended on a chivalry queue can be made executable by a process leaving the monitor.

6. When a process leaves a monitor and other processes are blocked on both the boundary and the chivalry queues, the latter queue has priority.

6. RESOURCES

Resources in Pascal-FC provide some of the features of the **`protected record`** of Ada 9X[9]. Resources are similar to monitors in that they provide compiler-guaranteed mutual exclusion of access to enclosed data, but condition synchronisation is effected by *barriers* on procedures, rather than by condition variables.

6.1. Declaration

The rules concerning the place of declaration of resources are the same as those for monitors: in particular, a resource may only be declared in a global declaration part. The syntax of a resource declaration is as follows:

```
resource_declaration ::=

   resource identifier;
      export_list
      resource_declaration_part
  [resource_body]
   end;

resource_declaration_part ::=

   {
   constant_declaration
 | type_declaration
 | variable_declaration
 | procedure_declaration
 | function_declaration
 | guarded_procedure_declaration
   }

guarded_procedure_declaration ::=

   full_guarded_procedure_declaration
 | deferred_guarded_procedure_declaration

full_guarded_procedure_declaration  ::=

   guarded procedure identifier[formal_part]
      when boolean_expression;
     [declaration_part]
   begin
      statement_sequence
   end;
```

```
deferred_guarded_procedure_declaration  ::=

   forward_guarded_procedure_header
   guarded_procedure_body


forward_guarded_procedure_header  ::=

   guarded procedure identifier[formal_part]
      when boolean_expression;forward;

guarded_procedure_body ::=

   guarded procedure identifier;
     [declaration_part]
   begin
      statement_sequence
   end;


resource_body ::=

   begin
      statement_sequence
```

NOTES

1. Resources may only be declared in a global declaration part.
2. Only identifiers that appear in the export list are in scope from outside the resource.
3. The identifiers in the export list must be the names of procedures or guarded procedures declared inside the resource.
4. Exported procedures must not be nested within other subprograms declared in the resource.
5. Guarded procedures, whether exported or not, must not be nested inside other subprograms.
6. The formal parameters of a guarded procedure are not in scope until *after* the guard expression.
7. Type and variable declarations involving semaphores, channels and processes are not permitted anywhere in a resource.
8. Where a deferred guarded procedure declaration is used, the header and the body may be separated by other declarations.

6.2. Calls to resources

A call to a resource is a call to an exported procedure (guarded or otherwise) of that resource. It has the form:

```
resource_call ::=

    resource_identifier.
          exported_procedure_identifier[(actual_parameters)]
```

As with monitors, if the procedure concerned is declared within the current resource, a shortened notation (consisting simply of the procedure identifier and the actual parameters) is permitted. This is semantically equivalent to the longer version given above.

If a guarded procedure is called whose guard evaluates to false, the calling process becomes "suspended" and leaves the resource. Any process leaving a resource must attempt to find a candidate which will inherit mutual exclusion on the resource. The candidate will be selected from among those suspended on guards which now evaluate to true. No particular queuing discipline is specified for guards, and the choice among the procedures with open guards is arbitrary. Processes leave resources when they complete an execution path through an exported procedure, when they become blocked on a guard in that resource, or when they requeue to a different resource (whether or not they block on a guard).

Nested calls from one resource to an exported procedure of another are permitted. In such a case, mutual exclusion on the current resource is retained.

NOTE

A call to a guarded procedure is not permitted anywhere inside a resource.

6.3. The requeue statement

The **requeue** is used to abandon the current guarded procedure and transfer the call to another, either within the current resource, or within another. Its syntax is:

```
requeue_statement ::=

    requeue [resource_identifier.]
          guarded_procedure_identifier[(actual_parameters)]
```

The optional identifier, if present, must be the name of a resource. If it is the name of the resource enclosing the **requeue** statement, it has no effect. The second identifier in the above syntax must be the name of a **guarded procedure**. If the associated guard is open, the procedure is executed as normal. If the guard is closed, the calling process must become "suspended" (and leave the resource), but it must attempt to find a candidate for awakening.

Control does *not* eventually return to the statement following a **requeue** statement: its execution causes the *abandonment* of the current guarded procedure.

NOTES

1. A **`requeue`** statement may only be used in a the statement part of a **`guarded procedure`** (ie, not within a subprogram nested within such a procedure).
2. The destination of a **`requeue`** must be a *guarded* procedure.
3. In the event that a **`requeue`** is made to a guarded procedure of another resource, mutual exclusion on the current resource is released, as a return will not be made to it.

6.4. An Example: the Alarm Clock

The following is a resource-based version of Hoare's[8] alarm clock example. A number of sleeper processes wish to slumber for various periods. Each time that the clock ticks, sleepers awake and check to see whether it is time to get up. If not, they go back to sleep. The example illustrates the use of the **`requeue`** statement.

```
program ALARMCLOCK;
const
  PMAX = 3;

resource ALARM;
  export
    SLUMBER, TICK;

  var
    NOW: integer;
    queue : integer;      (* takes values 1 or 2 *)
    freed1, freed2 : boolean;

  guarded procedure SLUMBER2(AL: integer) when freed2; forward;

  guarded procedure SLUMBER1(AL: integer) when freed1;
  begin
    if NOW < AL then
        requeue SLUMBER2(AL)
  end;  (* SLUMBER *)

  guarded procedure SLUMBER2;
  begin
    if NOW < AL then
        requeue SLUMBER1(AL)
  end;  (* SLUMBER *)
```

```
  guarded procedure SLUMBER(N: integer) when true;
  var
    ALARMCALL: integer;
  begin
    ALARMCALL := NOW + N;
    if NOW < ALARMCALL then
      if queue = 1 then
        requeue SLUMBER1(ALARMCALL)
      else
        requeue SLUMBER2(ALARMCALL)
  end;  (* SLUMBER *)

  procedure TICK;
  begin
    NOW := NOW + 1;
    if queue = 1 then
    begin
      queue := 2;
      freed1 := true;
      freed2 := false
    end else
    begin
      queue := 1;
      freed1 := false;
      freed2 := true
    end
  end;  (* TICK *)

begin  (* body *)
  NOW := 0;
  queue := 1;
  freed1 := false
end;  (* ALARM *)

resource SCREEN;
  export
    PRINT;

  procedure PRINT(N: integer);
  begin
    writeln('Process ',N:1,' awakes')
  end;  (* PRINT *)
end;  (* SCREEN *)
```

```
process DRIVER;
  (* provides the clock "ticks" *)
begin
  repeat
    sleep(1);
    ALARM.TICK
  forever
end;  (* DRIVER *)

process type SLEEPERTYPE(N: integer);
begin
  repeat
    ALARM.SLUMBER(n);
    SCREEN.PRINT(N)
    (* get up and go to work *)
  forever
end;  (* SLEEPERTYPE *)

var
  SLEEPERS:  array[1..PMAX] of SLEEPERTYPE;
  PLOOP: integer;

begin
  cobegin
    DRIVER;
    for PLOOP := 1 to PMAX do
      SLEEPERS[PLOOP](PLOOP)
  coend
end.
```

6.5. Process States and Transitions

This section summarises the effects on process states of the features described in this chapter.

1. A process that attempts to enter a resource that is already occupied becomes "suspended" at the resource boundary.
2. A process that calls a guarded procedure whose guard expression evaluates to `false` becomes "suspended" on the barrier of that procedure.
3. A process suspended on a procedure barrier may become executable when a process leaves the resource and the guard expression evaluates to `true`.
4. A process suspended on a resource boundary may become executable when a process leaves the resource and there is no process suspended on a guarded procedure of the resource whose guard evaluates to `true`

7. RENDEZVOUS BY CHANNEL

Semaphores, monitors and resources provide forms of inter-process communication based on *shared memory*. An alternative model for such communications is based on *message-passing*. Pascal-FC includes two forms of message-passing protocol. In both cases, communicating processes take part in a *rendezvous*. The essence of a rendezvous is that the process which arrives first is suspended until the other party arrives. When the rendezvous is complete, the processes resume their separate execution.

