# Reference Manual

# for the Programming Language

# Real-Time Pascal

PN: 99110141

RC Computer

PN: 99200389

**Keywords:**
Systems programming language, real-time, parallel processes, data communication network

**Abstract:**
This manual contains the third revised definition of the programming language Real-Time Pascal. The definition is implementation independent. The Real-Time Pascal programming language is a high level Pascal-like language, designed to express algorithms and their implementation as parallel cooperating processes, executing on a network of processing components.

**Date:**
March 1989

# TABLE OF CONTENTS

Page

# 1. INTRODUCTION

## 1.1 Goals

The principal goal of Real-Time Pascal is to be a programming tool particularly well-suited in situations characterized by the following two requirements:

- the software must provide rapid response to external events ("real-time"),
- programmers wish to utilize the organization of software into parallel cooperating processes as a fundamental structuring tool.

The major use of Real-Time Pascal has been, and is foreseen to remain, in the basic software of distributed processing systems and data communication network nodes:

- terminal emulation,
- layered protocol handling,
- local area networking services.

In these kinds of situations the two above-mentioned requirements are inherently present.

The programming of end-user applications, and in particular: programming by the end user, are not goals of Real-Time Pascal. High level run-time support functions, such as a general high level input/output system, are not included in the language. However, provided suitable tools are furnished along with support for the language itself, it may prove to be well suited for applications programming as well as systems programming.

Although Real-Time Pascal aims at low level programming it is very much a high level language. This is true in terms of syntax, in terms of programming facilities, and in particular in terms of the amount of consistency enforcement which is embodied in the language.

Real-Time Pacal has not been designed specifically for any particular machine. However, the feasibility and usefulness of an implementation on the Intel iAPXn86 processor series and RC3502 have been absolute requirements.

## 1.2 Main Features

In many respects Real-Time Pascal is, as one might anticipate, similar to standard Pascal /Pascal, ISO Pascal/. The major difference is that Real-Time Pascal includes facilities for starting and controlling multiple processes as well as for the orderly synchronization and intercommunication between such processes. Features which are basic to Real-Time Pascal but well-known from standard Pascal are not discussed in this section.

### 1.2.1 Processes

As in standard Pascal, a Real-Time Pascal *program* consists of declarations and definitions of data to be manipulated, and a description of actions to perform the desired manipulations. The execution of a Real-Time Pascal program is called a *process*. A process is said to be an *incarnation* of the program which is executed.

Real-Time Pascal is intended for compilation. The major ingredient of an implementation is the compiler which will transform source programs into object code, executable on some target machine. Throughout this document reference is made to *compiletime*, the time when a source text is being manipulated by a compiler, and *run-time*, the time when a dynamic system consisting of a number of cooperating processes is operative.

In addition to declarations of data and descriptions of actions a program (and this is where Real-Time Pascal departs from standard Pascal) may contain sub-programs, and a process may create, start and control incarnations of sub-programs of the program of which it is itself an incarnation. Sub-programs may be nested to any depth. In other words a number of Real-Time Pascal programs may constitute a *program tree*. The encloser relation between a program and a sub-program is carried directly over to the *parent* relation that exists between a process and a *child* process which it has created. Thus the dynamic set of active processes will exhibit a control structure reflecting the nested structure of the program tree.

An essential feature of Real-Time Pascal is that a number of special types and operations on variables of these types are directly tailored to perform synchronization and exchange of access to shared data between processes in a well-defined and secure fashion. In particular one important invariant is maintained a priori (i.e. without the programmer needing to worry about it): to every *message* there exists at any given time precisely one *reference*, allowing at most one process to access the message. Exchange of access to a message is achieved by passing the message via a *mailbox*.

The operations for process synchronization and message passing are available in Real-Time Pascal as predefined routines. This implies that an implementation of Real-Time Pascal will involve the construction of either:

- a software nucleus, i.e. a small operating system, which performs the message passing and process synchronization and scheduling functions, typically in a highly dedicated manner, or

- a run-time system providing a bridge to a general operating system which lends itself to supporting the type of process synchronization and inter-communication functions defined as part of Real-Time Pascal.

An operating system to be used for the latter kind of implementation must support the execution of multiple processes either in true parallel on a transparent multiprocessor system or in pseudo-parallel on a single processor. The operating system must perform process scheduling; it must also support the exchange of messages via mailboxes. In general it is necessary to critically evaluate a given operating system before it is used as a foundation for an implementation of Real-Time Pascal.

When a suitable general operating system is used to perform the Real-Time Pascal functions of process synchronization and message passing, the ability is opened up for Real-Time Pascal processes to cooperate with processes written in other, typically low-level, languages supported under that operating system. In order to make this possibility practically useful, the data formats and operating system calling sequences used by the Real-Time Pascal compiler in question must be well-documented.

One area in which the use of other languages in conjunction with Real-Time Pascal is particularly important is the direct interaction with peripheral devices and the processing of interrupts. So-called driver processes which perform tasks of these types will always be machine dependent and may, most often, be programmed in assembler or PL/M-type languages. Consequently no input/output instructions or interrupt syncrhonization functions have been incorporated into Real-Time Pascal. It is a simple matter, however, to extend a particular implementation with these functions, as is done in the implementation for RC3502.

### 1.2.2 Data Typing

Like standard Pascal, Real-Time Pascal is a strongly typed language. Types, in the abstract, provide important assistance to structured programmer thinking, and the enforcement of strong typing is a useful tool in the detection of many kinds of errors. A particular class of types, the so-called descriptive types, on the other hand, may be used in a very concrete fashion to describe the precise interpretation of bit-strings in the memory of the machine executing a Real-Time Pascal program. This feature is particularly useful when the precise representation of data is prescribed as part of the external specifications of a software project, e.g. a standard protocol for some aspect of a data communication function.

Another feature which stands apart from the classroom style of standard Pascal is that Real-Time Pascal allows the definition of families of conformant types, differing only in the values of type parameters which may determine e.g. the length of an array, but having the same structure. Types are also allowed to be dynamic, e.g. by having parameters which cannot be evaluated at compile-time. Both of these features support the construction of dynamically configurable software.

### 1.2.3 Data Access

The data items manipulated by a process may be allocated as *private* to the process, or they may be allocated as *shared,* implying that access to the data may be shared among several processes, as described above.

The handling of *variables* is based on the concepts of objects and types which are described in chapter 3. A private variable may be declared, in which case allocation and deallocation of memory for the variable is performed automatically in a *stack* according to the well-known discipline for block structured languages. A declared variable is accessed directly by name. A private variable may also be allocated dynamically in the so-called *heap* by an invocation of the predefined routine new. In this case the variable must be accessed through a pointer.

Each process has its own stack and heap, which are thus well-suited for private variables. However, a variable in the stack may be declared as shared, implying that it can only be accessed in a so-called *region.* A shared variable can be made accessible to a child process as a process parameter.

A variable is said to be *owned* by the process in whose stack or heap it is allocated. A variable may become known to processes other than the owner by being passed as a process parameter.

Messages are allocated neither in the stack nor in the heap of a process, but separate from both of these. Messages are organized in *pools.* A pool may contain a number of messages of equal size.

A message is not in itself a variable, but like an object it may occupy a number of consecutive bytes of memory, called the message buffer. A message buffer may be treated as a variable by superimposing a type onto it in a lock statement.

Messages are accessed through variables of the predefined type reference. A number of operations involving messages are available as predefined routines taking references as parameters. In particular it is possible to build *message stacks* (not to be confused with process stacks) and *message chains.*

Associated with a message is a set of *attributes,* one of which is the buffer size, i.e. the number of bytes the buffer occupies. The values of some of the attributes are accessible, and some may also be modified. A message with a buffer of size 0, called an *empty message,* may be used meaningfully in connection with message stacks, and/or for simple synchronization purposes. An empty message has a full set of attributes. Fig. 1 gives a sketch of how memory might be organized in an implementation of Real-Time Pascal. Clearly an architecture supporting segmentation will be helpful.

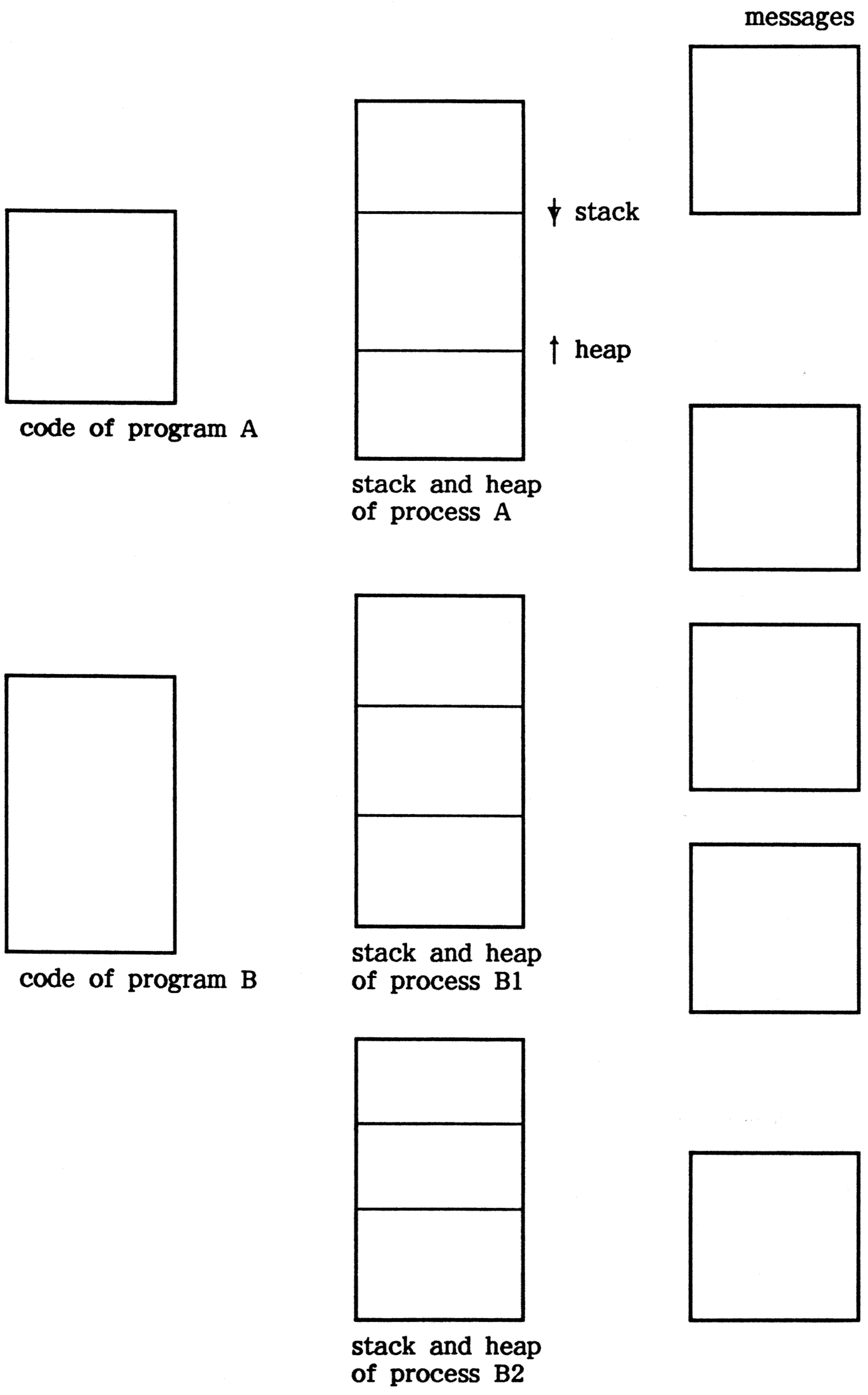

messages
stack
heap
code of program A
stack and heap of process A
code of program B
stack and heap of process B1
stack and heap of process B2

Fig. 1.1. Example of memory organization.

## 1.2.4 Distributed Systems

A principal area of intended use of Real-Time Pascal is the construction of distributed systems according to the general architectural principles described in /DSA/.

A resident module, in the sense of /DSA/, may consist of a number of cooperating Real-Time Pascal processes. Intercommunication at the so-called level i, may then take place using the Real-Time Pascal facilities for inter-process communication.

To support inter-module communication (at the so-called level d) the predefined type *port*, representing the concept of port as described in /DSA/ has been included in the language along with a set of predefined routines to perform inter-module communication (IMC) functions /DSA-IMC/.

Implementations may exist which support only a limited set of IMC functions or none at all.

## 1.2.5 Faults

The term *fault* is used throughout the following chapters to refer to violations of semantic rules which cannot be completely enforced at compile-time, i.e. violations which can in some cases only be detected when a program is executed.

The language allows partly programmer-defined handling of faults. By default the occurrence of a fault will cause the output of suitable diagnostic information.

## 1.2.6 Extensibility

The general representation of built-in functions of the language is that of predefined routines working on parameters of predefined, and often shielded, types.

When a desire for extensions to the defined language arises, it will be both natural and usually also easy to define such extensions in terms of one or more types and routines operating on parameters of these types. For example, a general high-level input/output system may be implemented in this way.

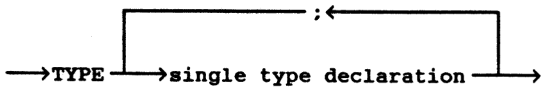
The distinction between built-in functions and such extensions will not appear very sharp at all, nor is it intended to. The only missing feature will be compiler supported protection of types one might wish to shield. Supporting a larger number of shielded types, however, requires very little in the way of compiler modification, and thus a future evolution of the language, e.g. toward supporting application programming, is at least feasible.

## 1.3 Syntax Diagrams

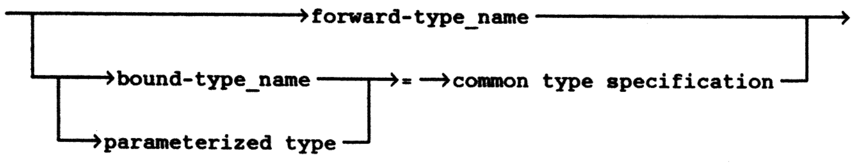
Each *syntax category* of the context-free syntax of Real-Time Pascal is defined by a *syntax diagram.* A syntax diagram consists of:
- the name of the defined syntax category followed by a colon,
- arrows, which may include branching,
- indications of occurrences of syntax categories, with the category names written in lower-case letters,
- language symbols written in upper-case letters (if text).

Example:

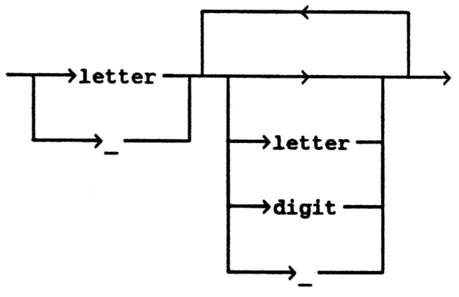
type declaration:



single type declaration:



name:



A source text or substring of a source text is a syntactically correct occurrence of a syntax category if it can be obtained by traversing the diagram defining that category, following the arrows. When an indication of an occurrence of a syntax category is encountered (must be entered through an arrow), the traversal rule is applied recursively. The result of a traverſal of a diagram is the sequence of lexical elements which have ultimately been encountered.

The names of syntax categories are used frequently in the descriptions of the semantics of language constructs to refer to particular occurrences of syntax categories. To make it clear that a sequence of words in the text is indeed a reference to such an occurrence it may be enclosed in single quotes.

Prefixes terminated by underscores are also used in names of syntax categories to make it easier to refer to a particular occurrence of a syntax category. They have no significance in the context-free syntax. For example 'bound-type_name' and 'parameterized-type_name' are syntactically equivalent and both defined by the diagram for 'name'. Hyphens and spaces are used exclusively as reading aids.

A prefix which occurs in several syntax diagrams may be understood as an indication of a context-sensitive syntax rule. Such rules, however, are all explained in the text describing the semantics of the relevant constructs.

A complete set of syntax diagrams is collected in appendix B. The page number on which a diagram is shown may be found by means of the catchword index, appendix D.2, since the category name appears here followed by a :.

## 1.4 Organization of this Manual

A rigorous definition of the Real-Time Pascal programming language is given in the following chapters. Each chapter is divided into sections, each dealing with a particular aspect of the language. The contents of a section are in general as follows:

- introductory remarks,
- syntax diagrams (some sections contain no diagrams),
- description in natural language of the semantics of the particular part of the language,
- optional notes, where specific consequences of the syntax or semantics may be pointed out,
- examples.

The notes and examples do not constitute part of the definition of the language.

# 2. LEXICAL ELEMENTS

A Real-Time Pascal source program is a string of characters which can be (uniquely) parsed as consisting of a sequence of suitably separated lexical elements. Separators are comments and nonprinting symbols. There are five categories of lexical elements:

- names,
- character literals,
- character strings,
- numbers, and
- language symbols.

Of these only names are defined in terms of syntax diagrams; the others are verbally described. All language symbols, some of which are keywords similar to names, as well as some additional names, are predefined as part of the language. The remaining lexical elements are programmer-specified.

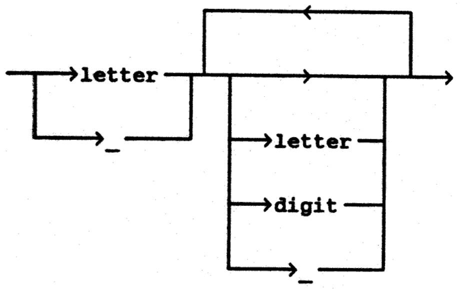
No separator may occur within a single lexical element. At least one separator must appear between any pair of consecutive lexical elements whenever this is necessary to provide unique delimitation.

An alphabetic character, a through z, is an occurrence of the category 'letter' which is referred to in the following. No destinction is made between the upper and lower case forms of the same letter, except in character literals or character strings.

## 2.1 Names

All declared entities, whether programs, routines, types, variables or merely constants, have a name.

**name:**



All characters in a name are significant. However, names used in conjunction with external linking may be abbreviated in an implementation/installation dependent fashion.

The keywords (cf. subsection 2.5.1) satisfy the syntax for 'name', but are explicitly excluded from the category. In addition a number of routines, types and constants exist (cf. Appendix C) with predefined names, i.e. these names can be redefined.

Examples:

step usage_count process_117 Very_Long_Identifier_Name

## 2.2 Character Literals

Character literals denote characters, which are values of the predefined type char. They are described in subsection 3.4.2.

## 2.3 Character Strings

Character strings denote values of string types. They are described in subsection 3.8.4.

## 2.4 Numbers

A number is a sequence of digits, possibly prefixed by a radix specification. A digit is a decimal digit, 0 through 9, or one of the letters A through F. A radix specification is the character with ordinal value 35 (in this manual shown as #) followed by one of the following letters: B, O, D, or H.

Numbers without a radix prefix are integer numbers in standard decimal notation; they denote values of the predefined type integer, cf. subsection 3.4.3.

Numbers prefixed with a radix specification are interpreted as follows:

| | |
|---|---|
| #B | binary, digits must be 0,1 |
| #O | octal, digits must be 0..7 |
| #D | decimal, digits must be 0..9 |
| #H | hexadecimal, digits must be 0..9, A..F |

Examples:

```
 #B1010
 #O777    #HCAFE    #D255    #D7913    -- four integers
```

## 2.5 Language Symbols

The predefined language symbols fall in two classes: keywords and special symbols.

### 2.5.1 Keywords

Keywords are reserved names, i.e. it is illegal to use them as names in declarations. Throughout this document some keywords are rendered in small letters and some in capitals, merely as a matter of style. The keywords are:

| | | | | |
|---|---|---|---|---|
| AND | END | IN | PROCEDURE | typesize |
| ARRAY | ENDLOOP | INSPECT | PROGRAM | UNTIL |
| AS | EXIT | LOCKBUF | RECORD | VAR |
| BEGIN | EXITLOOP | LOCKDATA | REGION | varsize |
| CASE | EXTERNAL | LOOP | REPEAT | WHILE |
| CONST | FOR | MOD | SET | WITH |
| CONTINUELOOP | FORWARD | NOT | SHARED | XOR |
| DIV | FUNCTION | OF | SHIFT | |
| DO | getswitch | OR | THEN | |
| DOWNTO | GOTO | OTHERWISE | TO | |
| ELSE | IF | PACKED | TYPE | |

### 2.5.2 Special Symbols

The special symbols are special graphic symbols or short sequences of such symbols. The following special symbols are defined as part of the language:

```
+  -  *   <    >  <>  <=  >=  (   )   (.  .)  (:  :)

↑  =  :=  :=:  .  ,   ;   :   ..  ***  !   ?   &   $

(*  *) <*  *>  --  '  "
```

## 2.6 Comments

Comments may be inserted in a source program in three forms:

1. (* comment *)
   All characters between the delimiters (* and *) are part of the comment, including any non-printing characters.

2. <* comment *>
   All characters between the delimiters <* and *> are part of the comment, including any non-printing characters.

3. -- comment end-of-line
   All characters from the delimiter -- up to the first occurrence of a carriage return, line feed, or form feed character are part of the comment.

Examples:

```
<* this is (* ... *) one comment *>
```

```
(* this is -- another
   comment *)

-- a third comment
```

## 2.7 Non-printing Characters

Non-printing characters which are not part of a comment are separators on their own. Any Real-Time Pascal compiler should allow space, tabulation, line and form feed, and carriage return characters.

# 3. TYPES AND OBJECTS

An *object* is a data entity, manifest during the execution of a program as occupying some amount of memory. With one exception (irregular sub-objects of objects of a descriptive type, cf. section 3.10) an object always occupies an integral number of successive bytes of memory. The number of bytes is called the *size* of the object and the address of the lowest addressed byte is called the *address* of the object. The *value* of an object is at any given time represented by the bit pattern present in the part of memory occupied by the object. An object may be a declared, and thus named, constant, variable, or parameter, or it may be a temporary anonymous object which exists only during the evaluation of some expression or the execution of certain kinds of statements.

Every object has a *type*. A type comprises a set of values which may be assumed by objects of the type. A number of predefined types exist as part of the language and additional types may be defined in type declarations. In particular it is possible to define structured types. Objects of structured types are composed of *sub-objects* of other (simpler) types. All objects of a type have the same size, called the size of the type.

Associated with a type is a set of *operations* applicable to values of the type. In the case of a structured type some of the operations provide access to the sub-objects of objects of the type.

An important relation between types is *compatibility* which plays a key role in determining when the assignment, explicit or by parameter passing, of a value to an object of some type is legal.

The structural aspects of a type are always obtainable from (the text of) the definition. However, the size of a type may be given by expressions which in general can only be evaluated at run-time. A type is said to be *established* when all expressions in the definition are evaluated and the actual layout of objects of the type and thus also the representation of values of the type, are determined. A type in whose definition all expressions are constant expressions and which can therefore be established at compile-time, is called a *static* type.

All types, whether explicitly specified or implicitly given by the context, are *classified* as:

- ordinal types,
- set types,
- pointer types,
- shielded types, or
- structured types.

Each class of types is described in a separate section of this chapter.

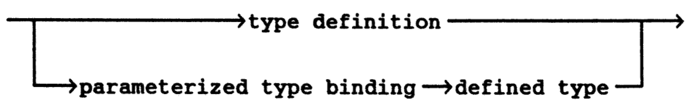
The language includes no *real* number types. It is a simple matter to extend the language or an implementation with a c'ass containing one or more real types.

Pointer types and shielded types are called *protected* types. The same is true of certain structured types, cf. section 3.8. Protected types cannot be used in conjunction with retyping (type conversion) in *with* or *lock* statements.

## 3.1 Specification of Types

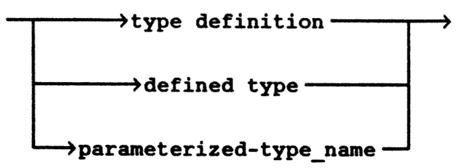
Type specifications, which include type definitions, are used in type declarations, in object declarations, in with and lock statements, and in the formal parameter lists of routine and program headings and of declarations of parameterized types. The form of type specification is called a common type specification. This form covers all types with no unbound parameters.

**common type specification:**



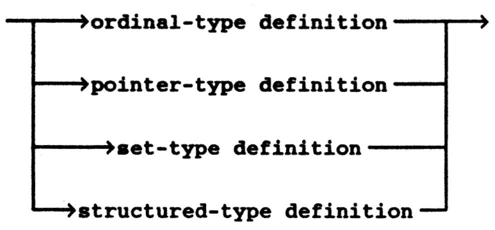
An augmented form of type specification is used in formal parameter lists of routines and programs where unbound parameterized types are allowed.

**formal type specification:**

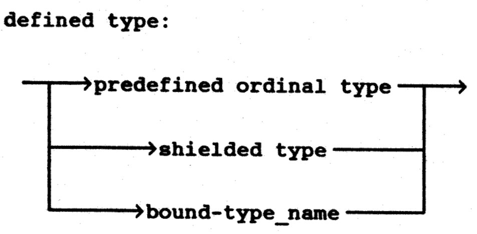


A type definition is the ultimate definition of any type which is not predefined.

**type definition:**



The form 'defined type' may be used in the specification of a type as either predefined or defined, bound, and named.



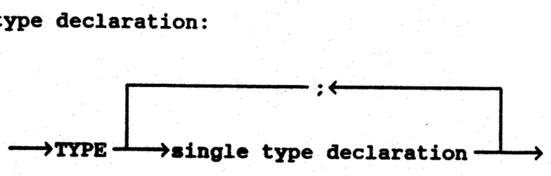
The missing details in the above description are given in the following sections:

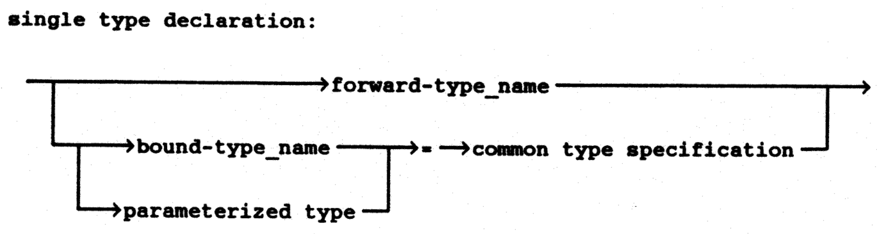
| syntactic category | section number |
|---|---|
| bound-type_name | 3.2 |
| parameterized-type_name | 3.2 |
| parameterized-type binding | 3.3 |
| ordinal-type definition | 3.4 |
| predifined ordinal type | 3.4 |
| set-type definition | 3.5 |
| pointer-type definition | 3.6 |
| shielded type | 3.7 |
| structured-type definition | 3.8 |

Note:
Of the five classes of types shielded types can only be predefined.

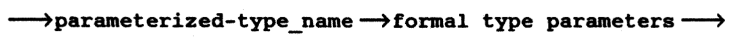
## 3.2 Declaration of Types

Types may be named and defined in type declarations:

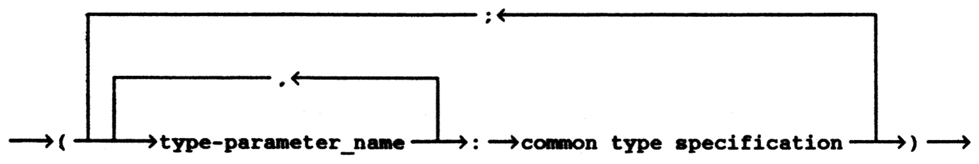




**parameterized type:**



**formal type parameters:**



The rules for defining types allow several types of use of (forward, bound, or parameterized) type names in 'defining type specifications'. However, no type name may be used on the right hand side of a 'single type declaration' until it has been introduced in a preceding declaration. In particular no type name may be used in its own 'defining type specification'. An exception to this rule is pointer types to the type itself.

