PN: 99000994

Edition: December, 1987

Author: Bo Bagger,
Jan Bardino.

Title:

RC3502/2 REAL-TIME PASCAL

Reference Manual

PN: 99200170

RC International

Keywords:

Real-Time Pascal, RC8000, RC3502, Multiprogramming.

Abstract:

This manual contains a description of the implementation of Real-Time Pascal on the RC3502 /2 computer.

Information of how to use the Real-Time Pascal compiler and related utility programs is included in appendices.

(This manual substitutes RCSL No.: 52-AA1167)

(200 printed pages).

PN: 99200170

TABLE OF CONTENTS PAGE

TABLE OF CONTENTS (continued) PAGE

TABLE OF CONTENTS (continued) PAGE

TABLE OF CONTENTS (continued) PAGE

TABLE OF CONTENTS (continued) PAGE

FOREWORD.

This edition of the RC3502/2 Real-Time Pascal Reference Manual is updated according to the separately released papers about changes and manual updates announced in SIS. Changes are marked with correction lines in the left margin.
This is an intermediate version of the manual reflecting release 3 og the RTP3502 compiler and release 3 of the run time system. With release 4 of the compiler, the language is changed in direction of Real-Time Pascal as defined in the language reference manual (ref 8.). At the same time RTP86 is changed in order to make the two dialects of the language nearly equal, making it possible to port programs from one implementation to the other with a minimum of effort. Old programs may be changed nearly automatically by a porting utility programme, which will be part of the release package.

1. INTRODUCTION 1.

This manual describes the RC3502 Model 2 implementation of the programming language Real-Time Pascal.

It should be noted, that the name of the language was changed from PASCAL80 to Real-Time Pascal in 1981.

The manual is structured in the following way:

Chapter 2 contains the differences and limitations of the RC3502/2 Real-Time Pascal language related to the PASCAL80 Report (ref. 1).

Chapter 3 describes the representation of objects at run time.

Chapter 4 describes all the available routines.

Chapter 5 describes the standard processes comprising the run time environment for RC3502/2 Real-Time Pascal programs.

The appendices B, C and D describe the use of the RC3502/2 Real-Time Pascal compiler including error messages.

Utilities for manipulation of source and object programs are described in appendices E and F.

Appendix G describes how a load file or autoload file is generated on an RC8000.

Appendix H contains a complete list of language symbols.

Appendices I and J contain a list of all predefined constants, types, and routines.

Appendix K contain a list of operator input/output type definitions and routine declarations.

Appendix L is a more comprehensive example using the OPERATOR input/output routines.

Appendix M contains a complete list of exception codes.

2. IMPLEMENTATION DETAILS AND LANGUAGE MODIFICATIONS 2.

The following is a list of modifications and details as compared to the PASCAL80 report of this version of the RC3502 Real-Time Pascal implementation. The reference is done by page and line number in the PASCAL80 Report (ref. 1).

Dynamic Types

The Real-Time Pascal compiler for RC3502 allows variables in the expressions defining the bounds of subrange types. It is demanded that variables (including function calls) are declared in a surrounding scope. Change of scope is related to the block stucture of the program, so that a new scope is started:

- at the start of a parameter list,
- at the start of local declarations, and
- by the key word "as" in the "lock"/"with"-statement.

The comprising expressions are evaluated once, namely at the definition of the type.

A "dynamic type" is:

subrange, where one or both bounds contain variables,

array, where index type and/or component type is dynamic,

record, where at least one of the fields is dynamic.

There are certain restrictions on the use of dynamic types:

It is allowed to use dynamic types as:

- component type in array definition,
- index type in array definition,
- the type of a field in a record,
- the type of a variable,

- the type of a lock-variable,
- the type of a formal VAR-parameter in an internal routine definition,
- for definition of "alias"-types.

It is not allowed to use dynamic types as:

- the element type of a "SET",
- the type of the buffer in a "POOL"-definition,
- the return type for a function,
- definition of a structured constant,
- the type of VALUE-parameters in a routine definition,
- the type of a formal process parameter, neither VAR nor VALUE,
- the type of a formal parameter in an external routine.

Restrictions in the use of dynamic variables:

- A dynamic array with "char" as component type is not compatible with a text constant.

- Dynamic array and record cannot be initialized simultaneously with a declaration.

Logical Operators

The logical operators OR, AND, XOR (new logical operator) and NOT may be applied on integer variables (16-bit objects) beyond the applications previously allowed. It is pointed out that NOT is 16-bits 1-complement, if the argument is integer (or subrange).

Page 2, Comment:

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 occurence of a NL character are part of the comment.

Examples:

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

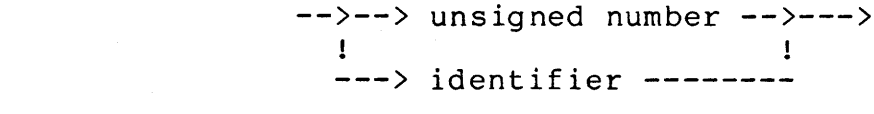
(* this is -- another comment *)

-- a third comment

Page 3, line 2 from the bottom:
"all" is replaced by "most".

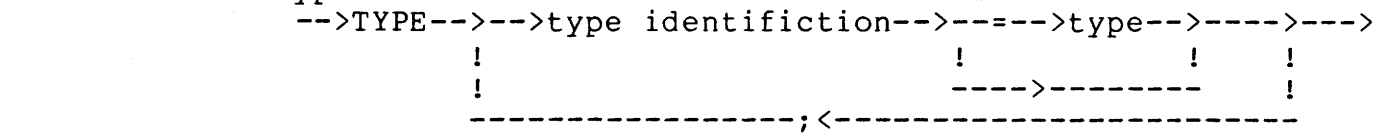
Page 6, Label is implemented as:

label:



Page 9, Forward Pointer Types.

type declaration:



A ´forward-typename´ is a type identification without an associated ´type´ it may only be used in the definition of pointer types and for every ´forward typename´ occurring in a ´type declaration´, the same name must be given a definition 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. cl-l

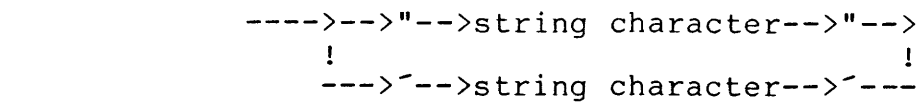
Page 9 and pages 41-42:
Parameterized types are not implemented, hence the predefined type string (n) (page 36) has no meaning. Instead of string (n) there is a predefined type:

```
alfa = array(1..12) of char;
```

and the routine heading of link is changed to:
FUNCTION link(external_name:alfa;process name): integer;

Page 16, Char value:

char value:



The string characters are the characters: sp .. ~

Page 17, 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 incarnation 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 (see below).

Example:

```
LOOP
  next_b:=read_next_buffer;
  prepare_buffer(next_b);
  send_buffer(next_b)
ENDLOOP
```

Jump Statements.

The statements described in this section serve to explicitly modify the order of execution of

statements by transferring control to an implicitly indicated statement, in the contra_ry to goto statements which transfer control to explicitly indicated statements.

Exitloop Statement.

Execution of an exitloop statement forces an enclosing repetitive statement to terminate.

exitloop statement:
--> EXITLOOP -->

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
```

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 the specific effect of the continue_loop statement is:

for statement:
Execution continues at "count and test".

loop statement:

Execution continues with the first statement after LOOP.

while statement:
The remainder of the (compound) 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.

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 process as if the compound statement had been exhausted.

Example:

```
BEGIN (* main program *)
  ...
  IF errors_detected THEN
    EXIT; (* terminate the incarnation *)
  ...
END
```

Page 19, Empty Subrange.

The number of elements in the value set of a subrange is ord(upperbound)-ord(lowerbound)+1. The number may be zero in which case the subrange is empty, but it may not be negative.

Page 19: The type real is not implemented.

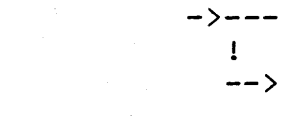
Page 26 Lockmes Statement.

The lock statement is the language construct which provides access to the actual contents of

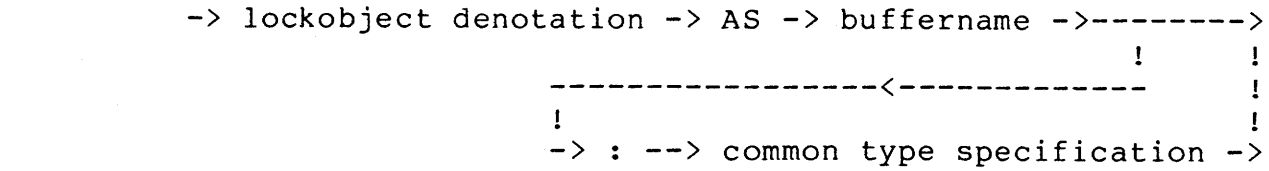
a buffer, i.e. to the top non-empty buffer in a message stack.

lock statement:
--> lockword --> lock definition --> DO --> statement -->

lockword:



lock definition:



The denoted object must be a variable of type reference. Its value must not be NIL. Otherwise a fault occurs. The message designated by the reference may be accessed in the statement following DO as an implicitly declared variable the name of which is specified by the ´buffername´. Either the whole buffer or only its data is accessible, depending on the ´lockword´.

If the ´lockword´ is LOCK the whole buffer is accessible as a variable of type
buffer= ARRAY(0..bufsize*2-1) OF 0..255

where the value of the parameter equals the size of the buffer. This is the type of the buffer, unless another type is superimposed on the buffer, by means of the specification of a local type.

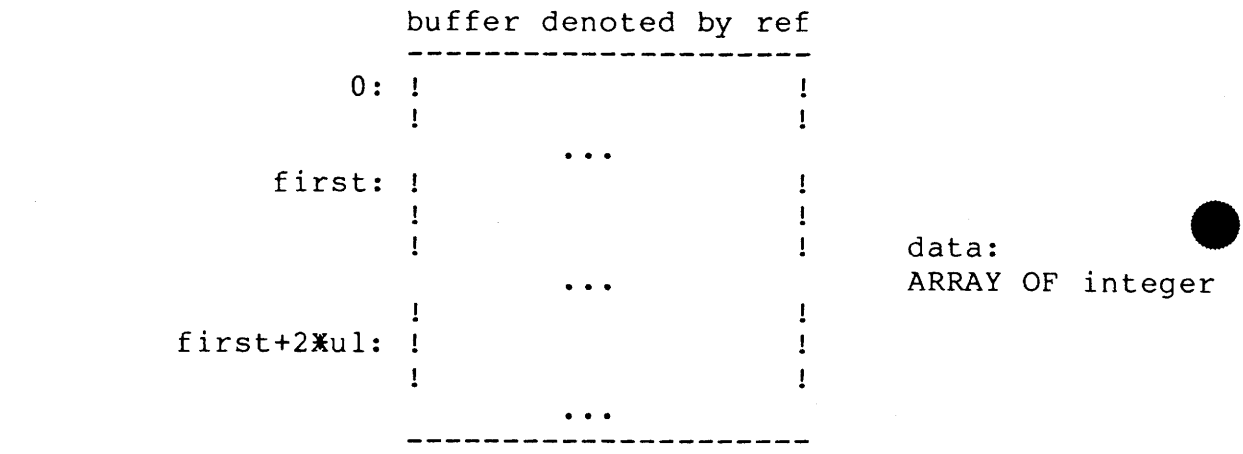
If the ´lockword´ is LOCKMES the data area is accessible as a variable of a type
dataarea= ARRAY(first..last) OF 0..255

where first and last values are equal to the buffer attributes with the same names. This is the type of the buffer, unless another type is superimposed on the data area, by means of the specification of a local type. The address of the variable as well as the parameter values are evaluated before the statement following DO is

executed.

Example:

```
LOCKMES ref AS d_part DO
  WITH d_part AS data: ARRAY(0..ul(ref)) OF integer DO
    ...
```



Page 31 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.

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.

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.

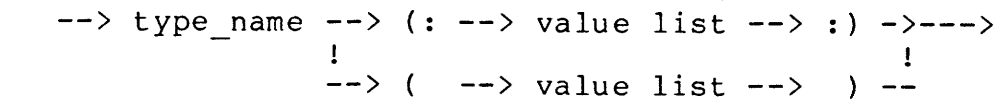
PROCEDURE new(VAR ptr: ptrtype)
A call of new causes memory for a variable of the base type of the type of the parameter ptr to be allocated on the heap of the calling process incarnation. If an initial value is defined for the variable or any components of it, the initialization takes place immediately after allocation. The value of the parameter becomes a pointer to the allocated variable.

Example:

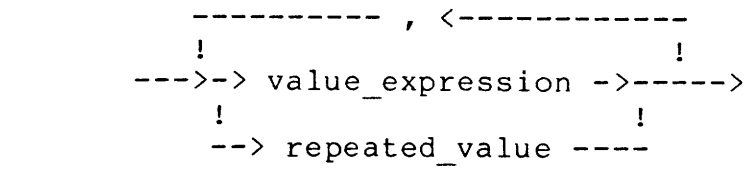
```
TYPE
  comp; compl; -- forward declarations
  compl_type= RECORD
    number: integer;
    compl_chain: compl;
    comp_chain: comp
  END (* RECORD *);
  comp_type= ARRAY (x..y) OF compl;
  comp= ↑comp_type;
  compl= ↑compl_type;
VAR
  structure_start: comp;
BEGIN
  ...
  new(structure_start);
  new(structure_start↑(x));
  new(structure_start↑(x)↑.compl_chain)
  ...
```

Besides the well known notation for constants of structured types the RTP86 notation is now allowed. Constants of a structured type may be denoted by lists of element or field values.

structured constant:



value list:



repeated_value:

The ´type_name´ 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 structured type is an array type the values in the list are the element values in index order. The number of values, possibly none, must equal the number of elements and the type of each value must be assignable to the element type. A ´repeated_value´, if present, is equivalent to a number of repeated occurrences of its ´value_expression´. 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 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. In the case of a record type a ´repeated_value´ must not occur.

If the ´value list´ is enclosed in (: :) then 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 constant´ occurs as a ´value_expression´ within a larger ´structured constant´

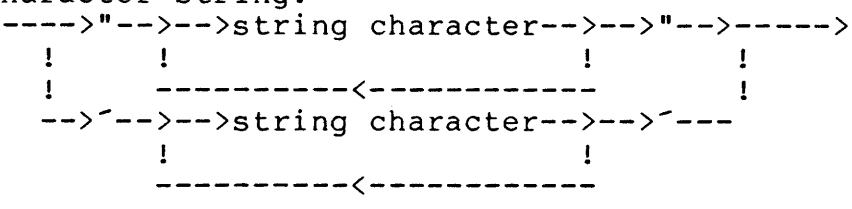
2. The ´structured constant´ occurs as an ´initialization_expression´ in a variable declaration.

Example:

```
CONST
  identity_3=matrix_3(:(:1,0,0:), (:0,1,0:), (:0,0,1:):);
VAR
  a4: matrix_4:= (:4✱✱✱(:4✱✱✱0:):); -- initially null
```

Page 36, page 64, Character string:

character string:



Concatenation Operator.
Individual non-graphic characters may be included in a character string by means of the concatenation operator &.

Example:

```
"✱✱✱ illegal input message" & BEL & NL
```

The string characters are the characters: sp .. ~

Page 37, With Statement.

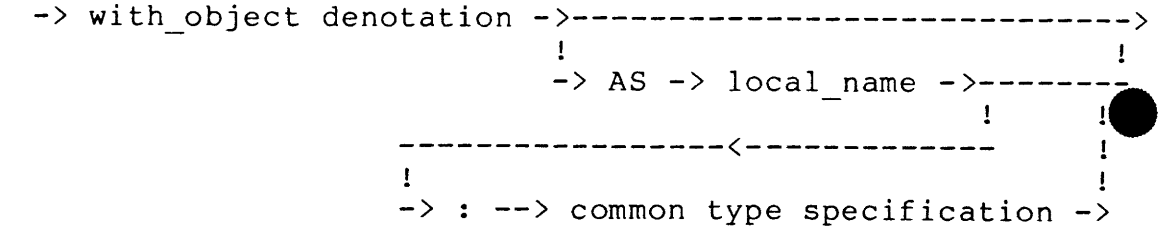
A with statement may be used for three purposes:

1. shorthand notation for field access in a record object,
2. object renaming,
3. object retyping.

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

with statement:
--> WITH --> with definition --> DO --> statement -->

with definition:



The denoted object is called the with-object. It must not be an irregular object (packed field). 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 (i.e. contain elements of type semaphore, pool, reference, shadow, process, or pointer), nor may the local type. The size of the local type must be less than or equal to the size of the with-object.

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:
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 buffers whose exact type is not known beforehand (some of the type information may be part of the buffer contents). 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 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
   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
   rec_field         or
   loc_var.loc_rec_field
```

Example:

```
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 -- abbreviaiton
    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
          ...
```

Page 45, Pool Initialization:
The allocation of memory for messages for a pool variable is performed during the initialization of an incarnation after it has been started and scheduled to run. One effect of this is that an incarnation may in unfortunate cases be successfully created and yet unable to run due to lack of memory. If the pool variable is declared as empty (e.g.: pool 0;) the allocation may be performed at any moment by the incarnation calling the INITPOOL routine.

Page 45, Typesize and Varsize Call.

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(--> type_name -->) -->

The 'type_name' must be the name of a 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
  arr_type=ARRAY(1..10) OF integer;
  rec_type=RECORD                    -- Note:
    f1: arr_type;                    -- type of f1 not compatible
    f2: ARRAY(1..10) OF integer      -- with type of f2
  END(*RECORD*);
  ptr_type=↑rec_type;
  mask_type= PACKED ARRAY(1..typesize(arr_type)) OF boolean
```

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. 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.

Page 46, Shift operator.

A SHIFT operator has been introduced with the same operator precedence as multiplication and division. SHIFT takes integer operands and produces integer result. Which is the value of left operand logical shifted the value of right operand toward more significant positions.

Examples:

The result of #HF SHIFT 4 is #HF0
the result of #HF0 SHIFT (-4) is #HF

Page 50, Frozen VAR Parameters.

The set of objects which may be used as actual parameters for VAR parameters of a frozen type

is extended with constants, both simple and structured. The parameters are passed by address transfer, i.e. the (expensive) copying of the objects is avoided.

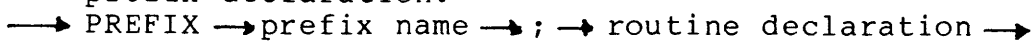
Example:
```
  PROCEDURE out_string(str: string)
```
may now be declared as
```
  PROCEDURE out_string(VAR str: !string)
```
without changes in the calls, but saving of about string length micro seconds of execution time for each call, and the stack appetite of the caller is reduced with approximately string length bytes.

Page 51, Process Parameters:
Process parameters of reference, pool, or shadow types are not allowed, neither as variables, values, nor components of structured parameters. Pointers may be passed only as values or frozen variables. Semaphores may be passed as variables, but not as values.

Page 52 and page 60:
Prefixed process declaration has not been implemented. As an alternative the compiler accepts "context´s", see appendix B.2.

Page 56, Incarnation Termination:
When an incarnation is terminated (by completing its compound statement) it is permanently descheduled and will not become running again. Garbage collection is performed, when the incarnation is removed by the father or an ancestor.

Page 60 Prefix declaration is implemented as:



The file containing the generated code for the routine may be specified as input to the cross linker when processes using the routine are cross linked for autoload preparation or may be

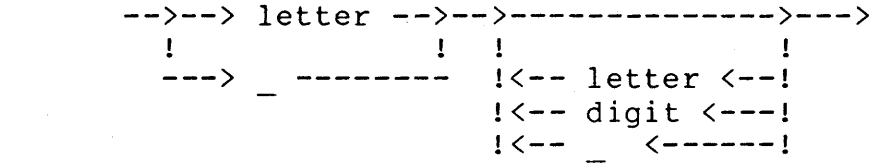
loaded directly in the RC3502 by the LOADER process.

Page 61, Language Symbols.

See appendix H of this manual for a complete list of language symbols.

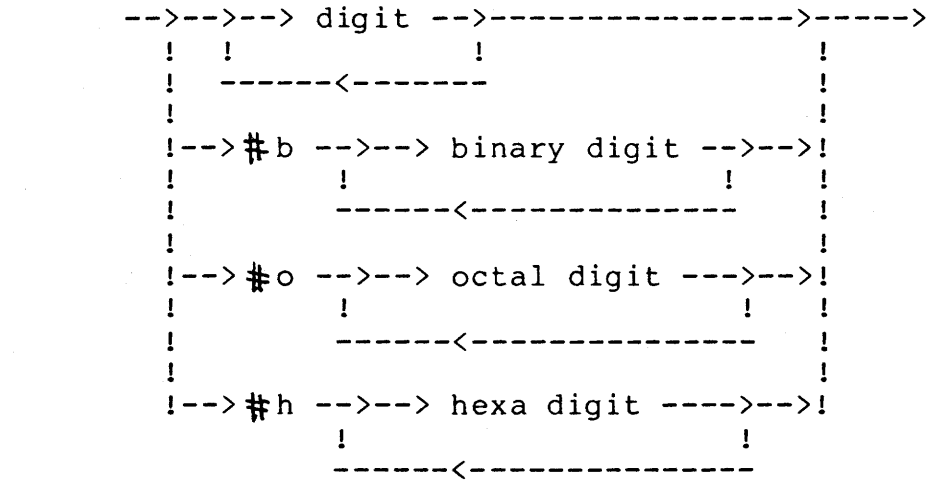
Page 63, Identifier:

identifier:



Page 63, Numeric value is implemented as:

numeric value:



binary digits are 0..1
octal digits are 0..7
hexa digits are 0..9 and a..f

3. REPRESENTATION AND LAYOUT OF VARIABLES 3.

In this chapter it is explained how values of the various types of Real-Time Pascal are represented in the RC3502 implementation and how memory is allocated and laid out to hold the values of the variables of a process incarnation.

3.1 Representation of Values 3.1

3.1.1 Enumeration Types 3.1.1

An enumeration type is defined as consisting of a finite, totally ordered set of values, corresponding to a set of ordinal values which is a subset of the integral numbers. The representation of a value of an enumeration type is the two´s complement representation of the corresponding ordinal value. If a type includes negative ordinal values, the representation of values of the type is always in 16 bits (a word). If the type includes only non-negative ordinal values, and n is th largest of these, then the representation is in log2(n+1) bits (at most 16). Examples:

- integer values (-32768..32767) are represented in 16 bits,
- boolean values (false, true) are represented in 1 bit,
- char values (see ref. 1) are represented in 7 bits,
- values of the subrange type -3..7 are represented in 16 bits,
- values of the scalar type (red, green, blue, orange, pink) are represented in 3 bits.

3.1.2 Shielded and Pointer Types 3.1.2

The representation of values of shielded and pointer types is not revealed.

3.1.3 Set Types

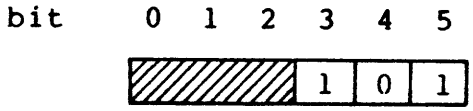
The representation of values of a set type uses a bit vector whose length depends on the base type. The base type must be an enumeration type which does not include negative ordinal values.

If the ordinal value set of the enumeration type T is the range m..n ($m \geq 0$) then values of the type set of T are represented in a bit vector with indices from 0 to n. The first m (possibly zero) of these bits are significant only in comparisons. If the element of T whose ordinal value is i ($m \leq i \leq n$) is a member of a particular value of type set of T, then bit i in the representation of that value is 1, otherwise it is 0. Example:

```
TYPE
     a=set of 3..5;

VAR
     b: a:=(.3,5.);
```

The value of b is represented as shown:



The shaded bits are significant only in comparisons.

3.1.4 Structured Types

Values of array or record types are vectors of values of enumeration, shielded, pointer and set types. The component values are represented as described for the component types.

3.1.5 Type Size

The size of a type T, denoted S(T), is the number of bits used to represent values of type T. The concept of size is only relevant for types which are affected by packing when used for components of structured types, and is therefore not defined for all types. For enumeration types S(T) is computed as described above.

Example: S(char)=7, S(integer)=16.

3.2 Memory Layout 3.2

The memory requirement of a type T, denoted M(T), is defined as the number of bytes which are allocated for a variable of type T.

For an enumeration type T, M(T) depends on S(T), as follows:

```
1 <= S(T) <=  8 : M(T)=1
9 <= S(T) <= 16 : M(T)=2
```

For shielded and pointer types, M(T) is the following:

```
M(reference)=  7
M(semaphore)=  7
M(shadow)=     6
M(pointer)=    3
M(pool)=       7
```

Memory requirement for structured types is described in connection with memory layout for these types in subsection 3.2.3.

3.2.1 Alignment 3.2.1

The memory of the RC3502 machine consists of a sequence of eight bit bytes. Each byte has an address. Two consecutive bytes is a word. The most significant halfword is the byte with the lowest address.

Memory allocation for a variable of a shielded type, or of a stuctured type containing components of shielded type(s) is physically word-aligned, i.e. the allocated memory starts on a word boundary, where the first byte has an even address. All other variables are byte-aligned.

3.2.2 Stack Frame

Memory for variables declared in process or routine blocks is allocated in stack frames in the data structure of the incarnation to which the variables belong. The general layout of an incarnation stack is shown in fig.1 .

In the portion of a stack frame used for declared variables memory is allocated from low to high addresses in the order of declaration of the variables in the program text.

An integral number of bytes is allocated for each variable. The number of bytes is determined by the memory requirement of the type of the variables. Because of physical word-alignment, unused bytes of memory may be left between variables (in fact also inside structured variables).

When more memory is allocated for a variable of an enumeration type than indicated by the size of the type, the variable is placed in the least significant bits of the allocated byte or word.

Example: Corresponding to the declarations:

```
VAR
     a: char;
     b,c: 0..7;
     d: integer;
     e: reference;
```

Memory is allocated within a stack frame as illustrated in fig.2 .

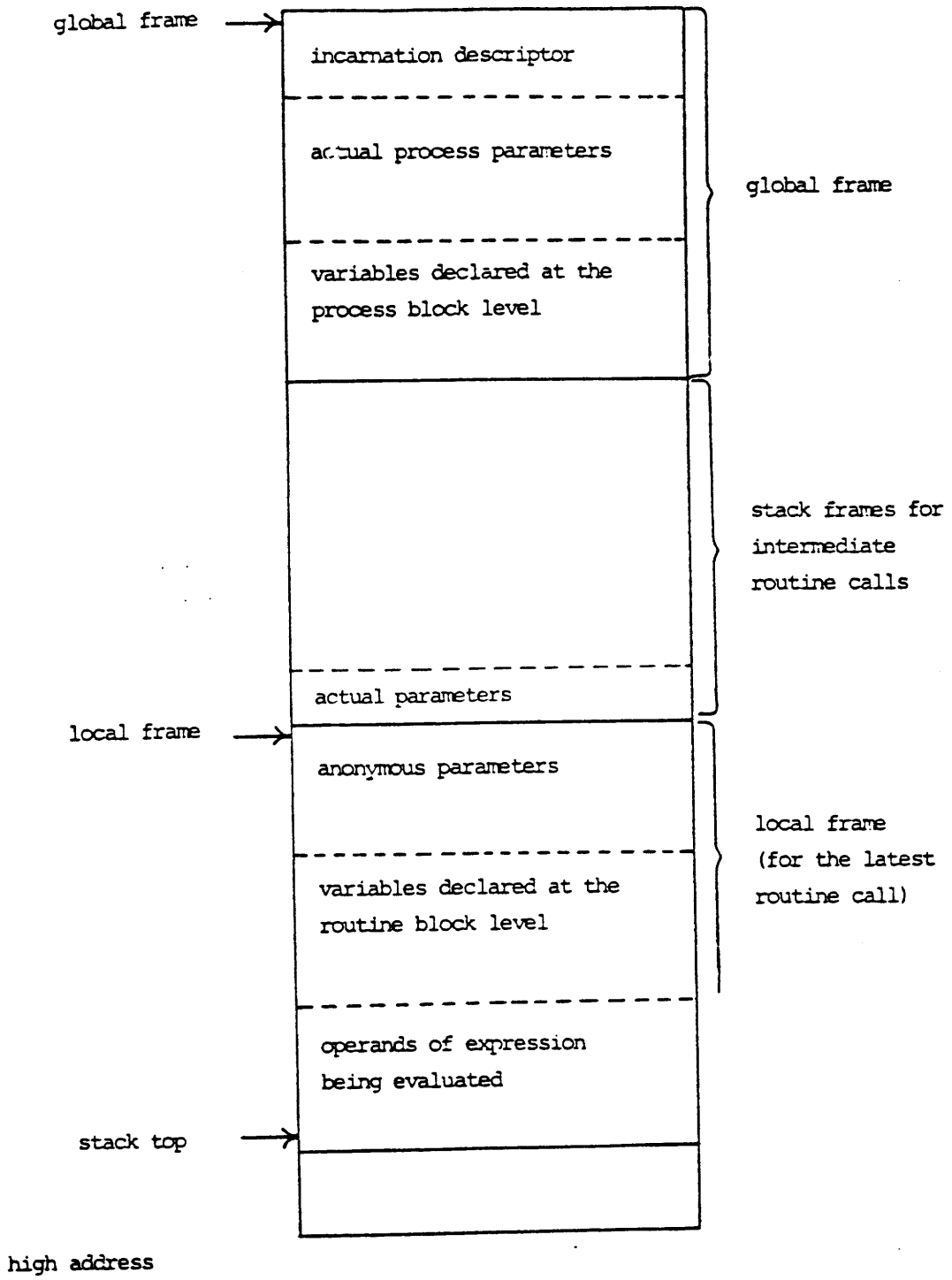


Fig. 1. Incarnation stack layout (snapshot)

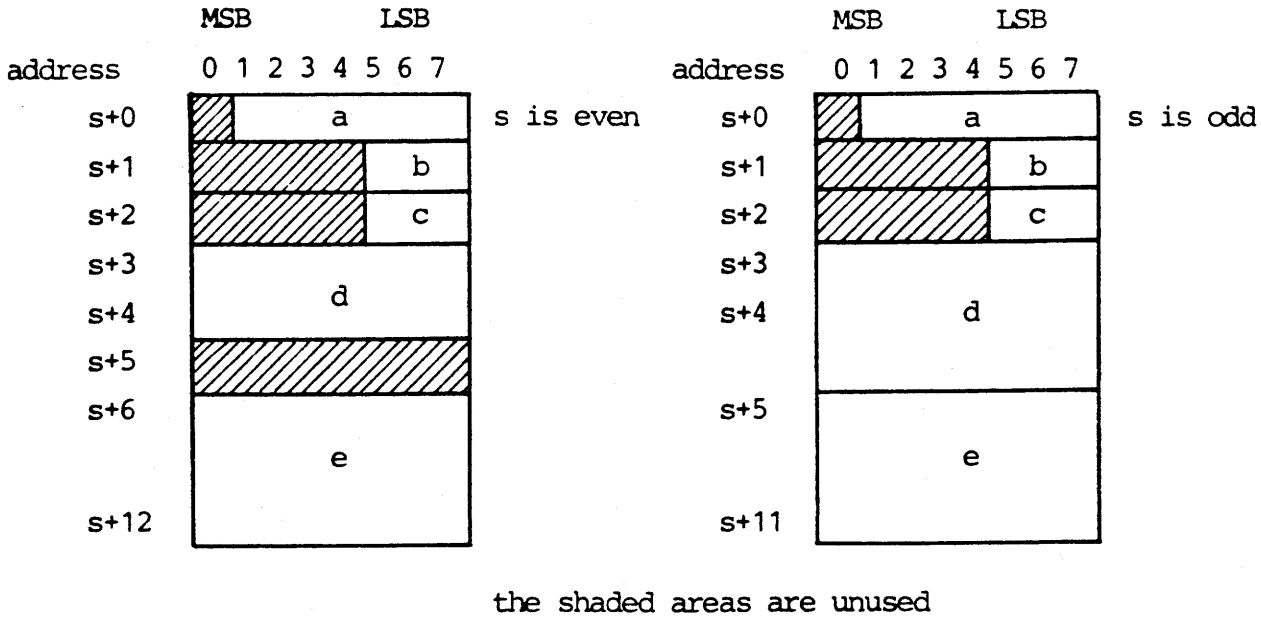


Fig. 2. Memory layout for simple declared variables in stack frame or record type (not packed).

The layout of actual parameters is similar to that of declared variables, with a minor modification: Since the smallest unit of memory pushed on the evaluation stack is a word, each actual parameter of a process or routine stack frame occupies an integral number of words. Similar to the case of declared variables, an actual parameter of an enumeration type of size less than 16 (bits) is placed in the least significant bits of the allocated word.

3.2.3 Structured Types

3.2.3

The memory requirement of a structured type is determined by the memory requirements of the component type(s) and by the necessary alignment.

The memory allocated for a variable of a structured type is sub-allocated for the components of the variable in a fashion which depends on whether the type is declared as packed or not.

For an array, memory is allocated from low to high addresses for the elements in index order, and for a

record, memory is allocated from low to high addresses for the fields in the order they are declared in the record type definition.

3.2.3.1 Arrays and Records, Not Packed

For an array or record type which is not packed, memory allocation for the components takes place in precisely the same fashion as allocation of memory for declared variables in a stack frame, i.e.:

- from low to high addresses,
- an integral number of bytes per component,
- alignment (relative to the beginning of the array or record and by implication also in absolut memory) as described in subsection 3.2.1,
- right justification of each enumeration type component within the allocated byte or word.

Example: With the record type definition:

```
TYPE
     r=RECORD
         a: char;
         b,c: 0..7;
         d: integer;
         e: reference
     END;
```

Fig. 2 (s is even) shows the layout of a variable of type r.

By summation (or multiplication) the memory requirement of an array or record type may be determined from the above rules for sub-allocation of memory for components. For the record type in the example M(r)=13, because the start address is even.

3.2.3.2 Packed Arrays and Records

In the memory allocated for a variable of a packed array or record type several consecutive components may be packed into a single byte or word. Only components of types with size less than 16 (bits) (for arrays: 6 bits) are candidates for packing. Packing of components always starts from bit 0 (MSB) of a byte, and each component is allocated as many bits as indicated by its size. When it is not pos-

sible to fit any more components without crossing two byte boundaries, packing stops and allocation is resumed from the next byte boundary, i.e. byte-alingment takes place. By this rule unused space may be left in the least significant bits of the byte, where byte-alignment took place.

Notice that in the current implementation only components of enumeration types are candidates for packing.

The memory requirement of a packed record or array type is always at least 1 byte.

Example: The layout of variables of the following packed record type is shown in fig. 3.

```
TYPE
     q=PACKED RECORD
         a: char;
         b,c: 0..7;
         d: integer;
         e: reference
     END;
```

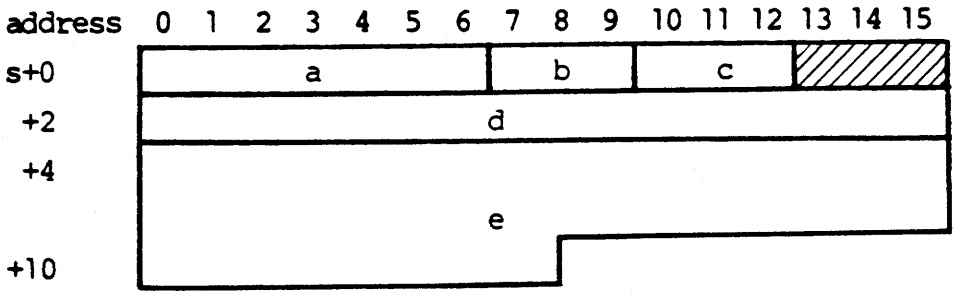


Fig. 3. Memory layout for packed record. M(q)=11

3.2.3.3 Set Types

The memory requirement of a set type is always an even number (of bytes), i.e. an integral number of words are allocated for each variable of a set type. The number of words used is the smallest number which will accomodate the bit vector used to represent values of the set type, cf. subsection 3.1.3. The bit vector is laid out with index 0 in the most sig-

nificant bit of the first word. The last word may contain an unused portion in its least significant bit positions. Example: See fig. 4.

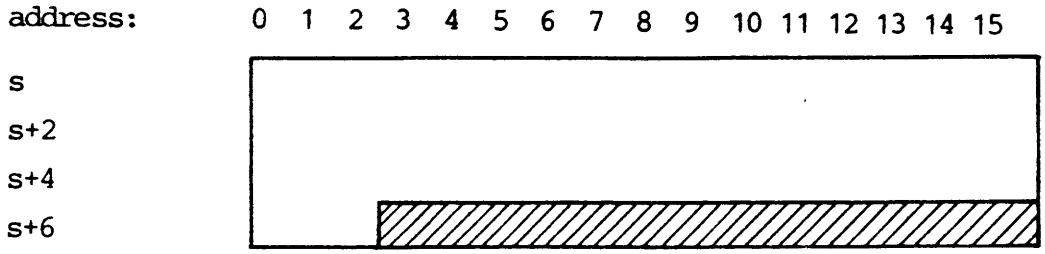


Fig. 4. Layout of a variable of the type set of 0..50; 13 bits are unused. The memory requirement of the type is 8.

4. AVAILABLE ROUTINES 4.

In the following sections all available routines are described. The routines are grouped together according to common functional characteristics:

Routines for Process Control,
Communication Routines,
Routines for Message Manipulation,
Conversion and Arithmetic Routines,
Miscellaneous Routines,
Routines for Operator Communication,
Driver Input/Output Routines.

In the descriptions one out of three attributes is listed:

Predefined: The routine is predefined and may be understood as part of the language.

Not predefined: The routine must be declared in the declaration part of a program or in a user defined environment.

Defined in IOENVIR: The routine is declared in the environment IOENVIR, which may be used in the compilation for access of the routine.

4.1 Routines for Process Control 4.1

This section describes the routines for control of processes and process incarnations. Exception handling is described, and a basic debug tool (TRACE).

4.1.1 Break 4.1.1

PROCEDURE break(VAR sh: shadow; excode: integer);

Predefined.

- stops the child and starts it in the exception procedure (see EXCEPTION). If 0 <= excode and excode <= break_by_father, the child will go into the exit state; otherwise the child will continue.

4.1.2 Create


```
FUNCTION create (incarnation_name: alfa;
                 process name (actual parameters);
                 VAR sh: shadow;
                 size: integer): integer;
```

Predefined.

- A new incarnation of the process linked to ´process name´ is created. The ´size´ parameter specifies the amount of storage for holding the runtime stack. The stack is initialized with the actual parameters and various administrative information. The incarnation name field in the stack is initialized to ´incarnation_name´ and the state to stopped.

The function returns the following results:

Results	Meaning
0	Ok, incarnation created
1	The shadow variable was not nil
2	The process was not linked
3	No storage or demanded size too small

The ´size´ parameter is indicated in words. The maximum size an incarnation stack can take is 32 K words, which will be allocated if ´size´ is negative.

If ´size´ equals zero, a compiler calculated default create size will be used.

Consult appendix B.1 option stack.<appetite> for calculation of the default create size.

Example 1. CREATE

```
PROCESS example1 (VAR mysem : semaphore);

CONST
  size = 238;
  ok   = 0;

VAR
  childsem : semaphore;
  sh       : shadow;

PROCESS example11 (VAR s1, s2 : semaphore);
BEGIN
```

```
    (* body of internal process *)
END;

BEGIN
  IF create ('child', examplell (mysem, childsem),
    sh, size) = ok THEN;
END; (* example1 *)
.
```

4.1.3 Exception

PROCEDURE exception (excode: integer);

- A declaration of such a procedure causes the runtime system to call the procedure, when an exception occurs.

If the user has not declared an exception procedure, a standard exception procedure will be called. The standard procedure may also be called as the procedure TRACE.

The standard exception procedure activates the exception process which produces output with the format:

```
system clock
incarnation name >> exception, excode=code: error text
gf= ....., top= ...., code= ...
called from: ....., ic= ...., line ...-... , date
   .
   .
   .
```

- 'system clock' is the current value of global date and time

- 'code' and 'error text' are the actual exception code and the meaning of the code; some of the texts include information about the operands which caused the exception. (The possible texts may be seen in appendix L).

- 'gf' and 'top' are stack references, and the number after 'code' is the instruction which caused the exception.

- The list of ´called from...´ is the dynamic chain of routine activations.

- ´ic´ and ´line ...-...´ are an identification of the calls.

- ´date´ is the compilation date of the modules in question.

Furthermore, if the process is external linked, printout of date, time, and version of the source is appended.

4.1.4 Link

4.1.4

```
FUNCTION link (external_name: alfa; process name): integer;
```

Predefined.

- The process identified by ´external_name´ is searched for in the LINKER catalog. If found the process identified by ´external_name´ is linked to ´process name´.

The function returns the following results:

Results	Meaning
0	Process linked
1	Process name with ´external_name´ was not found in the LINKER catalog.
3	Process with name ´external_name´ is in the LINKER catalog, but the number of parameters or the type of parameters do no match.
6	A process is already linked to process name.

Example 2. LINK

```
PROCESS example2;

CONST
  default_size = 0;
  ok           = 0;

VAR
  sem : semaphore;
  sh  : shadow;

PROCESS example21 (VAR s : semaphore);
EXTERNAL;
```

```
BEGIN
  IF link ('program21', example21) = ok THEN
    IF create ('child', example21 (sem),
      sh, default_size) = ok THEN
END; (* example2 *)
.
```

4.1.5 Pi3 get 4.1.5

```
FUNCTION pi3_get (process name;
     pageno : integer;
     VAR buffer : reference) : integer;
```

Not predefined.

- Page number 'pageno' is returned in the buffer in locations specified by the 'first' and 'last' attributes. 'Next' is initialized to the top index of the actual number of bytes transferred.

Results	Meaning
0	OK
1	The process is not linked.
2	Illegal kind. The process is not linked by a call of PI3LINK.
3	Last page. The buffer contains the last page of the program.
4	Illegal page. 'Pageno' is outside the range 1..lastpage.

4.1.6 Pi3 link 4.1.6

```
FUNCTION pi3_link (external_name : alfa;
     process name) : integer;
```

Not predefined.

- The program of kind DATA identified by 'externalname' is searched for in the LINKER catalog. If found, the program identified by 'externalname' is linked to 'process name'.

  The function returns the same results as the predefined function LINK.

4.1.7 Remove


```
PROCEDURE remove (VAR sh: shadow);
```

Predefined.

- ´Remove´ terminates execution of the child, which cannot be resumed. ´Remove´ also removes all incarnations controlled by the child, their children etc. All the resources of the child are deallocated.

The owner and answer semaphores of messages in all declared pool variables are initialized to a deallocation semaphore controlled by the ALLOCATOR. All messages in reference, semaphore and pool variables are returned with u2 = 1 (not processed).

The shadow variable must refer to a process incarnation (child), otherwise an exception occurs.

After the call the shadow variable is nil.

4.1.8 Setpriority


```
PROCEDURE setpriority (priority : integer);
```

Predefined.

- changes the priority of the calling process incarnation inside the priority classes II and III running on interruption level 0.

An exception occurs if:

- ´priority´ < minpriority
- ´priority´ > maxpriority
- called in a channel statement

4.1.9 Start


```
PROCEDURE start (VAR sh: shadow;
                 priority: integer);
```

Predefined.

- activates a child, which has been created by calling the CREATE routine, or which has been stopped by a call of the STOP procedure.

If the child is already started, the procedure call is dummy.

If ´priority´ = 0, the child enters the coroutine class (class II). If ´priority´ < 0, the child enters the time slice class (class III).

If the activation is actually a reactivation of the child, the child could have been waiting at a semaphore. In that case the WAIT statement will be repeated.

If the child was stopped at an interruption level greater than zero, the child is scheduled directly to the old interruption level and activated as if a timeout has occurred.

An exception occurs, if the shadow variable is nil.

4.1.10 Stop

PROCEDURE stop (VAR sh: shadow);

Predefined.

- stops a child. The associated subtree - if any - is not stopped.

If the child is already stopped, the procedure is dummy.

The child is deactivated and possibly removed from the semaphore where the child is waiting.

If the child is waiting for interrupt on an interruption level greater than zero, the child is removed from the interruption level.

If the shadow variable is nil, an exception occurs.

4.1.11 Trace 4.1.11

PROCEDURE trace (excode : integer);

Predefined.

- calls the standard exception routine and generates the same kind of output on the console as the equivalent exception. The program always continues execution after the call, so 'trace' may be used for testoutput generation.

4.1.12 Unlink 4.1.12

FUNCTION unlink (process name): integer;

Predefined.

- The link to the process linked to process name is deleted, if there exists a link and no incarnations of the process exist.

The function returns the following results:

Results	Meaning
0	Process unlinked successfully.
1	No process was linked to process name.
2	Incarnations of the process are existing.

Example 3. UNLINK

```
PROCESS example3;

CONST
  default_size = 0;
  ok           = 0;

VAR
  sem : semaphore;
  sh  : shadow;

PROCESS example31 (VAR s : semaphore);
EXTERNAL;

BEGIN
  IF link ('program31', example31) = ok THEN
    BEGIN
      IF create ('child', example31 (sem),
        sh, default_size) = ok THEN
```

```
        BEGIN
          remove (sh);
          IF unlink (example31) = ok THEN;
        END;
    END;
END; (* example3 *)
.
```

4.2 Communication Routines 4.2

These routines are used for communication between process incarnations. The state of semaphore, pool and reference variables may be tested, etc. Attention should be paid to the fact, that the routines ALLOC and RELEASE in section 5.3 also may infer communications equivalent to WAIT, SIGNAL, or RETURN.

4.2.1 Definetimer 4.2.1

```
PROCEDURE definetimer (onoff : boolean);
```

Predefined.

- ´definetimer (true)´ includes the incarnation in a chain, where count down of OWN.TIMER is done once per second. The call ´definetimer (false)´ removes the incarnation from the chain, and count down is stopped.

Timeout can only take place after the call ´definetimer (true)´. At most one call of definetimer can be processed each 50 msec.

4.2.2 Delete semaphore 4.2.2

```
FUNCTION delete_semaphore (
     var s_name : ! alfa) : integer;
```

Predefined.

- Searches the semaphore catalog of the calling incarnation, and deletes the entry, if found.

Results	Meaning
0	OK
1	Not found. No semaphore is cata-

logued under the specified name.

4.2.3 Locked


```
FUNCTION locked (VAR sem: semaphore):boolean;
```

Predefined.

- returns the value true, if the semaphore is locked, otherwise false.

4.2.4 Locktest


```
FUNCTION locktest (VAR r: reference): boolean;
```

Not predefined.

- returns the value true if the reference variable is locked, otherwise false.

4.2.5 Name semaphore


```
FUNCTION name_semaphore
  (VAR s: semaphore; s_name: alfa): integer;
```

Not predefined.

- Catalogues the semaphore under the specified name.

Results	Meaning
0	OK, the semaphore is catalogued under the specified name.
1	Overlap, a semaphore is already catalogued under the specified name.
2	No resources, the semaphore cannot be catalogued because of lack of memory resources.

4.2.6 Nil


```
FUNCTION nil (VAR r: ↑niltype): boolean;
```

Predefined.

- returns the value true, if the parameter ´r´ is nil, otherwise false. ´Niltype´ may be the types reference, shadow, or any pointer type, e.g. ↑semaphore.

4.2.7 Open


```
FUNCTION open (VAR sem: semaphore): boolean;
```

Predefined.

- returns the value true, if the semaphore is open, otherwise false.

4.2.8 Ownertest


```
FUNCTION ownertest (VAR p: pool 1;
                    VAR r: reference): boolean;
```

Predefined.

- returns the value true, if the message references by ´r´ originates from the pool ´p´, otherwise false.

An exception occurs if the reference variable is nil.

4.2.9 Passive


```
FUNCTION passive (VAR sem: semaphore): boolean;
```

Predefined.

- returns the value true, if the semaphore is passive, otherwise false.

4.2.10 Ref