In this chapter, we introduce the first of these schemes, which is similar to the model used in the language, occam 2[10]. In this scheme, messages are sent between processes by means of *channels*.

7.1. Channels

A channel is the intermediary for a rendezvous between *one* sender and *one* receiver. Channels are strongly typed: the declaration of a channel variable or channel type specifies a *base type*, and only data of this type may be sent or received via such a channel. The base type of a channel may be any of the types allowed in Pascal-FC (including structured types). The effects of sending and receiving channels, processes, semaphores and conditions are not defined by the language.

7.1.1. Declaration and use of channels

Channel objects are introduced by **var** declarations. A special form of type declaration has been introduced for this purpose, whose syntax is as follows:

```
channel_type  ::=

    channel of type
```

NOTES

1. Channel types and objects can only be declared in a global declaration part.
2. Channel objects may be passed as parameters to subprograms, but they must have been formally declared as **var** parameters.

7.1.1.1. Examples

Typical declarations involving channel types and objects are:

```
type
   chan = channel of integer;
   buffer = array [1..buffersize] of char;
   link = channel of buffer;

var
   ch1, ch2: chan;
   coms: array [1..10] of link;
```

7.1.2. Operations on channels

Two channel operations, *send* and *receive*, are defined. The syntax is as follows:

```
channel_operation  ::=

   send | receive

send  ::=

   channel_variable  !  expression

receive  ::=

   channel_variable  ?  variable
```

NOTES

1. Channels are strongly typed. In both send and receive operations, the right-hand operand must have an equivalent type to the base type for the left-hand operand.
2. The effects of sending and receiving processes, semaphores, conditions and channels are not defined.
3. It is a run-time error to have multiple senders or multiple receivers concurrently active on a given channel.

7.2. The type `synchronous`

For synchronisation-only communication, where it is not required to pass data from one process to another, a data type, `synchronous` has been introduced. This identifier may be used in type and object declarations. A pre-defined variable, `any`, is also implicitly declared in every Pascal-FC program.

Objects of this type have no values associated with them. The only allowable operations are the two channel operations. These are defined to have no effects on their `synchronous` operands. Hence, although the programmer can declare variables of this type, it is unnecessary to do so, as `any` can be used on the right-hand side of all channel operations involving `synchronous` channels.

7.3. An Example: Unbuffered Producer-Consumer

Because the send and receive operations effect a rendezvous, simple process synchronisation is straightforward. In the following program, a producer generates characters which are then consumed by a second process.

```
program procon1;

var link: channel of char;


process producer;

var local: char;

begin
     repeat
          (* generate character *)
          link ! local
     forever
end;  (* producer *)

process consumer;

var local: char;

begin
     repeat
          link ? local;
          (* consume character *)
     forever
end;  (* consumer *)


begin  (* main *)
     cobegin
          producer;
          consumer
     coend
end.
```

7.4. Process States and Transitions

This section summarises the effects on process states of the features described in this chapter.

1. A process that attempts a send or receive operation on a channel on which there is no pending call becomes "suspended" if the channel is not mapped to a source of

interrupts (see Chapter 11) and "awaiting interrupt" if it is so mapped.

2. A process blocked on a channel may become executable when another process has carried out the complementary channel operation on the channel or (in the case of a channel mapped to a source of interrupts) when an appropriate interrupt occurs.

8. ADA-STYLE RENDEZVOUS

A subset of the Ada inter-process communication facilities has been implemented in Pascal-FC. These include the provision of entries (but not entry families), entry calls (not conditional or timed) and **`accept`** statements. These features provide for a basic Ada-style rendezvous. A selective waiting construct is also provided, but this is described in Chapter 9.

8.1. Process Entries

The syntax of an entry declaration is as follows:

```
entry_declaration ::=

    entry identifier [formal_part] [mapping_indicator];
```

The formal parameters have the same form as for a subprogram and may, therefore, include value and **`var`** parameters. Whereas value parameters can only be used to pass information into the called process, **`var`** parameters may be used to pass information either way.

The significance of the mapping indicator will be considered in Chapter 11.

NOTE

An arbitrary number of processes may at any time be suspended on an entry. The language does not define any particular queuing discipline on entries.

8.2. The accept statement

The form of the **`accept`** statement is:

```
accept_statement ::=

    accept entry_identifier [formal_part] do
       statement
```

NOTES

1. An **`accept`** statement may only be placed in the statement part of a process (ie, not within a subprogram nested in a process).
2. The formal part must correspond exactly with the one given for that entry in the entry declaration: the number, type and mode of the parameters, and their identifiers, must match.
3. An **`accept`** statement opens a new scope in the same way as a subprogram, and the formal parameters are only visible within the enclosed statement.
4. An **`accept`** statement for an entry `E1` may be nested inside an **`accept`** for an entry `E2`, but not within an **`accept`** for `E1` itself (either directly or indirectly).

8.3. Entry calls

The form for a call on a process entry is:

```
entry_call  ::=

    process_variable.entry_identifier [(actual_parameters)]
```

NOTES

1. The actual parameters must be compatible in number, type and mode with the formal parameters declared for that entry.
2. It is a run-time error to make an entry call on a process that was never made executable, or has already terminated.

8.4. Use of process provides declaration

It is convenient to consider this feature here, because it was introduced to facilitate certain forms of process interaction when using the Ada-style rendezvous.

There are applications when an instance of a process type wishes to rendezvous with another instance of the same type. The processes may, for example, be elements in a "pipeline". In outline, the process declaration might be as follows:

```
process type p(pn: integer);

entry e1(n: integer);

begin
     ..

     (* call entry e1 of instance pn+1 *)

     ..
end;
```

The syntax for an entry call demands a process *variable* identifier, not a *type* identifier. Hence, in order to code the entry call in the statement part of the process type, variables of the type must already have been declared at this point. Until we have declared the process type, however, we cannot declare any process variables of that type. The circularity is resolved by use of the **`provides`** declaration, whose syntax was given in Chapter 3. The purpose of this form of declaration is to pre-declare the interface that the process has with other parts of the program: this consists of any parameters and entries. Once such a declaration has been made, variables of the type can be declared. Hence, in outline, the above requirements can be met by the following:

```
process type p(pn: integer) provides
   entry e1(n: integer);
end;

var
     elements: array[1..pipelength] of p;

process type p(pn: integer);

   entry e1(n: integer);

begin
     ..

     elements[pn+1].e1(k)

     ..

end;
```

8.5. Process States and Transitions

This section summarises the effects on process states of the features described in this chapter.

1. A process that attempts an **`accept`** for an entry on which there are no pending calls becomes "suspended" if the entry is not mapped to a source of interrupts (see Chapter 11) or "awaiting interrupt" if it is so mapped.
2. A process blocked at an **`accept`** statement may become executable when another process makes a call on that entry, or (in the case of an entry mapped to a source of interrupts) when an appropriate interrupt occurs.
3. A process that attempts an entry call on an entry for which there is no pending **`accept`** becomes "suspended" on that entry.
4. A process that makes a call on an entry for which there is a pending **`accept`** becomes "suspended" on that entry following the transfer of parameters to the called process and the unblocking of the process suspended at the **`accept`**.
5. A process suspended at an entry call may become executable following the completion of an **`accept`** statement for that entry. (including the transfer of **`var`** parameters back to the caller)

9. SELECTIVE WAITING

Selective waiting in Pascal-FC is accomplished by use of the **select** statement, which is similar in many ways to the Ada **select**. This structutre is not restricted to the Ada-style of inter-process communication, however: it can also be used with channel alternatives.