A 'forward-type_name' may only be used in the definition of pointer types and for every 'forward type_name' occurring in a 'type declaration', the same name must be given a definition (bound or parameterized) later within the declaration part of the same block.

These rules exclude recursion in the definition of structured types, but allow objects of a structured type to contain pointers to objects of the same type and also allow mutual pointers between several structured types.

A single type declaration without parameters associates the 'bound-type_name' with the type specified on the right hand side, called the *defining type*. That is, when the 'bound-type_name' is itself used as a type specification, e.g. in the declaration of an object, the type thus specified inherits the value set, the representation of values, the object layout, the applicable operations and the classification of the defining type. However, the type specified by the 'bound-type_name' is *not* compatible with the defining type. An exception to the latter rule occurs when the defining type is a set type (cf. 3.9). The defining type is established when the type declaration is elaborated (cf. chapter 6).

A parameterized 'single type declaration' introduces the 'parameterized-type_name' as denoting a family of mutually conformant types. The 'formal type parameters', i.e. the 'type-parameter_names' may be used on the right hand side of the declaration. The types specified for formal type parameters must be ordinal types. Specification of a particular type in a parameterized family of types is described in the next section. No type is established when a parameterized type declaration is elaborated.

A typesize call is similar in form to a function call, but the "parameter" is the name of a type. The construct allows the size of a type to be used in computations.

```
typesize call:

  ──→TYPESIZE──→(──→bound-type_name──→)──→
```

The 'bound-type_name' must be the name of a bound (not parameterized) type. The value of a typesize call is the size (number of bytes) of the named type as computed when the type was established. The type of a typesize call is integer.

Example:

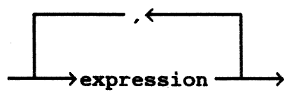
```
TYPE
  ptr_type;  -- forward announcement
  bound_typ= ARRAY (1..10) OF integer;
  rec_type= RECORD                    -- Note:
    f1: bound_typ;                    -- type of f1 not compatible
    f2: ARRAY(1..10) OF integer       -- with type of f2
    f3: ptr_type;                     -- one way of recursive def
    f4: ↑ rec_type;                  -- another recursive def.
  END(*RECORD*);
  ptr_type= ↑rec_type;
  mask_type= PACKED ARRAY(1..typesize(bound_typ)) OF boolean
```

## 3.3 Parameterized Types

A family of parameterized types may be defined in a type declaration (see the preceding section). A particular type in such a family, called a *bound parameterized type*, is obtained by binding values to the formal type parameters.

```
parameterized type binding:

  ──→parameterized-type_name──→(──→actual type parameters──→)──→
```

actual type parameters:

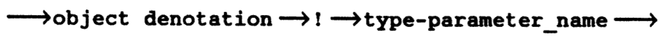


The 'parameterized-type_name' must occur in a preceding parameterized type declaration, i.e. it must denote a type family. The number of formal parameters in this declaration must equal the number of 'actual type parameters', and each actual parameter must be assignable to the type of the corresponding formal parameter.

The type specified by a 'parameterized type binding' is established according to the right hand side of the parameterized type declaration, i.e. the defining type specification, after all occurrences of the formal type parameters have been replaced with the values of the corresponding actual parameters. The properties of the defining type are inherited in the same fashion as in the case of a 'bound-type_name', cf. the preceding section.

The actual parameter values used to establish a bound parameterized type are attached to objects of the type. The parameter values are accessible whenever the object to which they are attached is visible.

selected type parameter:



The type of the denoted object must be a bound parameterized type. The 'type-parameter_name' must occur among the 'formal type parameters' of this type. The type of a 'selected type parameter' is the (ordinal) type specified for the 'type-parameter_name' and its value is the value of the corresponding actual parameter as evaluated when the object type was established.

Example:
```
TYPE
  column(rows: 1..100)= ARRAY(1..rows) OF integer;
  matrix(rows: 1..100)= ARRAY(1..rows) OF column(rows);
  matrix_10= matrix(10);
  ...
PROCEDURE invert(a:matrix);
VAR
  local_copy: matrix(a!rows);
  ...
  FOR i:=1 TO a!rows DO
    ...
```

## 3.4 Ordinal Types

Ordinal types are abstract types. For any ordinal type there exists a one-to-one mapping from the set of values of the type onto a finite interval of the integral numbers, yielding the *ordinal value* correspon-ding to each value of the type. It follows that the value set of an ordinal type is ordered by ordinal value and that every such value set has a first and a last element. By the ordering, every value, except the last one, has a *successor* and every value, except the first one, has a *predecessor*. Similarly, the relations *greater than* and *smaller than* are defined for pairs of values by the ordering.

The relational operators which produce results of the predefined type boolean apply to pairs of operands of any ordinal type. i.e. the two operands must be of the same ordinal type. Let oleft and oright denote the ordinal values of left and right operand, respectively. The relational operators are then defined in the following table:

| operator | result |
|---|---|
| = | true if oleft equals oright, otherwise false |
| <> | false if oleft equals oright, otherwise true |
| > | true if oleft is greater than oright, otherwise false |
| <= | false if oleft is greater than oright, otherwise true |
| < | true if oleft is smaller than oright, otherwise false |
| >= | false if oleft is smaller than oright, otherwise true |

For every ordinal type otype, there exist three predefined functions as described below:

FUNCTION succ(v: otype): otype

The result of a call of succ is the successor of the value of the parameter v, except if this value is the last one, in which case the call causes a fault.

FUNCTION pred(v: otype): otype

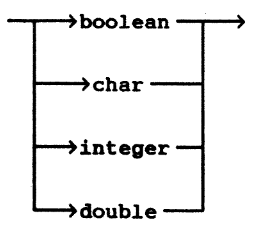
The result of a call of pred is the predecessor of the value of the parameter v, except if this value is the first one, in which case the call causes a fault.

FUNCTION ord(v: otype): integer

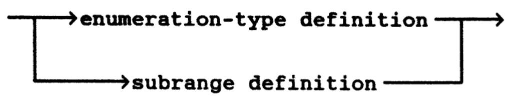
The result of a call of ord is the ordinal value corresponding to the value of the parameter v.

There are four predefined ordinal types and two ways to define new ordinal types. These are described in detail in the following subsec-tions.

predefined ordinal type:



ordinal-type definition:



## 3.4.1 The Type Boolean

The type boolean has two values which correspond to truth values and are denoted by the predefined value names *false* and *true*. The ordinal values are: ord(false)=0 and ord(true)=1.

AND and OR are two dyadic operators which take boolean operands and produce a boolean result. NOT is a monadic operator which takes a boolean operand and produces a boolean result. The results produced by these operators are in accordance with standard logical truth tables for conjunction, disjunction and negation, respectively. In addition the dyadic operator XOR is provided for boolean operands. Its result, which is also boolean, is defined by the formula

b1 XOR b2=(b1 AND NOT b2) OR (NOT b1 AND b2).

## 3.4.2 The Type Char

The values of type char are characters belonging to a character set derived from ISO-646 /ISO Char.Set/, i.e. an ASCII-like character code set.

The ordinal values of the type char span the interval 0..255. The non-graphic characters, with the ordinal values 0..32 and 127, are denoted by the predefined value names NUL, SOH, STX, ETX, EOT, ENQ, ACK, BEL, BS, HT, LF, VT, FF, CR, SO, SI, DLE, DC1, DC2, DC3, DC4, NAK, SYN, ETB, CAN, EM, SUB, ESC, FS, GS, RS, US, SP, and DEL. The characters with ordinal values in the range 32..126 are graphic characters and are denoted by character literals, i.e. the graphic symbol (letter, digit etc.) in question, betv'een single quotes. The set of available graphic characters and the corresponding ordinal values is implementation and/or installation dependent. No notation exists for characters with ordinal values in the range 128..255.

The graphic character symbols supported by an implementation/installation may also be used in character strings (cf. subsection 3.8.4).

There exists a predefined function which yields a character result:

FUNCTION chr(n: byte): char

The result of a call of chr is the character whose ordinal value equals the value of the parameter n.

Example:
```
CONST  single_quote=''';
TYPE   small_letter='a'..'z'
```

## 3.4.3 The Type Integer

The values of type integer are integral numbers. The ordinal value of such a number is the number itself. The range of the type integer, i.e. the interval spanned by its values is implementation dependent. As a natural extension of the language, double length integers, may be supported by an implementation, as a type named *double*, with the same operations as for the type integer. Positive integer values are denoted by integer numbers, cf. section 2.4.

There are ten dyadic operators which take integer operands and produce integer results:

| operator | description |
|---|---|
| + | addition |
| - | subtraction |
| * | multiplication |
| DIV | integer division (quotient truncated toward zero) |
| MOD | remainder of integer division, i.e. a MOD b = a-b*(a DIV b) |
| AND | bitwise logical and |
| OR | bitwise logical or |
| XOR | bitwise exclusive or |
| NOT | bitwise negation (one's complement) |
| SHIFT | bitwise logical shift toward more significant positions |

+ and - may also be used as monadic operators, implying an implicit left operand with value 0. When the result produced by an arithmetic operation falls outside the range of integer values supported by the implementation a fault occurs.

The result of the predefined function abs:

FUNCTION abs(n: integer): 0..maxint

is the absolute value of the parameter value. If this value falls outside the supported range the call causes a fault.

There are predefined procedures to increment and decrement variables of integer type (by 1 modulo $2^{16}$):

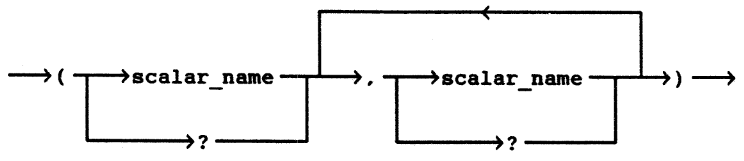
```
PROCEDURE inc(VAR v: integer)
PROCEDURE dec(VAR v: integer)
```

The predefined value names maxint and minint denote the largest and the smallest integer value, respectively, supported by the implementation.

## 3.4.4 Enumeration Types

An enumeration type is defined by explicitly naming its values:

enumeration-type definition:



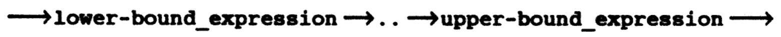
The names, at least two, given in the definition are used to denote the values of the type thus defined. Consider an enumeration type defined as ($e_0$, $e_1$, ..., $e_n$). This type has precisely n+1 distinct values with ordinal values in the interval 0..n. The ordinal value corresponding to $e_i$ is i, for i=0, 1, ..., n.

Example:
TYPE colours= (red, blue, green, yellow, pink)

## 3.4.5 Subrange Types

A subrange specifies a type compatible with an existing type, but with a constrained range of values:

subrange definition:



The lower and upper bound expressions must be of the same ordinal type, called the *base type* of the defined subrange type. The bounds are evaluated when the subrange type is established.

If the value of ord(upper bound)-ord(lower bound)+1 is greater than zero this is the number of elements in the value set of the subrange, otherwise the subrange is empty.

The same set of operators and predefined functions apply to values of the subrange type as to values of the base type, but objects of the subrange type are constrained to assume values in the range between the lower and upper bound values (inclusively).

There is one predefined subrange type, viz. *byte*, defined as:

byte= 0..255

There are predefined procedures to increment and decrement variables of byte type (by 1 modulo 256):

```
PROCEDURE inc(VAR v: byte)
PROCEDURE dec(VAR v: byte)
```

Note:
A subrange definition is the only place where an expression can occur in a type definition. Thus all dynamic types are built from subranges. Conversely, if all expressions in the subrange definitions of a type definition are constant expressions, then the defined type is static.

Example:
```
TYPE
  pos_int= 0..maxint;
  neg_int= minint..-1;
  codes= (nocode, ..., dummy_last_code);
  conv_table= ARRAY(succ(nocode)..pred(dummy_last_code))
              OF codes
```

## 3.5 Set Types

The set of values of a set type is the power set of the set of values of some ordinal type, called the element type of the set type. The available operators for sets correspond to the standard operators of mathematical set theory.

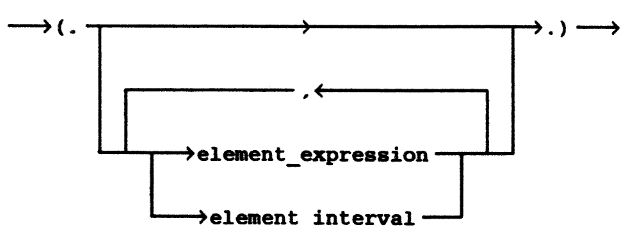
**set-type definition:**

```
  ──→SET ─→OF ─→common type specification ──→
```

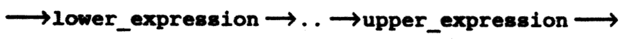
The common type specification specifies the element type which must be an ordinal type.

Values of set types are denoted by lists of set elements.

set denotation:



element interval:



All expressions occurring in a 'set denotation' must be of the same ordinal type. The type of the 'set denotation' is a set type whose element type is the type of the expressions.

The value of a 'set denotation' is evaluated by evaluating all the expressions. Their values determine the members of the set value. When an 'element interval' occurs all values in the closed interval from the value of 'lower_expression' to the value of 'upper_expression' are members. If the value of 'lower-expression' is greater than the value of 'upper-expression' the interval is empty.

When no expressions are present in a 'set denotation', i.e. (..), the value is the empty set. The empty set may occur whereever an operator of a set type is required. Occurring as an expression on its own, the empty set is assignable to any set type.

The operators applicable to values of set types are described in the following table, where st means some set type, et means the element type of st, lop means left operand, and rop means right operand. Notice that multiple occurrences of st in any one line of the table refer to compatible set types.

| operator | type of lop | type of rop | type of result | result |
|---|---|---|---|---|
| + | st | st | st | lop $\cup$ rop |
| * | st | st | st | lop $\cap$ rop |
| - | st | st | st | lop \ rop |
| IN | et | st | boolean | true if lop $\in$ rop, false otherwise |
| <= | st | st | boolean | true if lop $\subseteq$ rop false otherwise |
| >= | st | st | boolean | true if rop $\subseteq$ lop, false otherwise |
| = | st | st | boolean | true if lop $\subseteq$ rop and rop $\subseteq$ lop, false otherwise |
| <> | st | st | boolean | NOT lop=rop |

Note:
There is no operator to test for strong set inclusion.

Examples:
```
  digits= (. '0'..'9' .)
  letters= (. 'a'..'z', 'A'..'Z' .)
```

## 3.6 Pointer Types

The values of a pointer type are NIL (no pointer) and pointers to heap-allocated variables of a specified type, called the base type of the pointer type. A pointer may be used to access the variable it points to.

**pointer-type definition:**

$\longrightarrow \uparrow \longrightarrow$ **common type specification** $\longrightarrow$

The common type specification specifies the base type of the defined pointer type. The initial value of a pointer variable is NIL, i.e. it does not point to any variable.

A variable accessed through a pointer is called a designated variable.

**designated variable:**

$\longrightarrow$ **pointer_object denotation** $\longrightarrow \uparrow \longrightarrow$

The type of the denoted object must be a pointer type. The type of the designated variable is the base type of this pointer type. If the value of the denoted pointer object is NIL a fault occurs when an attempt is made to access the designated variable.

The comparison operators = and <> may be applied to pairs of operands of compatible pointer types. The result produced by the = operator is true if both pointers designate the same object, or if both have value NIL. Otherwise it is false. The result produced by the <> operator is the negation of the result of =.

There is a predefined function to test whether a pointer, of any pointer type ptrtype, is NIL.

FUNCTION nil(ptr: ptrtype): boolean

The result of a call of nil is true if the value of the parameter is NIL and false otherwise.

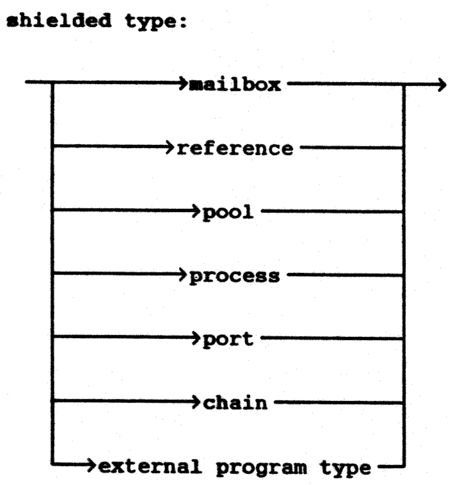
A variable is allocated on the heap and a pointer to it assigned to a pointer variable by a call of the predefined procedure new, where the parameter type ptrtype may be any pointer type.

PROCEDURE new(VAR ptr: ptrtype)
A call of new causes memory for a variable of the base type of the type of the parameter ptr to be allocated on the heap of the calling process. If an initial value is defined for the variable or any components of it, the initialization takes place immediately after allocation. The value of the parameter becomes a pointer to the allocated variable. If the claimed amount of memory is not available, the pointer becomes NIL after the call of new.

Example:
```
TYPE
  comp;  -- forward declaration
  comp1_type=
    RECORD
      number: integer;
      comp_chain: comp
    END(*RECORD*);
  comp_type= ARRAY(x..y) OF ↑comp1_type;
  comp= ↑comp_type;
VAR
  structure_start: comp;
  ...
  new(structure_start);
  new(structure_start↑(x));
  ...
  new(structure_start↑(x)↑.comp_chain)
  ...
```

## 3.7 Shielded Types

Shielded types are used in conjunction with control of offspring processes and with inter-process and inter-module communication. In order that the integrity of messages and of the data structures needed to administer multiple cooperating processes be preserved it is only possible to manipulate objects of shielded types by means of predefined routines. Accordingly, details of the representation of these types are not part of the reference definition of the language. Only predefined shielded types exist, seven in all.



Constants of shielded types do not exist. Variables of shielded types, except reference, can only be declared at the outer block level of a program, cf. subsection 6.2.1.

An object of type process may be used to control a child process, i.e. an incarnation of a sub-program. The value of a process object is either NIL or a reference to a child process, the initial value being NIL. Process objects may be manipulated by the predefined routines create, start, resume, stop and remove, as described in chapter 9.

The predefined function nil may be used to test whether a process variable has value NIL.

FUNCTION nil(VAR pr: process): boolean

The result of a call of nil is true if the value of the parameter pr is NIL, and false otherwise.

An object of type mailbox may be used to transfer access to a message stack from one process to another, using the predefined routines signal, wait and return, as described in chapter 9. The initial state of a mailbox is passive.

Objects of type external program are described in subsection 6.2.2. The initial state of an external program is unlinked.

An object of type port may be used in conjunction with intermodule communication as described in chapter 11. The initial state of a port is closed.

Messages are allocated using pools and accessed by means of objects of type reference. The value of a reference is either NIL (the initial value) or a reference to a message stack, called the *designated stack*. The top message of the designated stack is called the *designated message*. Message stacks, and the predefined procedures push and pop working on them, are described in detail in chapter 10.

The predefined function nil may be used to test whether a reference is NIL.

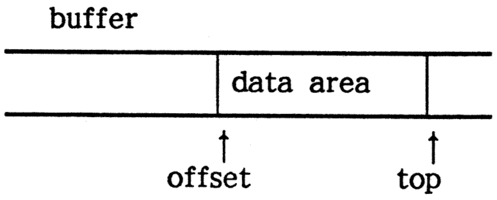
FUNCTION nil(VAR ref: reference): boolean

The result of a call of nil is true if the value of ref is NIL, and false otherwise.

Every message has fourteen attributes which are present even if the message is empty:

- home pool: the pool to which the message belongs,
- return address: mailbox to which the message may be returned,
- u1, u2, u3, u4: objects of type byte which may be read and written,
- size of the buffer, i.e. number of bytes,
- offset, top, byte count: objects of type 0..maxint which may be read and written; they describe the data area of the message buffer, see below,
- event kind: indicates how and why the message was placed in the mailbox from which it has last been received or that the message was removed from a pool; see details under the predefined function eventkind below,
- connection index, credit count, reason: used in conjunction with IMC functions, cf. chapter 11.

The values of the message attributes offset and top define an area within the buffer, called the *data area*, which comprises the (byte) locations from offset through top-1 relative to the beginning of the buffer.



The byte count attribute is used to indicate the size of a data unit which is located from the beginning of the data area, but which does not necessarily occupy the whole data area.

In order for the data area description to be consistent, offset must be less than or equal to top, which in turn must be less than or equal to the size of the buffer. In particular, if the message is empty, all three attributes must be zero.

In a message stack the topmost buffer attributes (size, offset, ...) will refer to the buffer of the topmost non-empty message, cf. section 10.1.

The concept of data area is used in conjunction with the IMC functions, cf. chapter 11, and is also intended as a basis for the establishment of practical conventions for the use of the language.

A pool is the home of a set of messages. Initially the set is empty. All messages belonging to the pool have the same size, and all are allocated from the same kind of memory. The set may grow by calls of allocpool/allocmempool, or shrink by calls of releasepool. At any point in time some subset of the set of messages which belong to a pool will be held in the pool, while the remaining part will be outside the pool, in use. By a call of alloc a message may be taken from the pool so it can be used (via a reference), and it may be put back in the pool by a call of release.

Whenever the set of messages belonging to a pool is empty, the message buffer size or/and memory kind attributes of the pool may be set or changed by a call of allocpool or allocmempool:.

```
FUNCTION allocpool(VAR p: pool; no_of_messages: 0..maxint;
         bufsize: 0..maxint): 0..maxint
FUNCTION allocmempool(VAR p: pool; no_of_messages,
         bufsize: 0..maxint; mem: mem_type): 0..maxint
```

The value of bufsize determines the buffer size of the messages which may belong to the pool p, and the value of mem determines the kind of memory to be used for these messages. The type of mem is the implementation dependent predefined enumeration type mem_type which is intended to reflect the various kinds of RAM used in multiprocessor systems which may include intelligent controllers with their own RAM resources.

If, at the time of call, the set of messages belonging to the pool p is non-empty, the values of the parameter bufsize and mem have no effect.

Subsequent to the possible setting of pool attributes as described above, an attempt is made to allocate memory for as many messages as indicated by the value of no_of_messages. These messages are subsequently placed in the pool p which is their home pool, ready to be taken out by calls of alloc (see below). If sufficient memory is not available the number of messages acquired may be smaller than requested. The actual number of messages is returned as the result of the function call.

Memory occupied by messages held in a pool may be deallocated by a call of releasepool. Depending on the implementation it may then be possible to reuse this memory for other purposes: program memory, stack, heap, or other pools.

```
FUNCTION releasepool(VAR p: pool;
          no_of_messages: 1..maxint): 0..maxint
```

The number of messages indicated by the value of no_of_messages are deallocated from the pool p and become free memory. If the requested number of messages is not present in the pool fewer messages may be deallocated. The actual number of deallocated messages is returned as the result of the function call.

A message is taken out from a pool by a call of the predefined procedure alloc:

```
PROCEDURE alloc(VAR r: reference; VAR p: pool; VAR ra: mailbox)
```

At the time of call the value of the parameter r must be NIL, otherwise a fault occurs. If the pool p is empty, i.e. all messages have been removed, the calling process will wait until a message becomes available. This occurs when a message is put back to the pool by another process (call of release, see below), or when additional memory is allocated for the pool (call of allocpool/allocmempool, see above). When several processes attempt to take out messages from an empty pool waiting takes place in a FIFO queue.

When a message becomes available it is removed from the pool, its return address becomes the mailbox indicated by the parameter ra, and r will designate a message stack consisting only of the removed message.

When a message has just been removed from its home pool its data area will be the whole buffer, i.e. offset=zero and top=size of the buffer. The attributes u1, u2, u3, u4 and byte count will all be zero. The value of the event kind attribute will be not_event, indicating the message does not represent a system event.

It is possible to specify a maximum time which a process is willing to wait for a message. This can be done by calling allocdelay instead of alloc, cf. subsection 9.2.2.

A message is put back in its home pool by a call of the predefined procedure release:

```
PROCEDRUE release(VAR r: reference)
```

At the time of call the parameter must not be locked (cf. section 5.9), and its value must not ›e NIL, nor may the designated message stack contain more than one message; otherwise a fault occurs. The message is put back in its home pool, and the value of r becomes NIL.

It can be tested whether a message belongs to a particular pool.

FUNCTION hometest(VAR ref: reference; VAR p: pool): boolean

The value of ref must not be NIL when hometest is called. If it is, a fault occurs. The result of a call of hometest is true if p is the home pool of the message designated by ref, otherwise it is false.

The event kind attribute of a message may be read in order to determine the kind of event which the message represents.

FUNCTION eventkind(VAR r: reference): event_type

If eventkind is called with a parameter with value NIL a fault occurs, otherwise the result is the value of the event kind attribute of the designated message. The result type is the predefined enumeration type

```
  event_type= (not_event, message_event, answer_event,
               process_removed, port_closed, disconnected, ?, ?,
               local_connect, remote_connect, reset_indication,
               reset_completion, credit, data_sent, data_arrived,
               data_overrun, ?, dummy_lcnct, dummy_rcnct,
               dummy_rindic, dummy_rcmpl, dummy_credit,
               dummy_sent, dummy_arrived).
```

The value not_event indicates the message has been obtained from a pool or its event kind has been reset. The value message_event indicates the message has been signalled from a process, cf. subsection 9.2.2. The value answer_event indicates the message has been returned by a process, cf. subsection 9.2.2. The value process_removed indicates the message has been returned from a process which was removed, cf. section 9.1. The remaining values indicate IMC events, cf. chapter 11.

The only way a process can modify the event kind attribute of a message while retaining access is by resetting it.

PROCEDURE resetevent(VAR r: reference)

If resetevent is called with a parameter with value NIL a fault occurs, otherwise the value of the event kind attribute of the designated message becomes not_event.

The u-attributes of a message may be read using the following four predefined functions:

```
FUNCTION u1(VAR r: reference): byte
FUNCTION u2(VAR r: reference): byte
FUNCTION u3(VAR r: reference): byte
FUNCTION u4(VAR r: reference): byte
```

If one of these functions is called with a parameter with value NIL a fault occurs. Otherwise the result is the indicated u-attribute of the designated message.

Similarly the u-attributes may be written using the following four predefined procedures:

```
PROCEDURE setu1(VAR r: reference; b: byte)
PROCEDURE setu2(VAR r: reference; b: byte)
PROCEDURE setu3(VAR r: reference; b: byte)
PROCEDURE setu4(VAR r: reference; b: byte)
```

If one of these procedures is called with a reference parameter with value NIL a fault occurs. Otherwise the value of the parameter b is assigned to the indicated u-attribute of the designated message.

The size of a buffer may be read using the predefined function bufsize:

```
FUNCTION bufsize(VAR r: reference): 0..maxint
```

If bufsize is called with a parameter with value NIL a fault occurs. Otherwise the buffer size of the designated message is returned as result.

