RCSL No: 42-i1542

Edition: November, 1980

Author: Bo Bagger Laursen

Title:

RC3502 - PASCAL80
Reference Manual.

A/S REGNECENTRALEN
af 1979

Keywords:

PASCAL80, RC8000, RC3502, Multiprogramming.

Abstract:

This is a description of the RC3502 implementation of the programming language PASCAL80.

The following are described: the runtime environment of a PASCAL80 process on the RC3502 machine, the predefined routines, and how to use the PASCAL80 compiler.

(76 printed pages)

Printed by A/S Regnecentralen af 1979, Copenhagen

42-i 1341

TABLE OF CONTENTS PAGE

TABLE OF CONTENTS (continued) PAGE

APPENDICES:

1. INTRODUCTION

This manual describes the RC3502 implementation of the programming language PASCAL80.

The manual is structured in the following way:

Chapter 2 describes the standard processes comprising the run time environment for RC3502-PASCAL80 processes.

Chapter 3 contains the differences and limitations of the RC3502 PASCAL80 language as defined in the PASCAL80 Report [2].

Chapter 4 describes the predefined types, constants, and variables used in the RC3502 implementation.

Chapter 5 describes all the predefined routines inclusive all input/output routines.

Chapter 6 describes the representation of objects in storage at run time.

The appendices B and C describes the use of the RC3502-PASCAL80 compiler, and the error messages from the compiler.

Appendix D describes how a load file is generated on RC8000.

Appendix E contains a complete list of language symbols.

2. THE RC3502 MACHINE 2.

2.1 Run Time Environment 2.1

After autoload and system initialization, the incarnation structure is:

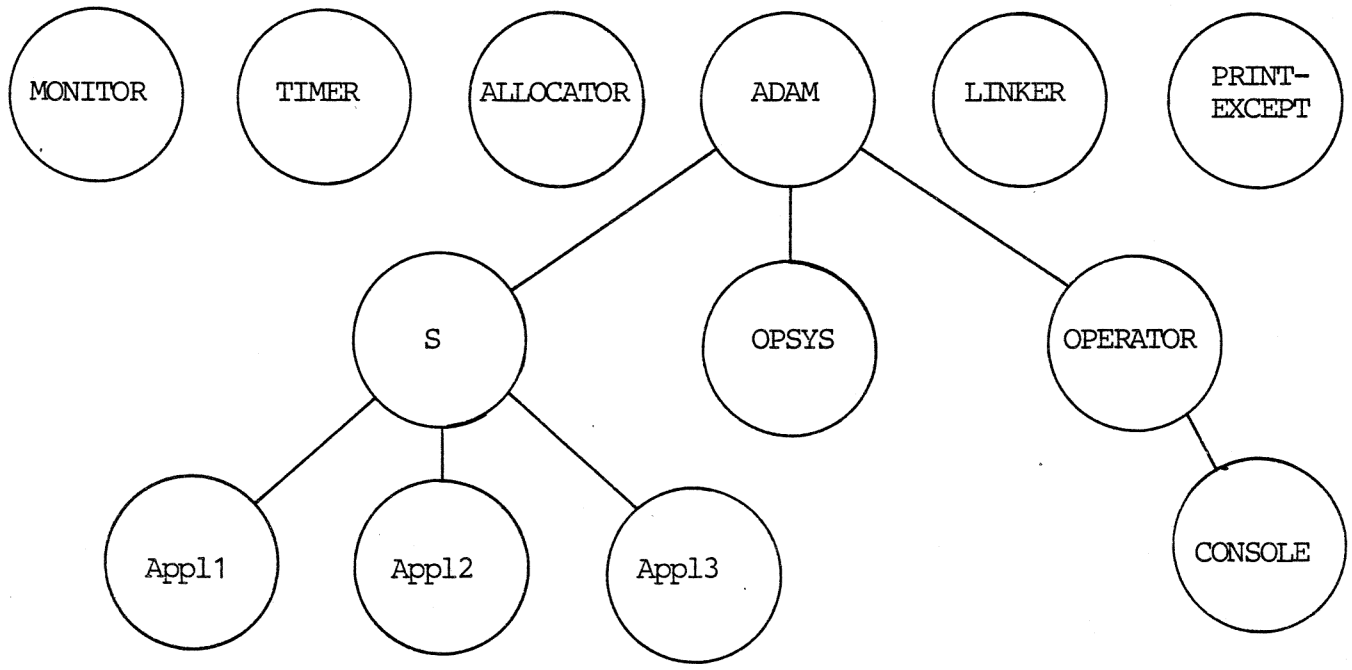


Figure 1: Incarnation structure after initialization.

MONITOR — controls short term scheduling (time slicing) and performs medium term scheduling (START, STOP).

TIMER — performs delay timing and time out of drivers.

ALLOCATOR — administers allocation and deallocation of RAM memory and I/O channels.

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

- OPERATOR
- 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 I/O to the debug console.

OPERATOR processes messages signalled to the operator semaphore.

OPSYS — is a command interpreter functioning as an interface between a human operator and ADAM.

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

S — is the root of all application incarnations. 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 OPERATOR with its own NEW_OPERATOR, and OPSYS by its own NEW_OPSYS.

2.2 Monitor Process 2.2

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 interrupt level greater than zero.

Class II and III incarnations are running on interrupt level 0. The incarnations in these classes are organized in active queues.

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 roundrobin algorithm with built-in priority.

The monitor is activated
- by the expiration of a time slice
- when incarnations call the routines BREAK, REMOVE, START, or STOP.

2.3 Driver Processes 2.3

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

CHANNEL <reference variable> DO <statements>;

construction or the standard input/output routines.