```
FUNCTION ref (VAR sem: semaphore): ↑semaphore;
```

Predefined.

- returns a pointer value to the pointed object ´sem´. ´Ref´ can only be applied to variables of type semaphore.

4.2.11 Return 4.2.11

PROCEDURE return (VAR r: reference);

Predefined.

- signals the message referenced by ´r´ to the anonymous answer semaphore.

An exception occurs if:

- the reference variable is nil.
- the reference variable is locked.

4.2.12 Search semaphore 4.2.12

FUNCTION search_semaphore (s_name: alfa): ↑semaphore;

Not predefined

- Retrieves a pointer value of a catalogued semaphore. The retrieval is directed from the leaves to the root of the incarnation tree, starting at the calling incarnation. A nil value is returned, if no catalogued semaphore with the specified name is found. ´

4.2.13 Sendlinker 4.2.13

PROCEDURE sendlinker (VAR r : reference);

Predefined.

- signals the message referenced by ´r´ to the LINKER process. No wait is performed in the procedure.

An exception occurs if:

- the reference variable is nil.
- the reference variable is locked.

User Fields	Message	Answer
u1	function	unchanged
u2	unused	result
u3	param	unchanged
u4	unused	unchanged

An unknown function is returned with result = 6.

Result = 6 is also returned, if the message cannot contain a proper result record.

4.2.13.1 Lookup name

4.2.13.1

User Fields	Message	Answer
u1	3	3
u2	unused	result
u3	unused	unchanged
u4	unused	unchanged

Databuffer
A message of the predefined type ´lookup_descriptor_segment´ (chapter 4).

The LINKER catalog is searched for a program with name ´name´. Some values of ´name´ have special effects, by sending output directly to the OPERATOR semaphore, before returning the request. If the name contains a wild character "*", the whole LINKER catalog is scanned. Programs are listed if the name satisfies a ´wild character compare´. (See the routine WILD_COMPARE). If the name is ´PROCESS´ all programs of kind process are listed. If the name is ´PROCEDURE´ or ´FUNCTION´, all programs of kind procedure or function are listed.

Results	Meaning
0	OK. A program with name ´name´ is found in the LINKER catalog. In the answer, the remaining fields of the type ´lookup_descriptor_segment´ are initialized.
1	No program with name ´name´ is found in the LINKER catalog.

4.2.13.2 Check 4.2.13.2

User Fields	Message	Answer
u1	7	7
u2	unused	result
u3	module	module
u4	unused	unchanged

Data buffer
A message of size 64 words (minimum).

A crc16 check is performed on the module.

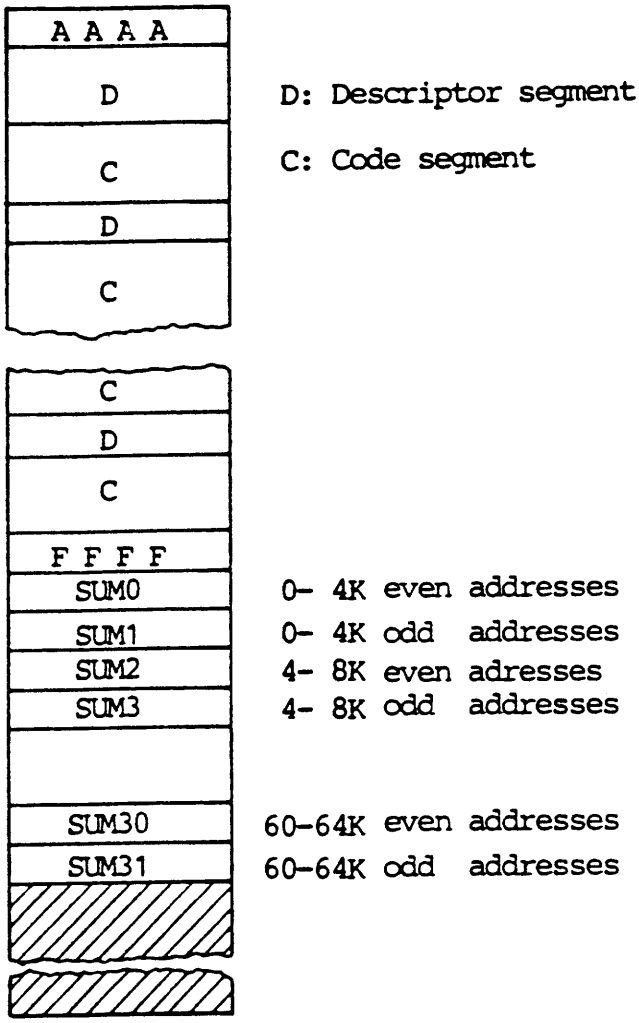
Note: The size of the message must be at least 64 words.

The polynomium used is:

$$x^{16} + x^{12} + x^{5} + 1$$

with remainder = -1 as initial value.

The module must have the format:



The sums include the ´AAAA´ and ´FFFF´ words. The sum words themselves are not included.

The sum check in the module and the computed sum are delivered in the message as:

```
ARRAY (1..32) OF
   RECORD
      promsum,
      expected: integer;
   END;
```

Results	Meaning
0	The check sum values are returned in the data buffer.
1	The module is empty.

4.2.14 Sendtimer


```
PROCEDURE sendtimer (VAR r : reference);
```

Predefined.

- signals the message referenced by ´r´ to the TIMER process. No wait is performed in the procedure.

An exception occurs if:

- the reference variable is nil
- the reference variable is locked

If the buffer cannot contain a message of the predefined type ´delaytype´ (chapter 4) when demanded, the message is refused immediately with u2=4 (unintelligible). Undefined functions are processed as a get clock message.

The subrecord consisting of the fields ´prev_date´, ´prev_time´, and ´prev_secs´ is called ´buffer time´.

4.2.14.1 Short Delay Message

User Fields	Message	Answer
u1	5 (1+4*1)	5
u2	delay1(<>0)	result
u3	delay2	0
u4	unused	unchanged

Data buffer
Not used.

Function
Returns the message after delay1 * 2 ** delay2 msec. ´delay1´ and ´delay2´ must fulfill the relation ´delay1*2**delay2 <= 65535´.

Results	Meaning
0	OK.
1	Not processed. The message is regretted.

4.2.14.2 Long Absolute Delay Message

User Fields	Message	Answer
u1	13 (1+4*3)	13
u2	unused	result
u3	unused	0
u4	unused	unchanged

Data buffer
The buffer is supposed to hold a message of type ´delaytype´ (chapter 4).

Function
The buffer time is set to ´buffer time + inc´. If the new buffer time is before global time, the buffer time is set equal to global time and the message is returned immediately, otherwise the message is returned when global time passes the new buffer time.

Results	Meaning
0	OK.
1	Not processed. The message is regretted.

4.2.14.3 Set Clock Absolute Message

User Fields	Message	Answer
u1	2 (2+4*0)	2
u2	unused	0
u3	unused	0
u4	unused	unchanged

Data buffer
The buffer is supposed to hold a message of type 'delaytype'.

Function
Global time is assigned from the buffer time. ('inc' is not used). The queue of long delay messages is scanned for eventual returns.

The message is returned immediately.

4.2.14.4 Get Clock Message

User Fields	Message	Answer
u1	1 (1+4*0)	1
u2	unused	0
u3	unused	0
u4	unused	unchanged

Data buffer
A message of type 'delaytype'.

Function
Global time is assigned to the buffer time. ('inc' is not used).

The message is returned immediately.

4.2.14.5 Long Relative Delay Message

User Fields	Message	Answer
u1	9 (1+4*2)	9
u2	unused	result
u3	unused	0
u4	unused	unchanged

Data buffer
A message of type 'delaytype'.

Function
The buffer time is set to 'global time + inc'. The message is returned, when global time passes the new value of the buffer time.

Results	Meaning
0	OK.
1	Not processed. The message is regretted.

4.2.14.6 Set Clock Relative Message

User Fields	Message	Answer
u1	6 (2+4*1)	6
u2	unused	0
u3	unused	0
u4	unused	unchanged

Data buffer
A message of type 'delaytype'.

Function
Global time is set to 'global time + inc', and the buffer time is set to this new value of global time. The message is returned immediately. The queue of long delay messages is scanned for eventual returns.

4.2.14.7 Set Clock Interval Message

User Fields	Message	Answer
u1	4 (0+4*1)	4
u2	unused	0
u3	new interval	old interval
u4	unused	unchanged

Data buffer
Not used.

Function
Controls the speed of global time. For every Real Time Clock interrupt, global time is incremented by ´global interval´ msec. Default = 50 msec. Only long delays are influenced by redefinitions of ´global interval´. Short delays and the RTC interrupt frequency are not disturbed.

New interval
´Global interval´ is set to ´new interval´ (msec.).

Old interval
´Global interval´ is returned, before set to ´new interval´ for reestablishment purposes.

4.2.14.8 Regret Message

User Fields	Message	Answer
u1	12 (0+4*3)	12
u2	unused	0
u3	unused	unchanged
u4	unused	unchanged

Data buffer
Not used.

Function
The delay queues are scanned. If a message is found with matching ´owner semaphore´, it is returned with result = 1 (not processed). The regret message is returned immediately. At most one message is regretted per ´regret message´.

4.2.15 Sensesem


```
PROCEDURE sensesem (VAR r : reference;
                    VAR s : semaphore);
```

Predefined.

- If a message is pending at the semaphore, it is delived in ´r´, otherwise ´r´ remains nil.

The caller will not be waiting.

An exception occurs if the reference parameter is not nil.

4.2.16 Signal


```
PROCEDURE signal (VAR r: reference; VAR s: semaphore);
```

Predefined.

- If the semaphore is passive or open, the message referred to by ´r´ becomes the last element of the semaphore´s sequence of messages. If the semaphore is locked, the message is handed over to the first incarnation waiting on the semaphore, and this incarnation is activated.

An exception occurs if:

- the reference variable is nil.
- the reference variable is locked.

4.2.17 Wait


```
PROCEDURE wait (VAR r: reference; VAR s: semaphore);
```

Predefined.

- If the semaphore is open, the first message is removed from the semaphore´s sequence of messages. If the semaphore is passive or locked, the incarnation waits and becomes the last element of the sequence of incarnations waiting on the semaphore. It is resumed by another incarnation calling SIGNAL or RETURN.

The reference parameter must be nil, otherwise an

exception occurs. After a call of ´wait´ it refers to a message.

4.2.18 Waitd 4.2.18

```
PROCEDURE waitd (delay : integer);
```

Predefined.

- waits for the expiration of a delay period.

The variable OWN.TIMER is initialized to ´delay´, and the wait is performed.

Note: Count down of OWN.TIMER only takes place if the routine ´definetimer´ has been called.

4.2.19 Waits 4.2.19

```
PROCEDURE waits (VAR r : reference; VAR s : semaphore);
```

Predefined.

- works as the procedure ´wait´.

4.2.20 Waitsd 4.2.20

```
FUNCTION waitsd (VAR r : reference;
                 VAR s : semaphore;
                 delay : integer   ): activation;
```

Predefined.

- waits for two kinds of event:

  1. The arrival of a message to the semaphore ´s´ (a_semaphore)
  2. Expiration of a delay period (a_delay)

The variable OWN.TIMER is initialized to ´delay´ before the wait is performed.

An exception occurs if the reference ´r´ is not nil.

Note: Count down of OWN.TIMER only takes place, if ´definetimer´ has been been called.

4.3 Routines for Message Manipulation 4.3

The routines in this section are used for allocation and deallocation of messages, besides general manipulation of message stacks and message attributes, such as the size of the associated buffer, the u-attributes, and the buffer index attributes, first, last, and next.

4.3.1 Alloc 4.3.1

```
PROCEDURE alloc(VAR r: reference; VAR p: pool 1;
                VAR s: semaphore);
```

Predefined.

- If the pool of messages is not empty, one of the messages is removed. If the pool is empty, the incarnation waits until a message is released to the pool by another process incarnation calling release. The answer semaphore of the allocated message becomes ´s´.

The reference variable must be nil, otherwise an exception occurs. After the call it refers to a message.

4.3.2 Bufsize 4.3.2

```
FUNCTION bufsize (VAR r : reference) : integer;
```

Predefined.

- If the ´bufsize´ function is called with a parameter with value NIL, an exception occurs. Otherwise the size in bytes of the designated buffer is returned as result.

4.3.3 Empty 4.3.3

```
FUNCTION empty (VAR r: reference): boolean;
```

Predefined.

- true, if the reference variable refers to an unstacked message (one header only), otherwise false.

4.3.4 First 4.3.4

```
FUNCTION first (VAR r : reference) : integer;
```

Predefined.

- If the ´first´ function is called with a parameter with value NIL or the message is a header message only, an exception occurs. Otherwise the result is the value of the ´first´ attribute of the designated buffer.
  NOTE: The ´first´ attribute is the first word in the buffer, according to the driver conventions. A data message always holds the ´first´ attribute. So the buffer is accessed without locking the reference variable.

4.3.5 Initpool 4.3.5

```
FUNCTION initpool (VAR p : pool 0;
                   number          ,
                   size_in_words : integer) : integer;
```

Not predefined.

- supplies the pool variable ´p´ with ´number´ messages of size ´size_in_words´. The result indicates the actual number of messages obtained from the ALLOCATOR. The ´owner´ semaphore in the messages is initialized to the anonymous pool semaphore. That means that the messages are returned to the pool variable by a ´release´ call. The function may be used to obtain a more flexible initialization of pool variables.

Example 4. INITPOOL

```
PROCESS example4;

CONST
  size = 37;

VAR
  driver,
```

```
    s      : semaphore;
    r      : reference;
    p      : pool 0;    (* note: p may be an empty pool *)

    FUNCTION initpool (VAR p : pool 1;
      no, size : integer) : integer;
    EXTERNAL;

  BEGIN
    IF initpool (p, 1, size) = 0 THEN
        (* no available memory now *)
    ELSE
      BEGIN
        alloc  (r, p, s);
          (* now r.answer = ref (s) *)
        signal (r, driver);
        wait   (r,s);
      END;
  END; (* example4 *)
  .
```

4.3.6 Initsem

4.3.6

```
FUNCTION initsem (VAR s : semaphore;
                  number    ,
                  size_in_words : integer) : integer;
```

Not predefined.

- supplies the semaphore ´s´ with ´number´ messages of size ´size_in_words´. The result indicates the actual number of messages obtained from the ALLOCATOR. The ´owner´ and ´answer´ semaphores in the messages cannot be updated. That means that the messages return to the ALLOCATOR by ´return´ or ´release´ calls.

The routine is intended for obtaining temporary resources from the ALLOCATOR. The messages should not be used for communication. INITPOOL should be used if communication is wanted. If you insist, the messages can only be used for direct wait/signal communication, not wait/return communication, unless a message header with other ´owner´ and ´answer´ semaphore values is pushed on top.

4.3.7 Last 4.3.7

```
FUNCTION last (VAR r : reference) : integer;
```

Predefined.

- If the ´last´ function is called with a parameter with value NIL or the message is a header message only, an exception occurs. Otherwise the result is the value of the ´last´ attribute of the designated buffer.
  NOTE: The ´last´ attribute is the second word in the buffer, according to the driver conventions. A data message always holds the ´last´ attribute. So the buffer is accessed without locking the reference variable.

4.3.8 Moveg 4.3.8

```
PROCEDURE moveg

  (count : integer;
   VAR frombyte : byte;
     fromindex  : integer;
   VAR tobyte   : byte;
     toindex    : integer);
```

Not predefined.

- moves ´count´ bytes starting at ´frombyte (fromindex)´ to ´tobyte (toindex)´. It is assumed that frombyte is ´frombyte (0)´, and tobyte is ´tobyte (0)´.

´Count´ is considered an unsigned number in the range 0..65535. So it fulfils

- ult (0, count).
- ult (count , -1).

´count = 0´ means no move.

WARNING: The move is performed with no check at all!!! - and only within the same 64K byte memory module. An exception occurs, if wrap around takes place. The use of dynamic types and assignment is advocated, instead of the use of ´moveg´.

NOTE: The move may be performed inside the same module or between different modules.

The move is in wordmode, if possible.

4.3.9 Next

4.3.9

FUNCTION next (VAR r : reference) : integer;

Predefined.

- If the ´next´ function is called with a parameter with value NIL or the message is a header message only, an exception occurs. Otherwise the result is the value of the ´next´ attribute of the designated buffer.
  NOTE: The ´next´ attribute is the third word in the buffer, according to the driver conventions. A data message always holds the ´next´ attribute. So the buffer is accessed without locking the reference variable.

4.3.10 Openpool

4.3.10

FUNCTION openpool (VAR p: pool 1): boolean;

Predefined.

- returns the value true if the pool contains a message, otherwise false.

4.3.11 Pop

4.3.11

PROCEDURE pop (VAR r1, r2: reference);

Predefined.

- The top message header from ´r2´ is removed. If the new top message and the old top message refer to the same data buffer, only the message header is removed. If not, the top message data is removed also. ´r1´ refers to the removed message.

Exceptions occur if:

- r1 is not nil before call
- r2 is nil before call
- r2 is locked before call

If empty (r2)=true before the call, r2 becomes nil after the call.

4.3.12 Push

PROCEDURE push (VAR r1, r2: reference);

Predefined.

- The header referred to by ´r1´ becomes the new top header of the stack. After the call, ´r2´ refers to the new stack.

If the new top message is a header message, the top message data of ´r2´ remains the same. After the call ´r1´ is nil.

The message accessible through ´r2´ (possibly nil) is called the stack.

Exceptions occur if:

- ´r1´ is nil before call
- ´r1´ is locked before call
- ´r2´ is locked before call
- not empty (r1) before call
- ´r1´ = ´r2´ before call

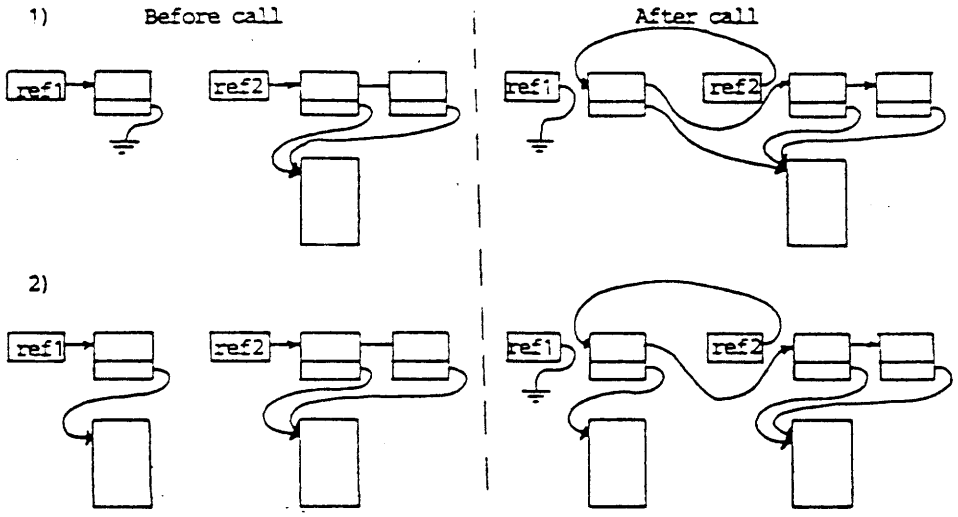


Fig. 5. Example on the behaviour of push

4.3.13 Release 4.3.13

PROCEDURE release (VAR r: reference);

Predefined.

- signals the message referenced by ´r´ to the anonymous owner semaphore.

  An exception occurs if:

- the reference variable is nil.
- the reference variable is locked.

4.3.14 Releasepool 4.3.14

```
FUNCTION releasepool (VAR p: pool 1;
                      number: integer): integer;
```

Not predefined.

- returns a maximum of ´number´ messages from the pool ´p´ to the ALLOCATOR. The actual number of buffers released is returned as the result.

4.3.15 Setfirst 4.3.15

```
PROCEDURE setfirst (VAR r : reference; val : integer);
```

Predefined.

- If the ´setfirst´ procedure is called with a reference parameter with value NIL, or the message is a header message only, an exception occurs. Otherwise the value of the ´val´ parameter is assigned to the ´first´ attribute of the designated buffer.
  NOTE: The ´first´ attribute is the first word in the buffer, according to the driver conventions. A data message always holds the ´first´ attribute. So the buffer is updated without locking the reference variable.

4.3.16 Setlast 4.3.16

```
PROCEDURE setlast (VAR r : reference; val : integer);
```

Predefined.

- If the ´setlast´ procedure is called with a reference parameter with value NIL, or the message is a header message only, an exception occurs. Otherwise the value of the ´val´ parameter is assigned to the ´first´ attribute of the designated buffer.
  NOTE: The ´last´ attribute is the second word in the buffer, according to the driver conventions.

A data message always holds the ´last´ attribute. So the buffer is updated without locking the reference variable.´

4.3.17 Setnext


```
PROCEDURE setnext (VAR r : reference; val : integer);
```

Predefined.

- If the ´setnext´ procedure is called with a reference parameter with value NIL, or the message is a header message only, an exception occurs. Otherwise the value of the ´val´ parameter is assigned to the ´next´ attribute of the designated buffer.
  NOTE: The ´next´ attribute is the third word in the buffer, according to the driver conventions. A data message always holds the ´next´ attribute. So the buffer is updated without locking the reference variable.

4.3.18 Setul


```
PROCEDURE setul (VAR r : reference; b : byte);
```

Predefined.

- If the ´setul´ procedure is called with a reference parameter with value NIL an axception occurs. Otherwise the value of the parameter ´b´ is assigned to the ul-attribute of the designated message. A call is equivalent to "r↑.ul:= b".

4.3.19 Setu2


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

Predefined.

- If the ´setu2´ procedure is called with a reference parameter with value NIL an axception occurs. Otherwise the value of the parameter ´b´ is assigned to the u2-attribute of the designated message. A call is equivalent to "r↑.u2:= b".

4.3.20 Setu3 4.3.20

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

Predefined.

- If the ´setu3´ procedure is called with a reference parameter with value NIL an axception occurs. Otherwise the value of the parameter ´b´ is assigned to the u3-attribute of the designated message. A call is equivalent to "r↑.u3:= b".

4.3.21 Setu4 4.3.21

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

Predefined.

- If the ´setu4´ procedure is called with a reference parameter with value NIL an axception occurs. Otherwise the value of the parameter ´b´ is assigned to the u4-attribute of the designated message. A call is equivalent to "r↑.u4:= b".

4.3.22 Tofrom 4.3.22

```
PROCEDURE tofrom (
     VAR toref : reference; toindex : integer;
     VAR fromref : reference; fromindex : integer;
     bytes : integer);
```

Not predefined.

- Moves ´bytes´ bytes from the buffer ´fromref´ starting at the location index ´fromindex´ to the buffer ´toref´ starting at the location index ´toindex´. Indexing in the buffers is according to the driver conventions. The ´bytes´ and indix parameters are considered integers in the range 0..65535.

Note: ´tofrom´ should be used instead of ´moveg´.

4.3.23 U1


```
FUNCTION u1 (VAR r : reference) : byte;
```

Predefined.

- If the 'u1' function is called with a parameter with value NIL, an exception occurs. Otherwise the result is the u1-attribute of the designated message. A call is equivalent to "r↑.u1".

4.3.24 U2


```
FUNCTION u2 (VAR r : reference) : byte;
```

Predefined.

- If the 'u2' function is called with a parameter with value NIL, an exception occurs. Otherwise the result is the u2-attribute of the designated message. A call is equivalent to "r↑.u2".

4.3.25 U3


```
FUNCTION u3 (VAR r : reference) : byte;
```

Predefined.

- If the 'u3' function is called with a parameter with value NIL, an exception occurs. Otherwise the result is the u3-attribute of the designated message. A call is equivalent to "r↑.u3".

4.3.26 U4


```
FUNCTION u4 (VAR r : reference) : byte;
```

Predefined.

- If the 'u4' function is called with a parameter with value NIL, an exception occurs. Otherwise the result is the u4-attribute of the designated message. A call is equivalent to "r↑.u4".

4.4 Conversion and Arithmetic Routines 4.4

4.4.1 The Predefined Routines Chr, Ord, Pred, Succ 4.4.1

4.4.1.1 Chr 4.4.1.1

FUNCTION chr(int: 0..127): char;

- returns the character with the ordinal value ´int´.

4.4.1.2 Ord 4.4.1.2

FUNCTION ord (x: otype): integer;

- returns the ordinal value of the variable ´x´ of type ´otype´, where ´otype´ is any ordinal type.

4.4.1.3 Pred 4.4.1.3

FUNCTION pred (x: otype): otype;

- gives the predecessor of the variable ´x´ of type ´otype´, where ´otype´ is any ordinal type.

4.4.1.4 Succ 4.4.1.4

FUNCTION succ (x: otype): otype;

- gives the successor of the variable ´x´ of type ´otype´, where ´otype´ is any ordinal type.

4.4.2 Double Arithmetics Routines 4.4.2

DOUBLEENV contains four constant declarations, besides declarations of the following double arithmetics routines:

```
double_one  = double (0, 1);
double_max  = double (maxint, -1);
double_min  = double (minint, 0);
```

```
double_zero = double (0, 0);
```

4.4.2.1 Double add

4.4.2.1

```
FUNCTION double_add (d1, d2 : double) : double;
```

Defined in DOUBLEENV.

- Addition in the range -2147483648..2147483647.

An exception occurs, if the result is outside this range.

4.4.2.2 Double div

4.4.2.2

```
FUNCTION double_div (d1, d2 : double) : double;
```

Defined in DOUBLEENV.

- Division in the range -2147483648..2147483647.

An exception occurs, if ´d2´ = ´double_zero´.

4.4.2.3 Double lt

4.4.2.3

```
FUNCTION double_lt (d1, d2 : double) : boolean;
```

Defined in DOUBLEENV.

- ´less than´ (´d1´ < ´d2´) in the range -2147483648..2147483647.

4.4.2.4 Double int

4.4.2.4

```
FUNCTION double_int (i : integer) : double;
```

Defined in DOUBLEENV.

- Converts a variable of type integer to a variable of type ´double´.

4.4.2.5 Double madd 4.4.2.5

FUNCTION double_madd (d1, d2 : double) : double;

Defined in DOUBLEENV.

- Addition in the range -2147483648..2147483647, but with no overflow exception.

4.4.2.6 Double mod 4.4.2.6

FUNCTION double_mod (d1, d2 : double) : double;

Defined in DOUBLEENV.

- Modulo operation in the range -2147483648..2147483647.

An exception occurs, if ´d2´ = ´double_zero´.

4.4.2.7 Double msub 4.4.2.7

FUNCTION double_msub (d1, d2 : double) : double;

Defined in DOUBLEENV.

- Subtraction in the range -2147483648..2147483647, but with no overflow exception.

4.4.2.8 Double mul 4.4.2.8

FUNCTION double_mul (d1, d2 : double) : double;

Defined in DOUBLEENV.

- Multiplication in the range -2147483648..2147483647.

An exception occurs, if the result is outside this range.

4.4.2.9 Double sub

4.4.2.9