9.1. The select statement

The syntax of the **select** statement is given below.

```
select_statement ::=

  [pri] select
      select_alternative
     {;or select_alternative}
  [else_part]
   end

select_alternative ::=

   channel_alternative
 | replicate_alternative
 | accept_alternative
 | timeout_alternative
 | terminate

channel_alternative ::=

  [guard]
   channel_operation
  [;statement_sequence]


guard ::=

   when boolean_expression =>


replicate_alternative  ::=

   for variable := expression to expression replicate
      channel_alternative
```

```
accept_alternative  ::=

  [guard]
   accept_statement
  [;statement_sequence]

timeout_alternative  ::=

  [guard]
   timeout integer_expression
  [;statement_sequence]

else_part ::=

   else statement_sequence
```

NOTES

1. A **select** statement containing accept alternatives can only be used in the statement part of a process (not in a subprogram nested within a process).
2. The t**timeout**, **terminate** and **else** parts are mutually exclusive. It is a compile-time error to mix these alternatives within a single **select** statement.
3. The variable in the **replicate** alternative (known as the "replicator index") and the two expressions must all be of equivalent ordinal types.
4. The value of the replicator index may be accessed anywhere in the replicate alternative. This type of alternative is intended to be used in association with arrays of channels. The index value is determined at run-time by the index of the channel with which a rendezvous is selected for that execution of the **select**, but it has no defined value on completion of the **select** statement.

9.2. Notes on the Semantics of the select Statement

9.2.1. Indivisibility

The execution of the **select** statement begins with the evaluation of guards. This phase is *not* indivisible. There follows an indivisible phase during which all channels or entries with open guards are checked for pending calls.

9.2.2. Order of checking for pending calls

The **select** statement exists in two basic forms: the **pri** and the "plain" forms. In the **pri** form, the order in which open alternatives are checked follows their *textual* order. In the "plain" form, the implementation may use *any* convenient order (including textual order).

9.2.3. Execution of select with all guards closed

It is a run-time error to attempt to execute a **`select`** statement in which there is no alternative with an open guard, unless there is an **`else`** part. Alternatives which have no guard (including all **`terminate`** alternatives) are considered always open.

9.2.4. The else part

The statements of the **`else`** part are executed if either of two conditions is satisfied:

(1) there are no open guards;

(2) there is at least one open guard, but none of the associated channels or entries has a pending call.

9.2.5. The terminate alternative

A process which becomes suspended on a **`select`** statement with a **`terminate`** alternative enters a special state ("termstate"). The process may become executable again if a call is made on one of the open alternatives. On the other hand, the process may make a direct transition to "terminated" if all other processes in the program are either terminated or themselves in termstate.

9.3. Examples

9.3.1. The select statement with channel alternatives

The following outline illustrates the use of the **`select`** statement with channels and the **`replicate`** alternative. It demonstrates the use of a server process (`SCREEN`) to enforce mutually exclusive access to the terminal screen.

```
program screenchan;

(*

Mutual exclusion using channels

*)

const
  max = 5;
type
  link = channel of synchronous;
var
  coms: array[1..max] of link;

process type clienttype(n: integer);
begin
    coms[n] ! any
end;
```

```
var
  clients:  array[1..max] of clienttype;

process screen;
var
  i: integer;
begin
  repeat
    select
      for i := 1 to max replicate
        coms[i] ? any;
        writeln('Message from process ',i);
    or
      terminate
    end
  forever
end;

var
  i: integer;

begin
  cobegin
    screen;
    for i := 1 to max do
      clients[i](i)
  coend
end.
```

9.3.2. The select statement with accept alternatives

The following is a solution to the bounded buffer problem using the Ada style of inter-process communication.

```
program pcon5;

(* buffered producer-consumer with ada rendezvous *)

process buffer;
  entry take(var ch: char);
  entry put(ch: char);

const
     buffmax = 4;
```

```
var
     store: array[0..buffmax] of char;
     nextin, nextout, count: integer;

begin
     nextin := 0;
     nextout := 0;
     count := 0;
     repeat
          select
          when count <> 0 =>
               accept take(var ch: char) do
                    ch := store[nextout];
               count := count - 1;
               nextout := (nextout + 1) mod (buffmax + 1);
          or
          when count <= buffmax =>
               accept put(ch: char)  do
                    store[nextin] := ch;
               count := count + 1;
               nextin := (nextin + 1) mod (buffmax + 1 );
          or
               terminate
          end  (* select *)
     forever
end;  (* buffer *)

process producer;
var
     local: char;
begin
     for local := 'a' to 'z' do
          buffer.put(local)
end;  (* producer *)
```

```
process consumer;
var
     local: char;
begin
     repeat
          buffer.take(local);
          write(local)
     until local = 'z';
     writeln
end;  (* consumer *)

begin
     cobegin
          producer;
          consumer;
          buffer
     coend
end.
```

9.4. Process States and Transitions

This section summarises the effects on process states of the features described in this chapter.

1. A process that attempts to execute a **`select`** on which there are no open alternatives with pending calls becomes blocked unless there is an **`else`** part. (In the special case that there are no open guards and no **`else`** part, a run-time error must be signalled).
2. A process that becomes blocked on a **`select`** with a **`terminate`** alternative enters the "termstate" state. It may return to the "executable" state if a call occurs on an open alternative or (in the case of a ,channel or entry mapped to a source of interrupts) when an appropriate interrupt occurs. It will prroceed directly to the "terminated" state if the run-time system detects that all processes are in "termstate" or are already "terminated".
3. A process that becomes blocked on a **`select`** with a **`timeout`** alternative is considered "delayed". It may become executable when the specified time has elapsed, or when a call occurs on an open alternative, or (in the case of a channel or entry mapped to a source of interrupts) when an appropriate interrupt occurs., whichever of these events occurs first.
4. A process that becomes blocked on a **`select`** with neither **`terminate`** nor **`timeout`** alternatives becomes "suspended" if none of the open-guarded alternatives is mapped to a source of interrupts, or "awaiting interrupt" if one or more such alternatives is so mapped.

10. TIMING FACILITIES

The timing facilities defined as part of Pascal-FC are primarily intended for implementations designed for real-time programming. However, some elementary implementation of these facilities will be provided by all versions of Pascal-FC. As this is one of the areas where there are important implementation dependencies, the relevant User Guide should be consulted.

10.1. The system clock

The timing facilities depend on a system clock, which will be provided as part of the Pascal-FC run-time environment. All timings in Pascal-FC programs are expressed in *system clock units.* The duration of a system clock unit is implementation-dependent, and need not necessarily be a constant real-time unit.

10.2. Outline of timing facilities

There are three timing facilities:

1. A standard function, `clock`, which can be used to examine the current system time in clock units;
2. A standard procedure, `sleep`, which can be used to delay a process for a specified number of clock units;
3. A **`timeout`** alternative to the **`select`** statement.

10.2.1. The `clock` function

This is a function of no arguments, returning an integer result. This represents the number of system clock units elapsed since some arbitrary zero (not necessarily the start of execution of the current program).

10.2.2. The `sleep` procedure

This procedure takes a single integer expression as an argument, which is the number of clock units for which the calling process should be delayed (the `sleep` procedure may also be called by the main program thread, which is not strictly a process in Pascal-FC).

NOTES

1. There is no guarantee that the process will indeed be suspended for *exactly* the time specified: the process should become "executable" when the period elapses, but there may not be a free processor.
2. If a negative or zero value is given as the argument to `sleep,` the calling process can be considered to make an instantaneous transition from "executable" to "delayed" and back to "executable". The scheduler must be invoked when such a call is executed.