The access to all 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 input/output channel the incarnation wants to control.

The system guarantees that at most one channel message is allocated per input/output channel.

When an incarnation executes a CHANNEL statement or calls an input/output routine, 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 this is also checked.

Before execution of the first statement in the CHANNEL construction, the incarnation has entered the class I priority class.

The statements in the CHANNEL construction are executed on the hardware priority level specified by the channel message.

After execution of the last statement in the CHANNEL construction, the incarnation reenters the priority class (II or III) which the incarnation left, when entering the CHANNEL construction.

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

This is true for all statements except those input/output routines, which clear the interrupt level, and thereby allow incarnations with less priority to execute instructions.

Therefore, the following recommendations should be followed:

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

- the statements in a CHANNEL construction 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 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 the input/output routines.

The channel messages may differ. This may be used to sense an input/output channel on another level than the level defined by the CHANNEL statement, or even outside a CHANNEL construction.

2.3.1 Time Out

2.3.1

Time Out of class I incarnations is performed by the TIMER Process which decrements once per second the standard variable OWN.TIMER in all class I incarnations.

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

2.4 Timer Process

2.4

The Timer Process

- returns messages after a specified interval (Delay Timing)
- controls time out of incarnations running on interrupt levels greater than zero.

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. Time out will only happen when running as a class I incarnation.

2.5 ALLOCATOR Process 2.5

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 are allocated memory, when a newborn process incarnation is started.

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

ALLOCATOR also controls access to all I/O channels. This is done by messages of kind CHANNELMESSAGE.

An I/O channel is allocated by calling the routine RESERVECH specifying

DEV - device number
MASK - facility mask.

A channel message is released by the statement

```
RELEASE (channel_message);
```

2.6 LINKER Process 2.6

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

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

2.7 ADAM Process 2.7

Immediately after upstart of the system the incarnation tree is:

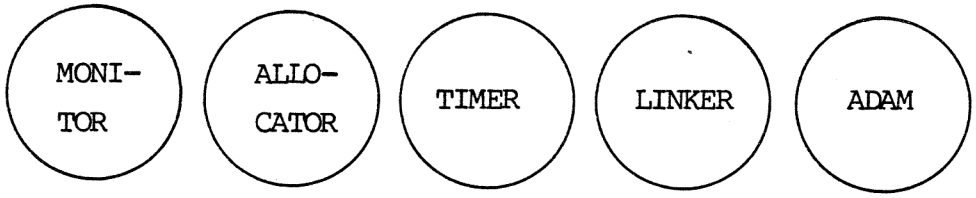


Figure 2: Snapshot of incarnation structure.
(This figure is not complete).

ADAM is the root of the dynamic tree of incarnations. ADAM creates and starts three incarnations:

1) OPERATOR, which performs input/output to the control microprocessor console, and processes messages from running incarnations.

2) OPSYS, which interprets commands from the console and converts the commands to ADAM control messages.

3) S is the root of all applications.

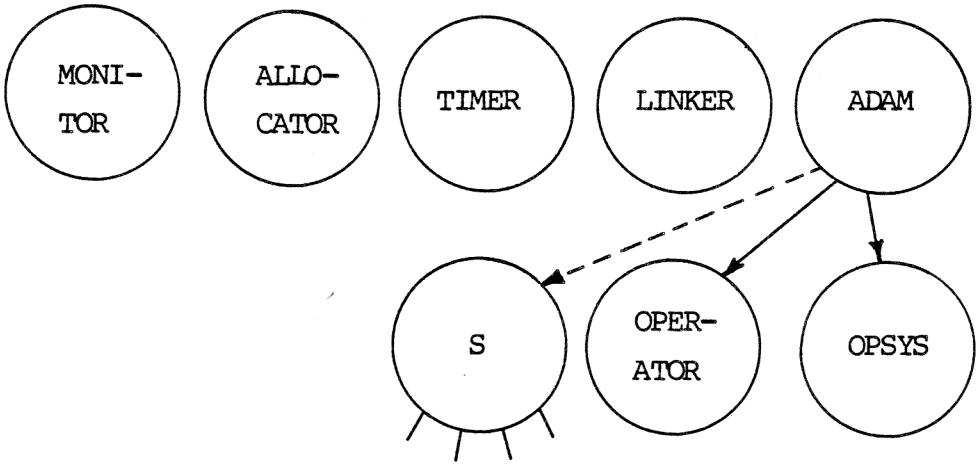


Figure 3: Snapshot of incarnation structure if S is included.

S is declared as an external process in ADAM.

The declaration of S in ADAM is:

```
PROCESS S(VAR sem_vector: system_vector);
EXTERNAL;
```

S is application dependent, and the formal parameters of the actual S must obey this declaration.

The parameter sem_vector is used to pass references to system semaphores like:

```
adam semaphore        [sem_vector(adamsem)↑]
allocator semaphore   [sem_vector(allocatorsem)↑]
operator semaphore    [sem_vector(operatorsem)↑]
```

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 complete different application tree.

2) The application tree may stop, remove, and unlink OPERATOR and OPSYS. A NEW_OPERATOR may be created and started as a child of S, e.g. to implement remote operator communication.

   NEW_OPERATOR must wait for the operator semaphore (at least all messages signalled to the operator semaphore must be released).