```
FUNCTION double_sub (d1, d2 : double) : double;
```

- Subtraction in the range -2147483648..2147483647.

An exception occurs, if the result is outside this range.

4.4.2.10 Double dec

4.4.2.10

```
PROCEDURE double_dec (VAR d : double);
```

Defined in DOUBLEENV.

- Decrements the parameter ´d´ by one in the range -2147483648..2147483647, but with no overflow exception.

4.4.2.11 Double inc

4.4.2.11

```
PROCEDURE double_inc (VAR d : double);
```

Defined in DOUBLEENV.

- Increments the parameter ´d´ by one in the range -2147483648..2147483647, but with no overflow exception.

4.4.3 Miscellaneous Arithmetic Routines 4.4.3

4.4.3.1 Abs 4.4.3.1

```
FUNCTION abs(x:integer): integer;
```

Predefined.

- The absolute value of the variable 'x' is delivered as result.

4.4.3.2 Crc16 4.4.3.2

```
FUNCTION crc16 (op1, op2 : integer) : integer;
```

Predefined.

- 'op1' represents the polynomium:

  f (x) =a15 * x↑15 + a14 * x↑14 + ... + a1 * x + a0

  where aj = op1.bitj.

  (Note: Bit 0 is the most significant bit).

- 'op2' represents the polynomium:

  g (x) = x↑16 + b15 * x↑15 + b14 * x↑14 + ... + b1 * x + b0

  where bj = op2.bitj.

  Note that x↑16 by convention is implicitly given.

  The routine delivers the remainder by the division:

  (f (x) * x↑8/g (x)

Example 5. CRC16

```
PROCEDURE example5;
    (* generate the crc16 remainder *)
    (* to be placed in the last *)
    (* positions of a buffer    *)

CONST
  quotient = -32768 + 8192 + 1;
```

```
    (* x**16 + x**15 + x**2 + 1 *)
  n         = 768;

VAR
  remainder ,
  i         : integer;
  buffer    : ARRAY (1 .. n + 2) OF byte;

BEGIN
  remainder := 0; (* init value *)
  FOR i := 1 TO n DO
    remainder := crc16 (remainder xor buffer(i), quotient);
    (* now remainder contains the crc16 remainder *)
    (* the least significant byte of remainder    *)
    (* is supposed to be sent first               *)
  buffer (n + 1) := remainder AND 255;
  buffer (n + 2) := swap (remainder) AND 255;
END; (* example5 *)
.
```

4.4.3.3 Increment mod 64k

4.4.3.3

```
PROCEDURE increment_mod_64k (VAR i : integer);
```

Not predefined.

- increments the parameter ´i´ by one with no overflow exception. The procedure is intended for statistical purposes - like i:= i + 1 -, where the access path to the parameter ´i´ is hard. The address of ´i´ is only calculated once.

4.4.3.4 Intel

4.4.3.4

```
FUNCTION intel (i : integer) : intel_integer;
```

Not predefined.

- converts an RC3502 integer to an INTEL integer by exchanging the least and most significant byte.

An ´intel_integer´ is defined as:

```
intel_integer = RECORD
                  low    : byte; (* least significant *)
```

```
                    high  : byte; (* most significant *)
                  END;
```

Example: See example under LAMBDA.

4.4.3.5 Lambda 4.4.3.5

FUNCTION lambda (ii : intel_integer) : integer;

Not predefined.

- converts an INTEL integer to an RC3502 integer by exchanging the least and most significant byte.

An INTEL integer is defined as:

```
intel_integer = RECORD
                  low  : byte; (* least significant *)
                  high : byte; (* most significant *)
                END;
```

Example 6. INTEL and LAMBDA

```
PROCEDURE example6;

TYPE
  intel_integer = RECORD
                    low, high : byte;
                  END;

  FUNCTION intel (i : integer) : intel_integer;
  EXTERNAL;

  FUNCTION lambda (ii : intel_integer) : integer;
  EXTERNAL;

VAR
  ii : intel_integer;
  i  : integer;

BEGIN
  ii.low  := 15;
  ii.high := 2;
  i       := lambda (ii);
    (* now i = 2*256 + 15 *)
  i       := 259;
  ii      := intel (i);
```

```
        (* now ii.low = 3 and ii.high = 1 *)
    END; (* example6 *)
    .
```

4.4.3.6 Madd 4.4.3.6

FUNCTION madd (a,b : integer) : integer;

Not predefined.

- addition, but with no overflow exception.

4.4.3.7 Mmul 4.4.3.7

FUNCTION mmul (a,b : integer) : integer;

Not predefined.

- multiplication, but with no overflow exception.

4.4.3.8 Msub 4.4.3.8

FUNCTION msub (a,b : integer) : integer;

Not predefined.

- Subtraction, but with no overflow exception.

4.4.3.9 Rotate 4.4.3.9

FUNCTION rotate (a, shifts : integer) : integer;

Not predefined.

- makes a cyclic shift of the parameter ´a´ ´shifts´ times. The result of the rotation is returned by the function. If ´shifts´ is positive, the shift is to the left; otherwise the shift is to the right.

4.4.3.10 Swap

FUNCTION swap (i : integer): integer;

Predefined.

- The conversion takes place by swapping the two bytes of the integer i. This effective way of swapping may be used for conversion between integers and the u-fields in message headers.

Example 7. SWAP

```
PROCESS example7;

VAR
  i ,
  j : integer;
  r : reference;
  s : semaphore;

BEGIN
  wait (r, s);
  WITH r DO
    BEGIN
        (* i := u2.u3 *)
      i  := swap (u2) OR u3;
        (* u3.u4 := j *)
      u3 := swap (j) AND 255;
      u4 := j AND 255;
    END;
END; (* example7 *)
.
```

4.4.3.11 Uadd

FUNCTION uadd (a, b : integer) : integer;

Not predefined.

- Addition in the range 0..65535.

An exception occurs, if the result is outside the range 0..65535.

4.4.3.12 Udiv 4.4.3.12

FUNCTION udiv (a, b : integer) : integer;

Not predefined.

- Division in the range 0..65535.

An exception occurs, if ´b´ = 0.

4.4.3.13 Ult 4.4.3.13

FUNCTION ult (a, b : integer) : boolean;

Not predefined.

- ´less than´ (´a´ < ´b´) in the range 0..65535.

4.4.3.14 Umod 4.4.3.14

FUNCTION umod (a, b : integer) : integer;

Not predefined.

- Modulo operation in the range 0..65535.

An exception occurs, if ´b´ = 0.

4.4.3.15 Umul 4.4.3.15

FUNCTION umul (a, b : integer) : integer;

Not predefined.

- Multiplication in the range 0..65535.

An exception occurs, if the result is outside the range 0..65535.

4.4.3.16 Usub 4.4.3.16

FUNCTION usub (a, b : integer): integer;

Not predefined.

- Subtraction in the range 0..65535.

An exception occurs, if the result is outside the range 0..65535.

4.5 Clock Routines 4.5

This section describes a number of predefined routines for operations on the predefined types ´clocktype´ and ´coded_inc´.

4.5.1 Getclock 4.5.1

FUNCTION getclock : clocktype;

Predefined.

- Returns the current value of the system clock. The type ´clocktype´ is predefined (see chapter 4). As opposed to the ´getclock´ request, when using the SENDTIMER procedure, the ´getclock´ function works without message communications with the TIMER.

4.5.2 Clock difference 4.5.2

FUNCTION clock_difference (
t1,t2 : clocktype) : coded_inc;

Predefined.

- Returns the difference between two clock values. If the difference exceeds 32 days, the value 31 days, 23 hours, 59 minutes, 59 seconds, and 999 milliseconds is returned.

4.5.3 Clock increment 4.5.3

```
FUNCTION clock_increment (
     t : clocktype;
     inc : coded_inc): clocktype;
```

Predefined.

- Returns the clock value ´t+inc´.

4.5.4 Clock less than 4.5.4

```
FUNCTION clock_less_than (
     t1,t2 : clocktype) : boolean;
```

Predefined.

- Returns the value ´true´ if ´t1´ precedes ´t2´, otherwise ´false´.

4.6 Miscellaneous Routines 4.6

4.6.1 Getid 4.6.1

FUNCTION getid : integer;

Not predefined.

- returns the value of the 16 bit ID register, IDR201. The register is normally installed on RC3502 when used as a node in a communications network.

4.6.2 Getlfgf 4.6.2

PROCEDURE getlfgf (VAR lf, gf : 32bittype);

Not predefined.

- delivers the current values of the stack pointers 'local frame' and 'global frame'. Together with the procedure 'print' it is possible to print specific areas of a stack.

Example 8. GETLFGF

```
PROCESS example8;

TYPE
  addr = RECORD
           base, disp : integer;
         END;

  PROCEDURE getlfgf (VAR lf, gf : addr);
  EXTERNAL;

  PROCEDURE print (VAR z : zone; base : byte;
    first_disp, last_disp, words_per_line : integer);
  EXTERNAL;

VAR
  lf ,
  gf : addr;
  z  : zone;

  PROCEDURE exception (code : integer);
  BEGIN
      (* the zone must be initialized for output *)
```

```
      getlfgf (lf, gf);
      print   (z, gf.base, gf.disp, lf.disp, l);
    END;

  BEGIN
      (* process body *)
  END; (* example 8 *)
  .
```

4.6.3 Memerrorlog

4.6.3

```
FUNCTION memerrorlog (baseword : integer) : integer;
```

Not predefined.

- returns an errorlog word from the memory module specified by the parameter ´baseword´, which is the base of the first 64K byte RAM module on MEM205. For further details consult "MEM205 General Information" (ref. 6).

4.6.4 Readram

4.6.4

```
PROCEDURE readram (VAR result: byte; index: integer);
```

Not predefined.

- returns the value of the byte with index ´index´ in the control microprocessor ram.

Relevant indices are:

```
 9: version
10: switches 0-7
11: switches 8-15
42: COM204 mask
43: COM204 result
```

(For further information, refer to ref. 4, appendix K).

4.6.5 Restart 4.6.5

PROCEDURE restart;

Not predefined.

- Performs a remove, create, and start of ADAM. All messages at OPERATOR and TIMER are deallocated. The programs in the LINKER catalog are not unloaded.

4.6.6 Setwatchdog 4.6.6

PROCEDURE setwatchdog (i : integer);

Not predefined.

- the "watchdog high" byte in the control microprocessor is initialized to ´i´.

  ´i´ = 1 ~ 640 msec.
  ´i´ = 255 ~ 163,200 msec.

´i´ = 0 disables the watchdog function.

4.6.7 Wild compare 4.6.7

FUNCTION wild_compare (
    name, key : alfa) : boolean;

Not predefined.

- The value ´key´ may contain wild characters, "*", where one wishes to specify zero or more occurrences of "I don´t care" characters. E.g.: ´a*b´ matches ´ab´, ´aab´, ´abb´, ´acdegfb´, and in fact any name that begins with an "a" and ends with a "b".

  The value ´true´ is returned if this comparison is successful, otherwise ´false´.

4.7 Routines for Operator Communication 4.7

This section describes a set of routines which may be used for input/output to the OPERATOR process or another character oriented input/output driver.

Communication takes place via variables of type zone.

The type zone is a simple implementation of the zone concept known from ALGOL8 and MUSIL.

Section 4.7.1 describes the initialization of variables of type zone.

Section 4.7.2 describes the output procedure OUTCHAR and a set of output procedures, which use the procedure OUTCHAR.

Section 4.7.3 describes the input procedure INCHAR and a set of input procedures, which use the procedure INCHAR.

IOENVIR contains declarations of the types and routines and is listed in appendix K.

Appendix L contains a more comprehensive example of OPERATOR communication.

The routines conform to the driver conventions (ref 3) regarding the use of the user u-fields and the buffer indices.

4.7.1 Initialization 4.7.1

The type zone is defined in the environment IOENVIR as:

```
zone = RECORD
          driver      ,
          answer      : ↑ semaphore;
          dataready   ,
          free        : semaphore;
          cur         : reference;
          u2val       ,
          state       : byte
          readstate   ,
          nextp       ,
          lastpos     : integer
       END;
```

´driver´ - points to the driver semaphore (e.g. the OPERATOR semaphore).

´answer´ - points to the semaphore, where answers arrive from the driver.

´dataready´ - holds the answers from the driver, if this semaphore is specified as ´answer´ in the call of openzone or openopzone. ´dataready´ is normally specified, when the zone is used for input from OPERATOR.

´free´ - holds the empty messages.

´cur´ - refers the message which is currently in use for reading input or writing output.

´u2val´ - whenever a message is signalled to the driver semaphore, the u2 field (result field) in the message is initialized to u2val. According to the driver conventions, some drivers utilizes the specific convention, that ´u2´ never takes the value 7 in answers from a driver, thereby distinguishing messages from application processes and answers from e.g. internal driver interruption processes.

´state´ - holds the result field (u2) from the message referred by ´cur´. State is updated when a message is taken from ´free´ as a new current message for output in the procedure outchar, or when a message is taken from ´dataready´ in the procedure opwait as a new current message holding input data.

´readstate´ - specifies the state of the zone when used for input after each call of one of the input routines.

Readstate is initialized to -1 after call of ´openzone´ or ´openopzone´.

The following interpretation of readstate holds:

´readstate´ = - 1
No input was ready (´cur´ = ´nil´) when an input procedure was called or the current input buffer became empty during call.

´readstate´ = 0
The call of an input procedure succeeded with no syntax errors.

´readstate´ > 0
This value range is intended for indication of a syntax error detected during the call of the input procedure. The routines in this manual do not deliver positive values.

´nextp´ - is the index of the next position in current message for reading or writing.

´lastpos´ - is the index of the last position in current message for reading or writing.

4.7.1.1 Openzone


```
PROCEDURE openzone (
            VAR z               : zone;
            driver              ,
            answer              : ↑ semaphore;
            bufs                : integer;
            VAR home            : pool 1;
            v1, v2, v3, v4      : byte);
```

Defined in IOENVIR.

Openzone prepares the zone ´z´ with messages for input/output.

Note: If OPERATOR communication is wanted, use OPENOPZONE.

´z´ - the zone which is initialized.

'driver' - a pointer to the driver semaphore, which is assigned to 'z.driver'.

'answer' - a pointer to a semaphore where answers arrive from the driver. 'answer' is assigned to 'z.answer'.

'bufs' - specifies the number of messages which the procedure will allocate the zone z. The messages are placed in 'z.free'.

'home' - the messages are allocated from the pool 'home'. The messages conform to the driver conventions and should be able to hold a variable of type:

```
RECORD
   first, last, next: integer;
   chars : ARRAY (6..max) OF char;
END;
```

'v1', 'v2', 'v3', 'v4' - the u-fields in the messages are initialized to 'v1', 'v2', 'v3', and 'v4'. 'v2' is used to reset the result field (u2), whenever a message is signalled to the driver.

4.7.1.2 Openopzone

4.7.1.2

```
PROCEDURE openopzone (
            VAR z              : zone;
            driver             ,
            answer             : ↑ semaphore;
            bufs               : integer;
            VAR home           : pool 1;
            v1, v2, v3, v4     : byte);
```

Defined in IOENVIR.

- Messages from the pool 'home' should be able to hold variables of type:

```
RECORD
   first, last, next: integer;
   name  : alfa;
   chars : ARRAY (firstindex..lastindex) OF char;
END;
```

´first´, ´last´, and ´next´ are the buffer indices, according to the driver conventions.

A type ´opbuffer´ is defined in IOENVIR with ´lastindex - firstindex - 1 = 80´.

The procedure initializes the field ´name´ to ´own.incname´ in all messages, whereafter it performs as the procedure openzone.

The OPERATOR process identifies the input and output messages by means of the parameter ´name´.

If the actual driver semaphore pointer is nil (an undefined semaphore pointer), the procedure will automatically open the zone with the operator semaphore as driver semaphore.

4.7.2 Output

If the process does not want to be activated, when an output message returns from the driver, it uses:

```
ref (z.free)
```

as the actual parameter ´answer´ in the call of ´openzone´ or ´openopzone´. In this way empty output messages return directly to the semaphore ´z.free´ as available messages for continued output.

If the process wants to supervise the answers, it uses:

```
ref ("mainsemaphore")
```

in the call of openzone or openopzone. The messages must be signalled to ´z.free´ afterwards.

4.7.2.1 Outend


```
PROCEDURE outend (VAR z: zone);
```

Defined in IOENVIR.

- signals he current message ´z.cur´ to the driver semaphore ´z.driver↑´.

4.7.2.2 Outchar

PROCEDURE outchar (VAR z: zone; ch: char);

Defined in IOENVIR.

- places the character ´ch´ in the current message ´z.cur´.

If no current message is available, a wait is performed on the semaphore ´z.free´.

If the message becomes full, the procedure OUTEND is called.

4.7.2.3 Outalfa

PROCEDURE outalfa (VAR z: zone; text: alfa);

Defined in IOENVIR.

- writes the variable ´text´ by calling outchar. The character ´#´ acts as a stop character.

4.7.2.4 Outtext

PROCEDURE outtext (VAR z: zone; text: alfa);

Defined in IOENVIR.

- works as ´outalfa´, which should be used instead of outtext.

4.7.2.5 Outfill

PROCEDURE outfill (VAR z: zone; filler: char; rep: integer);

Defined in IOENVIR.

- repeats the character ´filler´, ´rep´ times.

If ´rep´ is negative, no fill character is written.

4.7.2.6 Outinteger

PROCEDURE outinteger (VAR z: zone; i, pos: integer);

Defined in IOENVIR.

- writes the number ´i´ on decimal form. If the number occupies less than ´pos´ positions (including the ´-´ character, if negative), the number is prefixed a number of spaces to fill the ´pos´ positions. If the number occupies more than ´pos´ positions, all significant digits are written (inclusive a possible minus sign).

4.7.2.7 Outhex

PROCEDURE outhex (VAR z: zone; i, pos: integer);

Defined in IOENVIR.

- writes the number ´i´ in hexadecimal form. If pos is greater than four, pos -4 space characters are prefixed the number. If ´pos´ is less than four, the following is valid. If the number occupies more than ´pos´ positions, then number is printed as four hex characters. If the number occupies less than ´pos´ positions, the number is prefixed a number of zeroes to fill the ´pos´ positions.

4.7.2.8 Outdouble

PROCEDURE outdouble (VAR z: zone; d: double; pos: integer);

Defined in IOENVIR.

- writes the double ´d´ on decimal form. If the double occupies less than ´pos´ positions (including the ´-´ character, if negative), the double is prefixed a number of spaces to fill the ´pos´ positions. If the number occupies more than ´pos´ positions, all significant digits are

written (inclusive a possible minus sign).

4.7.2.9 Outaddr

PROCEDURE outaddr (VAR z : zone; a : 32bittype);

Not predefined.

- writes the three least significant bytes of the variable a in hexadecimal form with the layout:

  BB.DDDD

The routine is intended for output of RC3502 addresses.

4.7.2.10 Outdate

PROCEDURE outdate (VAR z: zone; date: coded_date);

Defined in IOENVIR.

- writes the parameter date with the layout:

  YYYY.MM.DD

The type ´coded_date´ is predefined and is used throughout the run time system, especially the timer system (see description of the procedure sendtimer)).

4.7.2.11 Outtime

PROCEDURE outtime (VAR z: zone; time: coded_time);

Defined in IOENVIR.

- writes the parameter time with the layout:

  HH.MM

The type coded_time is predefined.

4.7.2.12 Outnl

4.7.2.12

```
PROCEDURE outnl (VAR z: zone);
```

Defined in IOENVIR.

- writes the character nl and signals the message to the driver by calling outend.

4.7.2.13 Print

4.7.2.13

```
PROCEDURE print (VAR z: zone; base : byte;
                              first_disp,
                              last_disp,
                              words_per_line : integer);
```

Not predefined.

- prints the memory words:

  'base.first_disp' through 'base.last_disp'

with the layout:

```
<address> {<word hex>  <word decimal>
           <MSB decimal> <LSB decimal>
           <MSB char> <LSB char>   } words_per_line
                                     <nl>
                                   1
```

The procedure terminates if the 'state' field of the zone becomes not OK (<> 0).

4.7.2.14 Print descriptor

4.7.2.14

```
PROCEDURE print_descriptor (VAR z : zone;
                            VAR d : lookup_descriptor_segment);
```

Not predefined.

- prints selected fields from the predefined type lookup_descriptor_segment with the layout:

```
{PROCESS  } <name><date><time><no_of_pages>
{PROCEDURE}       <pagesize><last_page_length>
{FUNCTION }       <default appetite><no_of_params><nl>
```

A record of type lookup_descriptor_segment may be obtained by means of the ´sendlinker (lookupname)´ routine. See 5.2.10.

4.7.2.15 Printmessage


```
PROCEDURE printmessage (VAR z: zone;
                        VAR r: reference;
                        firstindex,
                        lastindex,
                        words_per_line: integer);
```

Defined in IOENVIR.

- prints the fields from the message header and the specified area of the data part. The message is supposed to be of type:

```
buffer = ARRAY (0..max) OF byte
```

The printed area is from ´buffer (firstindex)´ to ´buffer (lastindex)´.

4.7.3 Input

The input procedures consist of routines for communication with the driver (OPERATOR) process, i.e. OPIN and OPWAIT, besides a set of routines for converting the contents of an input buffer to variables of type ´char´, ´integer´, or ´alfa´.

If the process wants to be activated only when an input message returns from the driver, it uses:

```
ref (z.dataready)
```

as the actual parameter ´answer´ in the call of ´openzone´ or ´openopzone´.

More commonly, the process is also activated by other events. In that situation, it specifies:

```
ref (´mainsemaphore´)
```

as the actual parameter ´answer´ in the call of ´openzone´ or ´openopzone´ and uses ´opanswer´ to place the message in the input zone for further reading.

4.7.3.1 Opin

```
PROCEDURE opin (VAR z: zone);
```

Defined in IOENVIR.

- signals a message from ´z.free´, if any, to the driver semaphore ´z.driver↑´.

4.7.3.2 Opwait

```
PROCEDURE opwait (VAR z: zone; VAR inputpool: pool 1);
```

Defined in IOENVIR.

- is used to wait for specific input to the zone, which returns directly to ´z.dataready´, or to a mainsemaphore together with other messages. If a message is queued at ´z.dataready´, this message is taken. Otherwise, a wait is performed on ´z.answer↑´.

´opwait´ checks, that an arriving message originates from the ´inputpool´ when the zone was opened, and that (ul MOD 8) = 1 (read). Other arriving messages are queued temporarily in the zone until a zone message returns. The queued messages are put back in the mainsemaphore and the zone message prepared for later calls of INCHAR.

4.7.3.3 Opanswer


```
PROCEDURE opanswer (VAR r : reference; VAR z : zone);
```

Defined in IOENVIR.

- signals the message referenced by ´r´ to the semaphore ´z.dataready´.

The routine is intended for use, when the process waits at a main semaphore and sorts out all arriving input messages.

4.7.3.4 Optest


```
FUNCTION optest (VAR z: zone): boolean;
```

Defined in IOENVIR.

- is true if a message is queued at ´z.dataready´, otherwise false. This may be used to avoid a wait in the procedure opwait.

Example:

```
IF optest (z) THEN
   BEGIN
      opwait (z, inputpool);
      (* process inputdata from zone z *)
   END
   ELSE
      (* do something else *)
```

4.7.3.5 Inchar

PROCEDURE inchar (VAR z: zone; VAR ch: char);

Defined in IOENVIR.

- the next character from the current message ´z.cur´ is read.

If the message becomes empty, it is signalled to ´z.free´.

After the call the variable ´z.readstate´ indicates the state of the zone with the interpretation:

z.readstate	
0	Successful reading.
- 1	This buffer was empty before call. The character nl is returned.

4.7.3.6 Ininteger

PROCEDURE ininteger (VAR z: zone; VAR i: integer);

Defined in IOENVIR.

- reads a decimal number according to the pseudo syntax:

$$\left\{\text{<nondigit>}\right\}_0^n \left\{\begin{matrix}+\\-\end{matrix}\right\}_0^1 \left\{\text{<digit>}\right\}_1^n$$

Digits are read as long as the number is in the range -32768..32767.

z.readstate	
0	At least one digit is read.
- 1	No digit was met in the buffer. The buffer is empty and the value 0 is returned.

Examples: Input Result (i):

```
↓       ↓
a b c 1 2 x a b c d e                    12
↓         ↓
a b c + - - + 1 7 9 a b                  179
↓                   ↓
a b c + - 0 0 1 2 3 4 5 6 7             -12345
↓       ↓
a b c 3 2 7 6 8                          3276
```

(↓ indicates the value of nextp before and after the call).

4.7.3.7 Inhex