10.2.3. The timeout alternative to the select statement

One or more of the alternatives in a **`select`** statement may be **`timeout`** alternatives. If no calls arrive on any of the open entries or channels of the **`select`** before the period specified has expired, the **`timeout`** alternative becomes active (provided that it has an open guard), and any statements following it are executed. (If there are no statements, then the **`select`** statement is exited).

It is possible for a **`select`** statement to have several **`timeout`** alternatives with open guards. In such a case, the one with the *smallest* specified period will be the effective one. If several such alternatives become simultaneously due, the language does not specify which becomes effective.

Negative and zero values may be specified. The same remarks apply as for negative and zero arguments to the `sleep` procedure.

10.3. An example using `sleep` and timeout

The following program should be self-explanatory.

```
program sleeptest2;

(* illustrates timeout alternative to select *)

var
  coms: channel of synchronous;

process q;
var
  off: boolean;
begin
     off := false;
     repeat
          select
               coms ? any;
               writeln('received');
          or
               timeout 20;
               off := true;
               writeln('timed out')
          end
     until off
end;
```

```
process p;
var
  count: integer;
begin
     count := 0;
     repeat
               sleep(10);
               count := count + 1;
               write('sent  ');
               coms ! any
     until count = 10;
end;

begin
     cobegin
          q; p
     coend
end.
```

The following exemplifies the output from this program.

```
sent   received
sent   received
sent   received
sent   received
sent   received
sent   received
sent   received
sent   received
sent   received
sent   received
timed out
```

10.4. Process states, deadlock and the timing facilities

A process that becomes "delayed" by executing `sleep` or by becoming blocked on a **`select`** with a **`timeout`** alternative will eventually become "executable" again (when the time has elapsed). The scheduler, therefore, *must not* indicate that a program has become deadlocked as long as there is at least one "delayed" process.

11. LOW-LEVEL FACILITIES

The facilities described in this chapter are designed for implementations intended for real-time programming. Their purpose is to enable hardware device-drivers to be written, including the manipulation of I/O device registers and interrupt-handling. The relevant features are:

- The type, `bitset`, which is a useful type for modelling I/O device Control and Status Registers (CSRs).
- Record offset indicators, which are useful in modelling multi-register I/O devices.
- Mapping indicators, which are used to identify I/O devices and for interrupt-handling.

11.1. The type `bitset`

The values of this type are sets of `0 .. (n - 1)`, where the value of `n` is implementation-dependent. The *operators* for this type are the set operators familiar in standard Pascal. These implement union, intersection, set difference, and a test for set membership, as well as the assignment and relational operators.

11.1.1. Declaration

As `bitset` has been introduced as a standard type, objects of this type are introduced by **`var`** declarations. The following declaration illustrates the use of the type `bitset` to represent a device CSR.

```
var
      incsr: bitset;
```

11.1.2. Assigning values

A bitset literal can be used in expressions involving this type. Suppose, for example, that Bit 6 of the `incsr` declared above is the "interrupt enable" bit. The following assignment would set this bit to `1`, with all other bits cleared to zero:

```
incsr := [6]
```

The *empty set* notation can be used to clear all bits, as follows:

```
incsr := []
```

The set literal notation can be used to set several bits in a single statement. For example, to set bits 6,4,3,1 and 0, leaving the remainder cleared to zero:

```
incsr := [6,4,3,1,0]
```

If a large number of bits is to be set, the set literal notation can be tedious. Hence, Pascal-FC provides a shorthand in the form of a type transfer function, `bits`, which is described below in the section on type transfer functions.

11.1.3. Set union operator

In the following example, we wish to set bits 6,4,3,1 and 0, leaving others unaffected (which is not necessarily the same as leaving them cleared to zero). The following statement would achieve this:

```
incsr := incsr + [6,4,3,1,0]
```

Here, the *set union* operator ("+") has been used. In effect, it performs a bitwise *or* operation on its two operands.

11.1.4. Set intersection operator

A common requirement is to examine the state of several bits in a device register, ignoring the states of the others. For example, the following statement copies the least significant four bits of `incsr` into `temp.`

```
temp := incsr * [3,2,1,0]
```

The set intersection operator ("*") in effect performs a bitwise *and* operation on its two operands.

11.1.5. Set difference operator

In the following example, we wish to turn Bits 3 and 6 off, leaving the remainder unaffected. The set difference operator ("-") is used for this purpose, as follows:

```
incsr := incsr - [3,6]
```

11.1.6. Testing set membership

The state of an individual bit can be tested using the **`in`** operator. An expression involving this operator has the following form:

integer`_expression` **`in`** *bitset*`_expression`

The value of such an expression is of `boolean` type.

For example, we can test the value of Bit 7 and take appropriate action in the following way:

```
if 7 in incsr then
        action when Bit 7 is 1
else
        action when Bit 7 is 0
```

11.1.7. Relational operators and the type bitset

All the relational operators can be used with the `bitset` type. Operators such as ">" are used to test for inclusion of one `bitset` in another.

11.1.8. Type transfer functions

Two functions are provided for type transfer between bitsets and integers. The function `bits` maps integers to bitsets. As previously noted, this can be a useful shorthand notation when assigning values to bitsets. To set Bits 0 to 7, for example, we may write:

```
bs := bits(16#ff)
```

where `bs` is a bitset variable. The inverse function, `int`, maps from bitsets to integers. Hence:

```
i := int(bs)
```

where `i` is an integer variable.

NOTES

1. Because the number of bits in a bitset is implementation-dependent, the set of integers involved in these mapping functions will also vary between implementations.
2. The mapping between bitsets and decimal integers is implementation-dependent.
3. An expression of type `bitset` may not appear as a parameter to the `read` or `readln` procedures, but it *may* appear as a parameter to `write` or `writeln`. The format in which the set value is output is implementation-dependent.

11.2. Addressing Device Registers with Mapping Indicators

One of the applications of mapping indicators is to permit device registers to be modelled as variables in the Pascal-FC program. Such registers may then be modified and read by using assignment statements.

Suppose that a device-driver is required for a terminal. Consider the following declaration:

```
var
     inbuff: char;
```

The variable, `inbuff` will be placed by the compiler somewhere in the machine's memory map, but the programmer does not know where. If `inbuff` is a hardware register, the compiler must be forced to map the variable to a *specific* place. This is one of the applications of mapping indicators.

The syntax of a variable declaration in Pascal-FC Ci is:

```
variable_declaration  ::=

   var

   variable_list : type;
  {variable_list : type;}
```

```
variable_list  ::=

   identifier [mapping_indicator]
  {,identifier [mapping_indicator]}

mapping_indicator  ::=

   at integer_constant
```

The mapping indicator provides the necessary information for the compiler to carry out the mapping. The interpretation of this indicator is implementation-dependent: it may, for example, be a port address, or it may be a memory address in a system using memory-mapped I/O. An implementation is free to impose any restrictions on the use of such indicators, and is permitted to ignore them. The relevant User Guide will provide information on any such restrictions.

If the input character buffer for the terminal controller was located at hexadecimal 800001, then the following declaration could be written in Pascal-FC:

```
var
     inbuff at 16#800001: char;
```

References to `inbuff` would then read the character currently held in the input buffer.

The terminal controller output character buffer may reside at hexadecimal 800003. The following declaration expresses this:

```
var
     outbuff at 16#800003: char;
```

Assignments, such as:

```
outbuff := ’a’
```

can then be made.

In the event that the variable which has a mapping indicator is an array or record, the indicator specifies the *base* of the structure.

11.3. Use of Record Offset Indicators

Consider again the terminal controller used in previous examples. In addition to the two character buffers, let us suppose that there are also two control and status registers (one for input and one for output). Individual bits in the control and status registers have particular significance defined by the hardware design. As there is a requirement to manipulate individual bits, the control and status registers will be represented as bitsets.