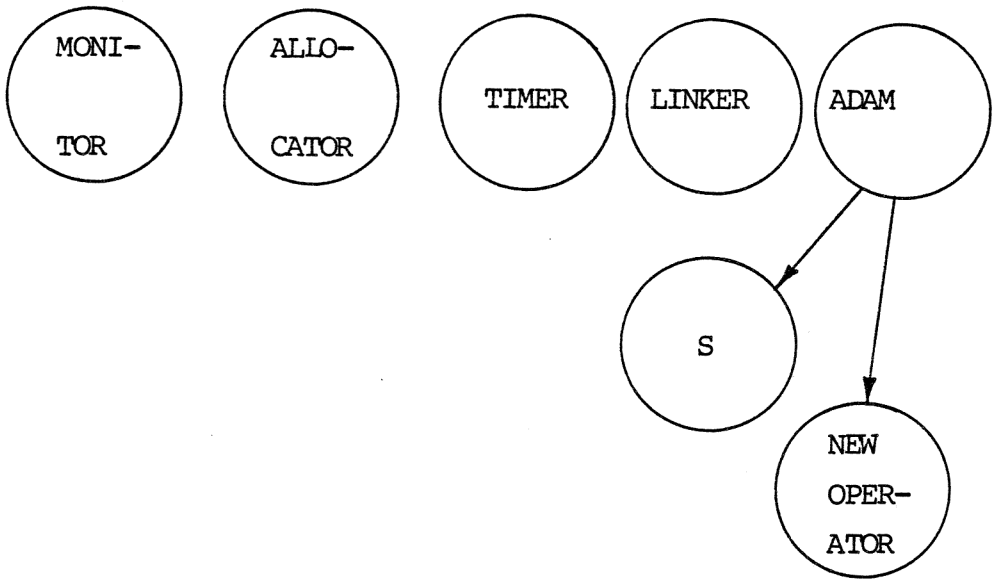


Figure 4: Incarnation structure when OPERATOR is replaced by NEW_OPERATOR.

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

```
adamtype=RECORD
            name1: alfa;
            name2: alfa;
            aux1 : integer
          END;
```

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 sem_vector: system_vector);
EXTERNAL
```

Result	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 does not match.

function=2 (CREATE)

An incarnation with name 'name2' of the process 'name1' is created with size 'aux1'. Only one incarnation per process can be created.

Results	Meaning
0	ok
6	an incarnation of process 'name1' is already created
7	no process with external name 'name1' is linked to a process declaration in ADAM
8	no storage or demanded size (aux1) is too small

function=3 (START)

The incarnation with name 'name2' is started with priority 'aux1'.

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)

The incarnation with name 'name2' is breaked with code aux1.

Results	Meaning
0	ok
14	unknown incarnation

2.8 OPERATOR Process 2.8

Messages signalled to the OPERATOR semaphore [sem_vector (operatorsem)↑] are interpreted as a data message of type

```
Buffertype = RECORD
                first: integer;
                last : integer;
                next : integer;
                name : alfa; (* 12 chars *)
                databuf: array (18..97) of char
             END;
```

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

```
    6 + alfalength <= first
and
    first <= last
and
    last < 6 + alfalength + 80
```

Messages to OPERATOR

read: u1 = 1

The message is queued up 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.

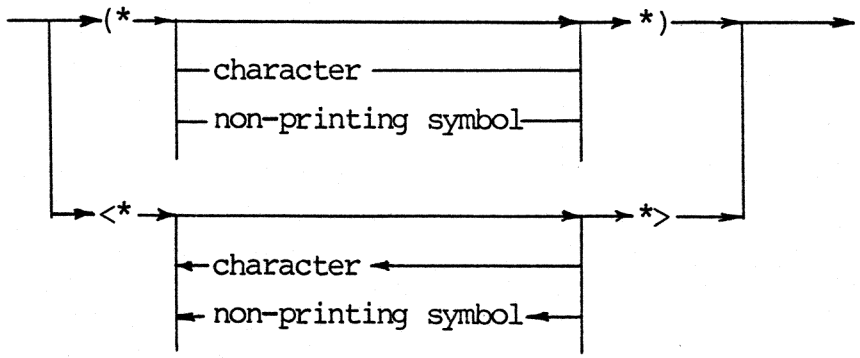
u1, u3, u4 are unchanged
u2 = result: 0 = ok
1 = not processed
2 = timeout
3 = perm error
4 = illegal message
5 = attention

"next" is undefined, unless result = ok

3. RC3502-PASCAL80 IMPLEMENTATION DETAILS AND LANGUAGE MODIFICATIONS 3.

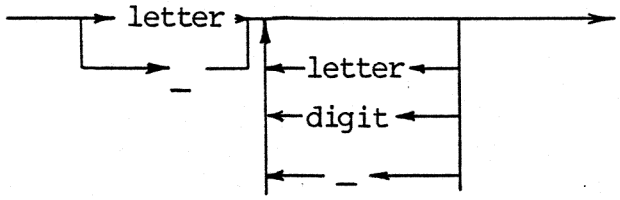
The following is a list of modifications and details as compared to the PASCAL80 Report of this first version of the RC3502-PASCAL80 implementation. The reference is done by page and line number in the PASCAL80 Report [2].

Page 2 Comment:

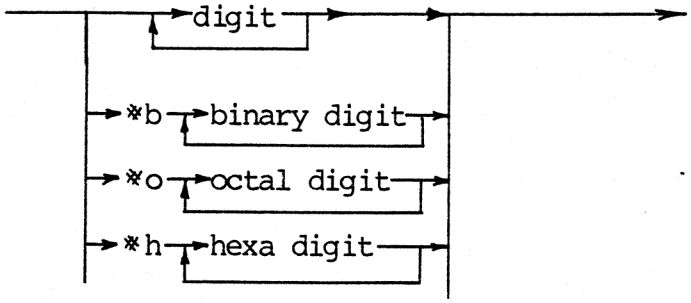


Page 3 line 2 from the bottom: "all" is replaced by "most".

Page 63 identifier



Page 63 numeric value is implemented as:



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

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:

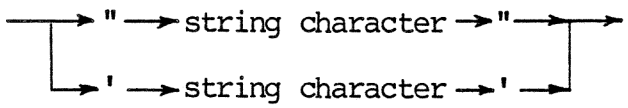
```
alfa = array(1..12) of char;
```

and the routine heading of link is changed to:

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

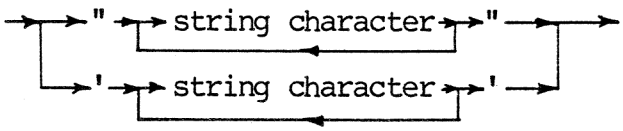
Page 19: The type real is not implemented.

Page 16 Char value:



The string characters are the characters: sp ..

Page 36, page 64 Character string:



The string characters are the characters sp ..

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 60 prefix declaration is implemented as:

prefix declaration:

PREFIX prefix name ; routine declaration

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

Page 19

The bounds min bound and max bound of a subrange type definition must be given by constant expressions.

Page 45, Pool Initialisation

The allocation of memory for messages for a pool variable is performed during the initialisation 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.

Page 51, Process Parameters

Process parameters of reference, pool, or shadow types are not allowed, neither as variables, values, or 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 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.

4. PREDEFINED CONSTANTS, TYPES, AND VARIABLES 4.

The following constants and types are predefined.