```
PROCEDURE inhex (VAR z: zone; VAR i: integer);
```

Defined in IOENVIR.

- reads a hexadecimal number according to the pseudo syntax:

$$\left\{ \text{<non hex digit>} \right\}_{0}^{n} \left\{ \text{<hex digit>} \right\}_{1}^{n}$$

z.readstate	
0	At least one hex digit is read.
-1	No hex digit was met in the buffer. The value 0 is returned and the buffer is empty

Hex digits are read as long as the number is in the range #h0000..#hffff.

4.7.3.8 Indouble


```
PROCEDURE indouble (VAR z : zone; VAR d : double);
```

Defined in IOENVIR.

- works as ININTEGER except that reading is in the range -2147483648..2147483647.

4.7.3.9 Inname

4.7.3.9

PROCEDURE inname (VAR z: zone; VAR name: alfa);

Defined in IOENVIR.

- reads a name of maximum 12 characters after the syntax:

$$\left\{\text{<not letter or \_>}\right\}_{0}^{n}\ \text{<letter or \_>}\ \left\{\text{<letter, digit or \_>}\right\}_{0}^{11}$$

Example:	Result:
a bc	a
_abc89_6c;	_abc89_6c
12ab	ab

The characters are delivered in the parameter ´name´ from left to right. ´name´ is not initialized by ´inname´, so ´name´ must be initialized before the call. The variable ´z.nextp´ may be used to calculate the number of characters read (inclusive leading blanks).

z.readstate	
0	At least one <letter or _> is transferred to ´name´.
- 1	No <letter or _> was met in the buffer. The buffer is empty and possibly a number of spaces was skipped.

4.7.3.10 Inwildname 4.7.3.10

```
PROCEDURE inwildname (
     VAR z : zone; VAR name : alfa);
```

Not predefined.

- Works as the routine INNAME except that the legal characters are extended with the wild character "*".

4.7.4 Advanced Use 4.7.4

Besides the ordinary use of the OPERATOR to send output and receive input from the console, additional facilities are offered. These services are requested by sending a message to the OPERATOR asking for the wanted facility.

4.7.4.1 Input - Output Mode 4.7.4.1

Normally OPERATOR sends output to the console and receives input from the console.

On request OPERATOR will send output to
- the console
- an incarnation requesting the output

On request OPERATOR will send input message to (receive input from)
- the console
- an incarnation requesting the input message (and generating input)

There are four i/o modes:

```
loc_i_glob_o   (0): input from console
                    output to request incarnation
loc_i_loc_o    (1): input from console
                    output to console
glob_i_loc_o   (2): input from request incarnation
                    output to console
glob_i_glob_o  (3): input from request incarnation
                    output to request incarnation
```

Default mode is ´loc_i_loc_o´.

The mode ´loc_i_loc_o´ is kept unchanged until a

change mode request is received.

The modes are reset to 'loc_i_loc_o' after 30 seconds, if no incarnation request the messages. Otherwise, the mode is kept for a new period of 30 seconds.

Section 4.7.4.2 on events describes how an incarnation can use the event mechanism to manipulate the input - output streams.

The change of i/o mode is requested by sending a change message to the OPERATOR semaphore.

	to OPERATOR	from OPERATOR
u1	0	unchanged
u2	-	ok
u3	wanted mode	unchanged
u4	-	unchanged

An incarnation requests the messages in accordance with the actual i/o mode by sending a request message to the OPERATOR

	to OPERATOR	from OPERATOR
u1	12	unchanged
u2	-	result
u3	-	unchanged
u4	-	unchanged

The request message is placed in a queue of waiting request messages.

The waiting input messages and/or output messages in accordance with i/o mode are stacked and the first request message is pushed upon the stack. Finally the stack is returned to the answer semaphore of the request message.

Results	Meaning
0	ok, some buffers are returned
1	not_processed, i/o mode was loc_i_loc_o

4.7.4.2 Events

Incarnations may send some event messages to the OPERATOR.

Event messages are queued until 4.7.4.3
- they are regretted (see section)
- the operator presses 'ESC' and types the 'name' of the event message.

Event messages can be used by incarnations to control the input - output streams as described in the following example:
An incarnation normally sends output to and recieves input from request incarnations, but sometimes the operator wants to send a command from the console. In this case, the incarnation:
- sends an event message to OPERATOR with the name set to 'command'
- requests 'glob_i_glob_o' mode continuosly.

When the operator presses 'ESC' and types 'command', the event message is returned. Now the incarnation can change i/o mode according to the need, and later continue with the 'glob_i_glob_o' mode.

	to OPERATOR	from OPERATOR
u1	4	unchanged
u2	-	result
u3	-	unchanged
u4	-	unchanged

Data buffer:
The buffer is supposed to hold a message of type 'opbuffer' (appendix K). The field 'name' is used to identify the message.

Results	Meaning
0	ok, no input message available when needed.
1	not_processed, the event message is regretted.

4.7.4.3 Regret

An incarnation can regret messages previously sent to OPERATOR by sending a regret message.

	to OPERATOR	from OPERATOR
u1	8	unchanged
u2	-	ok
u3	-	unchanged
u4	-	unchanged

The queues holding event messages, input messages and output messages are searched. Messages originating from the same pool as the regret message are returned with result ´not_processed´ and finally the regret message is returned.

4.7.4.4 Match

An incarnation may simulate input from the console by sending a match message to OPERATOR.

	to OPERATOR	from OPERATOR
u1	6	unchanged
u2	-	result
u3	-	unchanged
u4	-	unchanged

The buffer is supposed to hold a message of type ´opbuffer´ (appendix K). The field ´name´ is used as the key in searching the queue holding input messages. If an input message is found with matching ´name´ field, the data from the ´match´ message is copied to the input message, and both messages are returned.

Results	Meaning
0	ok, data copied
1	Notprocessed, no matching input message.

4.8 Driver Input/Output Routines 4.8

In the description of the input/output routines a device is considered as containing a number of registers:

- CONTROL
- STATUSIN
- STATUSOUT
- DATAIN
- DATAOUT

where information is transferred to/from by means of commands issued by the RC3502 machine.

The procedures INBYTEBLOCK, INWORDBLOCK, IOWBWC, OUTBYTEBLOCK, OUTWORDBLOCK interprete the actual datamessages as being of the pseudo type:

```
buffertype = ARRAY (0..65535) OF byte
```

The buffer indices, according to the driver conventions, are therefore treated as unsigned indices in the range 0..65535.

4.8.1 Clearinterrupt 4.8.1

```
PROCEDURE clearinterrupt;
```

Not predefined.

- clears the current interruption level and waits for an interrupt. If the process incarnation has status ´timedout´, the call has no effect.

4.8.2 Control 4.8.2

```
PROCEDURE control (control_word: 16bittype;
                   VAR chmsg: reference);
```

Not predefined.

- The contents of the parameter ´control_word´ are transferred to the CONTROL register in the device selected by the channel message chmsg. The current interrupt level is not cleared so the next statement is executed without waiting for interrupt from the device.

The type of ´control_word´ may be any type of size 16 bits.

An exception occurs if:

- the reference variable ´chmsg´ is nil.
- the reference variable ´chmsg´ is not a channel message.

4.8.3 Controlclr 4.8.3

PROCEDURE controlclr (control_word: 16bittype);

Not predefined.

- The contents of the parameter ´control_word´ are transferred to the CONTROL register in the device selected by the current interrupt level. If the process incarnation does not have status ´timedout´, the current interruption level is cleared, and the next statement is executed when an interrupt arrives from the device. If the process incarnation has status ´timedout´, execution continues immediately.

The type of control_word may be any type of size 16 bits.

An exception occurs if:

- the reference variable chmsg is nil
- chmsg is not a channel message

4.8.4 Ctrwaitid 4.8.4

FUNCTION ctrwaitid (c:16bittype; delay:integer): activation;

Not predefined.

- waits for two kinds of event:

  1. An interrupt (a_interrupt)
  2. Expiration of a delay periode (a_delay)

The variable OWN.TIMER is initialized to ´delay´ and the control word ´c´ is sent to the external device connected to the current interrupt level, before the wait is performed.

Note that delay activation only takes place, if the routine ´definetimer´ has been called.

4.8.5 Ctrwaitis 4.8.5

```
FUNCTION ctrwaitis (c:16bittype; VAR r:reference;
                                 VAR s: semaphore): activation;
```

Not predefined.

- waits for two kinds of event:

  1. An interrupt (a_interrupt)
  2. The arrival of a message to the semaphore s (a_semaphore)

The control word ´c´ is sent to the external device connected to the current interrupt level, before the wait is performed.

An exception occours if:

- the reference variable ´r´ is not nil.

4.8.6 Ctrwaitisd 4.8.6

```
FUNCTION ctrwaitisd (c:16bittype; VAR r:reference;
                                  VAR s:semaphore;
                                  delay:integer): activation;
```

Not predefined.

- waits for three kinds of event:

  1. An interrupt (a_interrupt)
  2. The arrival of a message to the semaphore s (a_semaphore)
  3. Expiration of a delay period (a_delay)

The variable OWN.TIMER is initialized to ´delay´, the control word ´c´ is sent to the external device connected to the current interruption level, and the wait is performed.

Delay activation only takes place, if a call of ´definetimer´ has been issued.

An exception occurs if:

- the reference variable is not nil.

4.8.7 Eoi

4.8.7

```
FUNCTION eoi: boolean;
```

Predefined.

- true if the EOI (End Of Information) status bit is 1 in the program status word in the incarnation descriptor. The EOI status bit is updated whenever a READ or WRITE data command is issued by the incarnation.

After a READ command eoi=true indicates that the device has responded with no data. After a WRITE command eoi=true indicates that the device has accepted the data and wants no more data.

4.8.8 Getbufparam

4.8.8

```
PROCEDURE getbufparam
  (VAR bufparam: 64bittype;
   first, last: integer;
   VAR msg: reference);
```

Not predefined.

- The parameter ´bufparam´ is supposed to be of

```
TYPE
  bufparamtype = RECORD
                    top, count: integer;
                    datastart : 32bittype;
                 END;
```

- returns the start address of the byte with index first in the data buffer referenced by msg. As a side effect

```
count := madd (usub (last, first), 1)
top := madd (last, 1)
```

  is returned.

The procedure is intended for initializing a DMA

controller with the start address and count for an input/output operation.

The following exceptions may occur

- reference = nil
- not data message
- size too small
- ult (last, first)

4.8.9 Inbyteblock


```
PROCEDURE inbyteblock
  (VAR next: integer;
   first, last: integer;
   VAR msg: reference);
```

Predefined.

- inputs a block of bytes to the databuffer specified by ´msg´, ´first´ and ´last´ from the device specified by the current interruption level. When the procedure terminates, ´next´ will be the index of the byte following the last byte input.

The procedure will terminate in two situations:

- when next = madd (last, 1);
- when eoi=true

If nothing is input ´next´=´first´.

The following exceptions may occur:

- the reference variable ´msg´ is nil
- the message ´msg´ is no data message
- size of ´msg´ is too small
- ult (last, first)

4.8.10 Inword


```
PROCEDURE inword (VAR word: 16bittype;
                  VAR chmsg: reference);
```

Not predefined.

- the contents of the DATAIN register in the device selected by the channel message ´chmsg´ is transferred to the parameter ´word´. If eoi=true after the call, the contents of ´word´ are undefined.

The type of ´word´ may be any type of size 16 bits.

An exception occurs in the following situations:

- the reference variable ´chmsg´ is nil
- ´chmsg´ is not a channel message

4.8.11 Inwordclr


```
PROCEDURE inwordclr (VAR word : 16bittype);
```

Not predefined.

- The contents of the DATAIN register in the device, selected by the current interruption level, is transferred to the parameter ´word´. If the process incarnation has status ´timedout´, execution continues immedeately. If the process incarnation does not have status ´timedout´, the current interruption level is cleared, so the next statement is executed when an interrupt arrives from the device.

The type of ´word´ may be any type of size 16 bits.

4.8.12 Iowbwc


```
PROCEDURE iowbwc (bufparam: 64bittype);
```

Not predefined.

- the parameter ´bufparam´ is supposed to be of

```
TYPE
     bufparamtype  = RECORD
                        top, count: integer;
                        datastart : 32bittype;
                     END;
```

The procedure outputs a block of words from a buffer specified by ´datastart´. The number of words will be ´count´ DIV 2 (count = 0 is interpreted as 32 K words).

The procedure is intended for simultaneous output of data residing in one message. The actual parameter to the procedure may be obtained by means of the procedure ´getbufparam´.

The procedure can be accessed in the following way:

```
TYPE
     bufparamtype = RECORD
                       top, count: integer;
                       datastart : 32bittype;
                    END;

PROCEDURE iowbwc (bufparam: bufparamtype);
EXTERNAL;
```

4.8.13 Inwordblock


```
PROCEDURE inwordblock
  (VAR next: integer;
   first, last: integer;
   VAR msg: reference);
```

Predefined.

- inputs a block of words to the data buffer specified by ´msg´, ´first´, and ´last´ from the device specified by the current interruption level.

The procedure terminates in two situations:

- when next = madd (last, 1)
- when eoi=true

If nothing is input ´next´=´first´.

First and last must specify an even number of bytes (usub(last, first) must be odd).

The following exceptions may occur:

- the reference variable ´msg´ is nil
- the message ´msg´ is the data message
- ´madd (usub (last, first), 1)´ is odd
- size of ´msg´ is too small
- ult (last, first)

4.8.14 Outbyteblock


```
PROCEDURE outbyteblock (VAR next: integer;
                        first, last: integer;
                        VAR msg : reference);
```

Predefined.

- outputs a block of bytes from the databuffer specified by ´msg´, ´first´, and ´last´ to the device specified by the current interruption level. When the procedure terminates, ´next´ will be the index of the byte following the last byte output. The procedure will terminate when ´next = madd (last, 1)´.

If nothing is output next=first.

The following exceptions may occur:

- the reference variable ´msg´ is nil
- the message ´msg´ is no data message
- size of ´msg´ is too small
- ult (last, first)

4.8.15 Outword


```
PROCEDURE outword (word: 16bittype;
                   VAR chmsg: reference);
```

Not predefined.

- The contents of the parameter ´word´ are transferred to the DATAOUT register in the device selected by the channel message ´chmsg´. The current interruption level is not cleared so the next statement is executed without waiting for interrupt from the device.

The type of ´word´ may be any type of size 16 bits.

An exception occurs in the following situations:

- the reference variable ´chmsg´ is nil
- ´chmsg´ is not a channel message

4.8.16 Outwordblock


```
PROCEDURE outwordblock
  (VAR next: integer;
   first, last: integer;
   VAR msg: reference);
```

Predefined.

- outputs a block of words from the data buffer specified by ´msg´, ´first´, and ´last´ to the device specified by the current interruption level.

The procedure terminates when ´next = madd (last, 1)´.

If nothing is output ´next=first´.

First and last must specify an even number of bytes (´usub (last, first)´ must be odd).

The following exceptions may occur:

- the reference variable is nil
- the message ´msg´ is no data message
- ´madd (usub (last, first), 1)´ is odd
- size of ´msg´ is too small
- ult (last, first)

4.8.17 Outwordclr


```
PROCEDURE outwordclr (word: 16bittype);
```

Not predefined.

- the contents of the parameter ´word´ are transferred to the DATAOUT register in the device selected by the current interruption level. If the process incarnation does not have status ´timedout´, the current interruption level is cleared, so the next statement is executed, when an interrupt arrives from the device. If the process incarnation has status ´timedout´, execution continues immediately.

The type of ´word´ may be any type of size 16 bits.

4.8.18 Reservech


```
FUNCTION reservech (VAR chmsg: reference;
                    level, mask: integer): integer;
```

Predefined.

- allocates the channel message to the interruption level specified by the parameter ´level´. The parameter ´mask´ is intended for specification of the actions the user wants to perform on the interruption level. The parameter is not used in this revision.

Results	Meaning
0	Reservation OK. ´chmsg´ refers to the allocated channel message.
1	The interruption level is already reserved or is not installed.
2	The reference variable ´chmsg´ is not nil before call.

4.8.19 Reserveextmem


```
FUNCTION reserveextmem(VAR r : reference;
class,size,memno,disp : integer) : integer;
```

Not predefined.

- allocates a data message in the area 80.0000 to 9e.ffff, called ´external RAM´. The start address is specified by ´memno´ and ´disp´. The buffersize is specified by ´size´ in words according to the size conventions. ´Class´ specifies the type of check, the routine performs during reservation.

The memory has the format (by convention):

```
mem = RECORD
        intr,
        reset,
        class,
        sizelsb,
        sizemsb,
        version,
        intrno : byte;
        spare : ARRAY (7..31) OF byte;
      END
```

If ´size´= 0 , the size is taken from the memory locations ´sizelsb´, and ´sizemsb´, which is the size in bytes on Intel integer form.

The following checks are performed:

```
class=1    No check.
class=2    mem.reset = 6.
otherwise mem.class = class.
```

Result	Meaning
0	Reservation ok. ´r´ refers to the specified memory. ´Messagekind´ is greater than zero and ´size´ is the actual size in words.
1	Already reserved. The memory is occupied, or the requested memory overlaps an area, which is already reserved.
2	The reference variable ´r´ is not nil before call.
3	Illegal ´memno´. Only modules in the range 0..15 are legal.

4 No memory. No physical memory is installed in the requested area.
5 Illegal class. The requested class does not match the location ´mem.class´. Note class1 and class2.
6 No resources. Message headers could not be allocated to describe the requested memory area.
7 Illegal size. ´size=0´ is returned from the memory locations ´sizelsb´, and ´sizemsb´.

4.8.20 Sense


```
PROCEDURE sense (VAR status_in: 16bittype;
                 status_out: 16bittype;
                 VAR chmsg: reference);
```

Not predefined.

- The contents of the parameter ´status_out´ are transferred to the STATUSOUT register in the device selected by the channel message ´chmsg´. As a response from the device the contents of the STATUSIN register are transferred to the parameter ´status_in´. The current interruption level is not cleared so the next statement is executed without waiting for interrupt from the device.

The type of ´status_in´ and ´status_out´ may be any type of size 16 bits.

An exception occurs in the following situations:

- the reference variable ´chmsg´ is nil
- chmsg is not a channel message

4.8.21 Senseclr


```
PROCEDURE senseclr (VAR status_in: 16bittype;
                    status_out: 16bittype;
                    compare   ,
                    mask : integer);
```

Not predefined.

- The contents of the parameter ´status_out´ are transferred to the STATUSOUT register in the device selected by the current interruption level. The contents of the STATUSIN register are transferred to RC3502. If STATUSIN AND ´mask´ = ´compare´, the original contents of the STATUSIN register are transferred to the parameter ´status_in´, and the next statement is executed without waiting for interrupt from the device. Otherwise the current interruption level is cleared and the procedure is repeated, when the next interrupt arrives from the device, unless the process incarnation changes status to ´timedout´.

The type of ´status_in´ and ´status_out´ may be any type of size 16 bits.

4.8.22 Setinterrupt

PROCEDURE setinterrupt (VAR ch: reference);

Not predefined.

- activates the interruption level controlled by the channel message ´ch´.

The following exceptions may occur:

- the reference variable chmsg is nil
- chmsg is not a channel message

4.8.23 Timedout

FUNCTION timedout : boolean;

Not predefined.

- true: The process incarnation has status ´timedout´. The status is cleared and OWN.TIMER initialized to zero.

  false: The process incarnation is not ´timed out´.

4.8.24 Waiti

```
PROCEDURE waiti;
```

Predefined.

- waits for an interrupt from an external device. If executed on interruption level 0 (class II or III), the caller will be permanently descheduled (~suicide).

4.8.25 Waitid

```
FUNCTION waitid (delay : integer) : activation;
```

predefined.

- waits for two kinds of event:

  1. An interrupt (a_interrupt)
  2. Expiration of a delay period (a_delay)

The variable OWN.TIMER is initialized to ´delay´ before the wait is performed.

Note that delay activation only takes place if the routine ´definetimer´ has been called.

4.8.26 Waitis

```
FUNCTION waitis (VAR r: reference;
                 VAR s: semaphore) : activation;
```

Predefined.

- waits for two kinds of event:

  1. An interrupt (a_interrupt)
  2. The arrival of a message to the semaphore s (a_semaphore)

An exception occurs if the reference ´r´ is not nil.

4.8.27 Waitisd


```
FUNCTION waitisd (VAR r : reference;
                  VAR s : semaphore;
                  delay : integer   ): activation;
```

Predefined.

- waits for three kinds of events:

    1. An interrupt (a_interrupt).
    2. The arrival of a message to the semaphore s (a_semaphore).
    3. Expiration of a delay period (a_delay).

An exception occurs if the reference ´r´ is not nil.

5. THE RC3502 MACHINE 5.

5.1 Run Time Environment 5.1

After autoload and system initialization, the incarnation structure is:

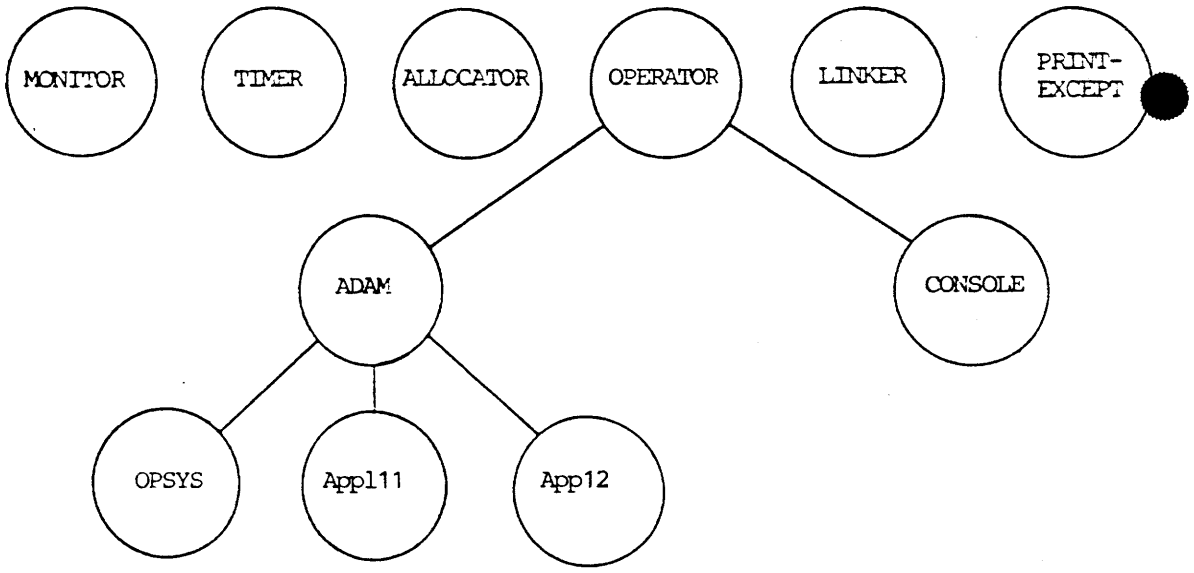


Fig. 6. Incarnation structure after initialization

MONITOR performs medium term scheduling (START, STOP).

TIMER performs delay timing, time out of incarnations, and controls a system clock for time stamping.

ALLOCATOR — administers allocation and deallocation of RAM memory and interruption levels.

ADAM — is the root of the dynamic tree of incarnations. ADAM automatically creates and starts a number of incarnations:

- FS
- IMCSTART
- OPSYS
- S

LINKER — administers a catalog of processes and routines, the LINKER catalog.

OPERATOR CONSOLE — is the interface between a human operator and the running incarnations. CONSOLE performs input/output to the debug console.

OPERATOR handles messages signalled to the operator semaphore.

OPSYS — is a command interpreter, functioning as an interface between a human operator and the Run Time processes.

EXCEPTION — prints a list of the dynamic chain of routine calls, when a process incarnation goes into a runtime error (exception).

S — If a process S exists in the LINKER catalog, an incarnation of S will be created and started. This will be the case when a process S is blasted in PROM or autoloaded.

S may replace OPSYS by its own NEW_OPSYS.

5.2 MONITOR Process 5.2

The main purpose of the Monitor is to control the set of active incarnations.

The active incarnations are divided into three priority classes:

Class I: High priority
Class II: Medium priority
Class III: Low priority

Scheduling of class I incarnations is managed by the hardware interrupt priority mechanism. Incarnations in class I are running on an interruption level greater than zero.

Class II and III incarnations are running on interruption level 0.

Scheduling of class II and III incarnations is also performed by the hardware.

The class II incarnations are scheduled according to internal priority in the class and round robin for a given priority.

The class III incarnations are scheduled after a time sliced round robin algorithm with built-in priority.

The monitor is activated, when incarnations call the routines BREAK, REMOVE, START, or STOP.

5.3 Driver Processes 5.3

By definition, a driver process is a process which contains the

CHANNEL <reference variable> DO <statements>;

construction or the standard input/output routines from chapter 4.8.

Access to most input/output routines and the CHANNEL statement is a reference variable which refers to a message of kind 'channel message'.

A channel message is obtained by calling the routine RESERVECH specifying the device or interruption

level, the incarnation wants to control.

The system guarantees that at most one channel message is allocated per device or interruption level.

When an incarnation executes a channel statement or calls an input/output routine, which demands a channel message, it is checked that the reference variable is not nil and that the reference variable refers to a message header of kind 'channel message'.

In the input/output routines, which do not wait for an interrupt from the device, this is also checked.

Before execution of the first statement in the CHANNEL statement, the incarnation has entered the class I priority class. An eventually 'timed out' status is cleared.

The statements in the CHANNEL statement are executed on the hardware interruption level specified by the channel message.

After execution of the last statement in the CHANNEL statement, the incarnation reenters the priority class (II or III) which the incarnation left, when entering the CHANNEL statement. An eventually 'timed out' status is cleared.

The user should be very much aware of the fact that the execution of incarnations in the priority classes II and III, besides all incarnations running at a hardware interruption level less than the interruption level of the user's incarnations, is disabled while executing statements in a CHANNEL statement.

This is true for all statements except those input/output routines, which wait for an interrupt, and thereby allow incarnations on lower interruption levels to execute instructions.

Therefore, the following recommendations should be followed:

- minimize the number of statements which are executed in a CHANNEL statement.

- the statements in a CHANNEL statement should mainly be input/output routine calls.

The system allows an incarnation to possess several channel messages, but it is emphasized that it is the channel message used in the CHANNEL statement that defines the interruption level, where the incarnation is sensitive for interrupts.

Therefore it is normally the same channel message which is used both in the CHANNEL statement and in the input/output routines.

The channel messages in a CHANNEL statement and an input/output routine may differ. This may be used to sense a device with another device address than the interruption level defined by the CHANNEL statement, or even outside a CHANNEL statement.

5.3.1 Time Out

5.3.1

Time Out is the activation of an incarnation with status ´timed out´.

Time Out of incarnations is performed by the TIMER Process which decrements once per second the standard variable OWN.TIMER in all incarnations, which have called the routine DEFINETIMER.

Time Out takes place, when the variable is decremented from 1 to 0.

The ´timed out´ status is cleared, by calling the routine TIMEDOUT or by exit of a CHANNEL statement.

5.4 TIMER Process 5.4

The Timer Process:

- returns messages after a specified interval (Delay Timing),
- maintains a system clock,
- controls time out of incarnations.

Delay timing is requested by calling the procedure SENDTIMER (see chapter 5).

Time out is requested by assigning the time out period in seconds to the variable OWN.TIMER, or by using one of the multiple wait functions specifying a delay. Count down of OWN.TIMER will only happen after a call of the routine DEFINETIMER.

5.5 ALLOCATOR Process 5.5

After startup of the system, ALLOCATOR controls the available memory.

Memory is allocated to contain the incarnation stack, when a process incarnation is created.

Variables of type POOL may be allocated memory, when a newborn process incarnation is started, or when the routine INITPOOL is called.

The memory possessed by a process incarnation is deallocated when the controlling father incarnation or an ancestor calls the REMOVE procedure.

Memory is allocated in the following order: c0, c2, .., fe, a0, a2, ..., be. This is done for hardware test purposes.

ALLOCATOR also controls access to all interruption levels. This is done by messages of kind CHANNELMESSAGE.

An interruption level is reserved by calling the routine RESERVECH specifying:

LEVEL - interruption level,
MASK - facility mask.

A channel message is released by the statement:

```
RELEASE (channel_message);
```

5.6 LINKER Process

LINKER administrates the LINKER catalog describing all programs (processes and routines) in the system. The programs are linked to physical memory (statically relocated) and all calls of external routines are resolved.

The LINKER processes link/unlink requests from running incarnations (see the routines LINK, UNLINK).

A LOOKUP function and a CRC16 check function may also be requested (see the SENDLINKER routine).

5.7 ADAM Process

Immediately after restart of the system the incarnation tree is:

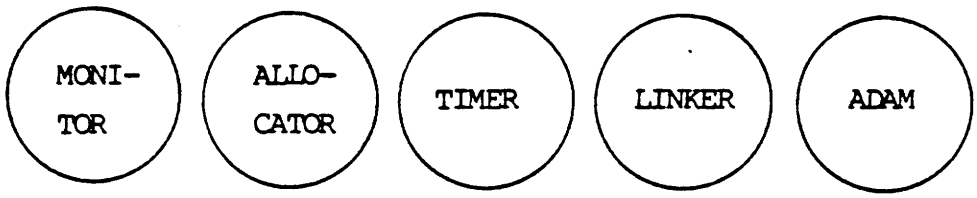


Fig. 7. Snapshot of incarnation structure
(This figure is not complete)

ADAM is the root of the dynamic tree of incarnations. ADAM creates and starts a number of incarnations during initialization:

1. OPSYS, which interprets commands from the console and converts the commands to ADAM, LINKER, or TIMER messages.

2. S, which is the root of an application.

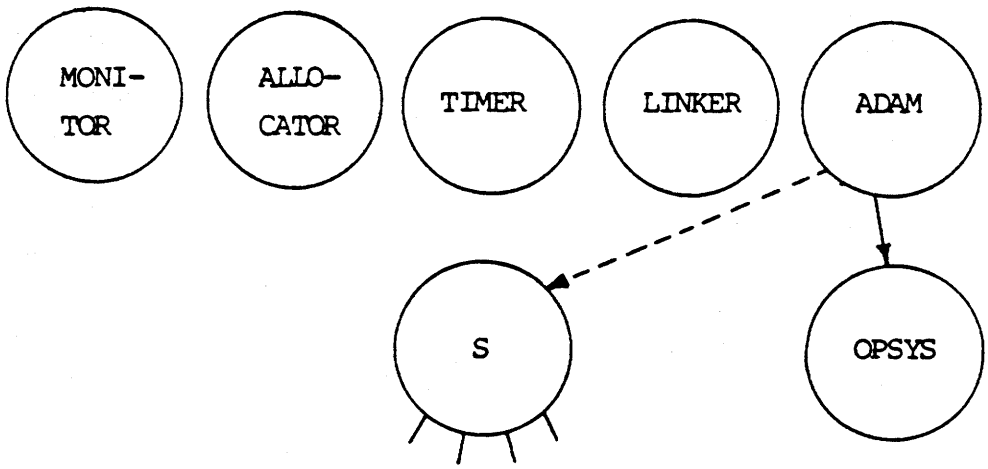


Fig. 8. Snapshot of incarnation structure if S is included.

ADAM contains declarations of two classes of programs. The first class has no parameters. The second has one parameter, the ´systemvector´:

```
s_type0 = EXTERNAL PROCESS;

s_type1 = EXTERNAL PROCESS (VAR sv: system_vector);
```

S is application dependent, and the formal parameters of the actual S must conform to one of these declarations.

ADAM controls a pool of process declarations, which conform to these declarations. These declarations may be used to request creation of new application sub trees with ADAM as the controlling father (see later).

The parameter ´sv´ may be used to pass references to system semaphores like:

```
adam semaphore      [sv (adamsem)↑]
operator semaphore  [sv (operatorsem)↑]
```

These semaphore might as well be obtained by the routine SEARCH_SEMAPHORE, because ADAM names the same semaphores during initialization:

systemvector	searchsemaphore
adamsem	'adamsem'
operatorsem	'operatorsem'
loadersem	'loadersem'
globalsem	'globalsem'
loaddriversem	'loaddriversem'
paxsem	'paxsem'
restartsem	'restartsem'

Furthermore these semaphores and new semaphores may be manipulated by a number of requests to ADAM.

ADAM may be requested to STOP and REMOVE any of the children and unlink the process by sending a message to the ADAM semaphore specifying the function to perform.

E.g.:

1. A human operator may stop, remove, and unlink the whole application tree (S) and start up a completely different application tree.

2. The application tree may stop, remove, and unlink OPSYS. A NEW_OPSYS may be created and started as a child of S.

Messages signalled to the ADAM semaphore are interpreted as a data message of type:

```
adamtype=RECORD
           name1: alfa;
           name2: alfa;
           aux  : integer;
         END;
```

Be aware of the difference between sending a link, create, start, stop, break, remove, or unlink message to ADAM and the call of the predefined routines LINK, CREATE, START, STOP, BREAK, REMOVE, UNLINK. The former operates on declarations in ADAM, the latter on declarations in the calling process incarnation.

Message headers to/from ADAM have the format:

	ADAM message	Answer
u1	function	unchanged
u2	not used	result
u3	not used	unchanged
u4	not used	unchanged

All messages which cannot hold a variable of type adamtype are returned with result = 1. An unknown function is returned with result = 15.

Function = 1 (LINK)

If ADAM has a free process declaration the process 'name1' is linked to a free process declaration. All process declarations in ADAM are:

```
PROCESS processname (VAR sv: system_vector);
EXTERNAL
```

Results	Meaning
0	ok
2	A process is already linked to a process declaration in ADAM with the external name 'name1'.
3	No free process declarations.
4	Process with name 'name1' does not exist in the LINKER catalog.
5	Process with name 'name1' exists in the LINKER catalog, but the number of parameters or the type of parameters do not match.

Function = 2 (CREATE)

An incarnation with name 'name2' of the process 'name1' is created with size 'aux'. More than one incarnation per process can be created.

Results	Meaning
0	ok
6	An incarnation of process 'name1' with name 'name2' is already created.
7	No process with external name 'name1' is linked to a process declaration in ADAM.
8	No storage or demanded size (aux) is too small.

Function = 3 (START)

The incarnation with name ´name2´ is started with priority ´aux´.

Results	Meaning
0	ok
9	Unknown incarnation.

Function = 4 (STOP)

The incarnation with name ´name2´ is stopped.

Results	Meaning
0	ok
10	Unknown incarnation.

Function = 5 (REMOVE)

The incarnation with name ´name2´ is removed.

Results	Meaning
0	ok
11	Unknown incarnation name.

Function = 6 (UNLINK)

The link to the process with external name ´name1´ is deleted.

Results	Meaning
0	ok
12	No process with external name ´name1´ is linked to a process declaration in ADAM.
13	ADAM still controls an incarnation of the process ´name1´.

Function = 7 (BREAK)

A break operation is performed upon the incarnation with name ´name2´ with exception code ´aux´.

Results	Meaning
0	ok
14	Unknown incarnation.

Function = 8 (NAME SEMAPHORE)

A semaphore is created and cataloged under the name ´name1´.

Results	Meaning
0	ok
16	Overlap.
17	No resources.´

Function = 9 (DELETE SEMAPHORE)

The semaphore catalogued under the name ´name1´ is deleted.

Results	Meaning
0	OK
18	Unknown semaphore

Function = 10 (RENAME SEMAPHORE)

The semaphore catalogued under the name ´name1´ is catalogued under the name ´name2´. The name ´name1´ is deleted from the semaphore catalog of ADAM.

Results	Meaning
0	OK
19	Overlap, a semaphore is already catalogued under the name ´name2´.
20	No catalogued semaphore with the name ´name1´ is found.

5.8 OPERATOR Process

5.8

Read and write messages signalled to the OPERATOR semaphore [sv (operatorsem)↑] should at least contain a data message of type:

```
Buffertype = RECORD
                first : integer;
                last  : integer;
                next  : integer;
                name  : alfa; (* 12 chars *)
                databuf : array (18..max) of
                char
             END;
```

The data part of the message follows the driver conventions. It is checked that the following assertions hold:

1. 18 <= first

2. first - 1 <= last

3. databuf (last) is accessible in the data message

OPERATOR identifies the input and output messages by the parameter 'name'.

Chapter 4.7 describes a set of routines for OPERATOR communication.

Messages to OPERATOR

control: u1 = 0

read: u1 = 1

The message is queued until it is "activated" by the human operator.

write: u1 = 2

The message is printed as soon as possible.

Answers from OPERATOR

All messages are returned, when the appropriate action has been performed.

```
u1, u3, u4 are unchanged
u2 = result: 0 = ok
             1 = not processed
             4 = unintelligible
```

"next" is undefined, unless result = ok.

A. REFERENCES

1. RCSL No. 52-AA964:
PASCAL80, Report

2. RCSL No. 42-i1539:
PASCAL80, User´s Guide

3. RCSL No. 31-D652:
PASCAL80, Driver Conventions

4. RCSL No. 99 0 00771
RC3502/2, Operating Guide

5. RCSL No. 52-AA1192:
RC3502/2, Reference Manual

6. RCSL No. 52-AA1182
MEM205, General Information

7. RCSL No. 52-AA1137
RC3502 LOADER, Reference Manual

8. RCSL No. 52-AA1174
Reference Manual for the Programming Language Real-Time Pascal

B. USE OF THE REAL-TIME PASCAL COMPILER B.

B.1 Call of the Compiler B.1

$$\left\{\text{<bin file>=}\right\}_{0}^{1}\ \text{rtp35022}\left\{\left\{\text{<s><option>}\right\}\left\{\text{<s><context>}\right\}\right\}_{0}^{\infty}\left\{\text{<s><option>}\right\}_{0}^{\infty}\left\{\text{<s><source>}\right\}_{0}^{1}$$

- <source> is a text file defining a process, a prefix or a library of routines/processes (se further appendix B.2). If no source is specified, the compiler reads the source from current input.

- <context> is a text file containing declarations of types, constants, and external routines. Contexts can be used for definition of libraries. The syntax is described in appendix B.2.

- <bin file> is a file descriptor describing the backing storage area where the object code is stored.

  If "<bin file>=" is omitted,
  "pass6code=" is assumed.

- <option>::= codesize.<size>
  |list.<yes or no>
  |stop.<pass nr>
  |spacing.<interval>
  |measure.<yes or no>
  |exception.<appetite>
  |short.<yes or no>
  |stack.<appetite>
  |codelist.<yes or no>
  |preserve.<yes or no>
  |survey.<yes or no>
  |set.switchname.switchvalue
  |version.<version>

- <size> is an unsigned integer in the range 0-15000. The size denotes the maximum number of bytes to be generated on one "program code page".

- <yes or no>::= yes | no

- list.yes means turn on listing of input (on current output).

  Listing only takes place after compilation of the standard environment.

  The list option is superfluous since the utility programs indent and cross may produce more readable listings (see the description of indent and cross in appendix E).

- measure.<yes or no>: consult appendix C: Performance Measurement.

- short.yes controls the compiler to use short encoded instructions in the generation of code whenever possible.

- stack.<appetite>
  This option overrides the default stack size, which the compiler estimates and outputs in the generated code. The default stack size is used when the create routine is called with size = 0.

  The "size" of a process incarnation may be calculated in the following way:

```
size:= dynamic appetite
     + exception code appetite
     + 10 (for dump of the register stack)
```

  The dynamic appetite is the static appetite as mentioned for the process in the compilation survey plus the dynamic appetite for the most hungry of the called routines. The dynamic appetite of a routine is the static appetite plus the dynamic appetite for the most hungry of the called routines etc. In this connection please note the standard routines, the appetite of which is specified in the package description.

  For recursive procedures the appetite has to be

multiplied by the maximum depth of the recursion.

The exception appetite amounts to a default value, which is specified in the package description. The appetite may be overruled by the ´exception´ option in the call of the compiler.

NOTE: If declarations of pool variables occur, a standard initialization procedure, _initpool_rc, is called.

The stack space of the children is not taken from the stack of the parent process. Create checks that there is memory for the incarnation descriptor, the actual process parameters, and for dump of the register stack, before the child is created.

- exception.<appetite>
  This option is used for specifying the amount of stack space an incarnation of the program will reserve for exception handling. The default value is the room necessary for a call of the standard exception procedure. For a user defined exception procedure the option value may be the appetite seen in the information list from the compilation of the routine, (see further under the ´stack´ option).

- codelist.yes means, list the generated code in symbolic form with information about instruction execution times, logical addresses, and some comments.

- preserve.yes means, do not remove intermediate work files after use

- survey.yes implies listing of some statistical information.
  Example:

```
finis       bb0 at 11 57
87.12.18.      11.56.      REAL-TIME PASCAL for RC3502/2     version 1987.12.09

    1
    2 process p;
    3 begin
    4 end;
    5 .

Information survey from REAL-TIME PASCAL, pass3, version 87.12.16
Max number of active names:                          631
Deepest nesting:                                       5
Deepest stack of names:                                5
Max number of active types:                           93
Max number of active strings:                         10
Max number of active operands:                        15
Page faults:                                          83

Information survey from REAL-TIME PASCAL, pass4, version 87.03.20
Deepest name stack:                                  129
Deepest operand stack:                                 2
Constant area size:                                    1
Deepest structure nesting:                             0
Page faults:                                          76


Information survey from REAL-TIME PASCAL, pass5, version 87.10.06

   name        headline beginline endline  appetite(words) default create-size

   p                3         4       4  :      45                        45
   exception      external, called  1 time(s)

Information survey from REAL-TIME PASCAL, pass6, version 87.03.20
Max number of active labels:                          67
Max number of labels at one instruction:               3
Max number of unsolved interdependencies:              1
Deepest block nesting:                                 4
Max number of unsolved jumps:                          0
Max number of unsolved references:                    10
Greatest case table or structured constant:            8
Elapsed time:                                   0.00.34
CPU time in seconds:                                  16


 code: 0 . 130  = 130 bytes


end of REAL-TIME PASCAL compilation for RC3502/2

end
blocksread = 62


end  40 sec  job bb0  log ah  date 1987.12.18  11.57.04
```

- set.switchname.switchvalue implies setting/defining a switch. The so defined switches may be used in the exptessions of IF and ELSEIF directives (see below).

- version.<version> implies setting of the version directive (see below).

```
<version>::=<integer>
           |<byte>.<byte>
           |<release>.<edition>.<variant>
```

- <interval> is the distance between two line number records. The line number records are used by the standard exception procedure to relate the address of a run time error to a line interval of the program.

  <pass nr>::= 1 | 3 | 4 | 5 | 6

- stop.<pass nr> terminates the translation after the pass specified.

  A short description of the passes:

```
pass 1 performs syntax analysis
pass 3 performs machine independent semantic
               analysis
pass 4 performs storage allocation and machine
               dependent semantic analysis
pass 5 performs symbolic code generation
pass 6 performs transformation of symbolic
               code to binary format
```

- default values:

  the call:

```
   rtp35022 inputfile
```

  is equivalent to:

```
pass6code=rtp35022 list.no codesize.512 spacing.52 stop.6,
measure.no exception.11 short.yes survey.no codelist.no,
stack.***  perserve.no inputfile
```

  ***: the default stack appetite is set according to a compiler generated value, computed as the sum of static appetites of the routines of and main body of the program.

Resource requirements

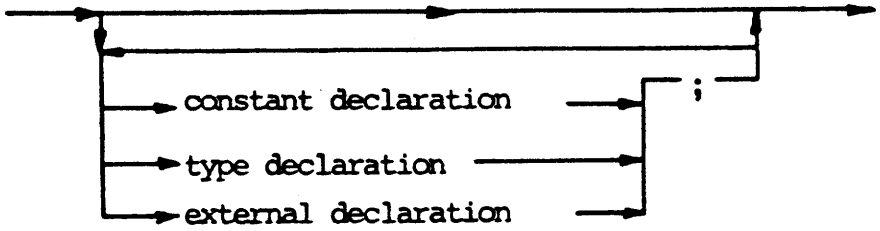
At least 45,000 hW memory (size 45000)
area 8
temp disc at least 6 slices and 6 entries.

B.2 Use of the Contexts

Specification of more input files to the Real-Time Pascal compiler is used for inclusion of (common) contexts (environments), i.e. constant and type definitions and declarations of external routines. The syntax is:

context:

context declarations:



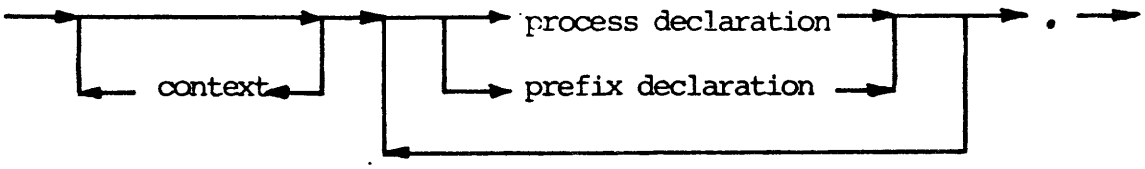
external routine declaration:

Compilation of libraries

It is allowed to compile several routines/processes in one compilation, thus saving environment handling

for each (small) routine. The code produced constitutes one RC8000 disc area which has the same format as a library built by "plibinsert". The source texts for the modules in question must be collected in one file. The syntax for a "program" is:

program:



B.3 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 and with parameters, if present, enclosed in parantheses. 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 directives to the RTP3502/2 compiler, default directives are followed by an '*'.

name	parameters	description
CODE	none	causes code generated for the following lines of source text to be listed,

NOCODE*	none	suppresses listing of generated code,
CREATESIZE	number	determines stack size of incarnations of the following program if the value of the sizeexpression of the create call is 0.
LIST	none	causes the following lines of source text to be listed,
NOLIST*	none	suppresses listing of source text,
VERSION	<version description>	16-bits integer as version identification, either one integer value, or 2 byte values, or a tripple denoting release, edition, and variant.
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 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 or from the call of the compiler), merely changes the

value.

The directives IF, ELSE, ELSEIF, and ENDIF provide the capability for conditional compilation. The expressions occurring in IF and ELSEIF directives 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
sl_1
$ELSEIF expr_2
sl_2
$ELSEIF expr_3
sl_3
...
$ELSEIF expr_n
sl_n
$ELSE
sl_n+1
$ENDIF
```

The only mandatory parts are the IF and ENDIF directive lines and the source line(s) sl_1. The effect is as follows: If the values of all the expressions are false (integer value zero) then the source lines sl_1, sl_2, ..., sl_n are excluded from compi_lation. Otherwise let k be the smallest number such that the value of expr_k is true (non zero integer value). Then sl_1, ..., sl_k-1, sl_k+1, ..., sl_n, and sl_n+1 (if present) are excluded from compilation.

Conditional compilation may be used at several nested levels. In the above terminology any of the sl_i may thus in_clude 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.

Undefined switches used in an expression are implicitly defined with value false/0. And a warning is given.

Directives occurring in comments are ignored, in particular those occurring in source lines excluded from compilation.

C. PERFORMANCE MEASUREMENT C.

It is possible to make some performance measurements on running Real-Time Pascal programs. The result of measuring includes the number of routine activations and the amount of time spent in the routines. The time indications are real-time, i.e. system overhead, time slicing and time for wait are included.

Measurement is initiated by a call of the procedure ´start_measure´ with a semaphore as parameter. All succeeding routine calls and returns trigger update of a table. The table is allocated and initialized by ´start_measure´.

The measurement is terminated by a call of ´stop_measure´ with the same semaphore parameter as ´start_measure´. ´Stop_measure´ signals the table to a system semaphore and a new call of ´start_measure´ is possible. Measurements may be started and stopped as long as it is possible to allocate an area of 2574 bytes to be used as measurement area.

The measurement results may be printed on the console by means of the process ´print_statis(tical_information)´. The order of results is FIFO. The output process deallocates the measurement areas after printing of the results.

There is an upper limit of 255 different routines per program, i.e. per process including internal processes. If more than 255 different internal and external routines are called the (statically) last routines will use one common table entry, and the output process will put ´**********´ instead of name.

The output format may be seen in the following example. The number of calls is modulo 64K and the time amounts are given as:

hours.minutes.seconds.hundredths of a second

Since the names are found in the code of the routines, the output process must be run before the routines and processes are unloaded.

Use of the measurement tools:

The processes to be measured must be compiled together with ´measureenv´ and with ´measure.yes´ in

the call of the compiler. ´Measureenv´ includes external declarations of ´start_measure´ and ´stop_measure´.

The necessary routines for measuring are supplied in ´measurelib´, which is required in order to load/link processes compiled with measure.yes.

The output process is supplied as ´bprintstat´.

Example 9. Traptest

```
PROCESS traptest( VAR sv : system_vector );

VAR
  step1, step2 : integer;
  measure_sem : semaphore;

  PROCEDURE empty_1;
  BEGIN END;

  PROCEDURE empty_2;
  BEGIN END;

BEGIN
  start_measure( measure_sem );
  FOR step1 := 1 TO 32 DO
    FOR step2 := 1 TO 1000 DO
      BEGIN
        empty_1;
        empty_2;
      END;
  stop_measure( measure_sem );
  start_measure(measure_sem);
  FOR step1 := 1 TO 33 DO
    FOR step2 := 1 TO 1000 DO
      BEGIN
        empty_2;
        empty_1;
      END;
  stop_measure(measure_sem);
END.
```

```
RUN  TRAPTEST
RUN  PRINT_STATIS
>print_statis
Performance measurement for traptest

name           called      time
-------------------------------------

traptest           01      0.00.31.45
start_measur       01      0.00.00.00
empty_1         32000      0.00.14.35
empty_2         32000      0.00.14.65
end
Performance measurement for traptest

name           called      time
-------------------------------------

traptest           01      0.00.31.65
start_measur       01      0.00.00.05
empty_1         33000      0.00.15.40
empty_2         33000      0.00.15.30
end
```

D. REAL-TIME PASCAL MESSAGES

D.