The following six predefined routines may be used to read and set the values of the attributes offset, top and byte count.

```
FUNCTION offset(VAR r: reference): 0..maxint
FUNCTION top(VAR r: reference): 0..maxint
FUNCTION bytecount(VAR r: reference): 0..maxint
```

If one of these three functions is called with a parameter with value NIL a fault occurs. Otherwise the result is the value of the indicated attribute of the designated message.

```
PROCEDURE setoffset(VAR r: reference; val: 0..maxint)
PROCEDURE settop(VAR r: reference; val: 0..maxint)
PROCEDURE setbytecount(VAR r: reference; val: 0..maxint)
```

If one of these three functions is called with a reference parameter with value NIL a fault occurs. Otherwise the value of the val parameter is assigned to that attribute of the designated message which is indicated by the procedure name.

Chains (linked lists) of message stacks may be built and manipulated by means of objects of the types reference and chain and the predefined routines chainenqueue, chaindequeue, chainup, chaindown, chainstart, chainreset as described in section 10.2. A chain object serves as a handle to such a list. Access to a list cannot be transferred via a mailbox.

The routines eventkind, resetevent, u1, u2, u3, u4, setu1, setu2, setu3, setu4, bufsize, offset, top, bytecount, setoffset, settop, and setbytecount may also be called with a parameter of type chain instead of reference. In this case the relevant attribute of the current message is accessed. The chain must not be empty; if so a fault occurs.

## 3.8 Structured Types

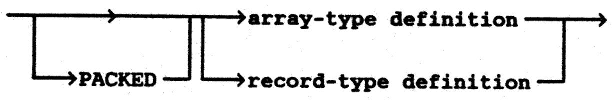
An object of a structured type is a structured collection of sub-objects of other (simpler) types. The value of such an object is a structured collection of values of the sub-objects. Structures may be arbitrarily deep, i.e. sub-objects of a structured object may themselves be of structured types. The types of the subobjects of objects of a structured type are called the *constituent types*. Sub-objects which are not of structured types are called *components*. The total set of components of a structured object are: those sub-objects which are themselves components plus the components of the remaining sub-objects.

If any component type of a structured type is protected (pointer or shielded) the structured type is also said to be protected.

The comparison operators = and <> may be applied to pairs of operands of the same structured type, provided they apply to all components. The result produced by the = operator is true if all component values are pairwise equal, otherwise false. The result produced by <> is just the opposite, i.e. true if any pair of component values are not equal.

A structured type is an array type or a record type, depending on the way it is built.

**structured-type definition:**



If the keyword PACKED is present the defined type is called a packed type. Objects of the type are also called packed. Packed types constitute a sub-class of the class of structured types. Packing indicates that the code generated by a compiler to access objects of the defined type should be optimized for compactness of object representation rather than execution time. An important sub-class of packed types is the descriptive types, cf. section 3.10.

A packed object may contain components which either do not start on a byte boundary or do not occupy a multiple of 8 bits (or both). Such a component is called an irregular object. It may not be used for retyping in a with or lock statement (cf. section 5.8) or as an actual parameter to be transferred by address (cf. section 6.2).

Array and record types are described in the following two subsections.

Note:
Unless a packed type is descriptive, cf. section 3.10, the precise effect of packing is not defined. In particular, if the type has constituent types which are not static, packing cannot be expected to have any effect.

## 3.8.1 Array Types

The sub-objects of an array are called elements. The elements are organized by indexing.

**array-type definition:**

```
——→ARRAY —→( —→common type specification —→) —→OF —→common type specification ——→
```

The type specified between the parentheses is called the *index type.* It must be an ordinal type. All elements of an object of the defined array type are of the type specified following OF, called the *element type.* Every object of the array type has precisely one element associated with each value of the index type.

The syntax ARRAY($t_1$, $t_2$, ..., $t_n$) OF element_type
is permissible as shorthand for

ARRAY($t_1$) OF ARRAY($t_2$) OF ... ARRAY($t_n$) OF element_type.

Similarly PACKED ARRAY($t_1$, $t_2$, ..., $t_n$) OF element_type

is legal shorthand for PACKED ARRAY($t_1$) OF PACKED ARRAY($t_2$) OF ... PACKED ARRAY($t_n$) OF element_type.

The elements of an array object are accessed by indexing.

**indexed element:**

```
——→array_object denotation —→( —→index_expression —→) ——→
```

The type of the denoted object must be an array type. The index expression must be assignable to the index type of this type. It is evaluated when access is made to the indexed element. the indexed element is that element of the array object which is associated with the value of the expression. The type of the indexed element is the element type of the array type.

The syntax $a(i_1, i_2, ..., i_n)$, where a denotes an array object, is permissible as shorthand for $a(i_1)(i_2) ... (i_n)$.

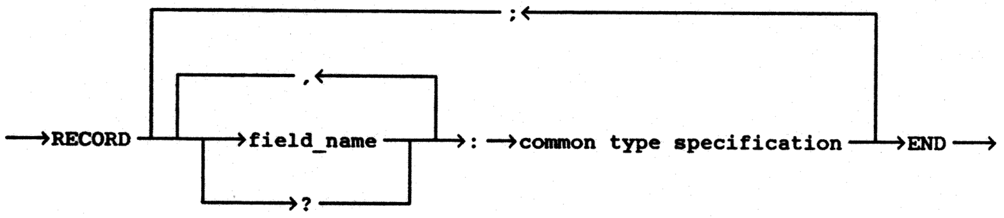
Note:
The above description implies that in general a range check is performed when an array element is accessed by indexing. A compiler may optionally allow index checking to be suppressed.

Example: (cf. section 3.3)
```
VAR
  amatrix: matrix_10;
  temp_column: column(10);
  ...
  temp_column:= amatrix(i); -- assignment of complete column
  amatrix(i,j):= amatrix(j,i); -- single component
  ...
```

## 3.8.2 Record Types

The sub-objects of a record are called fields. The fields are organized by naming.

record-type definition:



All the field names in a record type definition must be distinct. Each field name introduces and identifies a field. The type of the field, called the field type, is specified by the common type specification following :. A field type may not be specified by an 'enumeration-type definition'. An unused field of a record type may be specified explicitly, by means of a ?, called an unused-specification, in order to adjust the positioning of subsequent fields. An unused-specification is equivalent to the declaration of a field of the specified type, which must be a static ordinal type. The field is not accessible, i.e. it cannot be selected. Assignment to an unused field can only be made by assigning a value to the record object as a whole.

The fields of a record are accessed by selection by name.

**selected field:**

```
──→record_object denotation ──→ . ──→field_name ──→
```

The type of the denoted object must be a record type. The field name must occur in the definition of the record type. The selected field is that field of the record object which is identified by the field name. Its type is the specified field type.

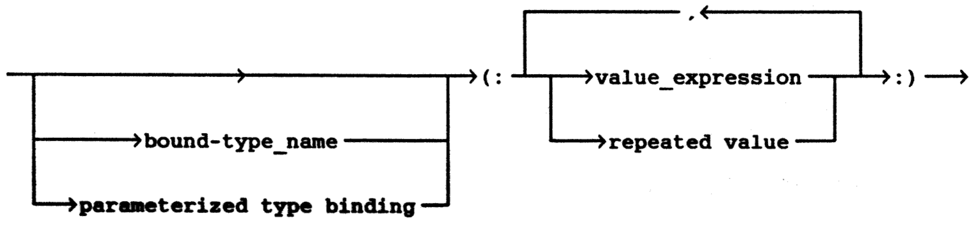
An abbreviated syntax for field access may be used in with statements, cf. section 5.8.

<u>Example:</u>
```
VAR
  rec: RECORD
    f1,f2: integer;
    f3: boolean
  END(*RECORD*);
  ...
  rec.f1:= rec.f1+rec.f2
  or
  WITH rec DO f1:= f1+f2
```

## 3.8.3 Notation for Values of Structured Types

Values of a structured type may be denoted by lists of element or field values.

**structured value:**



**repeated value:**

```
──→repetition_expression ──→*** ──→value_expression ──→
```

The 'bound-type_name' or 'parameterized type binding' specifies the type of the structured value, which must be a structured type. The construct between (: and :) is evaluated to a list of values, by evaluating the expressions in the order of occurrence. If the type is specified by a 'parameterized type binding' it is established before the value list is evaluated.

If the structured type is an array type the values in the list are the element values in index order. The number of values must equal the number of elements and the type of each value must be assignable to the element type; otherwise a fault occurs. A 'repeated value', if present, is equivalent to a number of repeated occurrences of its 'value_expression'; however, the expression is only evaluated once. The number is given by the ordinal value corresponding to the value of the 'repetition_expression' which must be a non-negative integer. If the number is negative a fault occurs.

If the structured type is a record type the values in the list are the field values in the order in which the field names occur in the definition of the record type. The number of values must equal the number of fields and the type of each value must be assignable to the corresponding field type; otherwise a fault occurs. If the record type contains explicit specified unused fields (cf. section 3.10) values must be given for these. They are implicitly set to value zero. In the case of a record type a 'repeated value' must not occur.

The type specification may be omitted when the type it specifies can be inferred from the context, i.e. in the following two situations:

1. The 'structured value' occurs as a 'value_expression' within a larger 'structured value'.

2. The 'structured value' occurs as an 'initialization_expression' in a variable or shared declaration.

Example: (cf. section 3.3)
```
CONST
  identity_3= matrix_3(:(:1,0,0:), (:0,1,1:), (:0,0,1:):);
  nul_list= list(:length***0:);
VAR
  a4: matrix(4):= (:4***(:4***0:):); -- initially null
```

## 3.8.4 String Types

The family of one-dimensional character string (array) types is predefined:

```
string(length: byte)= ARRAY(1..length) OF char
```

Values of types from the string family may be denoted by character strings. A character string is a string of graphic symbols, each representing a character value, enclosed in double quotes. The double quote character may itself be part of a character string. In this case it must be indicated by two adjacent occurrences of the " graphic symbol. Individual non-graphic characters may be included in a character string by means of the concatenation operator &. The character string must start with a, possibly empty, string surrounded by double quotes.

The specific type of a character string, i.e. the particular member of the string family of which it denotes a value, is determined by its length, i.e. the number of characters it consists of.

As an implementation feature character strings may be truncated or extended with SP characters when appropriate in the context, e.g. when occurring on the right hand side of an assignment statement where the corresponding component on the left hand side is of a string type with different length.

Examples:

```
"this is a string"
"x""x" (* a string of length 3 *)
"*** illegal input message" & BEL & CR & LF -- string(28)
```

## 3.9 Type Compatibility

The compatibility relations defined in this section are used to determine when a value may be assigned to an object, either explicitly when an assignment statement is executed or implicitly in connection with parameter passing.

The essential relation is *assignment compatibility*. However, the basis for that relation is the compatibility relation between types which is therefore defined first. The compatibility relation applies to established types.

Two types are said to be the same if:
- both are the same predefined type, or
- both are specified as the same bound-type_name, or
- both are specified as a binding of the same parameterized-type_name and the values of the actual type parameters are pairwise equal.

Two objects are said to be of the same type if:
- their types are the same (cf. above), or
- if they are variables introduced in the same list of variable names, cf. subsection 3.11.2, or
- if they are formal parameters of kind value introduced in the same formal name list, cf. section 6.1.

Two types are said to be compatible if:
- they are the same type (cf. above), or
- one is a subrange of the other (or both are subranges of the same type), or
- both are set types and the base types are compatible, or
- both are specified as ↑tname, where tname is a type name.

Type computation rules are defined for expressions (cf. chapter 4) so as to associate every expression with a type, either an explicitly specified type, or a predefined type determined implicitly by the structure of the expression. An expression exp of type $t_2$ is defined to be assigment compatible with the type $t_1$ if:

- $t_1$ and $t_2$ are compatible ordinal types and the value of exp is within the range specified for $t_1$, if any, or
- $t_1$ and $t_2$ are compatible set types and all members of the value of exp are within the range specified for $t_1$, or
- $t_1$ and $t_2$ are compatible pointer, shielded, or structured types, or
- $t_1$ and $t_2$ are both string types, cf. subsection 3.8.4.

Throughout this manual the shorthand form "assignable to" may be used instead of "assignment compatible with".

Example:
In each line below the types of a and b are compatible.

```
a: boolean;          b: boolean;
a: def_type;         b: def_type;
a: param_type(7);    b: param_type(7);
a: 1..10;            b: 2..15;
a: SET OF 1..10;     b: SET OF 2..15;
a: ↑ptrtype;         b: ↑ptrtype;
```

## 3.10 Object Layout

In general the definition of the language does not prescribe any specific layout for objects, and thus the way objects are laid out in memory and the way their values are represented depend on the implementation in question. In order to allow cooperation with processes whose programs are not written in Real-Time Pascal these aspects of an implementation must always be documented carefully.

By specifying a *descriptive type*, however, it is possible within certain limits for the programmer to explicitly determine the layout of objects. Descriptive types constitute a subclass of packed types, recursively defined as having constituent types which are all static ordinal types, or themselves descriptive.

This definition implies that all component types of a descriptive type are ordinal types. The representation of ordinal type components is presently described. Notice that the description only covers subobjects of objects of descriptive types.

A component of an ordinal type is represented as a binary number in a maximum of 16 bits. If the type includes negative values the two's complement representation is used. The number of bits used to represent objects of some type depends on the range of values. Let minval and maxval be the ordinal values corresponding to the first and last value of the type, respectively. Then the number of bits used for objects of the type is:

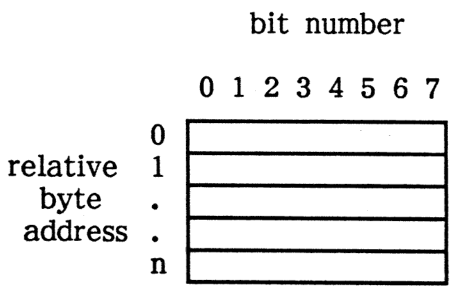
minval<0, maxval<0: $\log_2(-minval)+1$
minval<0, maxval>0: $\max\{\log_2(-minval), \log_2(maxval+1)\}+1$
minval>0, maxval>0: $\log_2(maxval+1)$

The numbers obtained by these formulae must be rounded up to obtain integral numbers.

As sub-objects of an object of a descriptive type need not occupy whole bytes the total object is considered as a bitstring rather than as a bytestring. The ordering of bits within such a bitstring is defined as follows:

- bits within separate bytes are ordered by byte address,
- bits within the same byte are ordered by significance, i.e. least significant bits first.

This implies that the whole bitstring will be ordered by significance, see the figure below.



When viewed in this fashion, i.e. the significance of bit i is the ith power of 2, bits are ordered as the characters on a text page.

Fig. 3.1. Bitstring ordering.

An ordering is also defined for the sub-objects of an object of a structured type: in the case of an array type by index; in the case of a record type by the order of the field names in the record type definition.

The following simple rule defines the layout of objects of a descriptive type: within the bitstring occupied by the object, subobjects are located contiguously and in order. Notice that this rule, like the definition of a descriptive type, is recursive. The following points complete the rule:

- every definition of a descriptive type is implicitly extended with an unused component at the end so that objects of the type occupy an integral number of bytes, if this property is not already satisfied by the type as stated,
- no component may cross two byte boundaries. When possible, an unused component is inserted to fill up a byte so that this situation is avoided.

Note:
If the comparison operator = and <> are applied to objects of structured types with unused components they are also applied to unused fields.

Example:
Consider the types

```
rec_type= RECORD
  a: integer;
  b: byte;
  c: 0..7;
  d: -3..4;
  e: -1000..1000;
  f: char;
  g: boolean;
  h: integer
END (*RECORD*)

arr_type= ARRAY (1..4) OF 0..2000 (* 11 bits *)
```

Typical layouts for objects of these types are shown to the left in the figure below; note that this representation is not specified by the language. However, if the keyword PACKED had been present before RECORD/ARRAY, the types would have been descriptive and their layout prescribed to be as shown to the right. The * marked fields are implicitly inserted unused components.

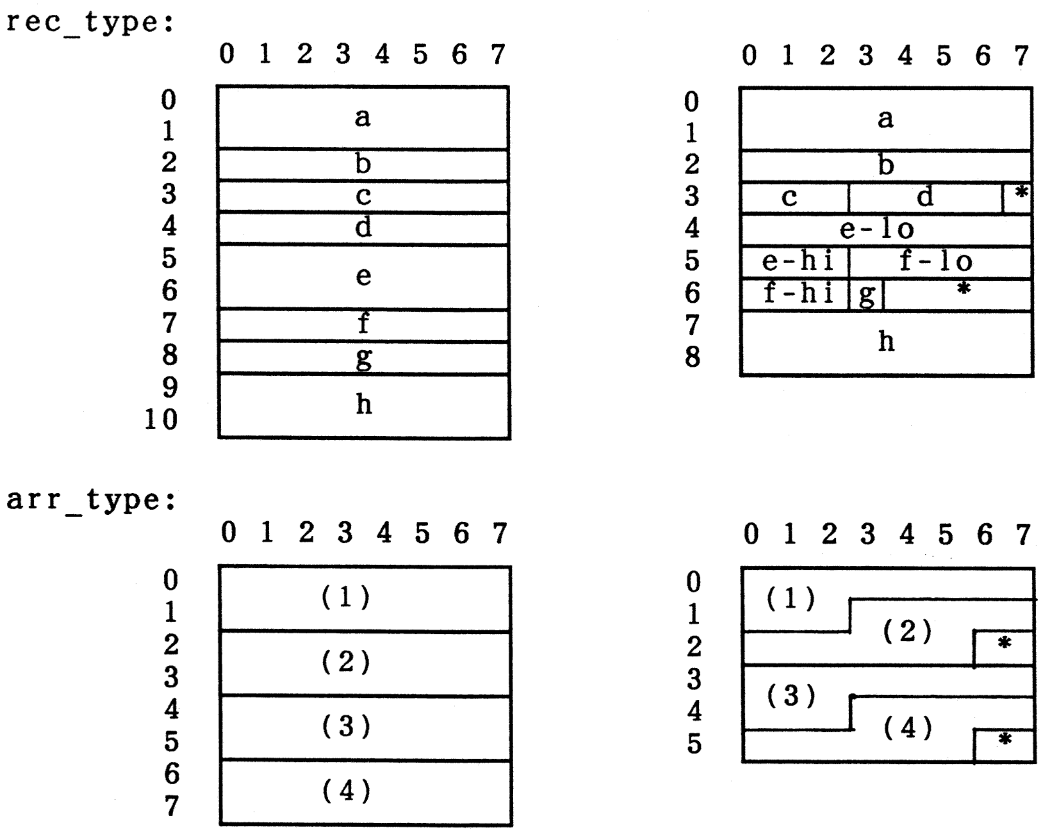


Fig. 3.2. Example of object layout.

## 3.11 Object Declarations

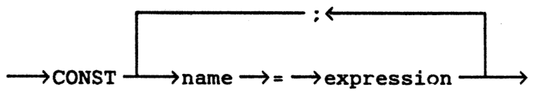
Declarations of objects may occur in the declaration part of a program or routine block, cf. chapter 6.

An object declaration inctroduces and names an object which may be used in the remainder of the block.

### 3.11.1 Constant Declarations

A constant declaration serves to name a constant object, i.e. an object whose value can be computed at compile-time and which cannot be altered dynamically by assignment.

**constant declaration:**

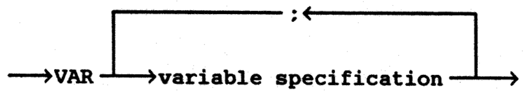


Each constant name introduced in a constant declaration denotes a constant object the type and value of which is determined by the expression following =. The expression must be a constant expression as described in chapter 4. The type of a constant must not be protected.

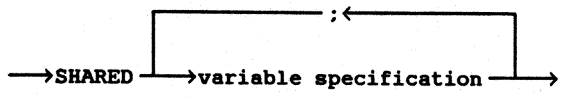
## 3.11.2 Variable Declarations

A variable or shared declaration serves to name one or more stack-allocated private or shared variables and optionally to specify initial values for instances of such variables.

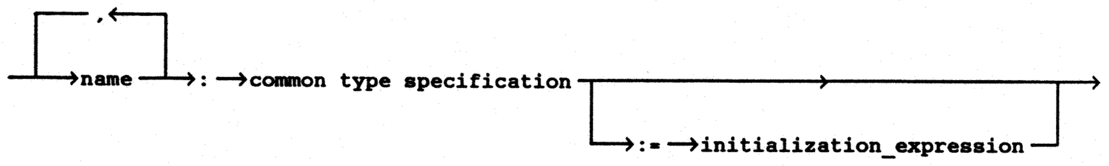
**variable declaration:**



**shared declaration:**



**variable specification:**



The variables introduced in a 'variable declaration' are private variables, i.e. they can only be accessed by the process in whose stack they are allocated. The variables introduced in a 'shared declaration' are shared variables which can only be accessed in region statements.

The type of each variable named in a variable or shared declaration is given by the first following type specification and the initial value by the expression following :=, if present. The type of the expression must be assignable to the specified type of the variable. When several variable names are listed, separated by commas, each name introduces a variable of the specified type. And each of them are given the initial value of the expression, if present. The value of the expression is only evaluated once. The type of a shared variable must not be process, external program, reference, pointer, or chain; nor may these types occur as component types in the types of shared variables.

When a variable or shared declaration is elaborated the specified types are established; memory is allocated on the stack for an instance of each of the named variables; the initialization expressions, if present, are evaluated; and the values of the expressions become initial values of the variables. An initialization expression must not contain function calls. However, the predefined functions abs, chr, ord, pred, and succ may be used.

Variables of components for which initial values or states are predefined are initialized accordingly. The initial value of a variable for which an initial value is neither predefined nor specified is undefined.

In a structured value occurring in an initialization expression the notation ? may be used for components of protected types. This makes it possible to combine the predefined initialization of these components with an explicitly specified initialization of the remaining components.

A varsize call is similar in form to a function call. It allows the size of a variable to be used in computations.

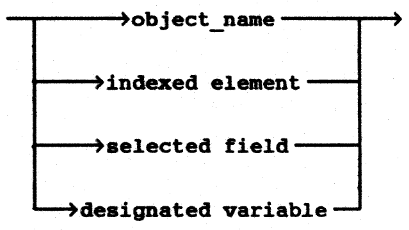
```
varsize call:

  ──→VARSIZE ──→ ( ──→variable_name ──→ ) ──→
```

The 'variable_name' must be the name of a variable or a record field, made visible by means of a with statement. It must have been introduced in a variable declaration. The value of a varsize call is the size (number of bytes) of the variable as computed when its type was established. The type of a varsize call is integer.

## 3.12 Notation for Objects and Values

An object which is declared or part of a (larger) declared object or accessed through a declared pointer, may be referred to by denotation.

**object denotation:**

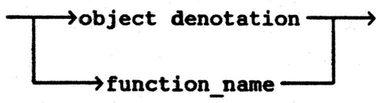


The denoted object is said to be *accessed* when: an assignment statement in which it appears on the left hand side is executed; or the factor which it constitutes in some expression is evaluated.

If the 'object denotation' is an 'object_name' its type is the type of the named object as determined by its declaration. The types of the other forms of denoted objects are described in previous sections of this chapter.

The target of an assignment may either be a variable object or the implicit result object associated with a function call.

**variable denotation:**



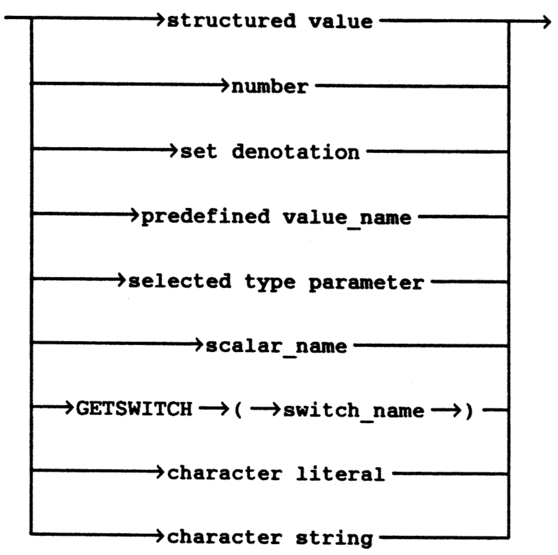
When an object denotation occurs as a variable denotation it must denote an object which is a variable or part of a variable in the stack or heap of a process, or superimposed on a message buffer in a lock statement. It must not be a constant or part of a constant.

When a function name occurs as a variable denotation it denotes the implicitly declared result object of an activation of the function it names. A function name may be used in this fashion only in the action part of the function, i.e. not in inner blocks.

The type of a variable denotation is the type of the denoted object or the result type of the function, whichever applies.

Values which are not the values of objects or sub-objects may be used in expressions.

value denotation:



A value denotation denotes the value of an anonymous object, the type and value of which is as described for the relevant one of the possible forms.

A getswitch call is similar in form to a function call. It allows the value of a compiler switch to be used in computations, i.e. transfer of values from the domain of switch names to the domain of program names (cf. chapter 12).

# 4. EXPRESSIONS

An expression describes either an object to be addressed or some computation to be performed by applying operators and functions, predefined as well as programmer-defined, to values of objects as they are at the time of computation and to constant values which may be denoted directly in the expression.

The operators of the language are divided in groups with different precedence. In order of increasing precedence the groups are: relational operators, addition-type operators, multiplication-type operators, and the negation operator.

All operators are described in chapter 3 in conjunction with the description of the types of the operands they apply to. Some operators exist in several semantically distinct versions, applicable to different types of operands and producing results in different ways depending on the operand types, e.g. <= may be used for integer (or in general: ordinal value) comparison, as well as for set comparison (inclusion).

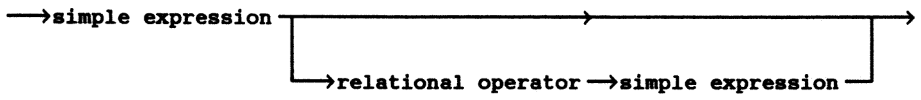
In section 4.1 reference is given, for each operator, to all sections where a version of that operator is described. The following selection rule is used in the application of operators occurring in expressions: If the types of the operand(s) provided for an operator correspond to one of the versions of that operator, then that version of the operator is selected. Otherwise the expression is illegal. The type of the result is determined according to the description of the selected version of the operator. The result may again be used as an operand of another operator and the selection rule may then be applied repeatedly.

An expression which is used as an actual parameter where the kind of the corresponding formal parameter is variable, shared or inspect (cf. chapter 6) must have the form of an 'object denotation'. Such an expression is called an *object expression*. The evaluation of an object expression stops when the address of the denoted object has been computed. In all other cases the evaluation of an expression proceeds until a value has been obtained, as described in the following section.

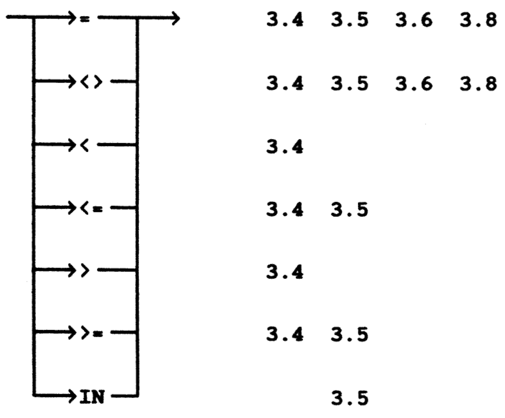
## 4.1 Evaluation of Expressions

The evaluation of an expression yields a type and a value. This section describes how the type and value are obtained from the types and values of the parts of the expression which constitute the operands at the various stages of computation.