There may be several different terminal controllers in a system, each with the same form, ie:

```
input csr at base address + 0
input character buffer at base address + 1
output csr at base address + 2
output character buffer at base address + 3
```

A suitable Pascal data structure for such an object would be a record. A record type could be declared as follows:

```
type slu =
record
   incsr: bitset;
   inbuff: char;
   outcsr: bitset;
   outbuff: char
end;
```

Individual instances could then be declared, and mapped onto the appropriate physical addresses, as follows:

```
var
     term1 at 16#800000, term2 at 16#800010: slu;
```

This example assumes that bitsets and characters occupy a single storage unit in the implementation concerned.

The requirements are often somewhat more complex. In the Motorola 68000, for example, the registers in a peripheral controller are likely to occupy odd addresses. The above type declaration would not be suitable for such a case, because the fields would be mapped onto *consecutive* addresses (ie, some odd and some even). Moreover, modern peripheral controller devices are often complex and contain a large number of registers. Only a small subset of these registers, located at widely differing offsets from the base address of the device, may be of interest in a particular application. Some facility is required to indicate that the record fields are not to be mapped to consecutive addresses, but to any arbitrary offsets, In Pascal-FC, an offset indicator may be included in the declaration of a record field to cater for this.

For example, the following would cater for the 68000-style device:

```
type slu =
record
   incsr at offset 1: bitset;
   inbuff at offset 3: char;
   outcsr at offset 5: bitset;
   outcsr at offset 7: char
end;
```

11.4. Interrupts

Three of the inter-process communication primitives (semaphores, channels and process entries) provided in Pascal-FC may be mapped on to the target machine’s interrupts. In each case, the target hardware is analogous to an implicit process which communicates with the software process which the programmer has written. The interpretation of the information is implementation-dependent. It may, for example, indicate an interrupt vector, but this is not a required interpretation.

11.4.1. Mapping semaphores to interrupt sources

In this case, a mapping indicator is used in the semaphore declaration, as follows:

```
var
     timsem at 64: semaphore;
```

where the supplied constant specifies the source of the interrupt. The software process must execute the semaphore `wait` operation. The hardware will, in effect, perform the corresponding `signal` operation when an interrupt is generated by the specified source.

11.4.1.1. Program example

The following is a simple example. Suppose that a hardware timer has been programmed to generate interrupts through vector 64 at the rate of one per second. On receiving an interrupt, the timer driver process outputs the current value of the counter.

```
program ticks1;

(* produce 1-second ticks using timer device *)

(* semaphore version *)

const
      intvec=64;



var
     rtclock at 16#800021: timregs;  (* a suitable record type *)
     timsem at intvec: semaphore;

procedure initialise;

(* set up timer to interrupt at 1Hz *)

begin
     suitable initialisation code
end;  (* initialise *)
```

```
process timer;


var
     local: integer;

begin
          local := 0;
          repeat
                    wait(timsem);
                    clear interrupt condition;
                    local := local + 1;
                    writeln(local)
          until local = 10;
     stop clock
end;  (* timer *)



begin  (* main *)
     initial(timsem,0);
     initialise;
     cobegin
          timer
     coend
end.
```

11.4.2. Mapping channels to interrupt sources

Mapping a channel onto an interrupt source is again accomplished by using a mapping indicator in a variable declaration. The software process which is intended to respond to the interrupt is then written to make a rendezvous on the channel concerned. The "other party" in the rendezvous is the hardware.

NOTES

1. The implementation may treat all such rendezvous as though the channel involved were of type `synchronous`, but this is not a requirement.
2. The language does not specify whether interrupt sources must be "senders" or "receivers".

11.4.2.1. Program example

The following is a channel version of the program used to illustrate the mapping of semaphores on to interrupts:

```
program ticks3;

(* produce 1-second ticks using timer device *)

(* channel version *)

const
      intvec=64;


var
     rtclock at 16#800021: timregs;  (* a suitable record type *)
     timchan at intvec: channel of synchronous;

procedure initialise;

(* set timer interrupt vector and preset *)

begin
     suitable initialisation code
end;  (* initialise *)


process timer;


var
     local: integer;

begin
          local := 0;
          repeat
                    timchan ? any;
                    clear interrupt condition;
                    local := local + 1;
                    writeln(local)
          until local = 10;
     stop clock
end;  (* timer *)
```

```
begin  (* main *)
     initialise;
     cobegin
          timer
     coend
end.
```

11.4.3. Mapping process entries to interrupt sources

In this case, a mapping indicator is used in the entry declaration. In effect, a rendezvous takes place between the software process which possesses the entry, and the hardware, which makes a call to the entry.

NOTE

The use of any parameters to the entry is implementation-dependent.

11.4.3.1. Program example

The following is another version of the program used hitherto for illustration:

```
program ticks3;

(* produce 1-second ticks using timer device *)

(* ada version *)

const
      intvec=64;


var
     rtclock at 16#800021: timregs;  (* a suitable record type *)
procedure initialise;

(* set timer interrupt vector and preset *)

begin
     suitable initialisation code
end;  (* initialise *)
```

```
process timer;

entry interrupt at intvec;


var
     local: integer;

begin
          local := 0;
          repeat
                    accept interrupt do
                         clear interrupt condition;
                    local := local + 1;
                    writeln(local)
          until local = 10;
     stop clock
end;  (* timer *)



begin  (* main *)
     initialise;
     cobegin
          timer
     coend
end.
```

11.5. Interrupts and process states

Processes that are blocked on semaphores, channels or entries are considered "awaiting interrupt" (unless they are considered "delayed"). Hence, the run-time system must not indicate deadlock as long as there is at least one process so blocked.

APPENDIX A - CHARACTER SET

The language does not define the set of characters that can appear in character or string literals, but (apart from those contexts), the character set consists of the following:

- The letters A to Z and a to z.
- The decimal digits 0 to 9.
- The space and horizontal tabulation character.
- An end-of-line marker, which is implementation-dependent.
- The following symbols:

```
( ) [ ] { } + - * / : ; , .  < > = ! ? # % '
```

APPENDIX B - RESERVED WORDS

accept	**and**	**array**
at	**begin**	**case**
channel	**cobegin**	**coend**
const	**div**	**do**
else	**end**	**entry**
export	**for**	**forever**
forward	**function**	**guarded**
if	**in**	**mod**
monitor	**not**	**null**
of	**offset**	**or**
pri	**procedure**	**process**
program	**provides**	**record**
repeat	**replicate**	**requeue**
resource	**select**	**terminate**
then	**timeout**	**to**
type	**until**	**var**
when	**while**	

APPENDIX C - PRE-DEFINED DATA TYPES

The following type identifiers are pre-defined:

```
char      integer   real        boolean
semaphore condition synchronous bitset
```

Semaphore, condition, synchronous and bitset types were considered respectively in Chapters 4, 5, 7 and 11. This Appendix provides a brief introduction to the first four types for readers not familiar with Pascal.

1. The type `char`

Objects and constant of type `char` may contain a single character value.

1.1. Set of values

The set of values for objects of this type is not defined by the language.

1.2. Operators

The set of permissible operators is:

1. Assignment.
2. Certain pre-defined subprograms (see Appendix D).
3. The relational operators:

   `< <= > >+ <> =`

   Operators such as "<" refer to the collating sequence of characters for the implementation, which is not defined by the language.

2. The type `boolean`

2.1. Set of values

The permissible values for objects of this type consists of the two pre-defined identifiers, `true` and `false`.

2.2. Operators

The operators consist of:

1. Assignment.
2. The binary operators **`and`** and **`or`**, and the unary operator **`not`**.
3. Certain pre-defined subprograms (see Appendix D).
4. The relational operators. For purposes of the inequality operators, `false` is defined to be "less than" `true`.

