RCSL No: 52-AA1072

Edition: October, 1981

Author: Bo Bagger Laursen

Title:

RC3502 REAL TIME PASCAL, Extensions to the Reference Manual

RCSL 42-i1592

A/S REGNECENTRALEN af 1979

Keywords:

REAL TIME PASCAL, RC3502, Software.

Abstract:

This is a description of new routines available in RC3502 Real Time Pascal, revision 5.

(30 printed pages).

Printed by A/S Regnecentralen af 1979, Copenhagen

RCSL 42-i1592

CONTENTS PAGE

CONTENTS (continued) PAGE

1. INTRODUCTION 1.

This is a description of predefined routines as of revision 5, which are not described in:

RC3502-PASCAL80 Reference Manual
RCSL: 42-i1542

or:

RC3502 REAL TIME PASCAL
Character Input/Output Routines
RCSL: 52-AA1056

Furthermore, a number of routines, which can be accessed by external declarations, are described.

2. PREDEFINED TYPES


```
TYPE

activation  = (a_interrupt, a_semaphore, a_delay);

adamsemtype = (allocatorsem, adamsem, operatorsem,
               loadersem,?,?,?,?,?,?,?,?,?,?,?,?,?);

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;

delaytype   = RECORD
                 prev_date         : coded_date;
                 prev_time         : coded_time;
                 prev_secs         : coded_secs;
                 inc : PACKED RECORD
                          days     : 0..63;
                          hours    : 0..23;
                          mins     : 0..59;
                          secs     : 0..59;
                          msecs    : 0..999;
                       END;
              END;
```

3. PREDEFINED ROUTINES 3.

'16_bit_type' or '32_bit_type' means that you can use any type, that occupies 2 or 4 bytes, and does not contain system types, i.e. pointer, semaphore, reference, shadow, pool.

3.1 crc16 3.1

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

- op1 represents the polynomium

$$f(x) = a_{15} x^{15} + a_{14} x^{14} + \ldots + a_1 x + a_0$$

where a_j = op1.bit$_j$.

(Note: Bit$_0$ is the most significant bit).

op2 represents the polynomium

$$g(x) = x^{16} + b_{15} x^{15} + b_{14} x^{14} + \ldots + b_1 x + b_0$$

where b_j = op2.bit$_j$.

Note that x^{16} by convention is implicitly given.

The routine delivers the remainder by the division

$$(f(x) * x^8)/g(x)$$

```
PREFIX example1;
  PROCEDURE example1;
      (* generate the crc16 remainder *)
  CONST
    quotient = -32768 + 8092 + 1; (* x^16 + x^15 + x^2 + 1   *)
    n        = 768;
  VAR
    remainder ,
    i         : integer;
    data      : ARRAY (1 .. n + 2) OF byte;

  BEGIN
  ! remainder := 0; (* init value *)
  ! FOR i := 1 TO n DO
  !   remainder := crc16 (remainder xor data(i), quotient);
  !   (* now remainder contains the crc16 remainder *)
  !   (* the least significant byte of remainder    *)
  !   (* is supposed to be sent first               *)
  ! data (n + 1) := remainder AND 255;
  ! data (n + 2) := swap (remainder) AND 255;
  END; (* example1 *)
```

□

3.2 definetimer

3.2

FUNCTION definetimer (onoff : boolean);

- time out can only take place after the call 'definetimer (true)'. This call 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.

□

3.3 inbyteblock (revised) 3.3

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

- works like the earlier 'inbyteblock', but with no channel message as parameter. This should be used instead of the old one.

□

3.4 inwordblock (revised) 3.4

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

- works like the earlier 'inwordblock', but with no channel message as parameter. This should be used instead of the old one.

□

3.5 outbyteblock (revised) 3.5

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

- works like the earlier 'outbyteblock', but with no channel message as parameter. This should be used instead of the old one.

□

3.6 outwordblock (revised) 3.6

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

- As the old one, but with no channel message as parameter.

□

3.7 sendlinker

PROCEDURE sendlinker (VAR r : reference);

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

The parameter must refer to a message (must not be nil), otherwise an exception occurs.

	message	answer
u1	function	unchanged
u2	not used	result
u3	param	unchanged
u4	not used	unchanged

An unknown function is returned with result = 6.

Function = 3 (LOOKUP NAME)

The message is supposed to hold a variable 'name' of type 'alfa', and the size must be at least 22 words to contain the OK answer.

The LINKER catalog is looked up for a program with name 'name'.