**expression:**

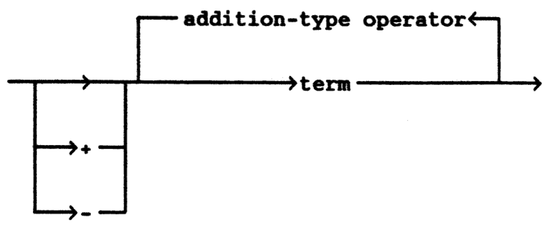


**relational operator:**



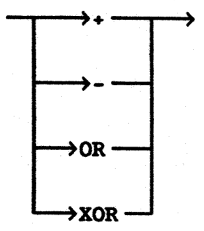
The expression is evaluated by evaluating the simple expression(s) in the order of occurrence. Then, if no operator is present, the type and value of the expression are the type and value of the (only) simple expression. Otherwise the appropriate version of the operator is applied to the values of the simple expressions; the result is the value of the expression, its type being boolean for all relational operators.

**simple expression:**



A leading + or - implies an implicit left term with type integer and value 0.

addition-type operator:



3.4.3 3.5

3.4.3 3.5

3.4.1 3.4.3

3.4.1 3.4.3

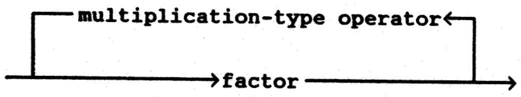
The evaluation of a simple expression proceeds from left to right. The leftmost term (possibly an implicit 0) is evaluated, yielding a *preliminary result*. The following is then performed repeatedly:

The preliminary result is used as left operand of the leftmost remaining operator. The leftmost remaining term is evaluated and used as right operand. The appropriate version of the operator is then applied and produces a new preliminary result.

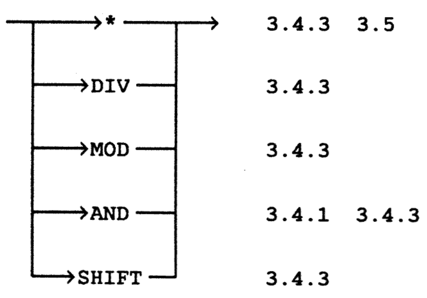
When no more operators and terms are left the simple expression has been completely evaluated; the final type and value of the preliminary result constitute the type and value of the simple expression.

There is one exception to the rule described above: if the left operand of the OR-operator is of type boolean and has value true, then the evaluation of the right operand is omitted; however, its type must still be boolean.

term:



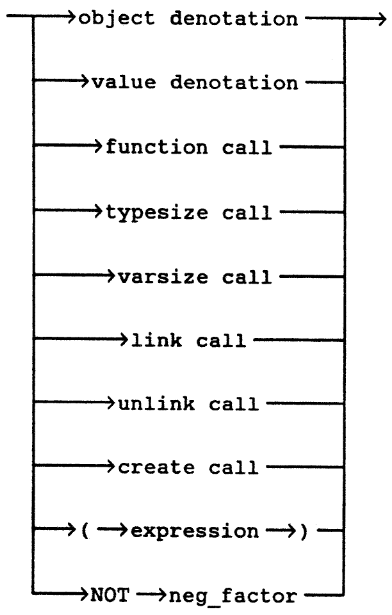
multiplication-type operator:



The evaluation of a term proceeds from left to right in the same fashion as for a simple expression (i.e. substitute 'factor' for 'term' and 'term' for 'simple expression' in the above description).

There is one exception to the general rule: if the left operand of the AND-operator is of type boolean and has value false, then the evaluation of the right operand is omitted; however, its type must still be boolean.

factor:



The type and value of a factor are obtained as described below for each of the possible forms:

object denotation:
The value of the factor is the value of the denoted object at the time of evaluation. The type of the factor is the type of the denoted object, except if this type is a subrange in which case the type of

the factor is the base type of the subrange. If the value of the object is undefined (not initialized) the effect of evaluating the factor is not defined.

value denotation:
The type and value of the factor are the type and value of the denoted value, cf. section 3.12.

| | |
|---|---|
| function call: | 4.2 |
| typesize call: | 3.2 |
| varsize call: | 3.11.2 |
| link call: | 9.1 |
| unlink call: | 9.1 |
| create call: | 9.1 |

The type and value of the factor are the type and value of the call, as described in the indicated section.

(expression):
The factor is evaluated by evaluating the expression. The type and value of the factor are the type and value of the expression.

NOT neg_factor:
the type and value of the factor are obtained by evaluating the neg_factor and applying the appropriate version of the NOT-operator to the result. The versions of the NOT-operator are described in subsections 3.4.1 and 3.4.3.

Note:
The precedence rules of Real-Time Pascal are those of standard Pascal which differ from the rules of other programming languages (e.g. ALGOL and PL/M languages).

Example:
As a consequence of the precedence rules the following is not a legal expression:

```
    0<x AND x<10
```

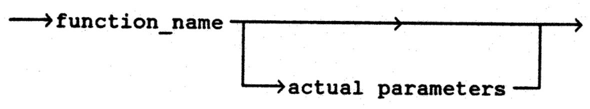
The expression should be written as:

```
    (0<x) AND (x<10)
```

## 4.2 Function Call

A function call causes a value to be computed by an activation of the indicated function.

function call:



The function name must be the name of a function, either predefined or programmer-defined. Evaluation of a function call takes place in two steps:

1. The actual parameters are evaluated in the order of occurrence.
2. An activation of the block associated with the function name is created and executed, cf. chapter 6.

The type of the function call is the result type of the function. The value of the function call is the value of the implicit result object associated with the activation of the function block when the execution of its action part terminates. If the value is undefined (no assignment) the effect of evaluating the function call is not defined.

Detailed rules for actual parameters are decribed in section 6.1.

## 4.3 Constant Expressions

Constant expressions can be evaluated at compile-time. Only constant expressions may be used in constant declarations. If all expressions used in a type definition are constant expressions the type is said to be static.

Constant expressions are recursively defined by the following restrictions.

1. All denoted objects must be constants.
2. Only the following (predefined) functions may be called: abs, chr, ord, pred, and succ.
3. Any typesize call must name a static type.
4. 'link call', 'unlink call', and 'create call' must not occur.
5. Factors of set types must not occur.

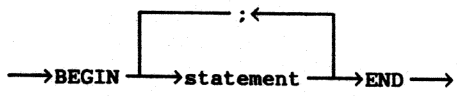
# 5. STATEMENTS

This chapter contains subsections describing the syntax and the use of the different statements which are included in the Real-Time Pascal language. Most of the statements are also found in standard Pascal and are well known language elements.

## 5.1 Compound Statement

The statements of a program describe the actions which are executed by an incarnation. These statements are collected in a compound statement.

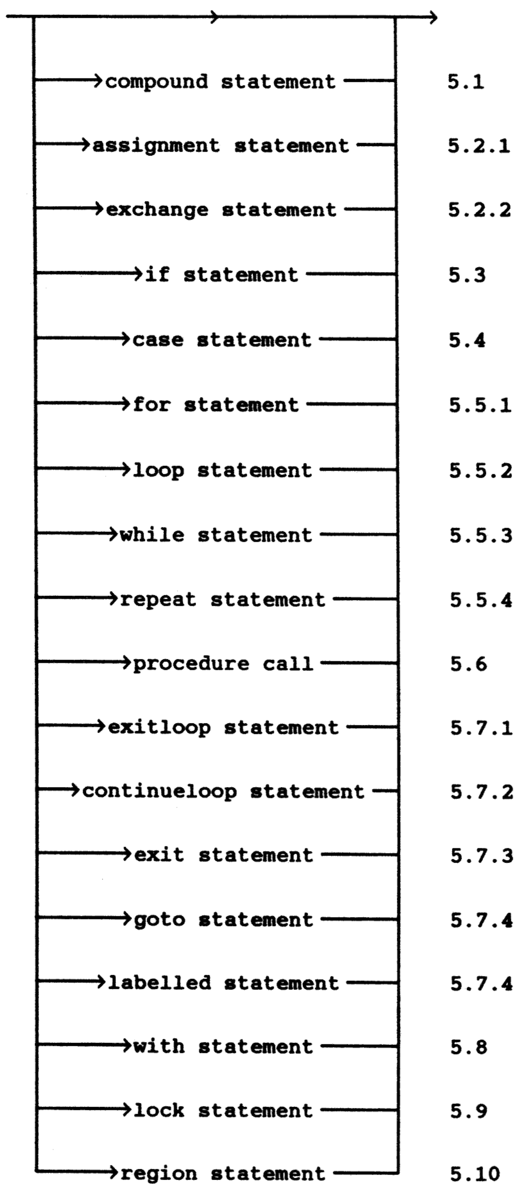
**compound statement:**



The statements are executed one at a time in the specified order. When all have been executed the compound statement has been completely executed or *exhausted.*

Below, all statement forms are given together with reference to their precise decription:

statement:



Note:
Statement may be empty.

## 5.2 Data Transfer Statements

Assignment and exchange statements are the basic building blocks for other types of statements. They serve to transfer values to objects.

### 5.2.1 Assignment Statement

The execution of an assignment statement causes the current value of a variable to be replaced with a new value specified by an expression. The right hand side expression must be of a type which is assignment compatible (cf. section 3.9) with the type of the variable.

```
assignment statement:

  ──→variable denotation ──→:= ──→expression ──→
```

The execution of an assignment statement takes place in four steps:

1) The variable denotation is evaluated as an object expression.

2) The right hand side expression is evaluated.

3) the run-time part of the type checking, including tests for range constraints, is performed. A fault occurs if some constraint is violated.

4) The value of the expression replaces the value of the variable.

Assignments cannot be made to variables of shielded types or of structured types with shielded component types.

Examples:

```
current_index:= current_index+1
catalog(current_index).author:= "Andersen H C"
matrix:= matrix_type(:(:1, 0, 0:),
                      (:0, 1, 0:),
                      (:0, 0, 1:):)
```

### 5.2.2 Exchange Statement

The values of two variables may be exchanged by the execution of an exchange statement.

```
exchange statement:

  ──→object denotation ──→:=: ──→object denotation ──→
```

The denoted objects must be variables, and they must be of the same type, cf. section 3.9. This type, or any of its components types, must not be mailbox, pool, chain, or port.

The execution of an exchange statement takes place in two steps:

1) The addresses of the left hand side and right hand side variables are evaluated, in that order.

2) The values of the objects located at the addresses evaluated in step 1 are interchanged.

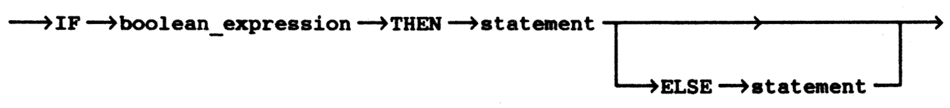
If the two variables are of type reference, external program, or process or if they have components of these types, the interchange takes place as an indivisible operation so that the integrity of references to messages and processes is preserved.

Examples:
```
current_message_ref:=:temp_message
matrix(i):=:matrix(j)
```

## 5.3 If Statement

An if statement selects for execution one of two (possibly empty) statements depending on the value of a condition. The expression specifying the condition must be of the predefined type boolean.

**if statement:**



The execution of an if statement takes place in two steps:

1) The value of the boolean_expression is evaluated.

2) If the expression evaluates to true the statement following THEN is executed. Otherwise the statement after ELSE (if present) is executed.

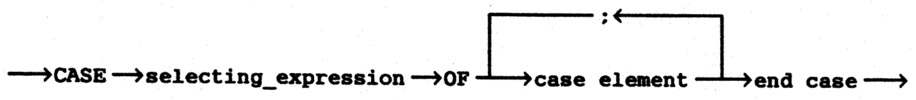
The ambiguous statement:
```
                IF e1 THEN IF e2 THEN s1 ELSE s2
```
is defined to be equivalent to:
```
                IF e1 THEN
                BEGIN
                  IF e2 THEN s1 ELSE s2
                END
```

Examples:
```
IF day=Saturday THEN
  day:= Sunday
ELSE
  day:= succ(day)

IF test THEN produce_test_record
```

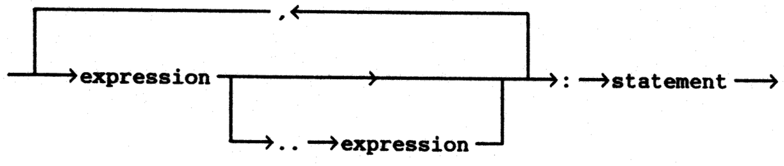
## 5.4 Case Statement

A case statement selects for execution one of a number of alternative statements, depending on the value of an expression. The expression must be of an ordinal type.

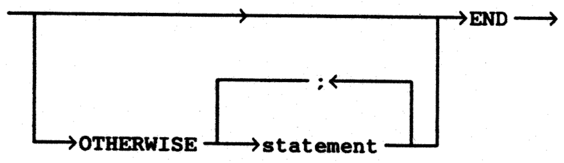
case statement:



case element:



end case:



A case element is a statement labelled by one or more constant expressions. These constant expressions, called case labels, must all be of a type compatible with that of the selecting expression. Consecutive values may be given as a range, e.g. first..last. All the case labels of one case statement must be distinct.

The execution of a case statement takes place in three steps:

1) Evaluation of the selecting expression.

2) The case element with the label corresponding to the value of the selecting expression is selected for execution.

   The "label" OTHERWISE (keyword) corresponds to all values of the type of the selecting expression which do not occur as case labels. A fault occurs if no case element corresponds to the value of the selecting expression.

3) Execution of the statement of the selected case element.

Note:
Upon completion of the selected statement the case statement is also completed.

Example:
```
CASE month OF
  January..May:          ....;
  October, December:     ....
OTHERWISE                ....
END (* of case *)
```

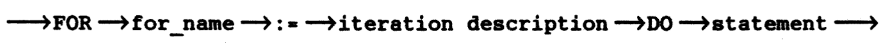
## 5.5 Repetitive Statements

A repetitive statement specifies that a statement is to be executed repeatedly, zero or more times.

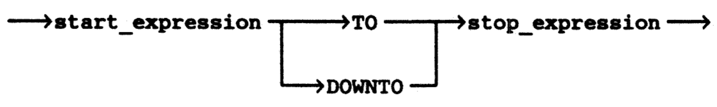
### 5.5.1 For Statement

A for loop may be used if a statement is to be executed a fixed number of times and/or elaborates on consecutive values of an object (iteration).

for statement:



iteration description:



The execution of a for statement takes place in five steps, where three may be repeated:

1) The start and stop expressions are evaluated and define an ordinal type, i.e. the expressions must be of the same type, which must be an ordinal type.

2) The *controlling variable* is allocated as an implicitly declared variable, local to the for statement. Its name is the 'for_name'. The type of the controlling variable is the type of the start and stop expressions. The initial value of the controlling variable is that of the start expression.

3) The termination condition is tested. That is, if TO is specified, execution of the for statement terminates when the value of the

controlling variable is greater than the value of the stop expression; if DOWNTO is specified, when the value of the controlling variable is less than the value of the stop expression.

4) The statement following DO is executed with the controlling variable acting as a constant (i.e. it must not appear on the left hand side of an assignment statement, nor may it be passed as a variable parameter of a routine call).

5) The value of the controlling variable is updated, except if it has already reached the upper or lower bound of the permissible range in which case execution of the for statement terminates immediately. The iteration can either be with increasing values of the controlling variable or with decreasing values. If TO is specified the ordinal value of the controlling variable is incremented in steps of one (succ). If DOWNTO is specified the iteration is with decreasing values (pred). The execution continues at step 3.

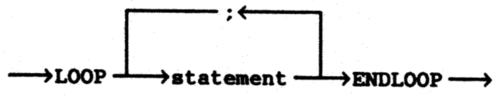
Note:
The two expressions are evaluated once, before the repetition. If the termination condition is satisfied before the very first repetition the statement (following DO) of the for statement is not executed at all.

Example:
```
FOR month:= January TO December DO
  FOR date:= 1 TO number_of_days(month) DO
    daily_activity(date,month)
```

## 5.5.2 Loop Statement

The loop statement constitutes an unconditional repetitive control structure, which may be used to define an "infinite" main loop of a program, only terminated in case of a fault or parent enforced process termination.

loop statement:



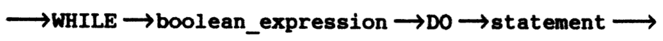
The loop statement specifies repeated execution of the statement sequence in the stated order. The loop may be left as the result of the execution of a jump statement (cf. section 5.7).

Example:
```
LOOP
  next_m:= read_next_message;
  prepare_message(next_m);
  send_message(next_m)
ENDLOOP
```

## 5.5.3 While Statement

Conditional repetition of a statement where the condition is checked before each execution may be achieved by means of a while statement.

**while statement:**



The value of the boolean_expression is evaluated before each execution of the statement. The test-and-execute sequence goes on as long as the evaluation yields true, when the result becomes false, possibly the first time, execution of the while statement terminates.

The while statement:
  WHILE exp DO st
is equivalent to the following combination of loop, if and exitloop statements:
  LOOP IF NOT exp THEN EXITLOOP; st ENDLOOP

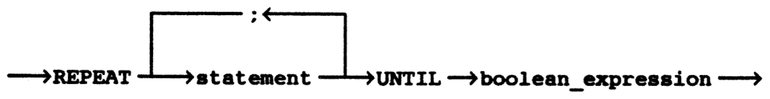
Example:

```
current_index:= first_index;
WHILE table(current_index)<>sought_element DO
  current_index:= current_index+1
```

## 5.5.4 Repeat Statement

Execution of a sequence of statements until some condition is satisfied may be achieved by means of a repeat statement.

**repeat statement:**



Every time the sequence of statements has been executed, the value of the boolean expression is evaluated and execution of the repeat statement terminates when the evaluation yields true.

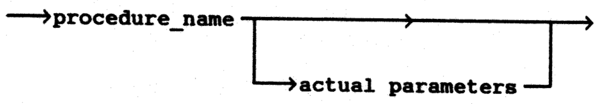
The repeat statement
  REPEAT s1; ..; sn UNTIL exp
is equivalent to the following special form of the loop statement:
  LOOP s1; ...; sn; IF exp THEN EXITLOOP ENDLOOP

Example:
```
REPEAT
  wait(ref, main_mailbox);
  final_message:= do_process(ref);
  return(ref)
UNTIL final_message
```

## 5.6 Procedure Call

A procedure call serves to establish a binding between actual and formal parameters, to allocate locally declared variables, and to invoke execution of the compound statement of the procedure block in its proper surroundings. A procedure call consists of the procedure name followed by a list of actual parameters. If the procedure is declared without formal parameters, the call consists of the procedure name only.

procedure call:



The execution of a procedure call takes place in two steps:

1) The actual parameters are evaluated in the order of occurrence.

2) An activation of the block associated with the procedure name is created, including parameter passing, and the action part of the block is executed (cf. chapter 6). When the execution terminates the procedure call is completed.

Detailed rules for actual parameters are decribed in section 6.1.

## 5.7 Jump Statements

The statements described in this section serve to explicitly modify the order of execution of statements by transferring control to an implicitly or explicitly indicated statement.

### 5.7.1 Exitloop Statement

Execution of an exitloop statement causes the execution of an enclosing repetitive statement (cf. section 5.5) to terminate.

exitloop statement:

The repetition exited is the innermost one. An exitloop statement may only appear within a repetitive statement, i.e. repeat, for, while or loop statement.

Example:
The generalized loop control structure may be constructed as a combination of a loop statement, an if statement, and an exitloop statement:

```
LOOP
  s_1_1; ...; s_1_n;
  IF bool_condition THEN EXITLOOP;
  s_2_1; ...; s_2_m
ENDLOOP
```

## 5.7.2 Continueloop Statement

The continueloop statement specifies that the remaining part of an iteration of a loop is to be skipped.

```
continueloop statement:

  ──→CONTINUELOOP──→
```

The statement applies to the innermost enclosing repetitive statement which must therefore exist. Depending on the kind of repetition statement (cf. section 5.5) the specific effect of the continueloop statement is:

for statement:
Step 4 terminates, and execution continues at step 5.

loop statement:
Execution continues with the first statement after LOOP.

while statement:
The remainder of the statement following DO is skipped. Execution continues with the evaluation of the loop condition.

repeat statement:
The remainder of the statement sequence up to UNTIL is skipped. Execution continues with the evaluation of the loop condition.

### 5.7.3 Exit Statement

An exit statement has the effect of a jump to the END of the compound statement of the enclosing block.

```
exit statement:

  ──→EXIT──→
```

Execution of an exit statement causes termination of the execution of the action part of the nearest enclosing routine or program block as if the compound statement had been exhausted.

Example:
```
BEGIN (* main program *)
  ...
  IF errors-detected THEN (* terminate the process *)
    EXIT;
  ...
END
```

### 5.7.4 Goto and Labelled Statement

The execution of a goto statement results in an explicit transfer of control to another statement specified by a label.

```
goto statement:

  ──→GOTO──→label_name──→
```

```
labelled statement:

  ──→label_name──→:──→statement──→
```

A labelled statement introduces the label name as denoting a label of the 'statement' following :.

A labelled statement is executed by executing the 'statement'.

Execution of a goto statement causes the normal order of execution of the statements within a compound statement to be broken. Execution is resumed at the labelled statement whose 'label_name' is identical to the one occurring in the goto statement. The label must be visible at the point where the goto statement occurs.

A goto statement cannot be used to transfer control from the outside into or from the inside out of the statement following DO of a for, with, lock, or region statement.

Note:
Goto into or out of a block is impossible.

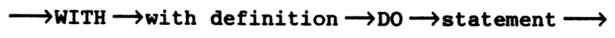
## 5.8 With Statement

A with statement may be used for three purposes:

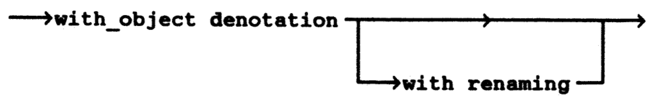
- shorthand notation for field access in a record object,
- object renaming,
- object retyping.

The latter is a facility allowing a programmer-defined type to be superimposed on a message buffer when applied in a lock statement.

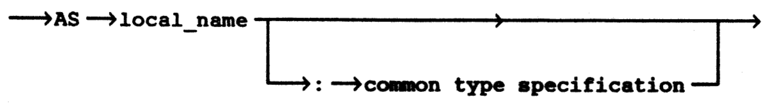
with statement:



with definition:



with renaming:



The denoted object is called the *with-object*. It must not be an irregular object (cf. section 3.8). The type specified by the 'common type specification', if present, is called the *local type*. The type of the with-object must not be protected, nor may the local type. The size of the local type must be less than or equal to the size of the with-object.

If the AS-part of a with statement is empty several with-objects may be listed. More specifically

WITH $d_1$, $d_2$, ..., $d_n$ DO st

is acceptable as shorthand for

WITH $d_1$ DO WITH $d_2$ DO ... WITH $d_n$ DO st

where the $d_i$ are object denotations.

A with statement is executed in three steps:
1. the denotation of the with-object is evaluated as an object expression, i.e. the address of the object is established;
2. the local type, if present, is established;
3. the statement following DO is executed observing the rules described below.

field access:
In the simple form of the with statement, i.e. without renaming, and if the type of the with-object is a record type and fname is a field name of this record type then

    fname

may be used as shorthand for

    obj.fname

where obj is an object denotation denoting the with-object.

renaming:
a non-empty AS-part is equivalent to a local declaration of an object, called the *local object*, with the same address as the with-object. The 'local_name' denotes the local object.

retyping:
The type of the local object is the local type, if specified; otherwise it is the type of the with-object.

Notes:
the with-object is not restricted to be of a record type, even (structured) constants are allowed.

The address of the object is evaluated only once before the statement following DO is executed.

The retyping of an object is a low-level facility of the language, intended for use in connection with messages whose exact type is not known beforehand (some of the type information may be part of the message buffer). But the facility may be used freely to achieve a relaxation of the otherwise rigorous object typing, which is one of the basic features of the language. This method demands an explicit and clear retyping stated where it is used in the program, in contrast to the standard Pascal solution using variant records.

The effect of assignment or exchange between partially overlapping objects is undefined.

Example:
Let rec_var be a record with a field named rec_field, and let local_type contain a field named loc_rec_field, then the fields may be accessed in the following ways in the statement following DO:

```
1) WITH rec_var DO (* the well-known standard Pascal form *)
   rec_var.rec_field or
   rec_field

2) WITH rec_var AS loc_var DO (* simple renaming *)
   rec_var.rec_field or
   loc_var.rec_field

3) WITH rec_var AS loc_var: local_type DO
   (* renaming and retyping *)
   rec_var.rec_field or
   loc_var.loc_rec_field
```

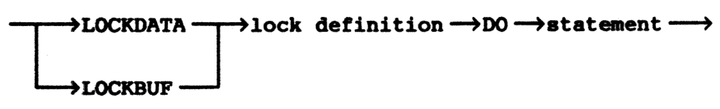
Exampel:

```
TYPE
  cat_record= RECORD
                title:  ....
                author: ....
              END;
VAR
  b_catalog: ARRAY(1..cat_size) OF cat_record;
  search_object: cat_record;
       ...
  WITH search_object AS s_o DO
    WHILE NOT found DO
      WITH b_catalog(current_index) DO
        IF author(* of b_catalog *)<>s_o.author then
          current_index:= current_index+1
        ELSE
          ...
```

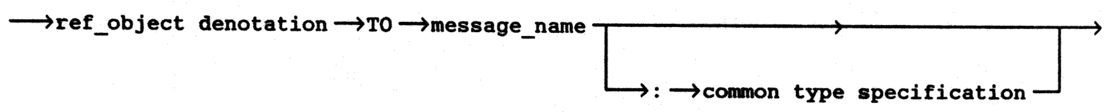
## 5.9 Lock Statement

The lock statement is the language construct which provides access to the actual contents of a message buffer, i.e. to the top non-empty message in a message stack.

lock statement:



lock definition:



The denoted object must be a variable of type reference or chain. If it is a reference its value must not be NIL, and if it is a chain it must not be empty. Otherwise a fault occurs. The message designated by the reference, or the current message of the chain, whichever applies, may be accessed in the statement following DO as an implicitly declared variable the name of which is specified by the 'message_name'. Either the whole buffer or only its data area is accessible, depending on the lockword (LOCKBUF or LOCKDATA).

If no retyping is specified for the buffer/data area the appropriate of the following types is assumed.

If the lockword is LOCKDATA the data area is accessible as a variable of a type belonging to the predefined family

```
dataarea(offset, top: 0..maxint)=
                PACKED ARRAY(offset..top-1) OF byte
```

where the parameter values are equal to the message attributes with the same names. The address of the variable as well as the parameter values are evaluated before the statement following DO is executed.

If the lockword is LOCKBUF the whole message buffer is accessible as a variable of a type belonging to the predefined family

```
buffer(bufsize: 0..maxint)=
                PACKED ARRAY(0..bufsize-1) OF byte
```

where the value of the parameter equals the size of the buffer.

If a lock statement is applied to a message stack with no non-empty message a fault occurs.

The following restrictions are imposed on the use of the locked variable while the statement following DO is being executed, including any routine calls made: If a reference, it must not be used as part of an exchange statement or as a parameter to signal, return, or release (cf. chapter 9), or be delivered to the IMC (cf. chapter 11). Whether a chain or a reference, it must not be used as a parameter to any of the message stack or chain manipulation routines (cf. chapter 10). However, it is legal to (dynamically) apply multiple locks to the same reference or chain (e.g. in a routine called from within a lock statement).