3. The type `integer`

3.1. Set of values

The set of permissible values is implementation-defined, but there must be an unbroken set from the most negative to the most positive. A pre-defined constant, `maxint` (of type `integer`) has the most positive value for the implementation.

3.2. Operators

The set of operators consists of:

1. Assignment.
2. The unary operators `+` and `-`, which have the usual interpretation.
3. The binary operators, `+ - * /` for addition, subtraction, multiplication and division with truncation respectively.
4. The integer division operators **`div`** and **`mod`** for quotient and modulus respectively.
5. Certain pre-defined subprograms (see Appendix D).
6. The relational operators, which have the usual interpretation.

4. The type `real`

Objects of this type are floating-point real numbers. An implementation is not required to support this type.

4.1. Set of values

The set of values is implementation-dependent.

4.2. Operators

The set of operators is similar to that for `integer`, except that the **`div`** and **`mod`** operators are not permitted, and the set of pre-defined subprograms is different (see Appendix D).

APPENDIX D - PRE-DEFINED SUBPROGRAMS

1. Mathematical Functions

Table D/1 lists the mathematical functions pre-defined in Pascal-FC. Note that those involving real arguments or results will not be included in an implementation that does not provide the type `real`

name	argument	result
abs	real/integer	real/integer
arctan	real	real
cos	real	real
exp	real	real
ln	real	real
odd	integer	boolean
sin	real	real
sqr	real/integer	real/integer
sqrt	real/integer	real

Table D/1: Mathematical Functions

2. Ordering Functions

All ordinal types are ordered sets of values. Table D/2 lists the ordering functions, which are applicable to ordinal types.

name	argument	result
ord	ordinal	integer
pred	ordinal	ordinal
succ	ordinal	ordinal

Table D/2: Ordering Functions

NOTES

1. `pred` is not defined for the first value of a type, and `succ` is not defined for the last.
2. When applied to integers, the `ord` function returns the value of the argument.
3. For types other than integer, `ord` returns the value 0 for the first member of the set.

4. The type `boolean` is the ordered set {`false`, `true`}.

3. Type Transfer Functions

Table D/3 sets out the functions provided for transfer between types.

name	argument	result
bits	integer	bitset
chr	integer	char
int	bitset	integer
round	real	integer
trunc	real	integer

Table D/3: Type Transfer Functions

4. Inter-process Communication

Table D/4 lists the subprograms concerned with inter-process communication, which have been described in the chapters specified in the table.

name	form	chapter
delay	procedure	5
empty	function	5
initial	procedure	4
resume	procedure	5
signal	procedure	4
wait	procedure	4

Table D/4: Inter-Process Communication Subprograms

5. Input and Output

Input and output facilities are provided by two boolean functions and 4 procedures.

5.1. The `eoln` and `eof` functions

In Pascal-FC, these are both `boolean` functions of no arguments. `eoln` returns `true` when the next character in the input stream is the end-of-line marker and `false` otherwise. `eof` returns `true` when the next character in the input stream is the end-of-file marker and `false` otherwise. The treatment of end-of-file is implementation-dependent.

5.2. The `read` and `readln` procedures

A call to the `read` procedure is of the form:

```
read(variable{,variable})
```

The only permissible types for the arguments are: `char`, `integer` and `real`.

A call to the `readln` procedure has the form:

```
readln[(variable{,variable})]
```

The same restrictions are applied to the types of the arguments. The `readln` procedure consumes characters, up to and including the next end-of-line character, after satisfying its arguments.

5.3. The `write` and `writeln` procedures

A call to these procedures has the form:

```
write[ln] [(output_value format{,output_value format})]

output_value  ::=

   expression | string

format  ::=

   :field_width_expression:decimal_places_expression
  | %base_expression

string  ::=

   '{printing_character}'
```

NOTES

1. The arguments to these procedures may be of types `char`, `integer`, `real`, `semaphore` or `bitset`, in addition to string literals.
2. The decimal places expression is only applicable to arguments of type `real`.
3. The implementation may apply restrictions to the use of based output, which is in any case only defined for numeric types.
4. The output format for arguments of type `bitset` is not defined.

6. Timing

This category consists of the procedure, `delay`, and the function,`clock`. These were described in Chapter 10.

7. Miscellaneous

7.1. The `random` function

This is a function of one integer parameter. If `n` is the parameter, the function returns a value in the range `0 .. abs(n)`.

7.2. The `priority` procedure

This procedure is provided to control process priority, and it was introduced in Chapter 3.

8. APPENDIX E - COLLECTED SYNTAX

accept_alternative

```
accept_alternative  ::=

  [guard]
   accept_statement
  [;statement_sequence]
```

accept_statement

```
accept_statement ::=

   accept entry_identifier [formal_part] do
      statement
```

actual_parameters

```
actual_parameters  ::=

   expression {,expression}
```

add_op

```
add_op  ::=

   + | - | or
```

anonymous_process_type_declaration

```
anonymous_process_type_declaration ::=

  [provides_declaration]
   process_body_declaration
```

array_subscript

```
array_subscript  ::=
```

  "["*ordinal*_expression{,*ordinal*_expression}"]"

array_type

```
array_type  ::=
```

  array index_type{index_type} **of** type

assignment_statement

```
assignment_statement  ::=

   variable := expression
```

base

```
base  ::=

   unsigned_integer
```

based_integer

```
based_integer  ::=
```

   base#*digit*_character{*digit*_character}

bitset_literal

```
bitset_literal  ::=
```

   "[""]"
 | "["*integer*_expression{,*integer*_expression}"]"

case_alternative

```
case_alternative  ::=

   case_label{,case_label}:statement
```

case_label

```
case_label  ::=

   ordinal_constant
```

case_statement

```
case_statement  ::=

   case ordinal_expression of
      case_alternative
     {;case_alternative}
   end
```

channel_alternative

```
channel_alternative ::=

  [guard]
   channel_operation
  [;statement_sequence]
```

channel_operation

```
channel_operation  ::=

   send | receive
```

channel_type

```
channel_type  ::=

   channel of type
```

character_literal

```
character_literal  ::=

   'character'
```

compound_statement

```
compound_statement  ::=

   begin
   statement_sequence
   end
```

concurrent_statement

```
concurrent_statement  ::=

   cobegin
   statement_sequence
   coend
```

constant

```
constant  ::=

   constant_identifier
 | integer_literal
 | real_literal
 | character_literal
```

constant_declaration

```
constant_declaration  ::=

   const
      identifier = constant;
     {identifier = constant;}
```

decimal_integer

```
decimal_integer  ::=

   [+|-] unsigned_integer
```

declaration_part

```
declaration_part  ::=

   {
   constant_declaration
 | type_declaration
 | variable_declaration
 | procedure_declaration
 | function_declaration
   }
```

deferred_guarded_procedure_declaration

```
deferred_guarded_procedure_declaration  ::=

   forward_guarded_procedure_header
   guarded_procedure_body
```

deferred_sequential_subprogram_declaration

```
deferred_sequential_subprogram_declaration  ::=

   procedure_header forward; procedure_stub
 | function_header forward; function_stub
```

else_part

`else_part ::=`

`else` `statement_sequence`

empty_statement

`empty_statement  ::=`

`{`*white_space*`_character}`

entry_call

`entry_call  ::=`

process`_variable.`*entry*`_identifier [(actual_parameters)]`

entry_declaration

`entry_declaration ::=`

`entry` `identifier [formal_part] [mapping_indicator];`

enumeration_type

`enumeration_type  ::=`

`(identifier_list)`

exponent_part

`exponent_part  ::=`

`["e"|"E"] [+|-] unsigned_integer`

export_list

```
export_list ::=

   export procedure_identifier_list;
  {export procedure_identifier_list;}
```