Results	Meaning
1	No program with name 'name' is found in the LINKER catalog.
0	OK. A program with name 'name' is found in the LINKER catalog. The answer contains a description segment of type

```
RECORD
   descriptor_length          , (* bytes *)
   no_of_pages                ,
   pagesize                   , (* bytes *)
   last_page_length           , (* bytes *)
   kind                       : integer; (* 1: process 2: procedure
   name                       : alfa;                  3: function *)
   entry_point                ,
   exception_point            ,
   exit_point                 ; 32_bit_type
   default_appetite           ,
   last_param_offset          ,
   no_of_params               : integer;
   date                       : coded_date;
   time                       : coded_time;
END;
```

function = 7 , param = module no (CHECK)

A crc16 check is performed on the module.

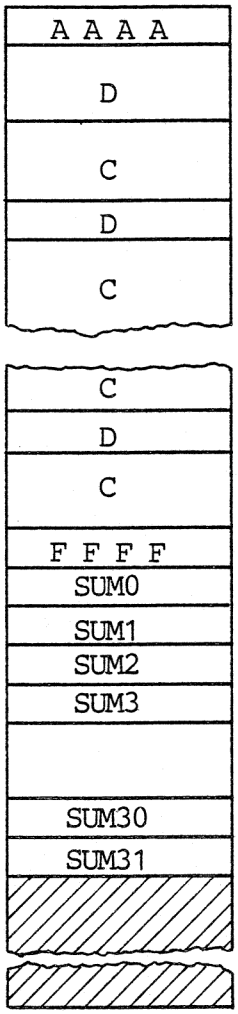
Note: The size of the message must be at least 64 words.

The polynomium used is:

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

with remainder = -1 as initial value.

The module must have the format:



D: Descriptor segment

C: Code segment

0- 4K even addresses
0- 4K odd addresses
4- 8K even adresses
4- 8K odd addresses

60-64K even addresses
60-64K odd addresses

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 message
1	The module is empty

□

3.8 sendtimer (revised) 3.8

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

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

  The parameter must refer to a message (must not be nil), otherwise an exception occurs.

	message	answer	
u1	unused	unchanged	
u2	unused	unchanged	(!)
u3	delay1	0	(!)
u4	delay2	0	(!)

  If u3 <> 0 then the message is returned after $delay1 * 2^{delay2}$ msec. (max. 255 * 256 = 65280 msec.).

  If u3 = 0 the message is supposed to hold a datamessage of type 'delaytype'.

The interpretation depends on the value of u4:

u3 = 0, u4 = 0

Delay-message.

Prev time is set to "prev time + increment". If prev time is before global time, prev time is set equal to global time and the message is returned immediately, otherwise the message is returned when global time passes this new value of prev time.

Note: This means that the message is ready to be sent to the timer once again, with all parameters unchanged.

u3 = 0, u4 = 1

Set clock message.

Global time is assigned from prev time. (Increment is not used).

The message is returned immediately.

Note: u3 and u4 are both zero at return, which means that the message is ready to be used as delay message.

u3 = 0, u4 = 2

Get clock message.

Prev time is assigned from global time. (Increment is not used).

The message is returned immediately.

Note: See Set clock note.

u3 = 0, u4 > 2

Not implemented yet!

(Works for the time being as get clock!).

3.9 setpriority 3.9

PROCEDURE setpriority (priority : integer);

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

□

3.10 swap 3.10

FUNCTION swap (i : integer): integer;

- works as 'lambda', but is predefined.

□

3.11 trace 3.11

PROCEDURE trace (code : integer);

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

□

3.12 waitd 3.12

PROCEDURE waitd (delay : integer);

- 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 call 'definetimer (true)' has been issued.

□

3.13 waiti

PROCEDURE waiti;

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

□

3.14 waitid

FUNCTION waitid (delay : integer) : activation;

- 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.

□

3.15 waitis

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

- 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 semaphore is locked or the reference 'r' is not nil.

□

3.16 waitisd 3.16

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

- works as 'ctrwaitisd' except that no controlword is output.

□

3.17 waits 3.17

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

- works as the procedure 'wait'.

□

3.18 waitsd 3.18

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

- 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 semaphore is locked or the reference 'r' is not nil. Note, that delay activation only takes place, if 'definetimer' has been called.

□

4. ZONE ROUTINES

In order to use the following routines you make an external declaration of the routines and compile with IOENVIR.

4.1 print


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

- prints the memory locations (word alligned)

'base.first_disp' through 'base.last_disp'

with the layout

<address> { <word hex> <word decimal>
<left byte decimal> <right byte decimal>
<left char> <right char> } words_per_line
<nl>
1

□

4.2 printmessage


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

- 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)'.

□

5. MISCELLANEOUS ROUTINES 5.

The following routines may be accessed by an external declaration.