If errors are detected during compilation, the compiler initialises the ok-bit to ok.no.

When the contexts and the source is spread over more files, the line number is set to 0, when a new file is to be compiled, or when a periode is encounted.

D.1 Messages from Pass 1

D.1

0 Illegal character or fatal error
1 Identifier expected
2 ´.´ expected (end of process, prefix or context)
3 ´;´ expected
4 Identifier expected
5 ´=´ expected
6 ´,´ or ´:´ expected
7 Error in declaration
8 Set element or ´.)´ expected
9 (structured) constant or ´:)´ expected
10 Error after ´&´, string or character constant name expected
11 Constant, variable or ´(<expression>)´ expected
12 ´(´ expected
13 ´)´ expected
14 Expression expected
15 Actual parameter expected
16 Expression expected
17 ´)´ or ´,´ expected
18 ´:)´ or ´,´ expected
19 ´.)´ or ´,´ expected
20 ´of´ expected
21 Identifier or ´?´ expected
22 ´PROCESS´ expected
23 ´array´ or ´record´ expected after ´packed´
24 ´..´ expected
25 ´end´ or ´;´ expected
26 ´;´ or ´)´ expected
27 ´:´ expected
28 ´.<field name>´ or ´;´ expected
29 Unsigned integer expected
30 ´begin´ expected
31 Error in for-variable specification
32 Error in for-variable specification
33 ´to´ or ´downto´ expected
34 Error in channel-variable specification
35 ´do´ expected
36 ´do´ or ´,´ expected
37 ´then´ expected

```
 38 Label expected ( name or integer )
 39 'else' expected
 40 ';', 'end', or 'otherwise' expected
 41 'end' or ';' expected
 42 'until' expected
 43 'endloop' or ';' expected
 44 Label declaration expected ( integer or name )
 45 Error in label declaration list  ( ',' or ';' expected )
 46 '=' expected
 47 '=' expected
 48 'process' or 'prefix' expected
 49 Only procedure or function declaration allowed in
    a prefix
 50 End of process expected
100 Error in real constant: digit expected
101 String did not terminate within line
102 Line too long, more than 150 characters.
103 Comment not terminated
104 export kind expected ( value, size, address, disp,
    or offset )
105 lock type expected
106 ':' met, changed to '='
107 String too long, remaining part of string skipped
108 Name after concatenation symbol ('&') is not name
    of char constant
109 Exitloop or continueloop statement not inside
    repetitive statement
110 Routines in environment must be EXTERNAL declarations
```

The following messages indicating fatal errors may appear from pass1 of the Real-Time Pascal compiler. The message will be preceded by the line just being parsed with an indication of error number 0 discovered.

E.g.:

```
917  21   if prod = 1305   then
                          ↑ 0
***  const 'chbufmax' too small.
```

In case no other errors are discovered a fatal error may indicate that one or more of the compiler tables are insufficient in size. But most often this kind of fatal errors appear in consequence of syntactical errors, and after correction of the marked errors the fatal error may disappear.

The messages are:

*** parse stack overflow. const 'stackmax' too small
Parsing of a too complicated syntactical construction.

*** end of file encountered
Input exhausted before the parsing of the process/ prefix has been successful.

*** recovery abandoned
The error recovery was unsuccessful.

*** reduction buffer overflow. const 'redumax' too small
Parsing of a too complicated syntactical construction.

*** const 'stringmax' too small
Literal text string too long.

*** const 'chbufmax' too small
Parsing of a too complicated syntactical construction.

*** const 'typebuffersize' too small
Too complicated type definition.

D.2 Messages from Pass 3

Error messages from pass 3 have the format:

*** pass 3 line <lineno>, <operand no> <error no> (<name>) (, in environment <env nr>)

where:

<lineno> is the line number where the error is detected,

<operand no> gives a hint of where in the line the error was.

Operands are: identifiers and numbers. (Note: The empty set does not count).

First operand in a line has number 1. (Note: If <operand no> is 0, the error occurred before the first operand in the line, maybe even in the last part of the previous line).

<error no> indicates which error was discovered. A list of error descriptions follows the error indications.

<name> is, if present, the name (or the type name) of the identifier which caused the error. For example the name of an undeclared identifier.

The error descriptions are:

1 identifier not declared
2 identifier used before declaration
3 identifier already declared at this level
4 label-identifier not declared at all
5 other identifier used as label-name
6 label defined several times at this level
7 label-identifier declared at surrounding level
8 use of a multiple defined label
9 a label-ident has been used in inner routine
10 goto leading into control-structure
11 goto out of lock- or channel statements
12 label is not defined
13 identifier is not a type-identifier
14 type of structured constant not specified
15 error in record or array etc.
16 constant is used in its own definition-expression
17 pool ... of <illegal type>, shielded components not allowed
18 illegal size´ing of pool type
19 illegal limit-types in subrange def
20 semaphore, external process and pool may only be declared at process level
21 processes inside functions/procedures forbidden
22 formal type may not be used in this context
23 functiontype may not contain shielded components
24 only variables of surrounding scope allowed
25 ´new´ paramlist may be empty or exact the same
26 forward-declared type/routine not followed by the real body
27 function-value has not been defined at all
28 locktype contains pointer-types
29 lock-/with-type contains shielded component(s)
30 operands not of same typename
31 for-variable/startvalue/endvalue not of compatible types
32 case-expression/caselabels not of compatible types
33 if-expression must be boolean type
34 until-expression must be boolean type
35 while-expression must be boolean type
36 short form of multiple locks not allowed
37 buffer name missing in lock statement
38 buffer type specification is missing
39 lock-variable must be reference type
40 channel-variable must be reference type
41 type of operand must be enumeration-type
42 varsizecall: identifier is not name of a variable
43 operand cannot be used as variable

44 <variable> in front of <.> is not a record
45 <name> after <.> is not a fieldname of <variable>
46 <variable> in front of <uparrow> is not a pointer
47 elements in set-value may not be of mixed types
48 illegal mixture of types in relation
49 illegal mixture of types in term or factor
50 illegal type for monadic operator
51 real occuring in expression
52 real-division of integers not impl
53 illegal operand kind in expression
54 too few actual parameters to routinecall (or strucrecord
55 error in routinecall
56 error in structured-record constant
57 typename in front of structured constant must be record or array
58 the ´***´ operator must only occur in structured-array constant
59 name in front of arglist is not of array-type
60 index-expression doesn´t match array-declaration
61 incompatible types in assignment
62 incompatible types in exchange
63 the statement is not a procedure-call
64 for-variable, with-variable or actual var-param is packed
65 operand may not be assigned to, type is shielded or frozen
66 operand may not be exchanged, type is semaphore, external process, pool or frozen
67 type must be: reference or shadow
68 formal and actual type must match exactly
69 actual and formal types are not compatible
70 formal is not frozen, therefore actual may not be frozen
71 the ´?´ may only occur in structured constants
72 incomp. types in structured array-constant
73 incomp. types in structured record-constant
74 incomp. types in var-initialization
75 repetition must be integer
76 value-export demands constant
77 offset-export demands variable
78 size-,disp-,addr-export demands constant or variable
79 disp-export demands ´fielding´
80 xor, integer-and, and integer-or not implemented on z80
81 a process may not be FORWARD declared
82 FORWARD specified type must be pointer

Fatal errors from pass 3

The following messages indicating fatal errors may appear from pass 3 of the Real-Time Pascal compiler. The messages will be preceded by:

*** error found in pass3, at sourceline: <line no>

The messages are:

max_nested_calls exceeded <limit>
More than <limit> nested routine calls and/or index specifications.

D.3 Messages from Pass 4

All error messages from pass 4 have the format:

*** pass 4 line <no>, <text>

where <no> is the line number where the error is detected.

<text> is one among the following:

subrange def.	Error in the definition of subrange type - lower bound > upper bound.
set def.	Error in the definition of set type - lower bound of the basis subrange type is negative.
pool def.	Error in the definition of pool type - number of elements is negative.
record size	Record type > 65536 bytes.
array size	Array type > 65536 bytes.
no init. or process in environment	Initialization of variables and process declarations in environment not allowed.
constant value	Value of constant outside interval bounds.
case label range	In a case label interval first > last.

constant	Syntax error in number.
set constant	Error in set constant, - negative constant, or an interval with first value < last value, or constant > 1023.
times	Wrong number of values in constant of array type.
not constant	Variable, or set constant used in expression, outside the statement part of a procedure or process.
stack	Variable(s) in a block occupies more than 65536 bytes.
overflow	Value of constant or constant expression outside the interval -32768..32767.
dynamic variable not allowed	Illegal use of dynamic string variable.
initialize dynamic type not allowed	Variables of dynamic types cannot be initialized in the declaration part.
dynamic type not allowed	Dynamic types are not allowed for: set, pool, process parameter, VALUE routine parameter, function result, and structured constant.
access to zero sized object	WITH-AS: size of new type exceeds that of old
compiler error <text>	Error in pass 4. Should be reported to the maintenance staff. <text> may be one of the following: addressing bit count for error

scalar error
wrong input
error code = <no>

Compilation terminated by <error>

where <error> is:

stack error: internal compiler inconsistency; should be reported.

wrong pass 4: wrong version of pass 4.

D.4 Messages from Pass 5

Errors detected by pass 5 are indicated by the text:

*** pass 5 line <line no>, <text>

where <text> is one of the following:

no labels in case statement
An empty case statement is not allowed.

+decl caselabel
Case labels are not unambiguous.

too many formal parameters
Too many formal parameters of process or routine.

rtp35022lib does not exist
The library of standard routines is not found, this error may also be indicated by the Pascal system message: "file does not exist: rtp35022lib", followed by a trace.

user is not allowed to call system-routines
Some system-routines are protected against user call.

WARNING, Call of PROCEDURE moveg.
********************** You are breaking the RTP security.
********************** Try to find another solution
********************** inside the RTP language !!
Fatal errors are indicated by:

*** pass 5 line <line no>, compilation terminated by <text>

where <text> is one of the following:

external routine table overflow
Use of too many external declared routines.

too many formal parameters
Too many formal parameters of process or routine.

operand stack overflow

pass 4 code error
Internal inconsistency, should be reported.

inconsistency in pass 5
Should be reported.

wrong pass 4/pass 5 combination

hardware stack overflow, should be reported.

The fatal error messages are followed by some internal pass 5 status information:

```
token       = <current code>
no of items= <no of read items>
stack       = stack size
hw stack size=<size of work area>
```

This status information is followed by the pascal error message: Break, and a trace.

Internal errors in pass5 during code generation (should be reported):

error in hw stack evaluation.

followed by a line with one of the messages.
- occurred in generating code for simple open function.
- occurred when generating code for arrayw with systemcomponents

- occurred when generating code for ref removal
- occurred when generating code for shadow removal

followed by the line:
hw_stack_size is: <size>

D.5 Messages from Pass 6

constant <name> too small.
Source line number = <line no>.

If <name> is "buffersize" the program contains a structure occupying more than 64k bytes.

If <name> is "descrl_size" there are too many formal parameters in the process or routine. The indicated source line is the last line of the program.

- Compiler error detected.

  Please inform the compiler group.

  Meaning: internal compiler inconsistency.

- Error in compilation.

  Use option codesize, with at least <number> as page size.

  Meaning: the program contains a structure which needs <number> consecutive bytes on the same code page.

E. REAL-TIME PASCAL SOURCE PROGRAM UTILITIES

E.1 Indent

Text formatting program

The program performs indention of source programs depending on the options specified in the call and on the keywords (reserved words) of Pascal/Real-Time Pascal.

Call:

$$\left\{\text{<outputfile>=}\right\}_{0}^{1}\ \text{indent <input file>}\left\{\text{<option>}\right\}_{0}^{\infty}$$

<option>::=	lines	line numbers are added
	mark	the blockstructure is made clear by means of ! between matching begin-end´s
	list	the same as: lines mark
	noind	the output will be left justified
	myind	the output indention is the same as the input indention
	lc	lists keywords in capital letters and identifiers in small (lower case) letters
	uc	both key words and identifiers are listed in upper case letters
	help	produces a list of legal options

Storage requirements:

The core store required for indent is 16000 hW (size 16000).

Error messages:

If errors are detected, the ok-bit is initialized to ok.no.

??? illegal input/filename
Input file must be specified.

call: "indent help", for help
An error is detected in the program call, a new call "indent help" will produce a list of the valid options.

** warning, end(s) missing
An error in the begin-end structure has been detected.

** premature end of file.
Comment or string not terminated.

E.2 Cross Reference Program

Produces a cross reference listing of the identifiers and numbers and a use count of the PASCAL/Real Time Pascal keywords used in the input text.

The cross reference list is made with no regard to the block structure of the program. The list is sorted according to the ISO-alphabet, i.e. numbers before letters, but with no difference between upper and lower case letters.

The occurrence list for an identifier consists of a sequence of Pascal/Real-Time Pascal line numbers. The occurrence kind is specified by means of the character following the line number:

* meaning the identifier or number is found in a declaration part.

= meaning the identifier is assigned to in the line specified.

: meaning the identifier or number occurred as a label.

blank all other uses.

<<<<<<<<<<<<<< in the list is a warning denoting that the name consists of more than 12 characters, which is the number of significant characters for PASCAL-identifiers.

Call:

<output file> = cross <input file>{<option>}$_0^3$

Available options:

bossline.<yes or no>	default is no, i.e. boss line numbers are not added.
title.<name>	default name is input-file name The name will appear in the page headings.
keywords.<yes or no>	default is yes, i.e. the keywords are listed separately with occurrence counts.
survey.<yes or no>	default is no, if yes is chosen information about the cross referencing will be produced after the cross reference tables.

The page format of CROSS can be seen in the tail of the catalog entry right after the date, i.e.:

```
cross = set 30 discl d.830315.0917 46.15 0 2.0 68 ; system
        ;  249 76 3 -8388607 8388605
```

After the date 46.15 appears. This means that the page format is:

46 lines/page and 15 columns in cross table.

Storage requirements:

The core store required for cross is at least 40000 hW (size 40000), but the requirements depend on the size of the input text.

Errormess:

If errors are detected, the ok-bit is initialized to ok.no.

??? illegal output-filename
Left hand side of the call must be a name.

??? output file must be specified

??? illegal input-filename
Input file must be specified.

??? yes or no expected
Option ´bossline´ must be ´bossline.yes´ or ´bossline.no´.

??? error in bracket structure, detected at line: xx
Missing ")" (´s)

??? error in blockstructure, detected at line: xx
Unmatched end.

***** warning: element queue ran full.
Some of the names of a list are not marked within the correct attribute (:, =, *).

E.3 RTP-Compress

Some compilation time may be saved if common contexts are compressed. Typically, contexts are compiled over and over, but only changed now and then. Hence, compilation time is spent on comments and superfluos blanks from contexts which are of minimal interest for the main program. Supplied with the compiler is a utility program called rtpcompress. This program may compact the text by removing comments and blanks. The syntax of call of the program is:

```
          1
(outfile=)  rtpcompress inputfile
          0
```

If <outfile> is omitted current output is chosen.

F. REAL-TIME PASCAL OBJECT PROGRAM UTILITIES

F.1 PLIBINSERT

Process or routine libraries for RC3502 may be constructed by means of the program plibinsert.

Plibinsert may collect a group of object programs into one object file. The cross linker may then include all the programs if specified as <obj file> in the call, or just include the programs referenced from an included object program if specified as <lib file>.

Plibinsert may also be used for updating an existing library. If the program to insert already is in the library the old one will be replaced by the new one.

The call is:

$$\left\{\text{<new lib>=}\right\}_{0}^{1}\ \text{plibinsert}\ \left\{\text{<new object>}\right\}_{0}^{\infty}\ \left\{\text{lib.<old lib>}\right\}_{0}^{1}$$

<new lib> is the updated library; default name is "lib".

<old lib> is the old library, possibly empty; default name is "oldlib".

<new object> is the name of a file containing object programs to be inserted into <old lib>, possibly replacing existing entries; default name is "pass6code".

F.2 PLIBLOOKUP

Pliblookup produces a catalog listing of the programs included in a library. Programs of imagefiles may also be listed. The call is:

pliblookup $\left\{ \text{<library>} \right\}_{0}^{1}$

<library> is the library to be scanned; default name is "lib".

F.3 PLIBALL

Pliball produces a catalog listing of the programs in a library. A list of all the external routines called by the programs is produced for each program in the library. The call is identical to the call of PLIBLOOKUP.

F.4 PLIBDELETE

Plibdelete may be used to remove programs from an existing library.

The call is:

$\left\{ \text{<new lib>=} \right\}_{0}^{1}$ plibdelete $\left\{ \text{<entry>} \right\}_{1}^{\infty}$ $\left\{ \text{lib.<old lib>} \right\}_{0}^{1}$

<new lib> is the updated library; default name is "lib".

<old lib> is the old library; default name is "oldlib".

<entry>::= entry.<number>
| <program name>

Program names consisting of more than 11 characters and names including underscore may only be specified as "entry.<number>"; the entry number may be found by means of a call of pliblookup.

F.5 PLIBEXTRACT

Plibextract may be used to extract programs from an existing library.

The call is:

$$\left\{\text{<new lib>=}\right\}_{0}^{1}\ \text{plibextract}\ \left\{\text{<entry>}\right\}_{1}^{\infty}\ \left\{\text{lib.<old lib>}\right\}_{0}^{1}$$

<new lib> is the result library containing the extracted entries; default name is "lib".

<old lib> is the old library; default name is "oldlib".

<entry>::= entry.<number>
| <program name>

Program names consisting of more than 11 characters and names including underscore may only be specified as "entry.<number>"; the entry number may be found by means of a call of pliblookup.

F.6 PLIBCONVERT

Plibconvert produces an RTP3502 library object file from a list of input files of arbitrary contents. The individual programs in the result file have the new kind DATA and the same name as the corresponding input file. The result file may be inspected by PLIBLOOKUP.

$$\text{<outfile>} = \text{plibconvert}\left\{\text{descriptor.}\left\{\begin{matrix}\text{yes}\\ \text{no}\end{matrix}\right\}\right\}_{0}^{1}\ \text{<infile>}$$

<outfile> is the resulting library object file.

<infile> is the name of a file of arbitrary contents.

descriptor defines whether the descriptor segments are generated for the input files during converting. 'descriptor.yes' is mandatory,

when the final receiver of the result file is the RC3502 LOADER. ´descriptor.no´ may be specified, if you want a transparent file transfer to the receiver (NOT the RC3502 LOADER).

Default: descriptor.yes.

G. LOAD OR AUTOLOAD FILE GENERATION ON RC8000 G.

The Real-Time Pascal compiler generates binary relocatable object programs. The object programs may be loaded by the RC3502 LOADER (ref. 4 and 7) from a paper tape reader, via an FPA, or from an application process, simulating an external device. Subsection G.1 describes this way of program load. The RC3502 LOADER performs the necessary linkage editing and allocation of memory for the programs.

Another way to load programs in the RC3502 is autoloading by means of the RC3502 BOOT program. In that case the necessary editing of the object programs must be performed by the CROSSLINK program on RC8000, before an autoload file is produced. This is described in subsection G.2.

G.1 How to Generate a Loadfile G.1

G.1.1 Generating an FPA Loadfile G.1.1

If 'bxxxx' is a binary object program, produced by the Real-Time Pascal compiler, a loadfile for later load from FPA is generated on RC8000 by the call:

<outfile> = crc16 { bxxxx }$_{1}^{\infty}$

If the name of the outfile is the name of a mainprocess, the CRC16 program outputs directly to the mainprocess according to the autoload protocol.

G.2 How to Generate an Autoloadfile

G.2.1 CROSSLINK

The CROSSLINK program will generate a file containing all the specified object programs and the necessary library routines. The file will be a coreimage, i.e. references between the programs are solved.

Call:

<outfile> = crosslink { <obj file> / <lib file> / <params> }$_{0}^{\infty}$

0

<outfile> contains the generated coreimage.

<obj file>::= <file name>
<file name> contains one or more object programs which (all) must be included in the coreimage.

<lib file>::= lib. <file name>
<file name> contains one or more object programs (routines) which may be included by CROSSLINK, if they are referenced from an included object program.

<obj file> and <lib file> can be output file(s) from the Real-Time Pascal compiler or may be generated by the Utility Program PLIBINSERT (se appendix D).

<params>::= {
- start.<base>.<displacement>
- descr. { yes / no }
- map. { yes / no }
- print. { yes / no } { .<words per line> } $_{0}^{1}$
- bind. { yes / no }

start

<base> and <displacement> specify where the first word of the coreimage is supposed to be autoloaded. The start-param may only occur before the first filename.

<base> }
<displ> } ::= { <integer>
h <hexadecimal digits>

Default is: hc0.h0100.

NOTE: If the coreimage is to be autoloaded by the BOOT program, start must be: start.hc0.h0100.

descr

defines whether the descriptor segments of the following programs are included in the coreimage or not.

NOTE: This option should not be used. Descr.no will generate a fatal error from CROSSLINK.

Default: descr.yes.

map

controls listing of the included programs. Each included program is listed with:

<programkind> <programname> <date time> <start of descriptor segment> <start of code segment> <length>

Default: map.yes.

<date time> is date and time of compilation of the program.

print

controls printing of the coreimage. Each line consists of:

<A>.<B> <C>.<D> <hexadecimal contents of coreimage words>

<A> and <B> is the absolute address (i.e. base and displacement) of the first word in the line.

<C> and <D> is the corresponding relative address (within the program).

bind

bind.yes secures that a new memory module always starts with a descriptor segment.

Default: bind.yes.

Bind.no will generate an error message from CROSSLINK if the programs are not for RC3502/2.

Example

```
coreimage = crosslink start.hc0.h0100,
            (blinker,
            bmonitor,
            ballocator,
            bexception (or bminiexcept),
system3502  btimer,
            boperator,
            bopsys,  )      may be substituted
            ,        }      by your own
            bloader, )      operating system
            userprocess 1, ..., userprocess n,
            lib. stdlib
```

will generate a coreimage in a file with the name coreimage containing (all) the object program(s) in the files explicitly mentioned, extended by the necessary object programs from the library file stdlib, which contains all standard runtime routines (link, create, ...).

The coreimage will start in memory module c0, displacement 256, as demanded by the BOOT program.

A supplementary loadermap is printed (see the example above).