expression

```
expression  ::=

   simple_expression {rel_op simple_expression}
```

factor

```
factor  ::=

   unsigned_integer
 | based_integer
 | unsigned_real
 | constant_identifier
 ! variable
 ! function_identifier [(actual_parameters)]
 | not factor
 ! bitset_literal
 | (expression)
```

field_declaration

```
field_declaration  ::=

   identifier[offset_indicator]
  {,identifier [offset_indicator]}
   : type
```

field_list

```
field_list  ::=

   field_declaration {;field_declaration}
```

field_selector

```
field_selector  ::=

   .record_field_identifier
```

formal_part

```
formal_part ::=

   ([var] identifier_list : type_identifier
   {;[var] identifier_list : type_identifier})
```

for_statement

```
for_statement  ::=

  for variable := expression to expression do
     statement
```

forward_guarded_procedure_header

```
forward_guarded_procedure_header  ::=

   guarded procedure identifier[formal_part]
      when boolean_expression;forward;
```

fractional_part

```
fractional_part  ::=

   .unsigned_integer
```

full_guarded_procedure_declaration

```
full_guarded_procedure_declaration  ::=

   guarded procedure identifier[formal_part]
      when boolean_expression;
     [declaration_part]
   begin
      statement_sequence
   end;
```

full_sequential_subprogram_declaration

```
full_sequential_subprogram_declaration  ::=

   sequential_subprogram_header
     [declaration_part]
   begin
      statement_sequence
   end;
```

function_header

```
function_header  ::=

   function identifier [formal_part]
   : type_identifier;
```

function_stub

```
function_stub  ::=

   function identifier;
     [declaration_part]
   begin
      statement_sequence
   end;
```

global_declaration_part

```
global_declaration_part  ::=

   {
   constant_declaration
 | type_declaration
 | variable_declaration
 | monitor_declaration
 | resource_declaration
 | procedure_declaration
 | function_declaration
 | process_type_declaration
 | process_object_declaration
   }
```

guard

```
guard ::=

   when boolean_expression =>
```

guarded_procedure_body

```
guarded_procedure_body ::=

   guarded procedure identifier;
     [declaration_part]
   begin
      statement_sequence
   end;
```

guarded_procedure_declaration

```
guarded_procedure_declaration ::=

   full_guarded_procedure_declaration
 | deferred_guarded_procedure_declaration
```

identifier

```
identifier  ::=

   letter{letter | digit}
```

identifier_list

```
identifier_list  ::=

   identifier{,identifier}
```

if_statement

```
if_statement ::=

   if boolean_expression then
      statement
  [else
      statement]
```

index_type

```
index_type  ::=

   "["ordinal_range{,ordinal_range}"]"
```

integer_literal

```
integer_literal  ::=

   decimal_integer
 | based_integer
```

main_statement_part

```
main_statement_part  ::=

   statement_sequence
  [;concurrent_statement
  [;statement_sequence]]
 | concurrent _statement
  [;statement_sequence]
```

mapping_indicator

mapping_indicator ::=

at *integer*_constant

monitor_body

monitor_body ::=

begin
statement_sequence

monitor_call

monitor_call ::=

*monitor*_identifier.*exported_procedure*_identifier
[(actual_parameters)]

monitor_declaration

monitor_declaration ::=

monitor identifier;
export_list
[declaration_part]
[monitor_body]
end;

mul_op

mul_op ::=

* | / | **div** | **mod** | **and**

null_statement

null_statement ::=

null

offset_indicator

offset_indicator ::=

at offset *integer*_constant

ordinal_range

ordinal_range ::=

*ordinal*_constant..*ordinal*_constant

procedure_call

procedure_call ::=

*procedure*_identifier [(actual_parameters)]

procedure_header

procedure_header ::=

procedure identifier [formal_part];

procedure_stub

```
procedure_stub  ::=

   procedure identifier;
     [declaration_part]
   begin
      statement_sequence
   end;
```

process_activation

```
process_activation ::=

   process_object_identifier[array_index][(actual_parameters)]
```

process_object_declaration

```
process_object_declaration ::=

   anonymous_process_type_declaration
 | process_variable_declaration
```

process_type

```
process_type  ::=

   process_identifier
  | process_array_type
```

process_type_declaration

```
process_type_declaration ::=

  [process_type_provides_declaration]
   process_type_body_declaration
```

process_type_body_declaration

```
process_type_body_declaration ::=

   process type identifier[formal_part];
       {entry_declaration}
       [declaration_part]
   begin
      statement_sequence
   end;
```

process_type_provides_declaration

```
process_type_provides_declaration ::=

  process type identifier[formal_part] provides
      entry_declaration
     {entry_declaration}
  end;
```

process_variable_declaration

```
process_variable_declaration ::=

   identifier_list : process_type;
```

program

```
program  ::=

   program_header
      global_declaration_part
   begin
      main_statement_part
   end.
```

program_header

```
program_header  ::=

   program identifier;
```

real_literal

```
real_literal  ::=

   [+|-] unsigned_real
```

record_type

```
record_type  ::=

   record
      field_list
   end
```

rel_op

```
rel_op  ::=

   < | <= | > | >= | = | <> | in
```

receive

```
receive  ::=

   channel_variable  ?  variable
```

repeat_limit

```
repeat_limit ::=

   until boolean_expression
 | forever
```

repeat_statement

```
repeat_statement ::=

   repeat
      statement_sequence
   repeat_limit
```

replicate_alternative

```
replicate_alternative  ::=

   for variable := expression to expression replicate
      channel_alternative
```

requeue_statement

```
requeue_statement ::=

   requeue [resource_identifier.]
        guarded_procedure_identifier[(actual_parameters)]
```

resource_body

```
resource_body ::=

   begin
      statement_sequence
```

resource_call

```
resource_call ::=

   resource_identifier.
        exported_procedure_identifier[(actual_parameters)]
```

resource_declaration

```
resource_declaration ::=

   resource identifier;
      export_list
      resource_declaration_part
  [resource_body]
   end;
```

resource_declaration_part

```
resource_declaration_part ::=

   {
   constant_declaration
 | type_declaration
 | variable_declaration
 | procedure_declaration
 | function_declaration
 | guarded_procedure_declaration
   }
```

select_alternative

```
select_alternative ::=

   channel_alternative
 | replicate_alternative
 | accept_alternative
 | timeout_alternative
 | terminate
```

select_statement

```
select_statement ::=

  [pri] select
      select_alternative
     {;or select_alternative}
  [else_part]
   end
```

send

```
send  ::=

   channel_variable  !  expression
```

sequential_subprogram_declaration

```
sequential_subprogram_declaration ::=

   full_sequential_subprogram_declaration
 | deferred_sequential_subprogram_declaration
```

sequential_subprogram_header

```
sequential_subprogram_header  ::=

   procedure_header
 | function_header
```

simple_expression

```
simple_expression  ::=

   [+|-] term {add_op term}
```

statement

```
statement ::=

     assignment_statement
   | procedure_call
   | for_statement
   | repeat_statement
   | while_statement
   | if_statement
   | case_statement
   | compound_statement
   | empty_statement
   | concurrent_statement
   | process_activation
   | monitor_call
   | channel_operation
   | select_statement
   | entry_call
   | accept_statement
   | resource_call
   | requeue_statement
   | null_statement
```

statement_sequence

```
statement_sequence ::=

     statement
    {;statement}
```

string

```
string  ::=
```

'{*printing*_character}'

term

```
term  ::=

   factor {mul_op factor}
```

timeout_alternative

```
timeout_alternative  ::=

  [guard]
   timeout integer_expression
  [;statement_sequence]
```

type

```
type  ::=

   type_identifier
 | enumeration_type
 | array_type
 | record_type
 | channel_type
```

type_declaration

```
type_declaration  ::=

   type
      identifier = type;
     {identifier = type;}
```

unsigned_integer

```
unsigned_integer  ::=

   decimal_digit{decimal_digit}
```

unsigned_real

```
unsigned_real  ::=

   unsigned_integer exponent_part
 | unsigned_integer fractional_part [exponent_part]
```

variable

```
variable  ::=

   variable_identifier{selector}
```

variable_declaration

```
variable_declaration  ::=

   var
      variable_list : type;
     {variable_list : type;}
```

variable_list

```
variable_list  ::=

   identifier [mapping_indicator]
  {,identifier [mapping_indicator]}
```

while_statement

```
while_statement  ::=

   while boolean_expression do
      statement
```