```
CONST
alfalength = 12;
stdpriority = -3;
maxint = 32767;
minint = -32768;
TYPE

bit = 0..1;
byte = 0..255;

alfa = ARRAY ( 1..alfalength ) OF char;

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

adamvector = ARRAY(adamsemtype) OF   semaphore;
system_vector = !adamvector;

incarnation_descriptor =
          RECORD
             :
            timer : integer;
             :
            incname: alfa;
             :
          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 refered 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	data message
> 0	> 0	false	> 1	≥ 1	
> 0	0	false	> 1	≥ 1	
0	0	true	1	0	header message
0	0	false	> 1	0	
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.

'Facility mask' originates from the call of RESERVECH (see chapter 5).

5. PREDEFINED ROUTINES 5.

The following routines are predefined unless explicitly mentioned.

If there is no functional description of the routine, the reader is requested to consult the PASCAL80 Report [2].

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, OUTBYTEBLOCK, OUTWORDBLOCK interprete the actual datamessages as being of type

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

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

```
FUNCTION alloc(VAR r: reference; VAR p: pool 100;
               VAR s: semaphore);
```

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

- stops the child and starts it in the exception procedure (see EXCEPTION)

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

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

- this 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 procedure must be declared in the declaration part of the process.
  The type of control_word may be any type of size 16 bits.
  An exception occurs if chmsg does not refer to a channel message.

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

- 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 cleared, so the next statement is executed when an interrupt arrives from the device.
  The procedure must be declared in the declaration part of the process.
  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

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

- a new incarnation of the process linked to processname 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:

result	meaning
0	call ok, incarnation creation
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.

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

```
FUNCTION eoi: boolean;
```

- 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.

```
PROCEDURE exception (excode: integer);
```

- if the user has not declared an exception procedure, a standard exception procedure will be called, when an exception occurs. The procedure may also be called as a normal procedure.

The standard exception procedure produces output with the format:

```
process name >> exception, excode=code: error text
gf= ..... , lf= .....
called from: ..... , ic= ...., line ...-... , date
   .
   .
   .
```

- The list of "called from ..." is the dynamic chain of routine activations.
- "gf" and "lf" are stack references
- "code" and "errortext" are the actual exception code and the meaning of the code, some of the texts include information about the operands which caused the exception.
- "ic" and "line ...-..." is an identification of the calls
- "date" is the compilation date of the modules in question.

```
PROCEDURE getbufparam
  (VAR i: RECORD
            top, count: integer;
            saddr:   datastart;
          END;
      first, last: integer;
   VAR msg: reference);
```

- returns the start address (saddr) of the byte with index first in the data buffer referenced by msg. As a sideeffect

```
count:= last+1-first
top:= 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

- the reference variable is nil
- the message is no data message
- size of message is too small
- last < first

The procedure must be declared in the declaration part of the process. The type of i may be any type of size 8 bytes.

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

- inputs a block of bytes to the databuffer specified by msg, first, and last from the device specified by the channel message chmsg. 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=last+1;
  - when eoi=true

  If nothing is input next=first.
  The following exceptions may occur
  - the reference variables chmsg is nil
  - chmsg is not a channel message
  - the reference variable msg is nil
  - the message msg is no data message
  - size of msg is too small
  - last < first

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

- 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 procedure must be declared in the declaration part of the process.
  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

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

- inputs a block of words to the data buffer specified by msg, first, and last from the device specified by the channel message chmsg.
  The procedure terminates in two situations
  - when next=last+1
  - when eoi=true

  If nothing is input next=first.
  The first word input will be the word indexed by first even if first is odd (note all indices are byte indices!).
  The word indexed by last will only be input if first is even and last is odd.
  The following exceptions may occur
  - the reference variable chmsg is nil
  - chmsg is not a channel message
  - the reference variable msg is nil
  - the message msg is the data message
  - size of msg is too small
  - last < first

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

- The process identified by external_name is looked up in the LINKER catalog.
  If found the process identified by external_name is linked to process name.

  The function returns the following results:

result	meaning
0	process linked
1	process with name '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 not match
6	a process is already linked to process name.

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

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

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

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

- returns the value true if the pool is not empty, false otherwise.