Example:

```
LOCKDATA ref TO data_part: my_data_type DO
    ...
```

## 5.10 Region Statement

The region statement provides access to shared variables and it is ensured that the access is exclusive.

```
region statement:

  ──→REGION ──→shared_object denotation ──→DO ──→statement ──→
```

The denoted object must be a shared variable. Associated with every shared variable is an *access count* which is initially zero. A region statement is executed in three steps.

1. Unless the process executing the region statement is already executing (dynamically inside) a region statement accessing the same shared variable it waits (is suspended) until the access count of the variable is zero. Subsequently the access count is incremented. If several processes wait for access to the same shared variable they observe a fifo discipline.

2. The statement following DO is executed. In this statement the shared variable may be accessed in the same fashion as an ordinary (private) variable.

3. The access count of the shared variable is decremented.

Example:

```
SHARED
  route_table: RECORD
    ...
  PROCEDURE close_down(node: node_ident);
    ...
    REGION route_table DO
      route_table.open_routes(node):= closed;
      ...
```

# 6. PROGRAMS AND ROUTINES

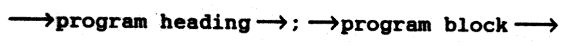
Programs and routines are very similar. Both have the general form of a heading followed by a block. Incarnations of program and routine blocks also exhibit fundamental similarities. In both cases the life of an incarnation has three stages: parameter passing, elaboration of declarations, and execution of an action part. The differences have to do with two aspects: control and environment.

An incarnation of a program block is a process on its own which lives autonomously except for the control exercised by its parent, whereas an incarnation, or *activation*, of a routine block is merely an episode in the life of a process. The activation, unless it chooses to loop infinitely or commits a fault, has no choice but to return to the point where it was called.

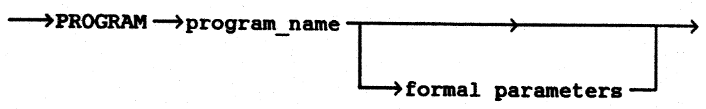
A process has no environment of data to access apart from predefined items and its parameters. A routine block activation, in addition to these, has its static surroundings: all the stack-allocated objects (including formal parameters) declared in enclosing blocks, except those which have been made invisible by redeclaration of their names, cf. chapter 8.

A program declaration may appear at the outer level of a 'compilation unit', cf. section 8.2, or in the declaration part of a program block. It serves to name and define a program.

**program declaration:**



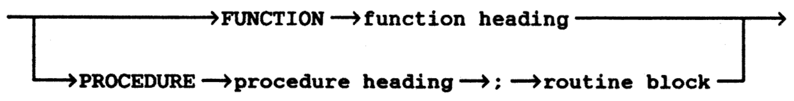
**program heading:**



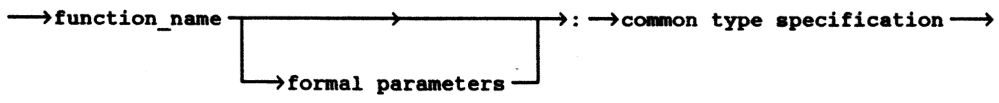
The heading specifies the name of the program and its formal parameters, if any.

A routine declaration may also appear at the outer level of a 'compilation unit', or in the declaration part of a block. It serves to name and define a routine.

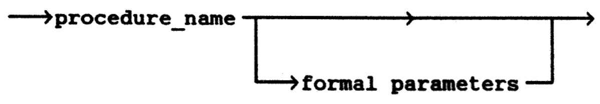
routine declaration:



function heading:



procedure heading:



The initial keyword in a routine declaration specifies whether the routine is a function or a procedure. In addition the declaration specifies the name of the routine, its formal parameters, if any, and in the case of a function, its result type, which must not be a protected type. The 'routine block' unless specified by one of the keywords EXTERNAL or FORWARD (see blow) is associated with the function or procedure name.

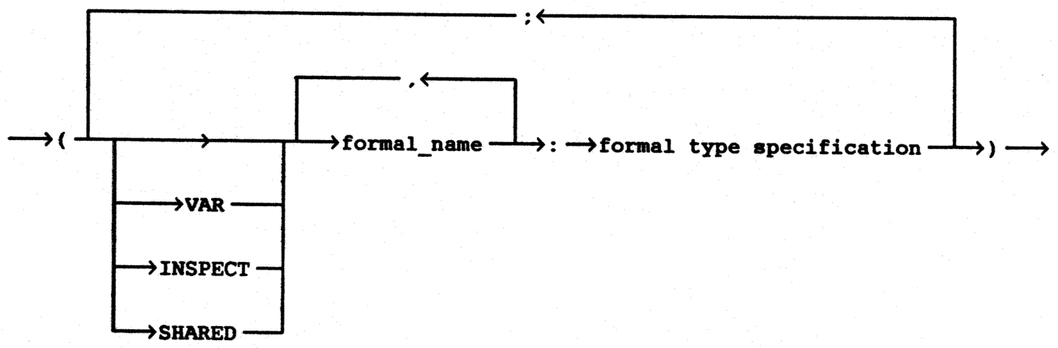
The parameters and blocks of programs as well as routines are described in the following two sections.

Examples:
PROGRAM router(INSPECT routs: route_table)
FUNCTION search_name(name: name_node): boolean
PROCEDURE insert_name(VAR pos: table_index; name: name_record)

## 6.1 Parameters

The 'formal parameters' of a program or routine heading specify the names, kinds and types of the formal parameters of the program or routine.

formal parameters:



Each 'formal_name' introduces one parameter. Several parameters may be named in a list, separated by commas. Such parameters have the same kind and type.

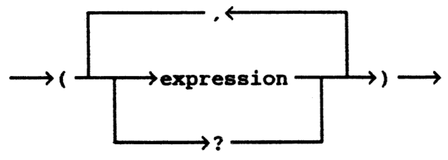
The kind of a parameter, which may be *variable, inspect, shared or value*, is specified by the (possible) keyword preceding its name. In the block of the program or routine each parameter acts as an object. The parameter kind determines how this object may be used:

| keyword | parameter kind | use of object |
|---|---|---|
| VAR | variable | as a private variable |
| INSPECT | inspect | as a constant |
| SHARED | shared | as a shared variable |
| none | value | as a private variable |

The type of each parameter, i.e. of the object which can be accessed in the block of the program or routine, is determined by the 'formal type specification' following the name of the parameter. If the 'formal type specification' is a 'parameterized-type_name', i.e. the name of a family of conformant types, the type of the formal parameter is determined for each incarnation of the program or routine by the type of the actual parameter.

An incarnation of a program or routine is created as a result of a 'create call', 'function call', or 'procedure call' being evaluated or executed. The call contains a description of the actual parameters to be bound to the formal parameters for the particular incarnation of the program or routine.

actual parameters:



Each actual parameter corresponds to the formal parameter in the same position in the 'formal parameters'. The number of actual parameters must equal the number of formal parameters. An actual parameter of kind variable, inspect, or shared must be of the same type as the corresponding formal parameter. An actual parameter of kind value need only be assignable to the formal parameter (cf. section 3.9). When the type of a formal parameter is specified as the name of a family of types the type of the corresponding actual parameter may be any type in that family.

The symbol ? may be used in place of an actual parameter expression when the parameter is not of kind value, regardless of the type of the formal parameter.

The binding of an actual parameter to the corresponding formal parameter takes place either by a value transfer ("call by value"), or by an address transfer ("call by reference") depending on the kind of the parameter. Value parameters are passed by value transfer, all other kinds by address transfer.

value transfer:
The value of the actual parameter becomes the initial value of the formal parameter which is allocated on the stack as an object local to the incarnation.

address transer:
The actual parameter is an object expression, cf. chapter 4. It must not denote an irregular object (cf. section 3.8). Evaluation of the actual parameter yields the address of an object of the parameter type. Throughout the life of the incarnation of the program or routine the formal parameter name will denote this object.

An actual parameter object passed to a process must be declared at the outer block level of the parent process, i.e. either it must itself be a process parameter or it must be declared in the program declaration part.

If the actual parameter is specified as ? there is no parameter object. If an attempt is made to access such a non-existing parameter object a fault occurs.

If the kind of the formal parameter is variable the actual parameter object must be a private variable or component of a private variable. If the kind of the formal parameter is shared the actual parameter object must be a shared variable. Conversely, if the actual parameter is shared, the kind of the formal parameter must also be shared.

The following restrictions apply to process parameters, i.e. to the formal parameters which occur in a program heading:
- parameters of kind variable must be of type pool, mailbox, or port;
- parameters (or components thereof) of pointer types must have mailbox as their base type, regardless of kind.

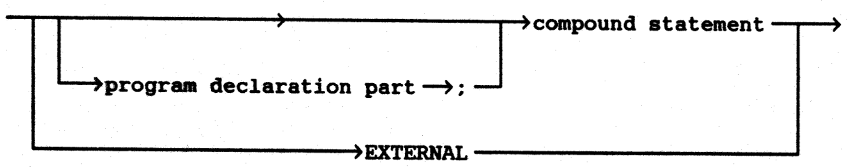
Note:
An actual process parameter of kind inspect may be a variable, and thus it may be changed by the parent process.

Example:
```
TYPE
   list= ARRAY(1..max_list_length) OF list_element;

   PROCEDURE handle_list(INSPECT lst: list);
   -- lst is of kind inspect to save time and space, and to
   -- allow the handling of constant lists
```

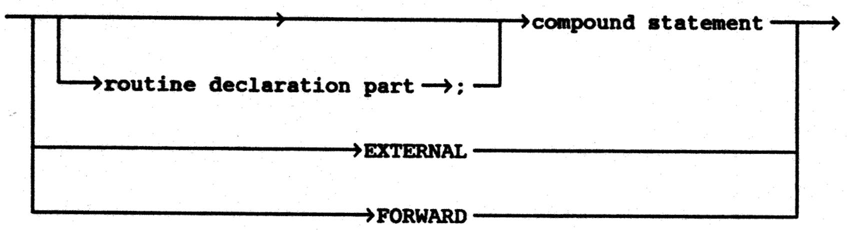
## 6.2 Incarnations of Blocks

A block, whether program or routine, consists of a declaration part and an action part which has the form of a compound statement.

program block:



routine block:



Blocks specified by one of the keywords EXTERNAL and FORWARD are described in subsection 6.2.2.

The declaration parts of program and routine blocks are slightly different in that certain forms of declarations may not occur in a routine block; see subsection 6.2.1.

An incarnation, whether of a program or routine, is created in the following three steps.

1) Allocation of the necessary (initial) amount of stack and heap. In the case of a program this means a whole new stack; in the case of a routine it means an *activation record* in the stack of the calling process.

2) Parameter passing, cf. section 6.1.

3) Elaboration of the declarations of the blocks, as described below.

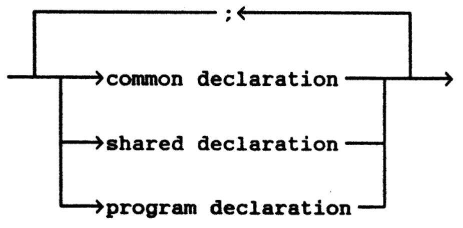
When an incarnation has been created the action part of the block can be executed, cf. function call (section 4.2), procedure call (section 5.6), and create call (section 9.1).

When execution of the action part of a routine terminates the values of all local reference variables must be NIL, otherwise a fault occurs.

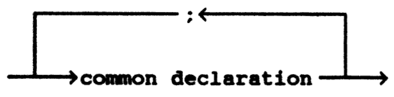
## 6.2.1 The Declaration Part

The declaration part of a block names and defines types, objects, routines and programs which are local to the block, i.e. not visible outside the block. The names introduced in the declarations may be used within the block to refer to the defined entities, cf. chapter 8.

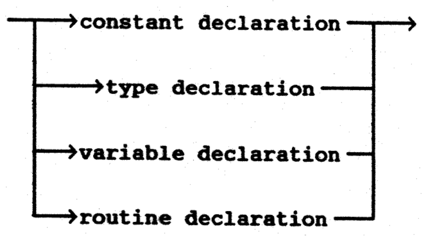
**program declaration part:**



**routine declaration part:**



common declaration:



Variables of shielded types, except reference, and variables which have components of these types may not be declared in a routine declaration part, i.e. such variables can only be declared in the outer block of a program.

The declarations in the declaration part of a block are elaborated in the order of occurrence. Elaboration of type declarations is described in section 3.2, elaboration of variable and shared declarations is described in subsection 3.11.2, and constant declarations need no elaboration at run-time.

The elaboration of a routine or program declaration causes all types defined in the 'formal parameters' to be established. When a program declaration is elaborated a *sub-program object* is allocated in the stack of the process being created, and associated with the program name specified in the declaration. Unless the program block is external, the sub-program object is linked (as if by an implicit link call, cf. chapter 9) to the block; otherwise it is initialized as having state unlinked.

The elaboration of the declaration part is performed for each incarnation of a block, and the names introduced in the declarations, when occurring in the remainder of the block, refer to those instances of the named entities which have been established or allocated when the particular incarnation of the block was created.

In the case of a function block elaboration of the declaration part, even if it is empty, includes establishment of the result type and allocation on the stack of an implicitly declared *result object*. The initial value of the result object is undefined, unless, it is of a pointer type, in which case the initial value is NIL.

Note:
The static environment of an internal program block is the same as that of the enclosing program block, i.e. only names in contexts specified for the compilation unit, cf. chapter 8, predefined names, and the names of formal parameters are known from the start of the block.

Elaboration of declarations is not the only time types may be established. Further types may be established when for, with, and lock statements are executed.

Example (of nested routines):

```
TYPE parity= (even, odd);

FUNCTION byte_parity(arg: byte): parity;

  FUNCTION even4bits(arg: 0..15): boolean;
    CONST table= (. 0, 3, 5, 6, 9, 10, 12, 15 .);
  BEGIN even4bits:= arg IN table END;

BEGIN (* byte parity *)
  IF even4bits(int(arg SHIFT (-4))) -- left half byte
    =even4bits(int(arg AND #HF)) -- right half byte
  THEN byte_parity:= even
  ELSE byte_parity:= odd
END
```

## 6.2.2 Forward and External Blocks

A forward announcement of a declaration containing the actual block of a routine may be given by using the keyword FORWARD in place of the routine block. When a forward announcement is used a declaration with an identical routine heading, i.e. all lexical elements identical, and an actual routine block (i.e. consisting of declaration part, which may be empty, and action part) must appear later in the same 'declaration part'. In this way it is possible to observe the rule of declaration before use, even for mutually recursive routines.

The block of a program or routine may be specified as external, i.e. separately compiled (cf. chapter 8), by the keyword EXTERNAL. An external program or routine may possibly be written in another programming language and compiled by a compiler for that language provided it is object code format compatible with the Real-Time Pascal compiler in question.

External sub-program objects may alternatively be declared as variables of an external program type

```
external program type:

  ──→EXTERNAL ──→PROGRAM ──→formal parameters ──→
```

The use of external program types is a simple way of defining more objects over the same sub-program heading. Either as variables of the same type, or as component type of an array, which may be dynamically sized depending on a program parameter (system configuration). Or a totally dynamic structure, by means of objects on the heap.

Note: the declaration
```
   PROGRAM prog(VAR mbx: mailbox); EXTERNAL
is equivalent to
   prog: EXTERNAL PROGRAM (VAR mbx: mailbox)
```

Example:
```
 TYPE
   child_type= EXTERNAL PROGRAM (VAR main_mbx: mailbox);
     ...
 VAR
   children: ARRAY (1..max_children) OF
               RECORD
                 ext_name, inc_name: string(12);
                 child: child_type;
                 child_handle: process;
                 ...

 WITH children(next) DO
 IF link(ext_name, child)=link_ok THEN
   IF create(inc_name, child(common_mbx), child_handle,
              0, stdpriority)=create_ok THEN
    ....
```

Due to the differences between routines and programs the linking of a program to a separately compiled block is somewhat different in the two cases.

During the execution of an incarnation of a program containing an external program declaration, the linking between the resulting sub-program object and the block of a separately compiled program is established as a result of the evaluation of an explicit link call, cf. chapter 9.

The association between the block of a separately compiled routine and the function or procedure name specified in an external routine declaration, i.e. the linking of the routine block to the program in which the external declaration occurs, must be established by a linkage editor before an incarnation of the program can be created. This can be done during a separate link-phase following compilation, or it can be done as a by-effect of program linking at run-time.

The amount and kind of checking of the agreement between the 'formal parameters' of an external program or routine declaration and the corresponding formal parameter specification of the separately compiled block, which is performed during linking, is implementation dependent. This is true for both cases of linking. In order to facilitate linking with programs written in other languages the parameter and result passing formats used by an implementation must be appropriately chosen and thoroughly documented.

Example (of mutually recursive routines):

```
PROCEDURE first(par1, par2: type1); FORWARD;

PROCEDURE second(par: par_type);
  ...
BEGIN
  ...
  first(act1, ct2);
  ...
END;

PROCEDURE first(par1, par2: type1);
  ...
BEGIN
  ...
  second(act);
  ...
END
```

## 6.2.3 The Action Part

Execution of the actions of a block means execution of its compound statement. This is described in chapter 5. Execution is *terminated* when the compound statement is exhausted, when an exit statement is executed, or when a fault occurs. When a process terminates it goes into a passive state where it remains until removed by its parent. When a procedure activation terminates its activation record is deallocated and a return is made to the caller, i.e. the procedure call is completed. When a function activation terminates its activation record is deallocated and the final value of the implicit result object is the value of the function call whose evaluation caused the activation.

# 7. FAULT HANDLING

A number of violations of the rules of Real-Time Pascal are referred to in this document as faults. Faults are errors which cannot, at least not in all cases, be detected at compile-time. Faults which are detected at compile-time cause the compiler to reject the source program. When a fault occurs at run-time, during the execution of an incarnation of a program, the following happens:

1. The *exception procedure* of the program is called with a fault code parameter indicating the kind of fault which occurred.
2. When (if) the exception procedure returns the process terminates and goes into a passive state as if execution of its action part had been completed.

Fault codes are implementation dependent and must be documented for each implementation.

## 7.1 Default Exception Procedure

Every implementation must include a default exception procedure which outputs a snapshot of the stack of the calling process and the fault code.

The default exception procedure may also be called to provide trace information about a process. Such an explicit call does not cause the process to terminate. The heading of the default exception procedure is:

```
PROCEDURE trace(fault: integer)
```

## 7.2 Programmer-defined Exception Procedure

When a procedure with the name exception and one integer-type parameter, i.e.

```
PROCEDURE exception(fault: integer)
```

is declared (internally) at the outer block level of a program, this procedure becomes the exception procedure of the program.

Note:
An ordinary call of a programmer-defined exception procedure does not cause a process to terminate.

# 8. NAMING ENVIRONMENTS

The rules described in this chapter serve to define for every point in a Real-Time Pascal source text the *naming environment* which is valid at that point. A naming environment is a set of names and a corresponding set of denotable program entities. In order for a name to be a member of a naming environment it must have an independent meaning, i.e. an occurrence of the name must denote a *program entity* irrespective of the syntactic context. A program entity is a value, a type, a family of conformant types, an object (which may in particular be a formal parameter), a label, a routine, or a (sub-)program. The visibility rules ensure that the entity denoted by a name is always uniquely determined.

In other words, the purpose of this chapter is to answer the question: When a name occurs at some point in a source text, what does it mean? and to ensure that the meaning is uniquely defined.

The unit of compilation for a Real-Time Pascal compiler is basically a sequence of routine and/or program declarations, referred to in the following as a *source text*. The visibility rules for names introduced in a source text are described in section 8.1, and the precise syntactic form of a compilation unit is described in section 8.2 along with a discussion of the role played by predefined names and names introduced in so-called contexts.

## 8.1 Visibility Rules

All names which occur in a source text must have one or more points of *introduction* each of which defines a program entity which can be denoted by the name within some region of the text, called the *visibility region* of the entity. The point of introduction may be a declaration, a formal parameter specification, a labelled statement, the AS-part of a with statement, the iteration description of a for statement, or a 'message_name' in a lock statement. Every introduction of a name has a *scope*, i.e. a region of text over which the introduction has an effect. The concept of scope serves as a tool in determining the visibility region of a named program entity.

Once the visibility regions of all program entities are known the determination of a naming environment proceeds as follows: Consider a name occurring at some point in a program. If the point is within the visibility region of a program entity whose name is the name under consideration, then the name denotes that program entity. Otherwise it has no meaning, i.e. it is not part of the naming environment.

There are two kinds of names which have no independent meaning, but whose meaning is dependent on the syntactic context, viz. the names of record fields and formal type parameters. However, because of the shorthand form of record field access allowed in with statements (cf. section 3.7) the names of fields of a with-object of a record type are treated as if introduced in the 'with definition'. Apart from this special case the names of record fields and type parameters are not members of naming environments.

Three kinds of rules which together make up the visibility rules are given below:

|   |   |
|---|---|
| - uniqueness rules: | rules which serve only to prevent ambiguous meanings of names, |
| - scope rules: | rules which determine the scope of an introduction of a name, |
| - exclusion rules: | rules which determine the visibility region of a program entity by explicit exclusion of sub-regions from the scope of the introduction of the name of the entity. |

## Uniqueness rules

The following names are said to be introduced *initially* in a block:

- the names of the formal parameters specified in the heading of the block,
- the names of program entities introduced in the declaration part of the block excluding the formal parameter lists and blocks of enclosed program and routine declarations, and
- the names of labels appearing in the compound statement of the block.

1) All names introduced initially in a block must be distinct.
2) All names introduced in the 'with definition' of a with statement must be distinct.

## Scope rules

1) The scope of a label or variable name is the compound statement of the block in which it is introduced.
2) The scope of any other name introduced initially in a block extends from the point of introduction to the end of the block. This is the rule which implies that in general a name must be introduced before use.
3) The scope of the 'for_name' introduced in a for statement, or of a name introduced in a with or lock statement is the 'statement' following DO in the for, with, or lock statement.
4) The scope of a record field name introduced as field access shorthand in a with statement is the statement following DO.

## Exclusion rules

An introduction of a name which occurs within the scope of a previous introduction of the same name is called a *reintroduction*.

The visibility region of a named program entity is the scope of the introduction of its name with the following possible exceptions:

1) Any inner program blocks.
2) The scope(s) of any reintroduction(s) of the name, i.e. reintroduction hides the entity denoted by the outer occurrence of the name.

Example:

Hiding of an entity by reintroduction of its name is illustrated below:

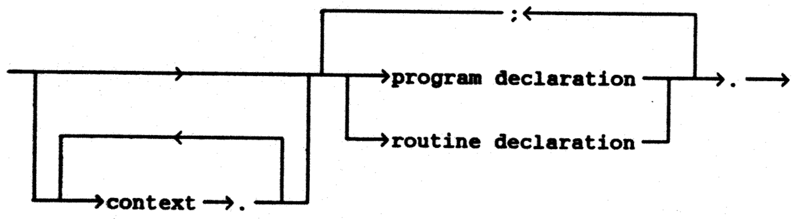
```
PROCEDURE p;
CONST
   n=2;
   ...
   PROCEDURE q;
   CONST
      n=5;
   BEGIN
      ... n ... (* has value 5 *)
   ...
END
```

## 8.2 Contexts and Predefined Names

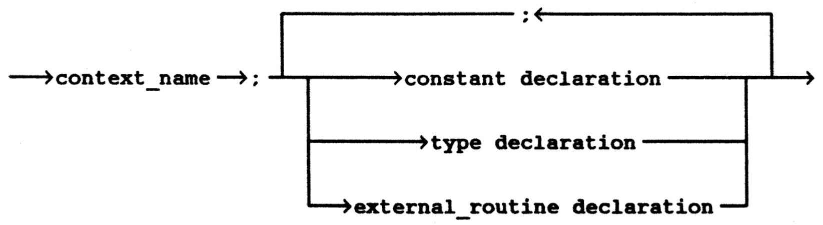
The initial naming environment of a compilation consists of the predefined names and entities and those introduced in contexts.

The unit of compilation is a sequence of program and/or routine declarations optionally preceded by one or more contexts. A compiler must allow contexts to be supplied as files separate from the source text proper.

compilation unit:



context:



All names introduced in one context must be distinct. The names introduced in contexts and the program entities defined at their points of introduction are treated as if they were introduced in a block surrounding the outer block(s) of the source text proper with one exception: the exclusion rule for inner program blocks does not apply. The scope of a name introduced in a context extends from the point of introduction to the end of the compilation unit. Routines declared in a context must be external, i.e. the actual block of a routine cannot be specified in a context.

The scope of names predefined as part of the language, cf. Appendix C, is the complete compilation unit. As with contexts the exclusion rule for inner program blocks does not apply to predefined entities. However, a predefined name may be hidden by reintroduction.

Note:
The exclusion rule for inner program blocks implies that types, constants, and routines which are to be common for several programs compiled as one unit must be specified in a context.

# 9. PROCESS CONTROL AND INTER-COMMUNICATION

This chapter describes the predefined language constructs available for control of offspring processes and for exchange of information between processes.

An incarnation of a sub-program is called a child of a (parent) process which is an incarnation of the program containing the sub-program declaration.

The navel string between the parent and the child is a variable of type process belonging to the parent. An arbitrary number of incarnations of a sub-program may be born; they are all controlled by the parent.

When a child is born it is supplied with actual parameters according to the formal parameter specification of the declaration, cf. section 6.1. Processes communicate via mailboxes or shared variables. A mailbox or a shared variable known by a parent may be made known as a parameter to its children (such a variable may either be owned by the parent or one of its ancestors). In this way a parent determines the communication paths of its children without, however, necessarily participating in the communication itself. Refer to section 5.10 for communication by means of shared variables and section 9.2 for mailbox communication.

## 9.1 Process Control

A sub-program declaration in a program implies the allocation of a sub-program object in the stack of an incarnation of the program. The states of a sub-program object are *linked* and *unlinked*, cf. section 6.2.1 for the initial state of a subprogram object.

The linking (i.e. change of state from unlinked to linked) is performed by a link call which is similar to a function call.

**link call:**

```
──→LINK──→(──→string_expression──→,──→program_name──→)──→
```

The type of the result of a link call is an implementation dependent predefined enumeration type
link_result= (link_ok, already_linked, external_not_found, ...).

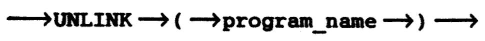
The purpose of a link call is to find a suitable program block matching the sub-program declaration. The value of the expression, which must be of a string type, is used to search for the program block in a fashion which is implementation dependent. The 'formal parameters' of the sub-program declaration may also be used in the match. If a program block is found it is linked to the sub-program object denoted by the 'program_name'.

The result of a link call indicates that the linking was successful or why it went wrong.

If the implementation and installation allows dynamic program load, the execution of a link call may involve the loading of a suitable program as well as the necessary linkage editing.