REFERENCES

1. ANSI, *Reference Manual for the Ada Programming Language*. 1983.
2. M. Ben Ari, *Principles of Concurrent Programming,* Prentice-Hall (1982).
3. M. Ben Ari, *Principles of Concurrent and Distributed Programming,* Prentice-Hall (1990).
4. R. E. Berry, *Programming Language Translation,* Ellis Horwood (1982).
5. G.L. Davies and A. Burns, ‘‘The Teaching Language Pascal-FC,’’ *The Computer Journal* **33**(2) pp. 147-154 (1990).
6. E.W. Dijkstra, ‘‘Co-operating sequential processes,’’ pp. 43-112 in *Programming Languages*, ed. F. Genuys,Academic Press (1968).
7. P. Brinch Hansen, ‘‘Structured Multiprogramming,’’ *CACM* **15**(7) pp. 574-578 (1972).
8. C.A.R. Hoare, ‘‘Monitors: an Operating System Structuring Concept,’’ *CACM* **17**(10) pp. 549-557 (1974).
9. Intermetrics, ‘‘Draft Mapping Rationale Document,’’ Ada 9X Project Report (August 1991).
10. INMOS Limited, *Occam Programming Manual,* Prentice Hall (1984).

CONTENTS

1 INTRODUCTION 2
1.1 Purpose of Pascal-FC 2
1.2 Historical Background 2
1.3 Scope of the Manual 2
1.4 Syntax Notation 2
2 PROGRAM STRUCTURE, DECLARATIONS AND STATEMENTS 5
2.1 Program 5
2.2 Declarations 6
2.2.1 Constant declarations 7
2.2.1.1 Character literals 7
2.2.1.2 Integer literals 7
2.2.1.3 Real literals 8
2.2.2 Type declarations 8
2.2.2.1 Enumeration types 9
2.2.2.2 Array types 9
2.2.2.3 Record types 10
2.2.3 Variable declarations 10
2.2.4 Procedure and function declarations 11
2.3 Statements 13
2.3.1 The assignment statement 13
2.3.2 The case statement 15
2.3.3 The compound statement 16
2.3.4 The empty statement 16
2.3.5 The for statement 16
2.3.6 The if statement 16
2.3.7 Procedure call 17
2.3.8 The null statement 17
2.3.9 The repeat statement 17
2.3.10 The while statement 17
2.4 Comments 18
3 PROCESSES 19
3.1 Process States 19
3.2 Process Declarations 20

3.2.1 Process Type Declarations 20
3.2.2 Process Object Declarations 21
3.3 Process Activation 21
3.3.1 The concurrent statement 22
3.3.2 Activating elements of an array of processes 22
3.4 Phases of Execution of a Pascal-FC Program 23
3.5 Process Scheduling and Priority 23
3.6 An Example: Multiple Update of a Shared Variable 24
3.7 Deadlock 25
4 SEMAPHORES 26
4.1 Declaration 26
4.2 Operations on Semaphores 26
4.2.1 The `initial` procedure 26
4.2.2 The `wait` procedure 26
4.2.3 The `signal` procedure 27
4.3 An Example: Multiple Update 27
4.4 Process States and Transitions 28
5 MONITORS 30
5.1 Declaration 30
5.2 Calls to monitors 31
5.3 Condition Variables 31
5.3.1 Declaration 31
5.3.2 Operations on conditions 32
5.3.2.1 The `delay` procedure 32
5.3.2.2 The `resume` procedure 32
5.3.2.3 The `empty` function 32
5.4 An Example: the Bounded Buffer 32
5.5 Process States and Monitors 34
6 RESOURCES 36
6.1 Declaration 36
6.2 Calls to resources 38
6.3 The requeue statement 38
6.4 An Example: the Alarm Clock 39
6.5 Process States and Transitions 41
7 RENDEZVOUS BY CHANNEL 42
7.1 Channels 42
7.1.1 Declaration and use of channels 42
7.1.1.1 Examples 42

7.1.2 Operations on channels 43
7.2 The type `synchronous` 43
7.3 An Example: Unbuffered Producer-Consumer 44
7.4 Process States and Transitions 44
8 ADA-STYLE RENDEZVOUS 46
8.1 Process Entries 46
8.2 The accept statement 46
8.3 Entry calls 47
8.4 Use of process provides declaration 47
8.5 Process States and Transitions 48
9 SELECTIVE WAITING 49
9.1 The select statement 49
9.2 Notes on the Semantics of the select Statement 50
9.2.1 Indivisibility 50
9.2.2 Order of checking for pending calls 50
9.2.3 Execution of select with all guards closed 51
9.2.4 The else part 51
9.2.5 The terminate alternative 51
9.3 Examples 51
9.3.1 The select statement with channel alternatives 51
9.3.2 The select statement with accept alternatives 52
9.4 Process States and Transitions 54
10 TIMING FACILITIES 55
10.1 The system clock 55
10.2 Outline of timing facilities 55
10.2.1 The `clock` function 55
10.2.2 The `sleep` procedure 55
10.2.3 The timeout alternative to the select statement 56
10.3 An example using `sleep` and timeout 56
10.4 Process states, deadlock and the timing facilities 57
11 LOW-LEVEL FACILITIES 58
11.1 The type `bitset` 58
11.1.1 Declaration 58
11.1.2 Assigning values 58
11.1.3 Set union operator 59
11.1.4 Set intersection operator 59
11.1.5 Set difference operator 59
11.1.6 Testing set membership 59

11.1.7 Relational operators and the type bitset 59
11.1.8 Type transfer functions 59
11.2 Addressing Device Registers with Mapping Indicators 60
11.3 Use of Record Offset Indicators 61
11.4 Interrupts 62
11.4.1 Mapping semaphores to interrupt sources 63
11.4.1.1 Program example 63
11.4.2 Mapping channels to interrupt sources 64
11.4.2.1 Program example 64
11.4.3 Mapping process entries to interrupt sources 66
11.4.3.1 Program example 66
11.5 Interrupts and process states 67
APPENDIX A - CHARACTER SET 68
APPENDIX B - RESERVED WORDS 69
APPENDIX C - PRE-DEFINED DATA TYPES 70
1 The type `char` 70
1.1 Set of values 70
1.2 Operators 70
2 The type `boolean` 70
2.1 Set of values 70
2.2 Operators 70
3 The type `integer` 71
3.1 Set of values 71
3.2 Operators 71
4 The type `real` 71
4.1 Set of values 71
4.2 Operators 71
APPENDIX D - PRE-DEFINED SUBPROGRAMS 72
1 Mathematical Functions 72
2 Ordering Functions 72
3 Type Transfer Functions 73
4 Inter-process Communication 73
5 Input and Output 73
5.1 The `eoln` and `eof` functions 73
5.2 The `read` and `readln` procedures 74
5.3 The `write` and `writeln` procedures 74
6 Timing 74
7 Miscellaneous 75

7.1 The `random` function 75
7.2 The `priority` procedure 75
8 APPENDIX E - COLLECTED SYNTAX 76
REFERENCES 98