```
FUNCTION ord (x: niltype): integer;
```

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

- outputs a block of bytes from the databuffer specified by msg, first, and last to the device specified by the channel message chmsg. When the procedure terminates next will be the index of the byte following the last byte output. The procedure will terminate in two situations
  - when next=last+1
  - when eoi=true

If nothing is output next=first.
The following exceptions may occur

- the reference variable chmsg is nil
- chmsg is not a channel message
- the reference variable msg is nil
- the message msg is no data message
- size of msg is too small
- last < first.

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

- The contents of the parameter word are transferred to the DATAOUT 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 procedure must be declared in the declaration part of the process.
  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

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

- outputs a block of words from the data buffer specified by msg, first, and last to the device specified by the channel message chmsg.
  The procedure terminates in two situations
  - when next=last+1
  - when eoi=true

  If nothing is output next=first.
  The first word output will be the word indexed by first even if first is odd (note all indices are byte indices!).

The word indexed by last will only be output if
first is even and last is odd.
The following exceptions may occur
- the reference variable chmsg is nil
- chmsg is not a channel message
- the reference variable is nil
- the message msg is no data message
- size of msg is too small
- last < first.

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

- the contents of the parameter word are transferred to the DATAOUT register in the device selected by the channel message chmsg. The current interrupt level is cleared, so the next statement is executed when an interrupt arrives from the device.
The procedure must be declared in the declaration part of the process.
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.

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

- 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

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

PROCEDURE pop (VAR r1, r2: reference);

- 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 becomes nil after the call if empty (r2)=true before the call.

FUNCTION pred (x: niltype): niltype;

PROCEDURE push (VAR r1, r2: reference);

- 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 data of r2 remains the same. After the call r1 is nil.

  The parameter r1 must refer to a message (must not be nil), and this message must have exactly one header, otherwise an exception occurs. The message accessible through r2 (possibly nil) is called the stack.

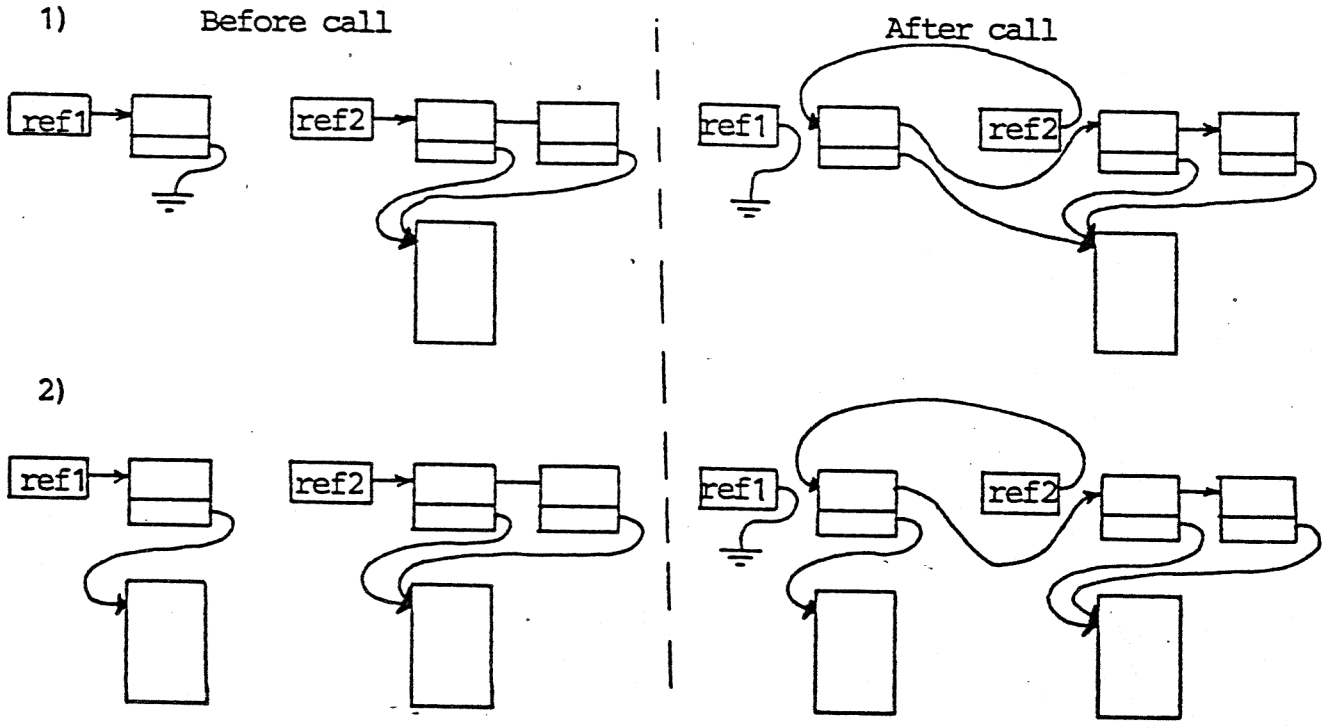


Figure 5: Example on the behavior of push.

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

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

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

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

- allocates the channel message to the I/O channel specified by the parameter channel. The parameter mask is extended for specification of the actions the user wants to perform on the channel. The parameter is not used in this revision.

result	meaning
0	reservation ok. chmsg refers to the allocated channel message.
1	the channel is already reserved
2	the reference variable chmsg is not nil before call.

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

```
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	resultcode=1
u3	delay 1	undefined
u4	delay 2	undefined

The message is returned after delay1*2↑delay2 msec.

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

- the contents in 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 interrupt level is not cleared so the next statement is executed without waiting for interrupt from the device.
  The procedure must be declared in the declaration part of the process.

The type of status_in 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

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

- takes a message from the semaphore, otherwise r remains nil.
  The caller will not be waiting.
  An exception occurs if the reference parameter is not nil.