'16_bit_type' or '32_bit_type' means that you can use any type, that occupies 2 or 4 bytes, and does not contain system types, i.e. pointer, semaphore, reference, shadow, pool.

5.1 clearlevel 5.1

PROCEDURE clearlevel;

- clears the current interrupt and waits for an interrupt. If the current level has status 'timed out', the call has no effect.

□

5.2 controlclr 5.2

PROCEDURE controlclr (control_word: 16_bit_type);

- works like the earlier 'controlclr', but with no channel message as parameter. This should be used instead of the old one.

□

5.3 ctrwaitid 5.3

FUNCTION ctrwaitid (c:16_bit_type, delay: integer): activation;

- 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' 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.

□

5.4 ctrwaitis


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

- 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 occurs if the semaphore is locked or the reference 'r' is not nil.

□

5.5 ctrwaitisd


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

- 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 interrupt level, and the wait is performed.

Delay activation only takes place, if a call of 'definetimer' has been issued. An exception occurs, if the semaphore is locked or the reference 'r' is not nil.

□

5.6 getlfgf

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

- 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.

```
PROCESS example2;
TYPE
  addr = RECORD
           !  base, disp : integer;
           END;

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

  PROCEDURE print (VAR z : zone;
    base, 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, 1);
  END;

BEGIN
!   (* process body *)
END; (* example 2 *)
```

□

5.7 increment_mod_64k

5.7

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

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

□

5.8 initpool

5.8

```
FUNCTION initpool (VAR p : pool 1;
                   number      ,
                   size_in_words : integer) : integer;
```

- 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 return to the pool variable by a 'release' call. The function may be used to obtain a more flexible initialization of pool variables.

```
PROCESS example3;
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);  (* r.answer := ref (s) *)
!    ! signal (r, driver);
!    ! wait   (r,s);
!    END;
END; (* example3 *)
```

□

5.9 initsem 5.9

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

- 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. If used for communication, they 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.

□

5.10 intel 5.10

```
FUNCTION intel (i : integer) : 16_bit_type;
```

- converts an RC3502 integer to an INTEL integer.

  Note: The function result must not be of type integer.

  Example: See example under 'FUNCTION lambda'.

□

5.11 lambda 5.11

```
FUNCTION lambda (ii : 16_bit_type) : integer;
```

- converts an INTEL integer to an RC3502 integer.

  This is the main purpose, but the conversion takes place by swapping the two bytes on top of the stack, and leaves the result on stack top. This effective way of swapping may be used for conversion between integers and the u-fields in message headers.

  Note: See also the predefined routine 'swap'.

```
PREFIX example4;
  PROCEDURE example4;
  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 := intel (i);
  ! i  := lambda (ii);
  END; (* example4 *)
```

```
PROCESS example5;
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; (* example5 *)
```

□

5.12 madd 5.12

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

- addition, but with no overflow exception.

□

5.13 moveb 5.13

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

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

  WARNING: The move is performed with no check at all!!! - and only within the same 64K byte memory module (wrap around takes place!).

  The whole move is in byte mode.

□

5.14 moveg 5.14

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

- works as 'moveb' except that the move is in word mode, if possible.

□

5.15 msub

5.15

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

- Subtraction, but with no overflow exception.

□

5.16 outwordclr

5.16

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

- As the old one, but with no channel message as parameter.

□

5.17 setinterrupt

5.17

```
PROCEDURE setinterrupt (VAR ch : reference);
```

- interrupts the level controlled by the channel message 'ch'.

□

5.18 timedout

5.18

```
FUNCTION timedout : boolean;
```

- true: The interrupt level has status 'timed out'. The status is cleared and OWN.TIMER initialized to zero.

  false: The interrupt level is not 'timed out'.

□

5.19 uadd

5.19

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

- Addition in the range 0..65535.

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

□

5.20 udiv 5.20

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

- Division in the range 0..65535.

An exception occurs, if b = 0.

□

5.21 ult 5.21

FUNCTION ult (a, b) : boolean;

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

□

5.22 umod 5.22

FUNCTION umod (a, b) : integer;

- Modulo operation in the range 0..65535.

□

5.23 umul 5.23

FUNCTION umul (a, b) : integer;

- Multiplication in the range 0..65535.

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

□

5.24 usub

FUNCTION usub (a, b) : integer;

- Subtraction in the range 0..65535.

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

□

RETURN LETTER

Title: RC3502 REAL TIME PASCAL, Extensions to the Reference Manual

RCSL No.: 52-AA1069

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

42 i 1288

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

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

Affix
postage
here

A/S REGNECENTRALEN
af 1979

Information Department
Lautrupbjerg 1
DK-2750 Ballerup
Denmark