If a new program block is to be linked to a sub-program object, the former link must be broken, i.e. the state of the sub-program object has to be changed from linked to unlinked. This is done by means of an unlink call, which is similar to a function call:

**unlink call:**



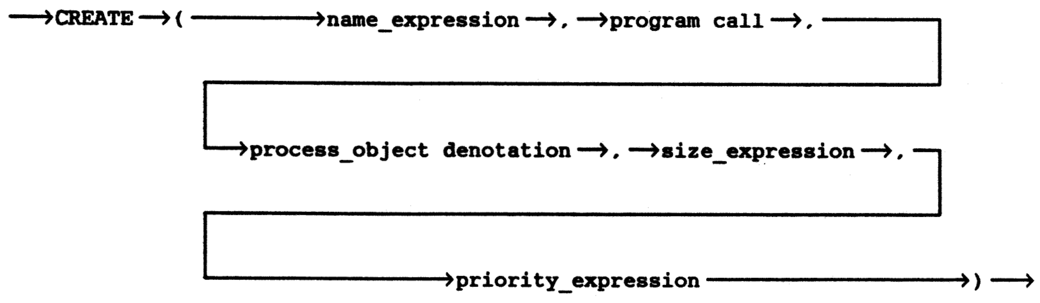
The result of an unlink call is of the implementation dependent predefined enumeration type
unlink_result= (unlink_ok, no_program_linked, existing_incarnations, ...)

After the call, if successful, the sub-program object may be linked anew (link call).

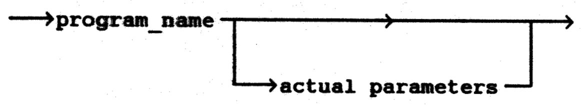
Processes may be created as incarnations of sub-programs in the linked state. When a process has been created it will begin to execute its actions. A process may become temporarily unable to execute actions for two reasons, which are independent of each other. It may be *waiting* for an event, cf. subsection 9.2.2, or *stopped*, cf. the predefined procedures stop and resume described below. A process is only able to execute actions if it is neither waiting nor stopped. The dynamic allocation of processor time to processes with the latter property is the scheduling function performed by the operating system or language-supporting nucleus.

A process is created by means of a create call.

**create call:**



program call:



The evaluation of a create call causes an incarnation of the program block linked to the sub-program object denoted by 'program_name' to be created as described in section 6.2. When a process has just been created it is neither stopped nor waiting.

The value of the 'name_expression', which must be of a string type, is attached to the created process for diagnostic purposes.

The 'process_object denotation' denotes the variable through which the calling process will control the child; it must be of type process. The value of this variable must be NIL before the call, otherwise a fault occurs. After the call it will be a reference to the child process.

The 'actual parameters' of the program call are bound to the corresponding formal parameters as part of the evaluation of the create call.

The initial size of the stack which is allocated for the created process is determined by the value of the 'size_expression' which must be of type 0..maxint. Size 0 means allocation of an area according to a value defined for the sub-program at compile-time, cf. chapter 12. The unit of measurement for stack size is implementation dependent.

The value of the 'priority_expression', of type prio_type, determines the execution priority of the created process. This quantity affects the way in which the process is scheduled for execution in a fashion which depends on the implementation. The type, prio_type, is an implementation dependent predefined enumeration type

    prio_type= (minpriority, stdpriority, maxpriority, ...)

The result of a create call has an implementation dependent predefined enumeration type

    create_result= (create_ok, no_memory, program_unlinked, ...)

If the create call was unsuccessful the result will indicate the reason. In this case no process will have been created and the value of the process variable remains NIL.

The predefined procedures stop, start, resume, and remove may be used to control a child, designated by a process variable passed as a parameter. If one of the procedures is called with this variable equal to NIL a fault will occur.

PROCEDURE stop(VAR pr: process)

The child process becomes stopped. If it is already stopped before the call there is no effect.

PROCEDURE start(VAR pr: process; prio: prio_type)

The priority of the child process is changed to the value of prio, and if the child process is stoppped a call of start makes it not stopped.

PROCEDURE resume(VAR pr: process)

If the child process is stopped a call of resume makes it not stopped; otherwise the call has no effect.

PROCEDURE remove(VAR pr: process)

A call of remove causes the child process to be removed, i.e. execution of its action part is terminated and all resources owned by the child are released and become free memory. The resources include stack, heap, and pools. Any ports owned by the process are closed (cf. section 11.1). Pools and messages are handled as follows:

1. All messages which have a pool owned by the child process as their home pool are marked for deallocation. A special treatment is given to a message stack whose top message is marked for deallocation, in the following situations:

   - A process calls release (cf. section 3.7) attempting to put the message in its home pool (in this case the message must be alone in the stack),

   - A process calls return (cf. subsection 9.2.2) attempting to place the stack in the return address mailbox of the top message,

   In both cases the special treatment is the following. First the top message is removed from the message stack and released to become free memory. Then the remainder of the stack, if non-empty, is placed in the return address mailbox of the new top message, except if this message is also marked for deallocation in which case the rule applies recursively. The event kind attribute of a message returned in this fashion becomes process_removed.

2. All message stacks accessed through reference or chain variables or presently placed in mailboxes owned by the child process are placed in the return address mailbox of the top message with the event kind attribute equal to process_removed. If the top message is marked for deallocation the above rule applies.

If the process to be removed has any children, these are removed before the process itself. This rule applies recursively. Thus, in effect, removal of a process means removal of that subtree of the complete process tree of which the process is the root.

After a call of remove the value of the parameter is NIL.

If a child process has a pool or a shared variable as parameter, it must not be removed by the parent process. If an attempt is made a fault occurs. However, if the parent process itself is removed, the child will also be removed by the recursion described above.

It is illegal to remove a child process while it is executing a region statement. An attempt to do so causes a fault.

Example:

```
link_res:= link("child1", my_child);
IF link_res<>link_ok THEN link_error(link_res);
create_res:= create("child_1_1", my_child(mbx1), process1, 0,
                    stdpriority);
IF create_res<>create_ok THEN create_error(create_res);
create_res:= create("child_1_2", my_child(mbx2), process2, 0,
                    stdpriority);
IF create_res<>create_ok THEN create_error(create_res);
  ...
stop(process6);
  ...
start(process2, highpriority); -- change priority
  ...
resume(process6);
  ...
remove(process4);
create_res:= create("child_2_4", my_child(alternative_mbx),
                    process4, 200, highpriority);
IF ...
  ...
```

## 9.2 Mailbox Communication

Messages may be passed among processes via mailboxes. A message is accessed through a variable of type reference or chain. At most one reference or chain variable holds a reference to a message at any time; thus mutually exclusive access to messages is ensured.

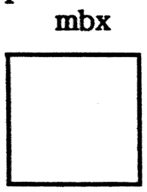
Access to the data contained in a message is only possible in a lock statement (cf. section 5.9).

The access right is exchanged between processes by means of the predefined routines signal, wait, waitdelay, and return as described in subsection 9.2.2.

### 9.2.1 Mailbox States

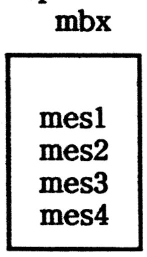
A mailbox may be regarded as a waiting room in one of the states passive, open, or locked:

passive:



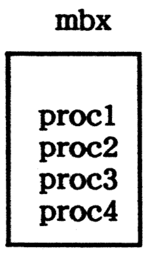
Neither messages to be accessed nor processes asking for access at the mailbox.

open:



A queue of messages (actually stacks) are ready for access, the first one will be removed from the queue when a process asks for a message, and the process will be given the access right.

locked:



A queue of processes are waiting for access to a message; the first one will get the access right to the next message arriving at the mailbox and will subsequently be removed from the queue.

There are three predefined boolean functions to inspect the state of a mailbox. However, it is not mandatory that these functions be implemented.

```
FUNCTION open(VAR mbx: mailbox): boolean
FUNCTION locked(VAR mbx: mailbox): boolean
FUNCTION passive(VAR mbx: mailbox): boolean
```

### 9.2.2 Communication and Synchronization Primitives

There are four predefined communication/synchronization primitives: signal, return, wait, and waitdelay. In addition the delay primitive may be used for purposes of synchronization or temporary process suspension.

Signal is performed by a call of the predefined procedure signal:

```
PROCEDURE signal(VAR ref: reference; VAR mbx: mailbox)
```

The value of ref must not be NIL or the reference locked (cf. section 5.9), otherwise a fault occurs. After the call, the message designated by ref is entered as the last element of the queue of messages belonging to mbx. If the mailbox is locked, the first process is removed from the list of waiting processes. This process is now allowed to complete the call of wait/waitdelay which caused its insertion in the list of the mailbox. The event kind attribute of the designated message becomes message_event.

The language provides two kinds of events which can be waited for:
- the arrival of a message at a specified mailbox.
- the expiry of a delay, specified as a number of milliseconds. An implementation need not, however, support such fine granularity. If the delay is specified as zero, the call (delay, waitdelay, or allocdelay) will return immediately.

A process may wait for one specific event or for the first one of two, one of each kind. It does so by calling one of the following routines.

PROCEDURE wait(VAR ref: reference; VAR mbx: mailbox)

The value of ref must be NIL prior to a call, otherwise a fault occurs. If the mailbox mbx is open, the first message stack in the list is removed and ref will designate this stack. If the mailbox state is locked or passive the process is suspended and entered as the last element of the list of waiting processes (FIFO strategy).

PROCEDURE delay(no_of_msecs: 0..maxint)

The process is suspended according to the value of the parameter.

```
FUNCTION waitdelay(VAR ref: reference; VAR mbx: mailbox;
                        no_of_msecs: 0..maxint): activation
```

The value of ref must be NIL prior to a call, otherwise a fault occurs. If the mailbox mbx is open, the first message stack in the list is removed and ref will designate this stack. If the mailbox state is locked or passive the process is suspended until one of the events eventually occurs. The result indicates the reason why the process is activated; its type is the (implementation dependent) predefined enumeration type

    activation= (a_mailbox, a_delay, ...).

Getting a message from a pool, cf. section 3.7, is similar to receiving a message from a mailbox. A special version of alloc is therefore available which allows the specification of a maximum delay which a process will tolerate.

```
FUNCTION allocdelay(VAR r: reference; VAR p: pool;
                    VAR ra: mailbox; no_of_msecs: 0..maxint):
                    activation
```

The description of alloc, cf. section 3.7, applies also to allocdelay with the modification that in case the delay specified by no_of_msecs expires before the calling process obtains a message the call will return the result a_delay and otherwise have no effect. If a message is obtained the result will be a_mailbox.

Return is performed by a call of the predefined procedure return:

PROCEDURE return(VAR ref: reference)

The call: return(ref)
is equivalent to: signal(ref, ret_address)
where ret_address denotes the return address of the message designated by ref, cf. section 3.7, except for the value of the event kind attribute which becomes answer_event.

Notes:
A mailbox may be inspected without having to wait if it is empty by calling waitdelay with zero delay.

An implementation must secure indivisibility of the communication primitives.

Example:
The following communication flow is possible by means of signal and return. Each process $p_i$ must know the (mailbox) address of process $p_{i+1}$. However, process $p_n$ does not know process $p_1$.

$$\begin{tikzcd}
p_1 \arrow[r, bend left, "\text{signal}"] & p_2 \arrow[r, bend left, "\text{signal}"] & \cdots \arrow[r, bend left, "\text{signal}"] & p_n \arrow[lll, bend left, "\text{return}"]
\end{tikzcd}$$

# 10. MESSAGE MANIPULATION

Two structures for organizing messages are supported by the language: stacks and chains. A message stack is the general form of a message, where a stack with only one message is a special case. Access to a stack may be transferred between processes via a mailbox (cf. section 9.2). The chain structure is intended for local organization of messages in a process. The elements of a chain are message stacks. They are accessed by a handle: a variable of type chain. The elements of a message stack are messages. They are accessed by a handle: a reference variable (or directly by a chain in the case of the current message). The manipulation of message stacks is described in section 10.1 and of chains in section 10.2.

## 10.1 Message Stacks

A stack is one or more messages, organized as a lifo list so that manipulation affects the youngest member only. Access to the data in a stack of messages is achieved by means of a lock statement (cf. section 5.9).

Access to the attributes of messages in a stack can only be made to those of the top (youngest) member. A special treatment is given to the buffer attributes when empty messages are pushed onto or popped from a stack, ensuring that these attributes will apply to the topmost non-empty message in the stack, since only non-empty messages can be accessed in a lock statement.

A stack may be manipulated by calling the predefined procedures push and pop. The parameters to these procedures must not be locked. Otherwise a fault will occur at the time of call.

PROCEDURE push(VAR new_top, stack_handle: reference)

The parameter new_top must designate a message stack with only one element, otherwise a fault occurs. The message stack designated by stack_handle, possibly an empty stack (i.e. the handle is NIL), is extended with the message designated by new_top as the new top element, i.e. the return address and the home pool of the stack becomes those of the new top element. This makes the flow control, as illustrated in the example below, possible. After the call stack_handle will designate the new stack, and the value of new_top will be NIL.

If the new top message is empty, and the stack was not empty before pushing, the message attributes, size of the buffer, offset, top, and byte count, are copied from the old to the new top message.

PROCEDURE pop(VAR popped_mes, stack_handle: reference)

The value of popped_mes must be NIL, and the value of stack_handle must not be NIL; otherwise a fault occurs. The result of a call of pop is: the top message in the stack is removed, popped_mes will designate a stack consisting of the removed message only, and stack_handle will designate the remaining part of the stack. If the stack had only one element its value will be NIL.

If the popped message is empty, its message attributes, size of the buffer, offset, top, and byte count, are all set to zero.

The depth of a message stack as well as the number of non-empty messages can be obtained by using the predefined functions stackdepth and bufcount.

```
FUNCTION stackdepth(VAR stack: reference): 0..maxint
FUNCTION bufcount(VAR stack: reference): 0..maxint
```

The value returned by stackdepth is the total number of messages in the designated stack, including empty ones, whereas bufcount yields only the number of non-empty messages (cf. lock statement, section 5.9). Both functions return the value 0 if the parameter has value NIL.

Example:
The following flow of messages may be achieved by means of the paired operations: (push, signal) and (pop, return).

$$\begin{tikzcd}
p_1 \arrow[r, bend left, "\text{(signal)}"] & p_2 \arrow[l, bend left, "\text{(pop, return)}"] \arrow[r, bend left, "\text{(push, signal)}"] & \cdots \arrow[l, bend left, "\text{(pop, return)}"] \arrow[r] & p_{n-1} \arrow[l] \arrow[r, bend left, "\text{(push, signal)}"] & p_n \arrow[l, bend left, "\text{(return)}"]
\end{tikzcd}$$

## 10.2 Message Chains

The initial state of a chain object is empty, i.e. the length of the chain is 0. Cyclic lists are built and manipulated by means of predefined routines only. The following actions may be performed on chains: insert an element, extract an element, change current message, update start point, and read the length.

Two elements of a chain have a special status:

- the *start point* of the chain is the first element put into the chain or an element explicitly selected as the start point.

- the *current message (stack)* is the element which may be extracted or accessed in a lock statement and the only element whose message attributes may be read or changed.

The predefined routines eventkind, resetevent, u1, u2, u3, u4, setu1, setu2, setu3, setu4, bufsize, offset, top, bytecount, setoffset, settop, and setbytecount which are described in section 3.7 may all be called with a chain object as parameter, in which case they apply to the current message stack (top message) of the chain.

The length of a chain is read by means of a call of the function chainlength:

```
FUNCTION chainlength(VAR ch: chain): 0..maxint
```

For all the routines described in the remainder of this section the parameter (chains and references) must not be locked, except for the chain of chainenqueue. Otherwise a fault occurs at the time of call.

A message stack is inserted into a chain by means of a call of chainenqueue:

```
PROCEDURE chainenqueue(VAR ref: reference; VAR ch: chain)
```

The value of ref must not be NIL, otherwise a fault occurs. The stack designated by ref becomes the new predecessor of current message of the list, and the value of ref becomes NIL.

The current message may be removed from a list by means of a call of chaindequeue:

```
PROCEDURE chaindequeue(VAR ref: reference; VAR ch: chain)
```

The value of ref must be NIL, otherwise a fault occurs. If the length of the chain ch is 0, a fault occurs. The current message is removed from the chain and will be designated by ref. The successor of the removed element becomes the new current message. If the start point is removed from a list with more than one element the successor of the removed element becomes the new start point.

The current message may be moved one step up or down the list, or moved to the start point by calling the procedures chainup, chaindown and chainstart respectively:

```
PROCEDURE chainup(VAR ch: chain)
PROCEDURE chaindown(VAR ch: chain)
PROCEDURE chainstart(VAR ch: chain)
```

The result of a call of chainup/chaindown/chainstart is that the successor/predecessor/start point becomes the new current message of the list.

The current message of a list may be made the new start point of a list by a call of chainreset:

```
PROCEDURE chainreset(VAR ch: chain)
```

Example:
Let ch be the handle of a sorted chain of messages:

```
chainstart(ch);
FOR count:= 1 TO chainlength(ch) DO
BEGIN
  chaindequeue(work_ref, ch);
  push(work_ref, result_stack)
END(*FOR*);
signal(result_stack, mail_center)
```

# 11. IMC FUNCTIONS

The concept of a resident module is introduced in /DSA/ as a well defined entity, which is self-contained in terms of resources. In Real-Time Pascal a resident is thus a tree of processes whose root owns the pool and port resources for the IMC services. However, the concept of resident is not reflected in the syntax of Real-Time Pascal or in rules enforced by the language.

Residents exist in a distributed environment and communicate with each other by using the standard inter module communication (IMC) services /DSA-IMC/. The embedding of these services into Real-Time Pascal is the scope of the present chapter. In the remainder of the chapter the software components at the IMC nodes plus the physical interconnection media which together provide the IMC services are referred to as the IMC network or just the IMC.

A resident gains access to IMC services via objects of the type port. Port names, in turn, establish the identification of residents towards the IMC network and towards one another. Port names consist of a maximum of 12 graphic characters. Communication between two residents takes place as transfers of strings of bytes from messages belonging to a sender to messages belonging to a receiver. An actual transfer within the IMC network takes place when both residents have delivered a message to the IMC.

The principal means of communication is connections between two ports. On a connection the IMC perform flow control, correct sequencing, and undamaged data transfer.

The relationship between invokation and completion of IMC functions is asynchronous. IMC services are invoked by calls of predefined service request routines. In most cases a call of a request routine causes the transfer of a message (or two) from the calling resident to the IMC. When the requested function has been carried out this message, called an *event* message, will be returned, i.e. to its return address mailbox, with relevant result information and possibly containing received data. The return of an event message to a resident is called an *IMC event*, or just an event. The interface between residents and the IMC has been designed so that no change, which is of significance to a resident, in the state of a port or connection endpoint can take place without the occurrence of an event, unless it occurs during a call of a request routine and as a direct consequence of this call. In other words, despite the asynchronous nature of the interface, residents always have complete up-to-date information about their ports and connections.

During the time-span from an event message has been transferred to the IMC until it is eventually returned it is said to be outstanding. An outstanding event message may be further characterized by the kind of event it is intended to retrieve. All IMC events correspond to values of the predefined enumeration type event_type which is given in section 3.7. All IMC events are described in the following sections.

In some cases the IMC will return an event message even though the intended event has not occurred, most often because circumstances change so that it never will. This is called a dummy event. As an example, an outstanding data message is returned as a dummy event when the connection to which it pertains is unexpectedly removed. A number of event_type values are used exclusively for dummy events. Each of these values indicates the event which the dummy event message had been set up to retrieve. The correspondence between dummy event kinds and intended event kinds is shown below:

| dummy event | intended event |
|---|---|
| dummy_lcnct | local_connect |
| dummy_rcnct | remote_connect |
| dummy_rindic | reset_indication |
| dummy_rcmpl | reset_completion |
| dummy_credit | credit |
| dummy_sent | data_sent |
| dummy_arrived | data_arrived |

In addition to request routines Real-Time Pascal provides routines which can decode event information by inspecting the IMC message attributes used to carry this information.

The IMC service request routines are described in the following three sections. With a single exception (connect, see subsection 11.2.2) the reference type parameters to these routines must not have value NIL or be locked when a call is made; otherwise a fault will occur. These parameters are used to transfer event messages to the IMC, and they always have value NIL after a call.

The IMC never changes the size, offset, top, or u-attributes of an event message. With the exception of actual data messages (send, receive, receiveall) IMC event messages may be empty.

The unit of data to be sent by the IMC (send) is always the contents of the data area of the send message in question. Similarly a received data unit (receive, receiveall) is always placed in the data area of the receive message, and the number of bytes it comprises is assigned to the byte count attribute.

The data area description of the message passed by a call of a data transfer routine must be consistent, cf. section 3.7. Otherwise a fault occurs.

## 11.1 Ports

An IMC port is represented in Real-Time Pascal as a variable of type port. The state of a port is either open, in which case the port is known in the IMC network by a name which is published according to its scope, or closed (or closing, see under closeport below).

An open port contains a number of connection end-points (cf. section 11.2). The number must not be greater than an implementation dependent maximum, cf. section 11.4).

The attributes of a port, i.e. name, scope, and number of endpoints, are defined when it is opened by a call of openport:

```
PROCEDURE openport(VAR p: port; VAR closemes: reference;
                   INSPECT name: string; scope: scope_type;
                   no_of_cons: 0..maxint; cntrl: control_type)
```
where p is the port to be opened. The parameter closemes designates the port_closed event message which will be outstanding while the port is open. The values of the name and scope parameters specify the name of the port and the range within the IMC network in which to publish the name. The value of no_of_cons is the number of connection end-points requested for the port. If this number is greater than the maximum allowed or if the length of the name is greater than 12 a fault will occur.

The type of the scope parameter is the predefined enumeration type

```
    scope_type= (anonymous, local, regional, global).
```

The type of cntrl is a predefined descriptive type

```
    control_type= PACKED RECORD
                    rcv_all,
                    get_credit,
                    ?, ?, ?, ?, ?, ?: boolean
                   END
```

For a further discussion of scopes, see /DSA-IMC/.

The call causes the port to be opened as requested. The state of the port must be closed when the call is made, otherwise a fault occurs. After the call the state of the port is open. The port_closed event will occur when it is eventually closed with one of the following reasons (cf. section 11.3):

| | |
|---|---|
| reason_ok: | the resident called closeport |
| reason_name: | a name-conflict arose |
| reason_resource: | resource problem somewhere in the IMC network. |

If the control parameter's field rcv_all has the value true, the general receive feature (cf. receiveall, subsection 11.2.2) is enabled for the port; otherwise it is not. And if the value of its field get_credit is true, the general flow control feature (cf. getcredit, subsection 11.2.2) is enabled for the port; otherwise it is not.

A port is withdrawn from the IMC network by a call of closeport:

```
PROCEDURE closeport(VAR p: port)
```

where p is the port to be closed. If the port is already closed the call has no effect. Otherwise it proceeds as follows. All connections on the port are removed with graceful completion of any feasible data transfers as described for disconnect, cf. subsection 11.2.2. The disconnected events occur with reason_closed. However, at the remote end of a connection the disconnected event occurs with reason_ok. Then all general receive messages are returned as dummy events, and finally the port_closed event occurs with reason_ok. The port is made unknown to the IMC network, its state changes to closed, and it may subsequently be re-opened, possibly with a different set of attributes. While connections are being removed and messages returned the port is said to be closing.

## 11.2 Connections

Connections are the principal means of data transfer between residents. A connection joins two connection end-points, each contained in an open port. Before a connection can be established one of the two residents must make a connection end-point available by calling getconnection (cf. subsection 11.2.2). The resident which calls getconnection is the passive part in the establishment of the connection and need not know the name of the remote port. The other resident is the active part which must specifically request that a connection be established by calling connect (cf. subsection 11.2.2).

An end-point within a port is identified in a request call by a pair (p, index) where p is a port and index is a number in the range 1..n, where n equals the value of no_of_cons given when the port was opened. This fact accounts for the first two parameters of all the request routines described in this section (except receiveall). If a request call is made with an index parameter which is out of range a fault occurs.

Notice that end-point indices are local to a port and not related to indices or even known at the remote end-points of connections.

Each kind of request call which may be made concerning a connection end-point is only permitted provided the end-point is in an appropriate state. The applicable states are:

| | |
|---|---|
| free: | the end-point is free for use by calling connect or getconnection. Free is the initial state of connection end-points contained in a newly opened port, |
| accept_remote: | getconnection has been called. The end-point will leave this state when a connection is established from a remote port, or when disconnect is called, |

| | |
|---|---|
| connected: | a connection has been or is being established, and data transfer may take place, |
| resetting: | reset has been called while the state was connected. The state will revert to connected when the resetting procedures have been completed, |
| disconnecting: | disconnect has been called, and the process of removing a previous connection is under way. The state will subsequently become free. |

If the end-point identification (p, index) given in a request call is such that the port is closed or closing, or the state of the end-point is free, the connection to which the call pertains is said to be *absent*. This phenomenon may occur if the event which informs the resident about the removal of a connection or the closing of a port has not been processed at the time of call.

It is a general rule that if a request call is made concerning a connection end-point which is in a wrong state for the call a fault occurs, except if the connection is absent.

### 11.2.1 Connection Administration

A connection end-point is made available for remotely initiated connection establishment by a call of getconnection:

```
PROCEDURE getconnection(VAR p: port; index: 1..maxint;
                        VAR compl, disc: reference)
```

If the port is closed or closing the message designated by compl is returned as a dummy event and the message designated by disc as a disconnected event with reason_closed, and the call has no further effect. Otherwise a call of getconnection is only permitted if the state of the end-point is free; after the call it is remote_accept. The message designated by compl will be returned as a remote_connect event when a connection has been established. The message designated by disc will be the disconnected message of the connection end-point. It will be outstanding until the end-point becomes free again.

In order to establish a connection between a local connection end-point and a remote end-point, which has been made available as described above, a resident must call connect:

```
PROCEDURE connect(VAR p: port; index: 1..maxint;
                  VAR compl, disc: reference;
                  INSPECT remote_name: string;
                  service: conn_service)
```

If the port is closed or closing the message designated by compl is returned as a dummy event and the message designated by disc as a disconnected event with reason_closed, and the call has no further effect. Otherwise the call is only accepted if the state of the indicated end-point is free.

The value of remote_name is the name of the remote port to which a connection is requested. If the length of the name is greater than 12 a fault occurs. The state of the (local) end-point becomes connected, i.e. data transfer calls may be made immediately following the call of connect. This does not imply that the actual path through the IMC network has been established at this time, nor in fact that it ever will be.

The value of the parameter compl may be NIL, i.e. the designated message is optional. If it is present it will be used to generate a local_connect event when the actual connection has been successfully established. This event is purely informative and not associated with any change of state. If the local_connect message is present, but establishment of the connection fails, the message will be returned as a dummy event.

The message designated by disc is the disconnected event message. It will be outstanding until the state of the connection end-point eventually reverts to free or the port is closed. If this happens because it turns out to be impossible to establish the actual connection the reason will be either reason_name, if a port with the specified name could not be found, or reason_resource, if the necessary resources were not available in the IMC network or at the remote port (no end-point with state accept_remote).

The service parameter selects the service class of the connection. Two service classes are defined: normal and high. The treatment of service classes depends on the implementation of the IMC network. The type of the service parameter is the predefined enumeration type

```
    conn_service= (cs_normal, cs_high).
```

A connection end-point may always be freed by a call of disconnect:

```
PROCEDURE disconnect(VAR p: port; index: 1..maxint)
```

If the indicated connection is absent or if the state of the connection end-point is disconnecting the call has no effect. Otherwise if the connection end-point is engaged in a connection, i.e. if its state is connected or resetting, the connection will be removed. This will be done gracefully, i.e. all data transfers for which credit is available are completed before the remaining outstanding messages are returned. At both ends of the connection all outstanding messages except the disconnected event messages are then returned as dummy events. While this takes place the state of the (local) end-point is disconnecting. Finally the disconnected events occur (at both ends) with reason_ok, and the state becomes free.