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

- The reference parameter must refer to a message (must not be nil), otherwise an exception occurs. The reference variable is nil after a call of signal.

  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 a semaphore is locked, the first incarnation waiting on the semaphore completes its wait call.

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

- 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.

  The monitor activates the child by placing it in an active queue according to <priority>. If <priority> $\geq$ 0, the child is placed in the coroutine class (Class II). If <priority> < 0, the child is placed in 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 instruction counter for the child was decremented when stopped, so that the WAIT statement will be repeated.

If the child was stopped at an interrupt level greater than zero, the child is scheduled directly to the old interrupt level and activated by a TIMEOUT interrupt.

An exception occurs if the shadow variable is nil.

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

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

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

The child is removed from the active queue or semaphore where the child is placed.

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

If the shadow variable is nil, an exception occurs.

```
FUNCTION succ (x: niltype): niltype;
```

```
FUNCTION unlink (process name): integer;
```

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

result	meaning
0	process unlinked successfully.
1	no process was linked to processname.
2	incarnations of the process are existing.

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

- The reference parameter must be nil, otherwise an exception occurs. After a call of wait it refers to a message.

  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 can be resumed by another incarnation calling signal or return.

6. REPRESENTATION AND LAYOUT OF VARIABLES 6.

In this chapter it is explained how values of the various types of PASCAL80 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.

6.1 Representation of Values 6.1

6.1.1 Enumeration Types 6.1.1

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 the largest of these then the representation is in $\log_2(n+1)$ 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 [2]) 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.

6.1.2 Shielded and Pointer Types 6.1.2

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

6.1.3 Structured Types

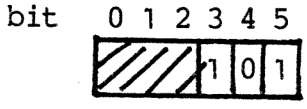
Values of array or record types are vectors of values of enumeration, shielded, pointer, and set types. The component values are represented as described for the component types.

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 not significant. 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 not significant.

6.1.4 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.

6.2 Memory Layout 6.2

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:

$1 \leq S(T) \leq 8$: M(T)=1
$9 \leq S(T) \leq 16$: M(T)=2

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

M(reference)= 8
M(semaphore)= 8
M(shadow)= 12
M(pointer)= 4
M(pool)= 8

Memory requirement for structured types is described in connection with memory layout for these types in subsections 6.2.3 and 6.2.4.

6.2.1 Word Alignment 6.2.1

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

Memory allocation for a variable of a shielded or pointer type, or of a structured type containing components of shielded or pointer type(s) is word-aligned, i.e the allocated memory starts on a word boundary.

6.2.2 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. 6.

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 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. 7.

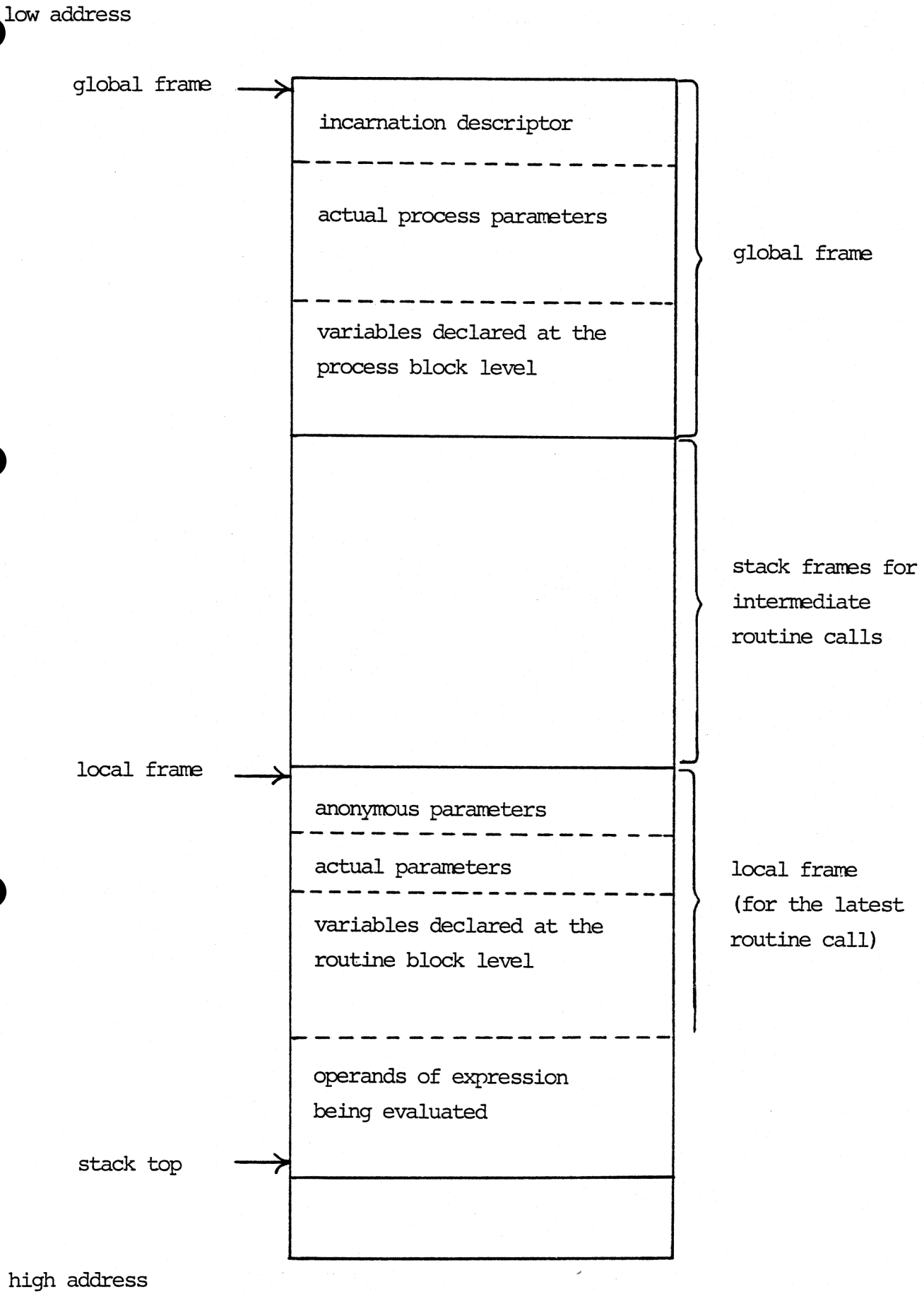


Figure 6: Incarnation stack layout (snapshot).