Example 10. Loadmap from CROSSLINK

```
*coreimage=crosslink system3502 lib.stdlib

   cross linker, version 83.08.23
    linking solved in: 2 scans

  kind       name            date     time     descr     code     length(hex)
-------------------------------------------------------------------------------
PROCESS    linker          1983.09.26 16.59    c0.0100   c0.012c   0ea8
PROCESS    monitor         1983.09.26 16.59    c0.0fa8   c0.0fd4   11ac
PROCESS    timer           1983.09.26 16.59    c0.2154   c0.2180   0bfc
PROCESS    allocator       1983.09.26 16.59    c0.2d50   c0.2d7c   0ad4
PROCESS    printexcept     1983.09.26 16.59    c0.3824   c0.3850   1062
PROCESS    adam            1983.09.26 16.59    c0.4886   c0.48b2   07c0
PROCESS    operator        1983.09.26 16.59    c0.5046   c0.5076   07e6
PROCESS    console         1983.09.26 19.22    c0.582c   c0.5864   072a
PROCESS    opsys           1983.09.26 16.59    c0.5f56   c0.5f80   165a
PROCESS    loader          1983.09.26 16.59    c0.77b0   c0.77e0   1eac
PROCEDURE  break           1983.09.26 17.11    c0.965c   c0.9690   00fc
FUNCTION   create          1983.09.26 17.11    c0.9758   c0.9798   0218
PROCEDURE  definetimer     1983.09.26 17.11    c0.9a50   c0.9a80   00b8
PROCEDURE  exception       1983.09.26 17.11    c0.9b08   c0.9b38   00ea
FUNCTION   initpool        1983.09.26 17.11    c0.9bf2   c0.9c2e   016c
FUNCTION   initsem         1983.09.26 17.11    c0.9d5e   c0.9d9a   0158
PROCEDURE  +initpool+rc    1983.09.26 17.11    c0.9eb6   c0.9eee   0156
FUNCTION   link            1983.09.26 17.11    c0.a00c   c0.a044   00e6
PROCEDURE  remove          1983.09.26 17.11    c0.a0f2   c0.a122   0324
FUNCTION   reservech       1983.09.26 17.11    c0.a416   c0.a452   00f2
PROCEDURE  sendallocato    1983.09.26 17.11    c0.a508   c0.a538   0074
PROCEDURE  sendlinker      1983.09.26 17.11    c0.a57c   c0.a5ac   0074
PROCEDURE  sendtimer       1983.09.26 17.11    c0.a5f0   c0.a620   0074
PROCEDURE  start           1983.09.26 17.11    c0.a664   c0.a698   00ee
PROCEDURE  stop            1983.09.26 17.11    c0.a752   c0.a782   00be
FUNCTION   unlink          1983.09.26 17.11    c0.a810   c0.a844   00d2
PROCEDURE  openopzone      1983.09.26 17.11    c0.a8e2   c0.a932   0178
PROCEDURE  alloc           1983.09.26 17.11    c0.aa5a   c0.aa92   0080
PROCEDURE  print+descri    1983.09.26 17.11    c0.aada   c0.ab0e   0146
PROCEDURE  print           1983.09.26 17.11    c0.ac20   c0.ac60   018c
PROCEDURE  outaddr         1983.09.26 17.11    c0.adac   c0.adec   00a2
PROCEDURE  outhex          1983.09.26 17.11    c0.ae4e   c0.ae86   010a
PROCEDURE  outdate         1983.09.26 17.11    c0.af58   c0.af8c   00c8
PROCEDURE  outtime         1983.09.26 17.11    c0.b020   c0.b054   00b0
PROCEDURE  outinteger      1983.09.26 17.11    c0.b0d0   c0.b108   0142
PROCEDURE  opin            1983.09.26 17.11    c0.b210   c0.b242   009c
PROCEDURE  opwait          1983.09.26 17.11    c0.b2ae   c0.b2e2   011a
PROCEDURE  ininteger       1983.09.26 17.11    c0.b3c8   c0.b3fc   014c
PROCEDURE  inhex           1983.09.26 17.11    c0.b514   c0.b548   012a
PROCEDURE  inname          1983.09.26 17.11    c0.b63e   c0.b672   00fa
PROCEDURE  outnl           1983.09.26 17.11    c0.b738   c0.b768   006e
PROCEDURE  outfill         1983.09.26 17.11    c0.b7a6   c0.b7de   0090
PROCEDURE  outalfa         1983.09.26 17.11    c0.b836   c0.b86a   00b8
PROCEDURE  outchar         1983.09.26 17.11    c0.b8ee   c0.b922   00fe
PROCEDURE  inchar          1983.09.26 17.11    c0.b9ec   c0.ba20   010a
PROCEDURE  outend          1983.09.26 17.11    c0.baf6   c0.bb26   009a
   end, cross linker.

end
blocksread = 53
*o c
```

Error messages from CROSSLINK

The possible error/warning messages are:

No outfile specified
Error in crosslink
Error in call of crosslink; parameter number NN
Illegal hexadecimal number in call; parameter number NN
Bad version, recompile NNNNN
yes or no expected after "descr."
Illegal contents of object or library file
Too many modules, (more than 200)
Illegal value of "start"-parameter; parameter number NN
Too many formal parameters, module: NNNNN
yes, no or yes.<number> expected after "print."
Unknown option; parameter number NN
Name expected after "lib."; parameter number NN
Inconsistency in loader map
yes or no expected after "map."
yes or no expected afer "bind."
"bind.yes" required by RC3502/2

G.2.2 Generating an FPA Autoloadfile

Assume coreimage has been generated by CROSSLINK.

The call:

<outfile> = crc16 coreimage

will generate a file with the correct format for the RC8000 AUTOLOAD-program.

If the name of the outfile is the name of a mainprocess, the CRC16 program outputs directly to the mainprocess according to the autoload protocol.

H. COMPLETE LIST OF LANGUAGE SYMBOLS

AND	ENDLOOP	LOOP	SHIFT
ARRAY	EXIT	MOD	THEN
AS	EXITLOOP	NOT	TO
BEGIN	EXPORT **)	OF	TYPE
BEGINBODY *)	EXTERNAL	OR	TYPESIZE
CASE	FOR	OTHERWISE	UNTIL
CHANNEL	FORWARD	PACKED	VAR
CONST	FUNCTION	POOL	VARSIZE
CONTINUELOOP	GOTO	PREFIX	WHILE
DIV	IF	PROCEDURE	WITH
DO	IN	PROCESS	XOR
DOWNTO	LABEL	RECORD	
ELSE	LOCK	REPEAT	
END	LOCKMES	SET	

+	-	*	/	"	'	<	>
<>	<=	>=	(	)	(.	.)	(:
:)	=	:=	:=:	.	,	:	;
***	(*	*)	<*	*>	--	!	?
..	$	&	↑	#			

*) reserved for internal use.
**) only for Z80 version.

I. PREDEFINED CONSTANTS, TYPES, AND VARIABLES I.

The following constants and types are predefined:

```
CONST

alfalength      = 12;

maxpriority     = 0;
minpriority     = -2;
stdpriority     = minpriority;

break_by_father = 47;

maxint          = 32767;
minint          = -32768;

TYPE

bit     = 0..1;
byte    = 0..255;

double =
RECORD
  ?, ? : integer;
END;

boolean = (false,true);

char    =(
(*            0    1    2    3    4    5    6    7    8    9  *)

(*   0 *)   nul, soh, stx, etx, eot, enq, ack, bel,  bs,  ht,
(*  10 *)    nl,  vt,  ff,  cr,  so,  si, dle, dc1, dc2, dc3,
(*  20 *)   dc4, nak, syn, etb, can,  em, sub, esc,  fs,  gs,
(*  30 *)    rs,  us,  sp,   ?,   ?,   ?,   ?,   ?,   ?,   ?,
(*  40 *)     ?,   ?,   ?,   ?,   ?,   ?,   ?,   ?,   ?,   ?,
(*  50 *)     ?,   ?,   ?,   ?,   ?,   ?,   ?,   ?,   ?,   ?,
(*  60 *)     ?,   ?,   ?,   ?,   ?,   ?,   ?,   ?,   ?,   ?,
(*  70 *)     ?,   ?,   ?,   ?,   ?,   ?,   ?,   ?,   ?,   ?,
(*  80 *)     ?,   ?,   ?,   ?,   ?,   ?,   ?,   ?,   ?,   ?,
(*  90 *)     ?,   ?,   ?,   ?,   ?,   ?,   ?,   ?,   ?,   ?,
(* 100 *)     ?,   ?,   ?,   ?,   ?,   ?,   ?,   ?,   ?,   ?,
(* 110 *)     ?,   ?,   ?,   ?,   ?,   ?,   ?,   ?,   ?,   ?,
(* 120 *)     ?,   ?,   ?,   ?,   ?,   ?,   ?, del );

alfa = ARRAY (1..alfalength) OF char;

adamsemtype = (adamsem, operatorsem, loadersem,
               globalsem, loaddriversem, paxsem,
```

```
                restartsem,
                ?,?,?,?,?,?,?,?,?,?,?,?,?,?,?,?,?,
                ?,?,?,?,?,?,?,?,?);
adamvector = ARRAY (adamsemtype) OF ↑ semaphore;
system_vector = !adamvector;

activation = (a_interrupt, a_semaphore, a_delay);

message_header =
             RECORD
               .
               .
             u1,u2,u3,u4 : byte;
             messagekind : !integer;
             size        : !integer;
               .
               .
          END;

reference = ↑ message_header;

coded_date = PACKED RECORD
                year_after_1900   : 0..127;
                month             : 0..12;
                day               : 0..31;
             END;

coded_time = PACKED RECORD
                compiler_version  : 0..31;
                hour              : 0..23;
                minute            : 0..59;
             END;

coded_secs = PACKED RECORD
                sec               : 0..59;
                msec              : 0..999;
             END;

code_inc   = PACKED RECORD
                days              : 0..31;
                hours             : 0..23;
                mins              : 0..59;
                secs              : 0..59;
                msecs             : 0..999;
              END;
```

```
delaytype  = RECORD
                prev_date          : coded_date;
                prev_time          : coded_time;
                prev_secs          : coded_secs;
                inc                : coded_inc;
             END;

clocktype = RECORD
              date : coded_date;
              time : coded_time;
              secs : coded_secs;
            END;

incarnation_descriptor =
            RECORD
               .
               .
              timer : integer;
               .
               .
              incname: alfa;
               .
               .
            END;

lookup_descriptor_segment =
            RECORD
               name                : alfa;
               descriptor_length   , (* bytes *)
               no_of_pages         ,
               pagesize            , (* bytes *)
               last_page_length    , (* bytes *)
               kind                : integer;
                                     (* 1: process
                                        2: procedure
                                        3: function
                                        6: data *)
               default_appetite    ,
               last_param_offset   ,
               no_of_params        : integer;
               date                : coded_date;
               time                : coded_time;
            END;
VAR
  own: incarnation_descriptor;
```

The variable OWN.INCNAME is initialized by the CREATE routine.

The values of the predefined type integer constitutes

the subrange minint..maxint.

In the RC3502 implementation the following relations between the fields in a message header referred by the reference variable r hold:

size	message kind	empty(r)	# elements in message header stack	# elements in message data stack	
> 0	> 0	true	1	1	
> 0	> 0	false	> 1	≥ 1	data message
> 0	0	false	> 1	≥ 1	
0	0	true	1	0	header
0	0	false	> 1	0	message
facility mask	< 0	true	1	0	channel message

The ´size´ field specifies the number of words (16 bits) of the associated message data. The maximum size a message can take is 32 K words, where SIZE = -32768 (minint).

´Facility mask´ originates from the call of RESERVECH (see chapter 5).

J. PREDEFINED ROUTINES J.

```
FUNCTION  abs(x : niltype) : niltype; EXTERNAL;

PROCEDURE alloc(VAR r : reference; VAR p : pool 1 ;
  VAR s : semaphore); EXTERNAL;

PROCEDURE break(VAR sh : shadow; excode : integer); EXTERNAL;

FUNCTION bufsize(VAR r : reference) : integer; EXTERNAL;

FUNCTION  chr(int : 0..127) : char; EXTERNAL;

FUNCTION clock_increment(t : clocktype;
  inc : coded_inc) : clocktype; EXTERNAL;

FUNCTION clock_less_than(t1, t2 : clocktype) : boolean;
EXTERNAL;

FUNCTION clock_difference(t1, t2 : clocktype) : clocktype;
EXTERNAL;

FUNCTION  crc16 ( op1,op2 : integer ) : integer; EXTERNAL;

FUNCTION  create (incarnationname : alfa;
  proces  : processrec;
  VAR sh  : shadow;
  storage : integer) : integer;
EXTERNAL;

PROCEDURE definetimer(onoff:boolean); EXTERNAL;

FUNCTION delete_semaphore(var s_name : !alfa) : integer;
EXTERNAL;

FUNCTION  empty(VAR r : reference) : boolean; EXTERNAL;

FUNCTION  eoi : boolean; EXTERNAL;

PROCEDURE exception(excode : integer); EXTERNAL;

FUNCTION first(var r : reference) : integer; EXTERNAL;

FUNCTION getclock : clocktype; EXTERNAL;

PROCEDURE inbyteblock
  (VAR next : integer;
  first,last : integer;
  VAR msg : reference);
EXTERNAL;
```

```
PROCEDURE inwordblock
  (VAR next : integer;
  first,last : integer;
  VAR msg : reference);
EXTERNAL;

FUNCTION last(var r : reference) : integer; EXTERNAL;

FUNCTION  link(external_name : alfa;
  VAR proces : process_descriptor) : integer; EXTERNAL;

FUNCTION  locked(VAR s : semaphore) : boolean; EXTERNAL;

FUNCTION name_semaphore(VAR s : semaphore;
  VAR s_name : !alfa) : integer; EXTERNAL;

FUNCTION next(VAR r : reference) : integer; EXTERNAL;

FUNCTION  nil(VAR r : ↑ niltype) : boolean; EXTERNAL;

FUNCTION  open(VAR s : semaphore) : boolean; EXTERNAL;

FUNCTION  openpool(VAR p : pool 1) : boolean; EXTERNAL;

FUNCTION  ord(x : niltype) : integer; EXTERNAL;

PROCEDURE outbyteblock
  (VAR next : integer;
  first,last : integer;
  VAR msg : reference);
EXTERNAL;

PROCEDURE outwordblock
  (VAR next : integer;
  first,last : integer;
  VAR msg : reference);
EXTERNAL;

FUNCTION  ownertest (VAR p : pool 1;
  VAR r : reference) : boolean;
EXTERNAL;

FUNCTION  passive(VAR s : semaphore) : boolean; EXTERNAL;

PROCEDURE pop(VAR r1, r2 : reference); EXTERNAL;

FUNCTION  pred(x : niltype) : niltype; EXTERNAL;

PROCEDURE push(VAR r1, r2 : reference); EXTERNAL;
```

```
FUNCTION  ref(VAR s : semaphore) : ↑ semaphore; EXTERNAL;

PROCEDURE release(VAR r : reference); EXTERNAL;

PROCEDURE remove(VAR sh : shadow); EXTERNAL;

FUNCTION  reservech (VAR ch_msg : reference;
  level, mask : integer) : integer;
EXTERNAL;

PROCEDURE return(VAR r : reference); EXTERNAL;

FUNCTION search_semaphore(VAR s_name : !alfa) : ↑ semaphore;
EXTERNAL;

PROCEDURE sendlinker(VAR r : reference); EXTERNAL;

PROCEDURE sendtimer(VAR r : reference); EXTERNAL;

PROCEDURE sensesem(VAR r : reference; VAR s : semaphore);
EXTERNAL;

PROCEDURE setnext(VAR r : reference; val : integer);
EXTERNAL;

PROCEDURE setpriority(priority : integer); EXTERNAL;

PROCEDURE setu1(VAR r : reference; val : byte); EXTERNAL;

PROCEDURE setu2(VAR r : reference; val : byte); EXTERNAL;

PROCEDURE setu3(VAR r : reference; val : byte); EXTERNAL;

PROCEDURE setu4(VAR r : reference; val : byte); EXTERNAL;

PROCEDURE signal(VAR r : reference; VAR s : semaphore);
EXTERNAL;

PROCEDURE start(VAR sh : shadow; priority : integer);
EXTERNAL;

PROCEDURE stop(VAR sh : shadow); EXTERNAL;

FUNCTION  succ(x : niltype) : niltype; EXTERNAL;

FUNCTION  swap (i : integer) : integer; EXTERNAL;

PROCEDURE trace (code : integer) ; EXTERNAL;

FUNCTION  unlink(VAR proces : process_descriptor) : integer;
```

```
EXTERNAL;

FUNCTION u1(VAR r : reference) : byte; EXTERNAL;

FUNCTION u2(VAR r : reference) : byte; EXTERNAL;

FUNCTION u3(VAR r : reference) : byte; EXTERNAL;

FUNCTION u4(VAR r : reference) : byte; EXTERNAL;

PROCEDURE wait(VAR r : reference; VAR s : semaphore);
EXTERNAL;

PROCEDURE waiti; EXTERNAL;

PROCEDURE waits (VAR r : reference; VAR s : semaphore);
EXTERNAL;

PROCEDURE waitd(delay : integer); EXTERNAL;

FUNCTION  waitid(delay : integer) : activation; EXTERNAL;

FUNCTION  waitis( VAR r : reference;
  VAR s : semaphore) : activation;
EXTERNAL;

FUNCTION  waitsd (VAR r : reference;
  VAR s : semaphore;
  delay : integer) : activation;
EXTERNAL;

FUNCTION  waitisd (VAR r : reference;
  VAR s : semaphore;
  delay : integer) : activation;
EXTERNAL;

PROCEDURE ___exit___rc; EXTERNAL;

PROCEDURE _initpool_rc (VAR s : semaphore;
  number, size : integer);
EXTERNAL;
```

K. IOENVIR K.

```
CONST

  linelength       = 80;
  firstindex       = 6 + alfalength;
  lastindex        = firstindex + linelength - 1;

TYPE

  opbuffer =
  RECORD
  !  first ,
  !  last  ,
  !  next  : integer;
  !  name  : alfa;
  !  chars : ARRAY (firstindex..lastindex) OF char
  END;

  zone =
  RECORD
  !  driver    : ↑semaphore;     (*  operator process       *)
  !  answer    : ↑semaphore;     (*  answers returns here   *)
  !  dataready : semaphore;      (*  buffers with data      *)
  !  free      : semaphore;      (*  free buffers           *)
  !  cur       : reference;      (*  current buffer         *)
  !  u2val     : byte;           (*  u2 to driver           *)
  !  state     : byte;           (*  resultcode from answer *)
  !  readstate : integer;        (*  0:ok,>0:error,-1:cur=nil *)
  !  nextp     : integer;        (*  pointer into databuf   *)
  !  lastpos   : integer;        (*  last position in buffer*)
  END;

  PROCEDURE openzone ( VAR z: zone; driv, answ: ↑semaphore;
    bufs: integer; VAR home: pool 1; v1, v2, v3, v4: byte );
  EXTERNAL;

  PROCEDURE openopzone ( VAR z: zone; driv, answ: ↑semaphore;
    bufs: integer; VAR home: pool 1; v1, v2, v3, v4: byte  );
  EXTERNAL;

  PROCEDURE outend ( VAR z: zone);
  EXTERNAL;

  PROCEDURE outchar ( VAR z: zone; t: char);
  EXTERNAL;

  PROCEDURE outalfa ( VAR z: zone; VAR text: !alfa);
  EXTERNAL;
```

```
PROCEDURE outfill ( VAR z: zone; filler: char; rep: integer);
EXTERNAL;

PROCEDURE outinteger ( VAR z: zone; num, pos: integer);
EXTERNAL;

PROCEDURE outhex ( VAR z: zone; num, pos: integer);
EXTERNAL;

PROCEDURE outdouble ( VAR z: zone; d: double; pos: integer);
EXTERNAL;

PROCEDURE outnl(VAR z : zone);
EXTERNAL;

PROCEDURE outaddr(VAR z : zone; a : addr);
EXTERNAL;

PROCEDURE outdate(VAR z : zone; date : coded_date);
EXTERNAL;

PROCEDURE outtime(VAR z : zone; time : coded_time);
EXTERNAL;

PROCEDURE opin ( VAR z: zone);
EXTERNAL;

PROCEDURE opanswer ( VAR msg: reference; VAR z: zone);
EXTERNAL;

FUNCTION optest ( VAR z: zone): boolean;
EXTERNAL;

PROCEDURE opwait ( VAR z: zone; VAR inputpool: pool 1);
EXTERNAL;

PROCEDURE inchar ( VAR z: zone; VAR t: char);
EXTERNAL;

PROCEDURE ininteger ( VAR z: zone; VAR num: integer);
EXTERNAL;

PROCEDURE inhex ( VAR z: zone; VAR num: integer);
EXTERNAL;

PROCEDURE indouble ( VAR z: zone; VAR d : double);
EXTERNAL;

PROCEDURE inname ( VAR z: zone; VAR name: alfa);
EXTERNAL;
```

```
PROCEDURE printmessage (VAR z: zone;
  VAR r: reference;
  firstindex,
  lastindex,
  words_per_line: integer);
EXTERNAL;
```

L. OPERATOR INPUT/OUTPUT EXAMPLE L.

```
PROCESS testiol (VAR sv : system_vector);
    (* single stream communication with operator *)
    (* where all input lines terminated by <nl> *)
    (* are echoed on the console                *)
CONST
  readcode          = 1;
  writecode         = 2;
  message           = 7;
  no_of_opbuffers   = 2;
  no_of_inbuffers   = 1;
  no_of_outbuffers  = no_of_opbuffers - no_of_inbuffers;
VAR
  t         : char;
  oppool    : pool no_of_opbuffers OF opbuffer;
  keyboard ,
  display   : zone;
BEGIN
  openopzone (display, sv(operatorsem),
  ref(display.free), no_of_outbuffers, oppool,
  writecode, message, 0, 0);

  outfill (display, '.', 10);
  outalfa (display, ' Welcome to ');
  outalfa (display, own.incname);
  outfill (display, '.', 10);
  outchar (display, nl);

    (* now open the zone keyboard for operator input *)
  openopzone (keyboard, sv(operatorsem),
  ref(keyboard.dataready), no_of_inbuffers, oppool,
  readcode, message, 0, 0);

  REPEAT
    outalfa (display, 'input <nl>:');
    outend  (display);
    opin    (keyboard);
    opwait  (keyboard, oppool);
    REPEAT
      inchar  (keyboard, t);
      outchar (display , t);
      IF t = nl THEN
        BEGIN
          outchar (display, bel);
          outend  (display);
        END;
    UNTIL keyboard.readstate < 0;
  UNTIL false
END.
```

```
PROCESS testio2 (VAR sv : system_vector);
    (* the program prints the global time *)
    (* every 30 seconds and echoes all    *)
    (* input lines from the keybord on    *)
    (* the console . The communication    *)
    (* is implemented as two logical      *)
    (* streams                            *)
CONST
  readcode          = 1;
  writecode         = 2;
  message           = 7;
  no_of_opbuffers   = 2;
  no_of_inbuffers   = 1;
  no_of_outbuffers  = no_of_opbuffers - no_of_inbuffers;
  stream0           = 0;
  stream1           = 1;
  longdelay         = 13;
  getclock          = 1;

VAR
  t          : char;
  oppool     : pool no_of_opbuffers OF opbuffer;
  delaypool  : pool 1 OF delaytype;
  keyboard   ,
  display    : zone;
  mainsem    : semaphore;
  r          : reference;

  PROCEDURE init_timercom;
  BEGIN
    alloc (r, delaypool, mainsem);
    r↑.u1 := getclock;
    sendtimer (r);
    wait (r, mainsem);
    LOCK r AS d : delaytype DO
      WITH d.inc DO
        BEGIN
          days  := 0;
          hours := 0;
          mins  := 0;
          secs  := 30;
          msecs := 0;
        END;
    r↑.u1 := longdelay;
    r↑.u4 := stream0;
    sendtimer (r);
      (* from now on this is input on stream 0 *)
  END; (* init_timercom *)
```

```
  PROCEDURE init_opcom;
  BEGIN
      (* open outputzone for operator output *)
    openopzone (display, sv(operatorsem),
    ref(display.free), no_of_outbuffers, oppool,
    writecode, message, 0, 0);
      (* print welcome text on display *)
    outfill (display, '.', 10);
    outalfa (display, ' Welcome to ');
    outalfa (display, own.incname);
    outfill (display, '.', 10);
    outchar (display, nl);
      (* now open the zone keyboard for operator *)
      (* operator on stream 1                    *)
    openopzone (keyboard, sv(operatorsem),
    ref(mainsem), no_of_inbuffers, oppool,
    readcode, message, 0, stream1);
  END; (* init_opcom *)

BEGIN (* main program *)

  init_timercom;
  init_opcom;

    (* first output to operator *)
  outalfa (display, 'input <nl>:');
  outnl   (display);

    (* first input to operator *)
  opin (keyboard);

  REPEAT

    wait (r, mainsem);

    CASE r↑.u4 OF
      stream0:
        BEGIN
          LOCK r AS d : delaytype DO
            BEGIN
              outdate (display, d.prev_date);
              outchar (display, sp);
              outtime (display, d.prev_time);
              outnl   (display);
            END;
          sendtimer (r);
        END;
```

```
      streaml:
        BEGIN
          opanswer (r, keyboard);
          opwait   (keyboard, oppool);
          REPEAT
            inchar  (keyboard, t);
            outchar (display , t);
            IF t = nl THEN
              BEGIN
                outchar (display, bel);
                outend  (display);
              END;
          UNTIL keyboard.readstate < 0;
          outalfa (display, 'input <nl> :');
          outnl   (display);
          opin    (keyboard);
        END;
    END; (* case *)
  UNTIL false

END;
.
```

M. EXCEPTION CODES M.

Code (Hex)	Meaning
1	- parity error
2	- registerstack error
3	- undefined opcode
4	- odd number of bytes
5	- stack overflow
6	- pointer = nil
7	- signal: reference = nil
	- push: first param = nil
	- pop: second param = nil
	- lock: reference = nil
	- reference = nil
8	- wait: reference <> nil
	- pop: first param <> nil
9	- push: param locked
	- pop: second param locked
	- signal: reference locked
	- reference locked
A	- lock overflow
B	- arithmetic overflow
C	- index out of bounds
	- subrange out of bounds
D	- illegal zonestate
E	- field overflow
F	- move wraparound
10	- push: identical arguments
11	- push: first param not empty
12	- lock: size error
	- size too small
13	- last < first
14	- lock: not data message
	- not data message
15	- not channel message
16	- word block i/o: odd number of bytes
17	- block i/o at level 0
18	- setcr: first limit negative
19	- setad: truncation error
1A	- no resources
1B	- file does not exist
1C	- try to read past eof
1D	- wrong answer
1E	- setpriority: illegal priority
1F	- pool: no core
20	- shadow = nil
21	- arithmetic overflow
22	- system error
23	- system error

24	- illegal switch in case construction
25	- upper limit in call of succ
26	- lower limit in call of pred
27	- with: size error
28	- lockmes: size error
29	- local reference variable not nil at routine exit
2A	- local shadow variable not nil at routine exit
2B-2E	- system error
2F	- break by father

N. INDICES N.

N.1 Survey of Figures N.1

N.2 Survey of Examples N.2

N.3 Catchword Index

RETURN LETTER

Title: RC3502/2 REAL-TIME PASCAL Reference Manual PN: 99000994

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

RC International

Information Department
Lautrupbjerg 1
DK-2750 Ballerup
Denmark