If the state of the connection end-point is accept_remote when disconnect is called the remote_connect message will be returned as a dummy event, the disconnected event will occur with reason_ok, and the state of the end-point becomes free.

The only reason for removal of a connection which has not been discussed above (this includes removal when a port is closed, cf. section 11.1) is failure in some component of the IMC network. If a connection is broken for this reason the disconnected event will occur with reason_network after all other outstanding messages have been returned as dummy events.

## 11.2.2 Connection-based Data Transfer

The routines for data transfer on connections and for control of data messages: send, receive, receiveall, getcredit, reset, and getreset, are described in this subsection.

The following rule holds for calls of all these routines except receiveall: If the connection indicated by the first two parameters is absent the message designated by the third parameter is returned as a dummy event and the call has no further effect. Otherwise the call is only permitted if the state of the end-point is connected (or resetting in the case of getreset).

A receive message is made available for a specific connection by a call of receive:

```
PROCEDURE receive(VAR p: port; index: 1..maxint;
                   VAR datames: reference)
```

The designated message becomes an outstanding receive message for the connection. At the remote end of the connection the call may cause either a credit or a data_sent event to occur. If both kinds of event message are outstanding only the data_sent event will occur. When a unit of data has been transferred from a remote send message to the receive message the latter is returned as a data_arrived event. If necessary, the received data unit is truncated to fit into the data area of the receive message, and in this case the event will be data_overrun.

If the general receive feature is enabled for a port (cf. the cntrl parameter of openport, section 11.1) a general receive message for the whole port, not dedicated to a particular connection, may be delivered to the IMC by a call of receiveall:

```
PROCEDURE receiveall(VAR p: port; VAR datames: reference)
```

If the port is closed or closing the designated message is returned as a dummy event and the call has no further effect. If the general receive feature is not enabled a fault occurs. Otherwise the message becomes an outstanding general receive message. It may be used for any connection within the port. It will eventually be returned as data_arrived or data_overrun, depending on whether the received data unit had to be truncated to fit into the receive data area. The index of the connection end-point to which the data belong will be an attribute of the message (cf. section 11.3). In case of network failure which causes a connection on the port to be removed the message may be returned as a dummy event.

A send message containing a unit of data to be sent on a specified connection is delivered to the IMC network by a call of send:

```
PROCEDURE send(VAR p: port; index: 1..maxint;
               VAR datames: reference)
```

The designated message becomes an outstanding send message. If a receive message is available at the remote end, or when one becomes available (call of receive or receiveall), the indicated data unit is transferred to the receive data area. Then the send message is returned as data_sent, and the receive message at the remote end as data_arrived (or data_overrun).

If the general flow control feature is enabled for a port (cf. the cntrl parameter of openport, section 11.1), flow control information pertaining to a specified connection can be obtained from the IMC by a call of getcredit:

```
PROCEDURE getcredit(VAR p: port; index: 1..maxint;
                    VAR credmes: reference)
```

The designated message becomes an outstanding credit message. It is returned as a credit event when there is one or more outstanding receive messages at the remote end of the connection (call of receive) for which credit has not been given previously. The number of such receive messages is passed as the message attribute credit count (cf. section 11.3). However, a credit event can only occur provided there is at least one outstanding reset_indication message at the connection end-point, cf. getreset.

Data and credit messages which are outstanding at an end-point of a connection may be taken back without breaking the connection by calling reset:

```
PROCEDURE reset(VAR p: port; index: 1..maxint;
                VAR compl: reference)
```

In the same graceful fashion as when a connection is removed all data transfers for which credit is available are carried out first. Then the remaining data and credit messages are returned as dummy events, but only at the local end-point. The return of receive messages may amount to the taking back of credit which has already been given to the remote resident. In this case the resulting negative credit is passed as the credit count attribute of a reset_indication event.

Following a call of reset the state of the (local) connection end-point will be resetting. The message designated by compl is the reset_completion event message. This event occurs after all outstanding data and credit messages have been returned. When it occurs the state of the end-point reverts to connected.

When a connection end-point has been reset the IMC has also reset its account of credit previously passed to the resident. Credit information obtained before resetting is therefore no longer valid.

If credit information is used it is necessary to know when credit is taken back as a result of resetting of the remote end-point. A reset_indication message which is used to carry this information is made available to the IMC by a call of getreset.

```
PROCEDURE getreset(VAR p: port; index: 1..maxint;
                   VAR indic: reference)
```

The call is permitted if the state of the connection end-point is connected or resetting, but does not change the state. The message designated by indic becomes an outstanding reset_indication event message. That is, when a reset occurs at the remote end of the connection causing loss of credit this message is returned as reset_indication with a credit count equal to the number of credits that have been taken back.

At least one reset_indication message must be outstanding before a credit event can occur, cf. getcredit. Therefore a call of getreset may trigger a credit event.

## 11.3 IMC Message Attribute Decoding

Whenever a message is returned as an event the event kind attribute is set to indicate the kind of event. This attribute may be read using the predefined function eventkind, cf. section 3.7. Depending on the event kind the attributes index, credit count, and reason may be meaningful. These attributes may be read using the three functions described below. A fault will occur if one of these functions is called with a parameter with value NIL.

```
FUNCTION index(VAR r: reference): 0..maxint
```

gives the index in the relevant port of the connection end-point to which the event pertains. It is defined for messages with event kind local_connect, remote_connect, disconnected, reset_completion, reset_indication, data_sent, data_arrived, data_overrun, credit, and for dummy events. If the event is dummy and there is no applicable index the result will be 0.

```
FUNCTION reason(VAR r: reference): reason_type
```

gives the reason for an event. It is relevant for messages with event kind port_closed or disconnected. The result is of the predefined enumeration type

```
    reason_type= (reason_ok, reason_name, reason_resource,
                  reason_closed, reason_network).
```

All uses of reason values are explained in the preceding sections.

FUNCTION creditcount(VAR r: reference): 0..maxint

If the event kind is credit the result is the number of receive messages available at the remote end-point of the connection. If the event kind is reset_indication the result is the number of credits which have been taken back by a call of reset at the remote end-point.

## 11.4 Miscellaneous Routines

At each node in an IMC network, the IMC imposes a restriction on users of the IMC services, namely a maximum number of connections to any one port. This limitation can be obtained by calling the predefined routine maxconnections:

FUNCTION maxconnections: 0..maxint

# 12. COMPILER DIRECTIVES

Directives to a Real-Time Pascal compiler may be regarded as lexical separators. They have the general form:

    $ directive-name parameters end-of-line

given on a separate line. Alternatively directives may be supplied as parameters in the call of the compiler. Some of them must be specified before the first line of actual source text is met, viz. either in the compiler call or as $-directive lines in front of the outermost program/routine heading.

The following table lists the mandatory directives. An implementation may support additional directives. The first three directives which are all used to control the appearance of a compiler listing must appear before the first line of actual source text.

| name | parameters | description |
|---|---|---|
| PAGELENGTH (default 45) | number | maximum number of lines per page of listing, |
| PAGEWIDTH (default 120) | number | maximum number of characters per line of listing, |
| EJECT | none | force the start of a new page of the listing |
| TITLE | "char.string" | the character string is placed in the title field of the header line of each page of the listing, |
| SUBTITLE | "char.string" | the character string is placed in the subtitle line of each page of the listing, |
| CODE | none | causes code generated for the following lines of source text to be listed, |
| NOCODE (default) | none | suppresses listing of generated code, |
| CREATESIZE | number | determines stack size of incarnations of the following program(s) if the value of the size_expression of the create call is 0, cf. section 9.1, (implementation dependent) |
| INDEXCHECK (default) | none | causes checking of subrange constraints before indexing to be included in code generated for the following lines of source text, |

| name | parameters | description |
|---|---|---|
| NOINDEXCHECK | none | switches off index checking, (implementation dependent) |
| LIST | none | causes the following lines of source text to be listed, |
| NOLIST (default) | none | suppresses listing of source text, |
| RANGECHECK (default) | none | causes checking of subrange constraints before assignment to be included in code generated for the following lines of source text, |
| NORANGECHECK | none | switches off range checking, (implementation dependent) |
| ACCESSCHECK | none | causes code to be generated for every access to a formal parameter object which is not of kind value, to check the existence of the actual parameter (not specified as ?). |
| NOACCESSCHECK (default) | none | switches off access checking. |
| SET | switch assignment list | see below, |
| IF | expression | see below, |
| ENDIF | none | see below, |
| ELSE | none | see below, |
| ELSEIF | expression | see below, |

Switches are compile-time variables which can - except from parameters of the transfer function getswitch (cf. 3.12) - only be used in the expressions occurring in IF and ELSEIF directives. A switch assignment list consists of one or more switch assignments separated by commas. A switch assignment has the form (number must be integer, no radix allowed):

```
    name = number
```

A switch assignment either introduces and sets the value of a switch, or, if the switch has been introduced in a previous switch assignment (SET directive), merely changes the value.

The directives IF, ELSE, ELSEIF, and ENDIF provide the capability for conditional compilation. The expressions occurring in IF and ELSE-IF directive must be boolean expressions obeying the standard rules of the language. The operands must be switches and integer numbers. The following operators may be used: <, <=, >=, >, =, <>, AND, OR, XOR, NOT.

The directives for conditional compilation may be used to selectively exclude blocks of lines of source text from compilation, i.e. to cause such lines to be treated as comments. When listed, each excluded line will be marked with
-- appearing at the beginning of the line.

The directive IF and ENDIF must always be used in matching pairs. ELSE and ELSEIF may optionally be used in conjunction with IF and ENDIF. A use of conditional compilation takes the following form:

$IF expr$_1$

st$_1$

$ELSEIF expr$_2$

st$_2$

$ELSEIF expr$_3$

st$_3$

...

$ELSEIF expr$_n$

st$_n$

$ELSE

st$_{n+1}$

$ENDIF

The only mandatory parts are the IF and ENDIF directive lines and the source line(s) st$_1$. The effect is as follows: If the values of all the expressions are false then the source lines st$_1$, st$_2$, ..., st$_n$ are excluded from compilation. Otherwise let k be the smallest number such that the value of expr$_k$ is true. Then st$_1$, ..., st$_{k-1}$, st$_{k+1}$, ..., st$_n$, and st$_{n+1}$ (if present) are excluded from compilation.

Conditional compilation may be used at several nested levels. In the above terminology any of the st$_i$ may thus include repeated use of conditional compilation.

Notes:
Switches and variables in the program text belong to separate name spaces and cannot be confused. The same names may be used.
Directives occurring in comments are ignored, in particular those occurring in source lines excluded from compilation.

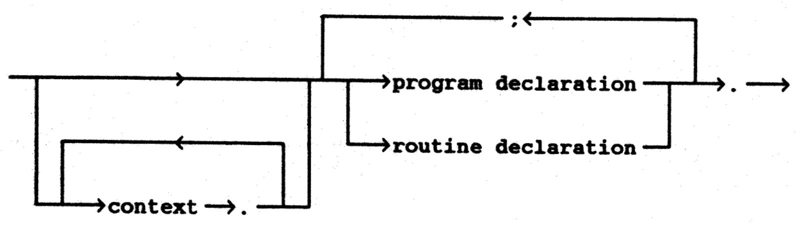
# A. REFERENCES

| | |
|---|---|
| /Pascal/ | The Programming Language Pascal<br>Acta Informatica, 1, 35-63, 1971<br>N. Wirth |
| /ISO Pascal/ | ISO International Standard ISO/IS 7185:<br>Specification for Computer Programming Language Pascal |
| /ISO char.set/ | ISO International Standard ISO/IS 646:<br>7-bit Coded Character Set for Information Processing Interchange |
| /DSA/ | RCSL No 42-i1982:<br>Distributed System Architecture<br>A Conceptual Framework for Systems Design |
| /DSA-IMC/ | RCSL No 42-i1983:<br>DSA Inter Module Communication<br>Functional Description |
| /Platon/ | Platon, A High Level Language for Systems Programming<br>RECAU, R-75-59<br>Jørgen Staunstrup and Sven Meiborg Sørensen |

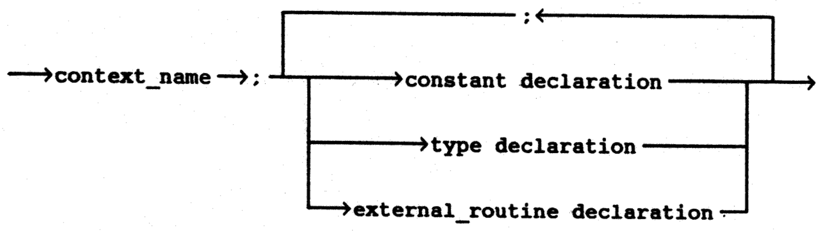
# B. SYNTAX DIAGRAMS

This appendix contains all the syntax diagrams of the definition of Real-Time Pascal in an attempted natural top-down reading order. All syntax diagram titles may be found in the catchword index (appendix D) marked with a trailing :.

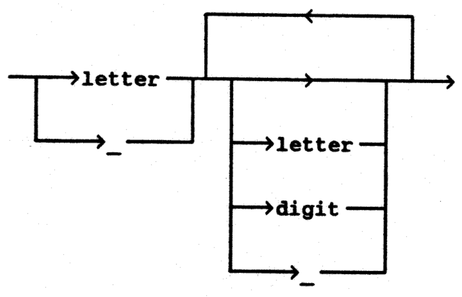
**compilation unit:**



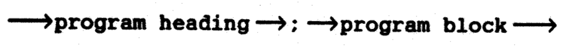
**context:**



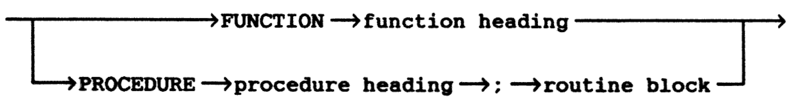
**name:**



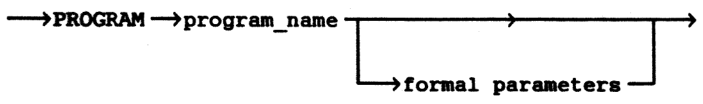
**program declaration:**



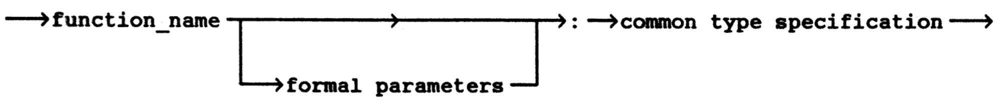
**routine declaration:**



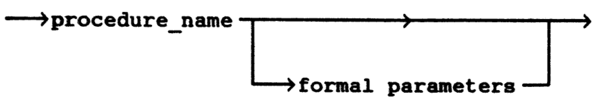
**program heading:**



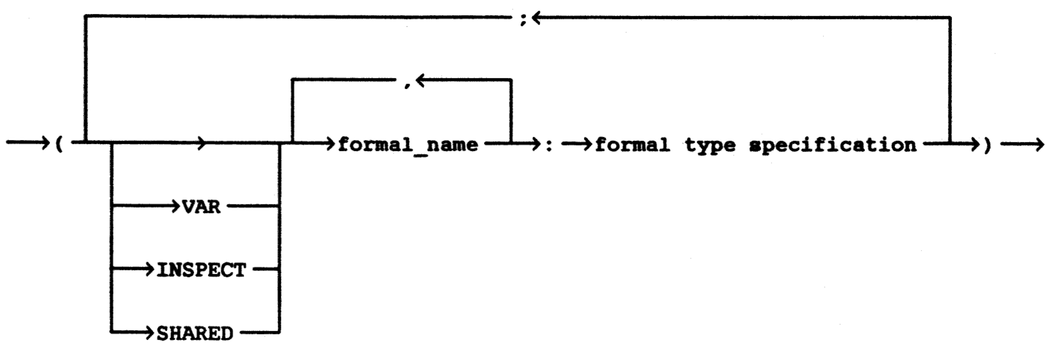
**function heading:**



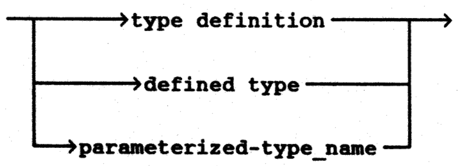
**procedure heading:**



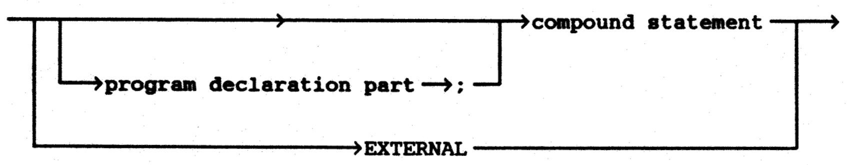
**formal parameters:**



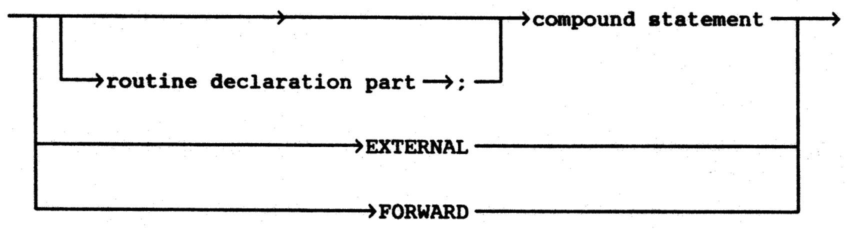
formal type specification:



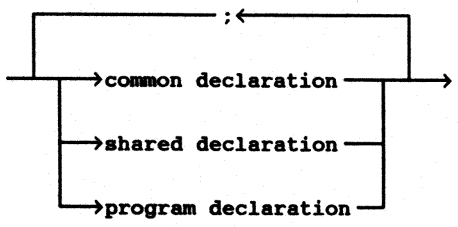
program block:



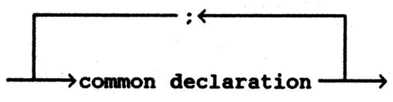
routine block:



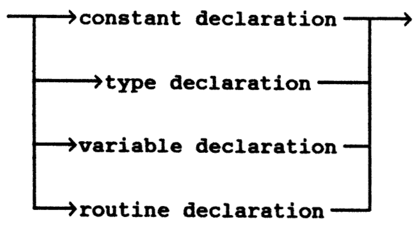
program declaration part:



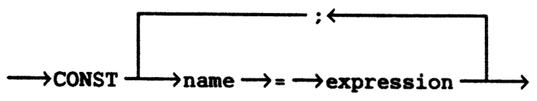
routine declaration part:



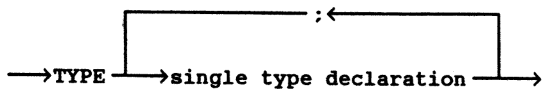
common declaration:



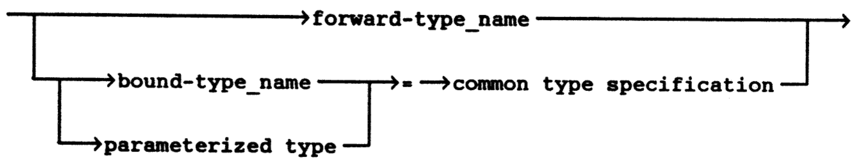
constant declaration:



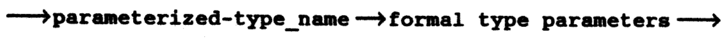
type declaration:



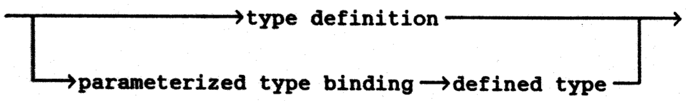
single type declaration:



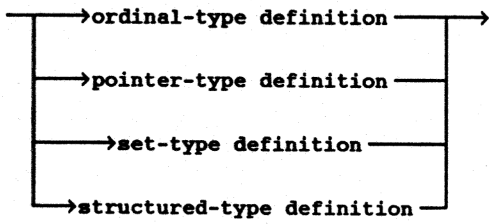
parameterized type:



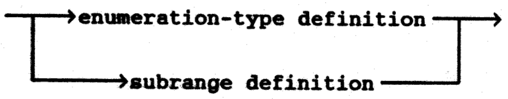
**common type specification:**



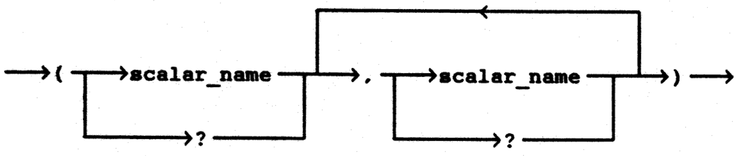
**type definition:**



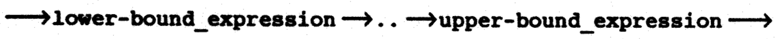
**ordinal-type definition:**



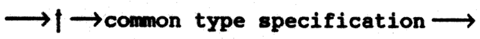
**enumeration-type definition:**



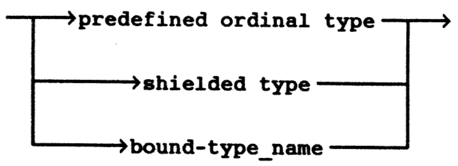
**subrange definition:**



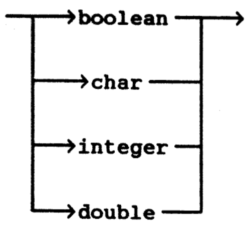
**pointer-type definition:**



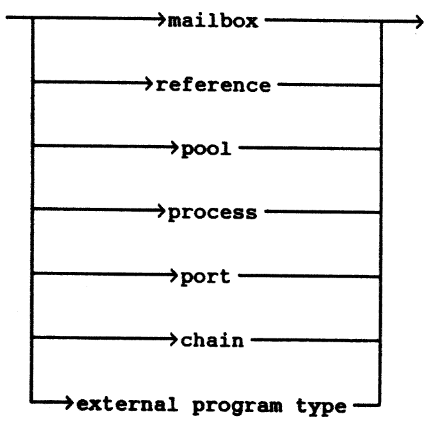
**defined type:**



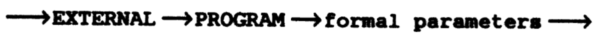
**predefined ordinal type:**



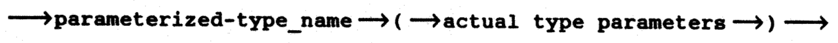
**shielded type:**



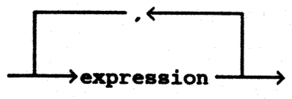
**external program type:**



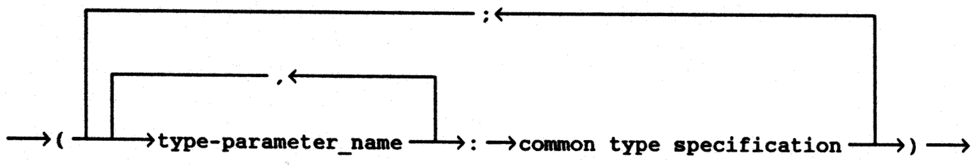
**parameterized type binding:**



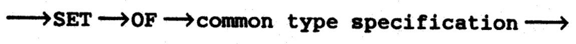
**actual type parameters:**



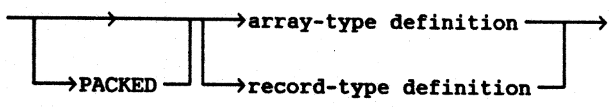
**formal type parameters:**



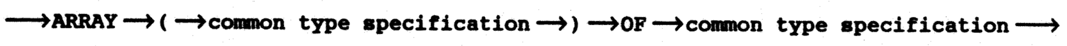
**set-type definition:**



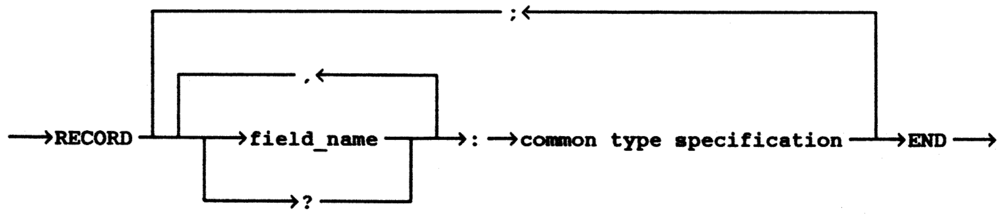
**structured-type definition:**



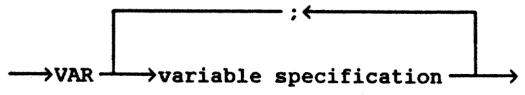
**array-type definition:**



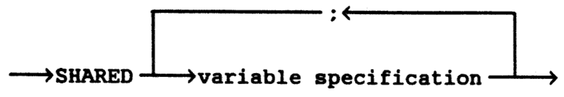
**record-type definition:**



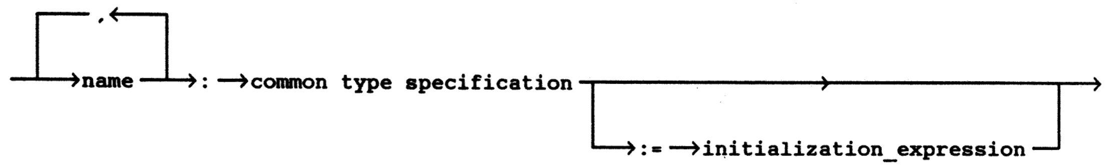
**variable declaration:**



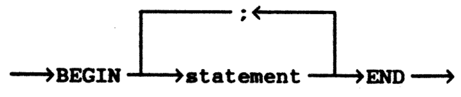
**shared declaration:**



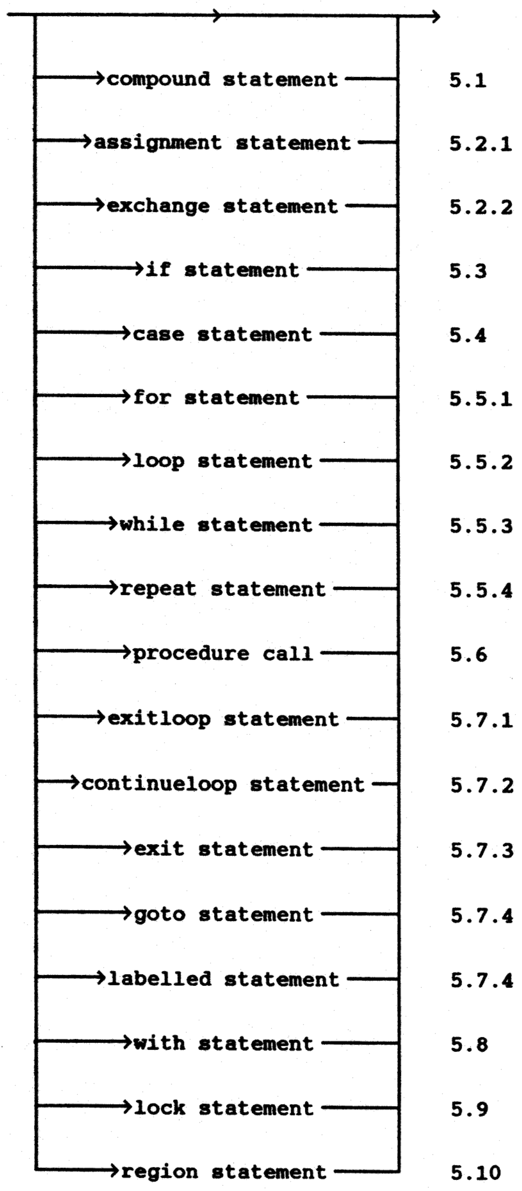
**variable specification:**



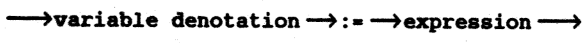
**compound statement:**



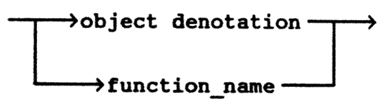
**statement:**



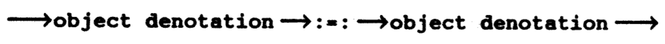
**assignment statement:**



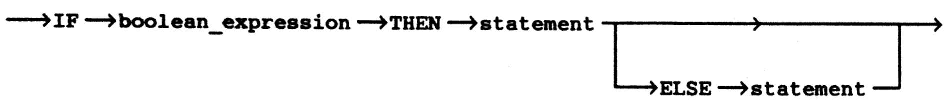
**variable denotation:**



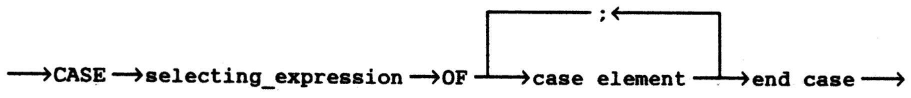
**exchange statement:**



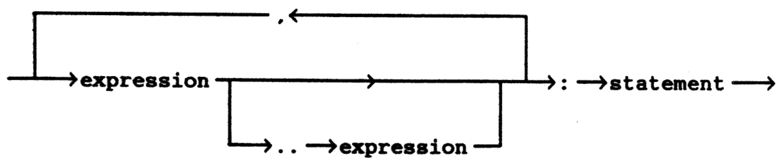
**if statement:**



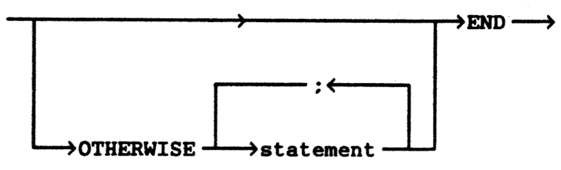
**case statement:**



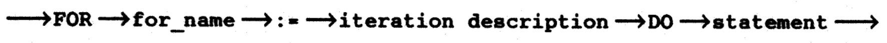
**case element:**



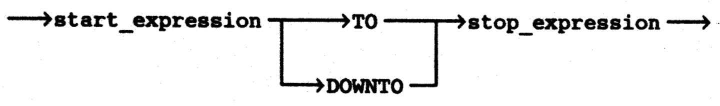
**end case:**



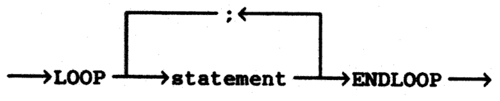
for statement:



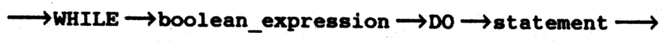
iteration description:



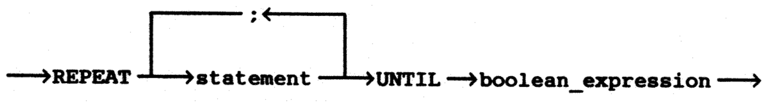
loop statement:



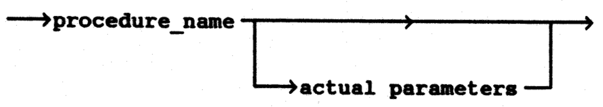
while statement:



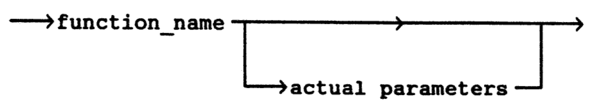
repeat statement:



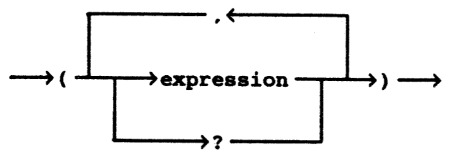
procedure call:



**function call:**



**actual parameters:**



**exitloop statement:**

⟶EXITLOOP⟶

**continueloop statement:**

⟶CONTINUELOOP⟶

**exit statement:**

⟶EXIT⟶

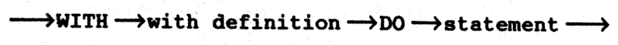
**goto statement:**

⟶GOTO⟶label_name⟶

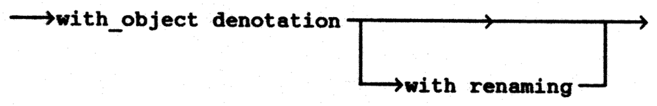
**labelled statement:**

⟶label_name⟶:⟶statement⟶

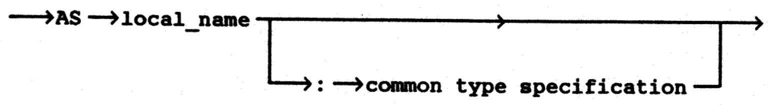
**with statement:**



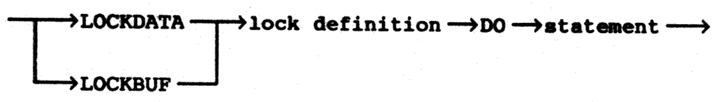
**with definition:**



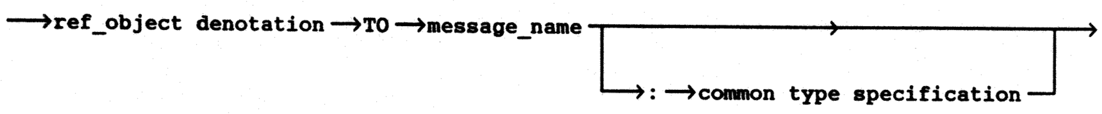
**with renaming:**



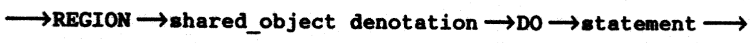
**lock statement:**



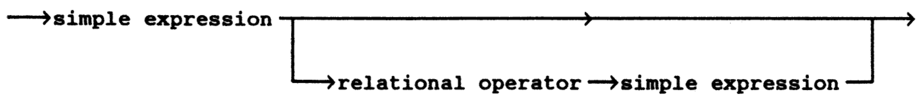
**lock definition:**



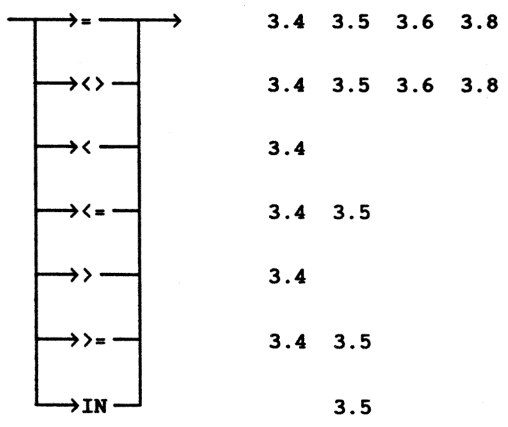
**region statement:**



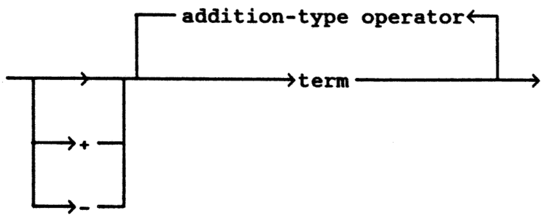
**expression:**



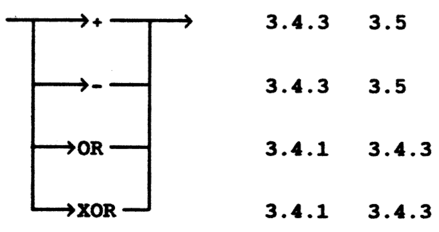
**relational operator:**



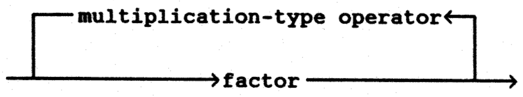
**simple expression:**



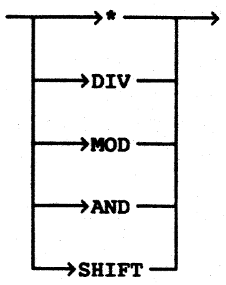
**addition-type operator:**



term:



multiplication-type operator:



3.4.3 3.5

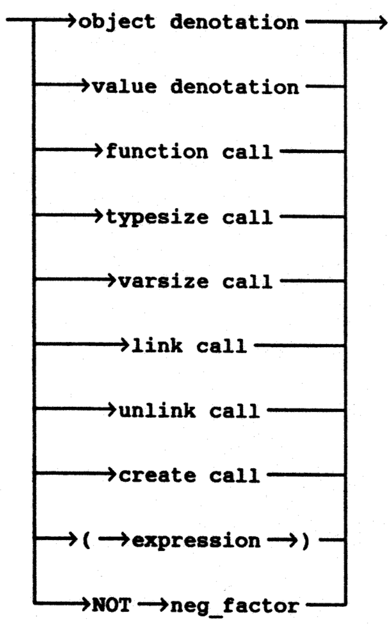
3.4.3

3.4.3

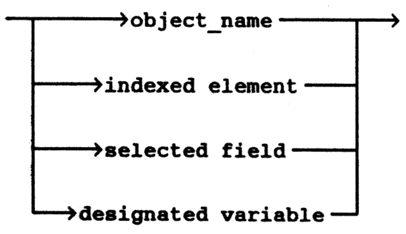
3.4.1 3.4.3

3.4.3

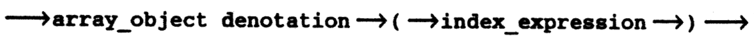
factor:



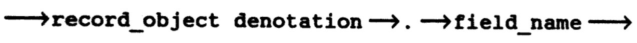
**object denotation:**



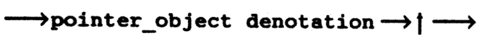
**indexed element:**



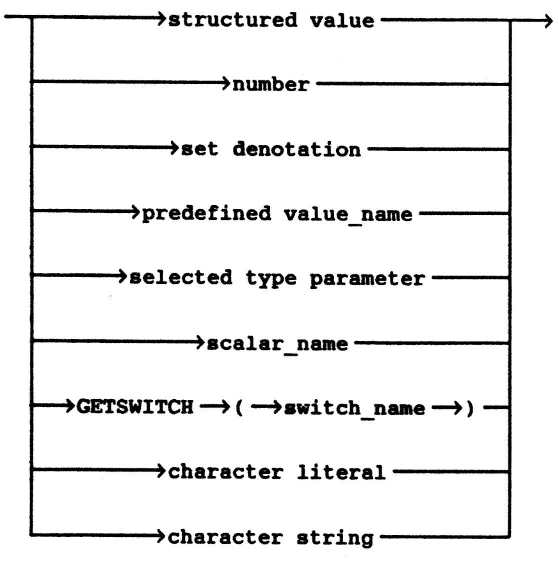
**selected field:**



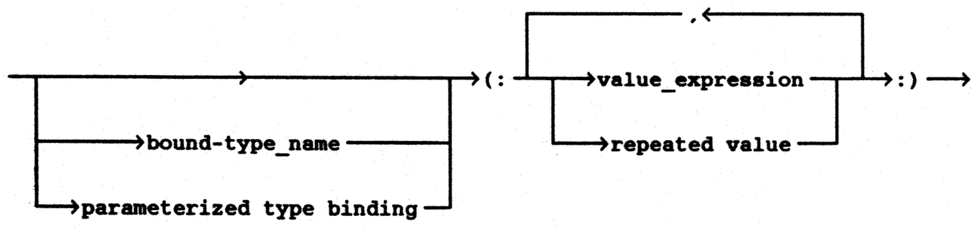
**designated variable:**



**value denotation:**



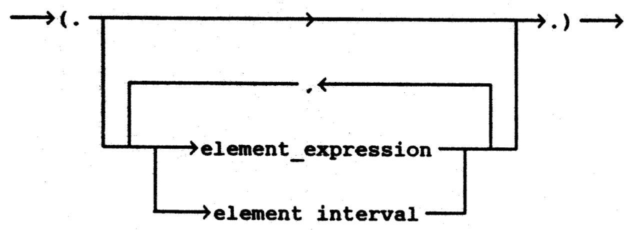
**structured value:**



**repeated value:**

⟶repetition_expression ⟶*** ⟶value_expression ⟶

**set denotation:**



**element interval:**

⟶lower_expression ⟶.. ⟶upper_expression ⟶

**selected type parameter:**

⟶object denotation ⟶! ⟶type-parameter_name ⟶

**typesize call:**

⟶TYPESIZE ⟶( ⟶bound-type_name ⟶) ⟶

**varsize call:**

⟶VARSIZE ⟶ ( ⟶variable_name ⟶ ) ⟶

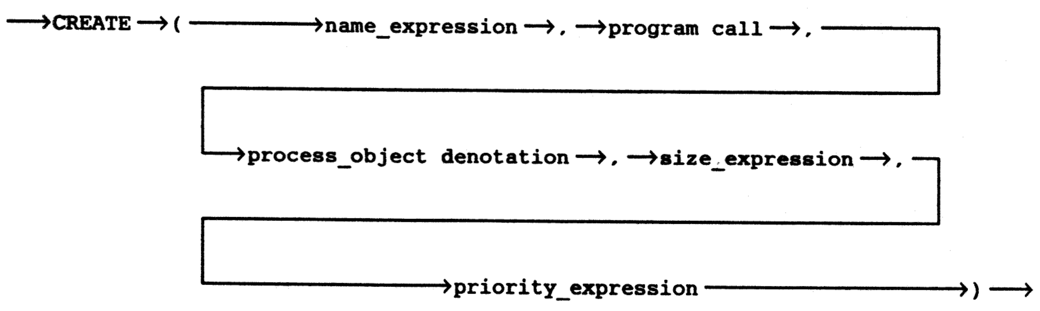
**link call:**

⟶LINK ⟶ ( ⟶string_expression ⟶ , ⟶program_name ⟶ ) ⟶

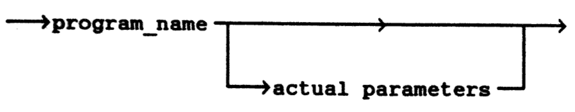
**unlink call:**

⟶UNLINK ⟶ ( ⟶program_name ⟶ ) ⟶

**create call:**



**program call:**



# C. PREDEFINED ENTITIES

In section C.1 a list of predefined entities is given in alphabetical order and with reference to the sections where the entities are described. The types and pseudo-functions which are intrinsic to the language, denoted by keywords rather than predefined names, are listed in sections C.2 and C.3.

## C.1 Routines, Types, and Constants

| section | kind | definition |
|---|---|---|
| 3.4.3 | FUNCTION | abs(n: integer): 0..maxint |
| 9.2.2 | TYPE | activation= (a_mailbox, a_delay) |
| 3.7 | PROCEDURE | alloc(VAR r: reference; VAR p: pool;<br>VAR ra: mailbox) |
| 9.2.2 | FUNCTION | allocdelay(VAR p: pool; VAR ra: mailbox;<br>VAR r: reference;<br>no_of_msecs: 0..maxint): activation |
| 3.7 | FUNCTION | allocmempool(VAR p: pool; no_of_messages,<br>bufsize: 0..maxint; mem: mem_type): 0..maxint |
| 3.7 | FUNCTION | allocpool(VAR p: pool; no_of_messages: 0..maxint;<br>bufsize: 0..maxint): 0..maxint |
| 5.9 | TYPE | buffer(bufsize: 0..maxint)=<br>PACKED ARRAY(0..bufsize-1) OF byte |
| 10.1 | FUNCTION | bufcount(VAR stack: reference): 0..maxint |
| 3.7 | FUNCTION | bufsize(VAR r: reference): 0..maxint |
| 3.7 | FUNCTION | bufsize(VAR ch: chain): 0..maxint |
| 3.7 | FUNCTION | bytecount(VAR r: reference): 0..maxint |
| 3.7 | FUNCTION | bytecount(VAR ch: chain): 0..maxint |
| 10.2 | PROCEDURE | chaindequeue(VAR ref: reference; ch: chain) |
| 10.2 | PROCEDURE | chaindown(VAR ch: chain) |
| 10.2 | PROCEDURE | chainenqueue(VAR ref: reference; VAR ch: chain) |
| 10.2 | FUNCTION | chainlength(VAR ch: chain): 0..maxint |
| 10.2 | PROCEDURE | chainreset(VAR ch: chain) |
| 10.2 | PROCEDURE | chainstart(VAR ch: chain) |
| 10.2 | PROCEDURE | chainup(VAR ch: chain) |
| 3.4.2 | FUNCTION | chr(n: byte): char |
| 11.1.2 | PROCEDURE | closeport(VAR p: port) |
| 11.2.1 | PROCEDURE | connect(VAR p: port; index: 1..maxint;<br>VAR compl, disc: reference;<br>INSPECT remote_name: string;<br>service: conn_service) |
| 11.2.1 | TYPE | conn_service= (cs_normal, cs_high) |
| 11.1 | TYPE | control_type= PACKED RECORD<br>rcv_all,<br>get_credit,<br>?, ?, ?, ?, ?, ?: boolean<br>END |
| 9.1 | TYPE | create_result= (create_ok, no_memory, ...) |
| 11.3 | FUNCTION | creditcount(VAR r: referenc): 0..maxint |

| section | kind | definition |
|---|---|---|
| 5.9 | TYPE | dataarea(offset, top: 0..maxint) = PACKED ARRAY(offset..top-1) OF byte |
| 3.4.3 | PROCEDURE | dec(VAR v: integer) |
| 3.4.5 | PROCEDURE | dec(VAR v: byte) |
| 9.2.2 | PROCEDURE | delay(no_of_msecs: 0..maxint) |
| 11.2.1 | PROCEDURE | disconnect(VAR p: port; index: 1..maxint) |
| 3.7 | FUNCTION | eventkind(VAR r: reference): event_type |
| 3.7 | FUNCTION | eventkind(VAR ch: chain): event_type |
| 3.7 | TYPE | event_type= (not_event, message_event, answer_event, process_removed, port_closed, disconnected, ?, ?, local_connect, remote_connect, reset_indication, reset_completion, credit, data_sent, data_arrived, data_overrun, ?, dummy_lcnct, dummy_rcnct, dummy_rindic, dummy_rcmpl, dummy_credit, dummy_sent, dummy_arrived) |
| 11.2.1 | PROCEDURE | getconnection(VAR p: port; index: 1..maxint; VAR compl, disc: reference) |
| 11.2.2 | PROCEDURE | getcredit(VAR p: port; index: 1..maxint; VAR credmes: reference) |
| 11.2.2 | PROCEDURE | getreset(VAR p: port; index: 1..maxint; VAR indic: reference) |
| 3.7 | FUNCTION | hometest(VAR ref: reference; VAR p: pool): boolean |
| 3.4.3 | PROCEDURE | inc(VAR v: integer) |
| 3.4.5 | PROCEDURE | inc(VAR v: byte) |
| 11.3 | FUNCTION | index(VAR r: reference): 0..maxint |
| 9.1 | TYPE | link_result= (link_ok, already_linked, external_not_found, ...) |
| 9.2.1 | FUNCTION | locked(VAR mbx: mailbox): boolean |
| 11.4 | FUNCTION | maxconnections: 0..maxint |
| 3.4.3 | CONST | maxint |
| 3.7 | TYPE | mem_type= (...) |
| 3.4.3 | CONST | minint |
| 3.6 | PROCEDURE | new(VAR ptr: ptrtype) |
| 3.6 | FUNCTION | nil(VAR ptr: ptrtype): boolean |
| 3.7 | FUNCTION | nil(VAR pr: process): boolean |
| 3.7 | FUNCTION | nil(VAR ref: reference): boolean |
| 3.7 | FUNCTION | offset(VAR r: reference): 0..maxint |
| 3.7 | FUNCTION | offset(VAR ch: chain): 0..maxint |
| 9.2.1 | FUNCTION | open(VAR mbx: mailbox): boolean |
| 11.1 | PROCEDURE | openport(VAR p: port; VAR closemes: reference; INSPECT name: string; scope: scope_type; no_of_cons: 0..maxint; cntrl: control_type) |

| section | kind | definition |
|---|---|---|
| 3.4 | FUNCTION | ord(v: otype): integer |
| 9.2.1 | FUNCTION | passive(VAR mbx: mailbox): boolean |
| 10.1 | PROCEDURE | pop(VAR popped_mes, stack_handle: reference) |
| 3.4 | FUNCTION | pred(v: otype): otype |
| 9.1 | TYPE | prio_type= (minpriority, stdpriority, maxpriority, ...) |
| 10.1 | PROCEDURE | push(VAR new_top, stack_handle: reference) |
| 11.3 | FUNCTION | reason(VAR r: reference): reason_type |
| 11.3 | TYPE | reason_type= (reason_ok, reason_name, reason_ressource, reason_closed, reason_network) |
| 11.2.2 | PROCEDURE | receive(VAR p: port; index: 1..maxint VAR datames: reference) |
| 11.2.2 | PROCEDURE | receiveall(VAR p: port; VAR datames: reference) |
| 3.7 | PROCEDURE | release(VAR r: reference) |
| 3.7 | FUNCTION | releasepool(VAR p: pool; no_of_messages: 1..maxint): 0..maxint |
| 9.1 | PROCEDURE | remove(VAR pr: process) |
| 11.2.2 | PROCEDURE | reset(VAR p: port; index: 1..maxint; VAR compl: reference) |
| 3.7 | PROCEDURE | resetevent(VAR r: reference) |
| 3.7 | PROCEDURE | resetevent(VAR ch: chain) |
| 9.1 | PROCEDURE | resume(VAR pr: process) |
| 9.2.2 | PROCEDURE | return(VAR ref: reference) |
| 11.1 | TYPE | scope_type= (anonymous, local, regional, global) |
| 11.2.2 | PROCEDURE | send(VAR p: port; index: 1..maxint; VAR datames: reference) |
| 3.7 | PROCEDURE | setbytecount(VAR r: reference; val: 0..maxint) |
| 3.7 | PROCEDURE | setbytecount(VAR ch: chain; val: 0..maxint) |
| 3.7 | PROCEDURE | setoffset(VAR r: reference; val: 0..maxint) |
| 3.7 | PROCEDURE | setoffset(VAR ch: chain; val: 0..maxint) |
| 3.7 | PROCEDURE | settop(VAR r: reference; val: 0..maxint) |
| 3.7 | PROCEDURE | settop(VAR ch: chain; val: 0..maxint) |
| 3.7 | PROCEDURE | setu1(VAR r: reference; b: byte) |
| 3.7 | PROCEDURE | setu1(VAR ch: chain; b: 0.255) |
| 3.7 | PROCEDURE | setu2(VAR r: reference; b: byte) |
| 3.7 | PROCEDURE | setu2(VAR ch: chain; b: byte) |
| 3.7 | PROCEDURE | setu3(VAR r: reference; b: byte) |
| 3.7 | PROCEDURE | setu3(VAR ch: chain; b: byte) |
| 3.7 | PROCEDURE | setu4(VAR r: reference; b: byte) |
| 3.7 | PROCEDURE | setu4(VAR ch: chain; b: byte) |
| 9.2.2 | PROCEDURE | signal(VAR mbx: mailbox; VAR ref: reference) |
| 10.1 | FUNCTION | stackdepth(VAR stack: reference): 0..maxint |
| 9.1 | PROCEDURE | start(VAR pr: process; prio: prio_type) |
| 9.1 | PROCEDURE | stop(VAR pr: process) |
| 3.8.4 | TYPE | string(length: byte)= ARRAY(1..length) OF char |
| 3.4 | FUNCTION | succ(v: otype): otype |

| section | kind | definition |
|---|---|---|
| 3.7 | FUNCTION | top(VAR r: reference): 0..maxint |
| 3.7 | FUNCTION | top(VAR ch: chain): 0..maxint |
| 7.1 | PROCEDURE | trace(fault: integer) |
| 9.1 | TYPE | unlink_result= (unlink_ok, no_program_linked, existing_incarnations, ...) |
| 3.7 | FUNCTION | u1(VAR r: reference): byte |
| 3.7 | FUNCTION | u1(VAR ch: chain): byte |
| 3.7 | FUNCTION | u2(VAR r: reference): byte |
| 3.7 | FUNCTION | u2(VAR ch: chain): byte |
| 3.7 | FUNCTION | u3(VAR r: reference): byte |
| 3.7 | FUNCTION | u3(VAR ch: chain): byte |
| 3.7 | FUNCTION | u4(VAR r: reference): byte |
| 3.7 | FUNCTION | u4(VAR ch: chain): byte |
| 9.2.2 | PROCEDURE | wait(VAR ref: reference; VAR mbx: mailbox) |
| 9.2.2 | FUNCTION | waitdelay(VAR ref: reference; VAR mbx: mailbox; no_of_msecs: 0..maxint): activation |

## C.2 Language Intrinsic Types

| section | name |
|---|---|
| 3.4.1 | boolean |
| 3.4.5 | byte |
| 3.7, 10.2 | chain |
| 3.4.2 | char |
| 3.4.3 | double |
| 3.7 | external program |
| 3.4.3 | integer |
| 3.7 | mailbox |
| 3.7 | pool |
| 3.7, 11 | port |
| 3.7, 9.1 | process |
| 3.7 | reference |

## C.3 Language Intrinsic Pseudo-function

| section | description |
|---|---|
| 9.1 | create(INSPECT inc_name: string; program call; VAR pr: process; size: 0..maxint; priority: prio_type): create_result |
| 3.12 | getswitch(switch_name): integer |
| 9.1 | link(INSPECT name: string; VAR prog: program): link_result |
| 3.2 | typesize(type_name): 0..maxint |
| 3.11.2 | varsize(variable_name): 0..maxint |
| 9.1 | unlink(VAR prog: program): unlink_result |

# D. INDICES

## D.1 Survey of Figures

## D.2 Catchword Index

Reference Manual for the Prgogramming Language Real-Time Pascal

PN: 99110141

**RETURN LETTER**

Title: Reference Manual for the Programming Language Real-Time Pascal    RCSL No.: 99110141

A/S Regnecentralen af 1979/RC Computer A/S maintains a continual effort to improve the quality and usefulness of its publications. To do this effectively we need user feedback, your critical evaluation of this manual.

Please comment on this manual's completeness, accuracy, organization, usability, and readability:

________________________________________

________________________________________

________________________________________

________________________________________

Do you find errors in this manual? If so, specify by page.

________________________________________

________________________________________

________________________________________

________________________________________

How can this manual be improved?

________________________________________

________________________________________

________________________________________

________________________________________

Other comments?

________________________________________

________________________________________

________________________________________

________________________________________

________________________________________

Name: ____________________ Title: ____________________

Company: ____________________________________

Address: ____________________________________

Date: __________

Thank you

PN: 99200176

.............................. Fold here ..............................

.................. Do not tear - Fold here and staple ..................

Affix
postage
here

REGNECENTRALEN
af 1979

Information Department
Lautrupbjerg 1
DK-2750 Ballerup
Denmark