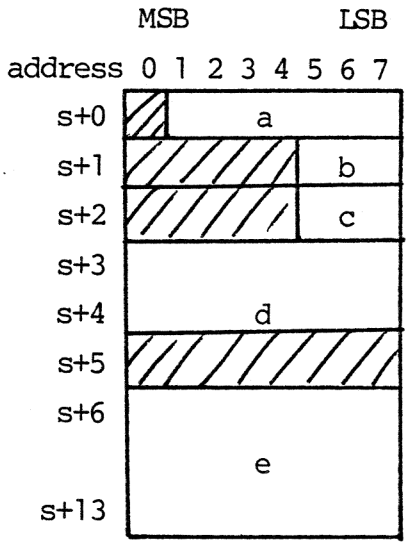


Figure 7: Memory layout for simple declared variables in stack frame or record type (not packed).

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. Similarly 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.

6.2.3 Structured Types

6.2.3

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

6.2.3.1 Arrays and Records, Not Packed 6.2.3.1

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,
- word-alignment (relative to the beginning of the array or record and by implication also in absolute memory) as described in subsection 6.2.1,
- 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. 7 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)=14.

6.2.3.2 Packed Arrays and Records 6.2.3.2

In the memory allocated for a variable of a packed array or record type several consecutive components may be packed into a single 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 word, and each

component is allocated as many bits as indicated by its size. When it is not possible to fit any more components without crossing a word boundary packing stops and allocation is resumed from that word boundary. By this rule unused space may be left in the least significant bits of a word in which one or more components are packed.

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 2 (bytes).

Example: The layout of variables of the following packed record type is shown in fig. 8.

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

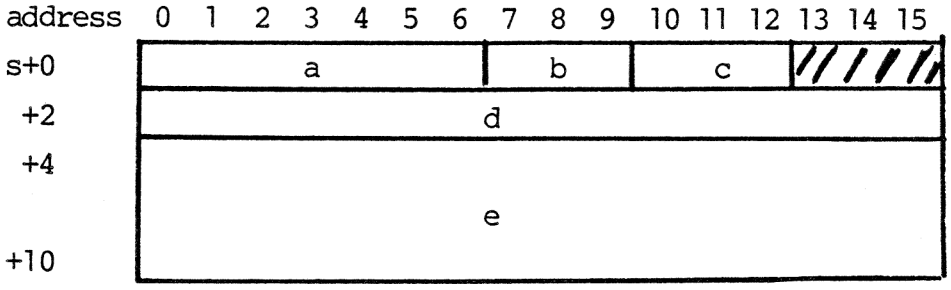


Figure 8: Memory layout for packed record. M(q)=12.

The memory requirement of a set type is always an even number (of bytes), i.e. an integral number of words is 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 6.1.3. The bit vector is laid out with index 0 in the most significant bit of the first word. The last word may contain an unused portion in its least significant bit positions. Example: see fig. 9.

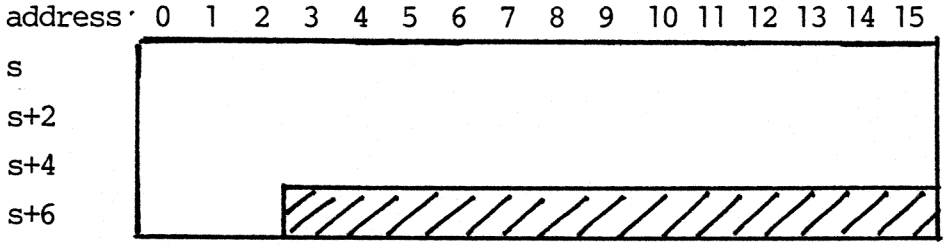


Figure 9: Layout of a variable of the type set of 0..50; s is even, and 13 bits are unused. The memory requirement of the type is 8.

A. REFERENCES

[1] RCSL No 42-i1577: PASCAL80, Introduction

[2] RCSL No 52-AA964: PASCAL80, Report

[3] RCSL No 42-i1539: PASCAL80, User's Guide

[4] RCSL No 31-D617: PASCAL80, Driver Conventions

[5] RCSL No 52-AA1000: RC3502-PASCAL80, Generating Guide

[6] RCSL No 52-AA996: RC3502-PASCAL80, Installation Guide

[7] RCSL No 31-D627: RC3502, Introduction

[8] RCSL No 52-AA1016: RC3502, Operating Guide

[9] RCSL No 52-AA972: RC3502, Reference Manual

B. USE OF THE PASCAL80 COMPILER B.

B.1 Call of the Compiler B.1

$$\left\{\begin{array}{l}\text{<bin file>=}\end{array}\right\}_{0}^{1}\ \text{pascal80}\left\{\left\{\begin{array}{l}\text{<s><option>}\end{array}\right\}\left\{\begin{array}{l}\text{<s><context>}\end{array}\right\}\right\}_{0}^{\infty}$$

$$\left\{\begin{array}{l}\text{<s><option>}\end{array}\right\}_{0}^{\infty}\left\{\begin{array}{l}\text{<s><source>}\end{array}\right\}_{0}^{1}$$

- <source> is a text file defining a process or a prefix. 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 ends.

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

- <option>::= codesize.<size>
  | list.<yes or no>
  | stop.<pass nr>
  | spacing.<interval>

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

- <yes or no>::= yes | no
  list.yes means turn on listing of input (on current output). The list option is superfluous since the utility programs indent and cross may produce more readable listings (see the description of indent and cross).

- <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:

```
pascal80 inputfile
```

is equivalent to:

```
pass6code =pascal80 list.no codesize.512 spacing.52 stop.6
inputfile
```

Resource requirements

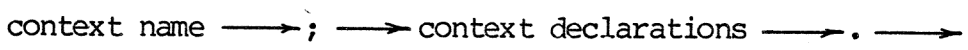
at least 90000 hW memory (size 90000)
area 8
temp disc at least 350 segm. and 8 entries

B.2 Use of the Contexts

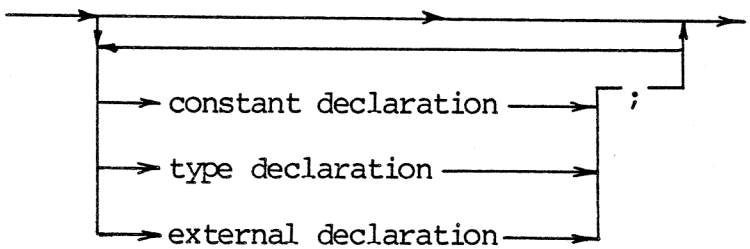
B.2

Specification of more input files to the PASCAL80 compiler is used for inclusion of (common) contexts (environments), i.e. constant and type definitions and declaration of external routines. The syntax is:

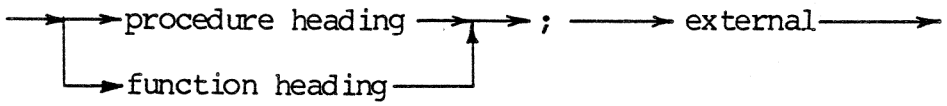
context:



context declarations:



external routine declaration:



C. PASCAL80 ERROR MESSAGES C.

C.1 Messages from Pass 1 C.1

0 Illegal character or fatal error
1 Context expected
2 Idenfier expected
3 ';' expected
4 Identifier expected
5 ',' or ':' expected
6 Error in declaration
7 Set element or '.)' expected
8 Constant, variable, or '(<expression>)' expected
9 Expression expected
10 Actual parameter expected
11 Expression expected
12 'of' expected
13 '(' expected
14 Identifier or '?' expected
15 ')' expected
16 Only 'array', 'record', and 'set' can be packed structure
17 '..' expected
18 ')' or ',' expected
19 ')' or ',' expected in array declaration
20 'end' or ';' expected
21 ',' or '.)' expected
22 ';' or ')' expected
23 ':' expected
24 Unsigned integer expected
25 ':' expected
26 'begin' expected
27 Error in for-variable specification
28 'to' or 'downto' expected
29 'of' expected
30 Error in channel-variable specification
31 'do' or ',' expected
32 'do' expected
33 'then' expected
34 Label expected (name or integer)

35 ',' or ':' expected
36 'else' expected
37 ';', 'end' or 'otherwise' expected
38 'until' expected
39 Label definition expected (integer or name)
40 Error in label list (',' or ';' expected)
41 '=' expected
42 Environment specification expected
43 'process' or 'include' expected
44 '.' expected
45 'process' or 'prefix' expected
46 Only procedure or function declaration allowed in a prefix
47 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

The following messages indicating fatal errors may appear from pass1 of the PASCAL80 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 small syntactical construction.

***end of file encountered
Input exhausted before the process/prefix has been satisfied.

***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 'maxnamenodeindex' too small
Too many very long names.

***const 'maxnameheads' too small
Too many different names.

***const 'typebuffersize' too small
Too complicated type definition.

C.2 Messages from Pass 3 C.2

Error messages from pass 3 have the format:

***pass 3 line <lineno>, <operand no> <text>

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).

```
undeclared,
(*identifier not declared*)

inconsistent_use,
(*identifier used before declaration*)

double_declaration,
(*identifier already declared at this level*)

label_not_declared,
(*label-identification not declared at all*)

not_label_name,
(*other identifier used as label-name*)

multiple_defined_label,
(*label defined several times at this level*)

label_not_locally_declared,
(*label-identifier declared at surrounding level*)

erroneous_label,
(*use of a multiple defined label*)

label_used_from_inner,
(*a label-ident has been used in inner routine*)

label_used_outside_scope,
(*goto leading into control-structure*)

label_defined_outside_lock_or_channel,
(*goto out of lock- or channel statements*)
```

```
not_typename,
(*identifier is not a type-identifier*)

recursive_use_of_type,
(*error in record or array etc.*)

recursive_constant_use,
(*constant is used in its own definition-expression*)

illegal_pool_type,
(*pool ... of <illegal type>*)

pool_cardinality_must_be_integer,
(*illegal size'ing of pool type*)

subrange_elems_must_be_enumeration,
(*illegal limit-types in subrange def*)

type_may_only_be_used_at_process_level,
(*nb: semaphore, pool*)

process_only_allowed_at_processlevel,
(*processes inside functions/procedures forbidden*)

external_only_at_processlevel,
(*restriction on 'external'*)

illegal_formal_type,
(*formal type may not be used in this context*)

illegal_function_type,
(*functiontype may not contain: semaphore etc.*)

paramlist_changed_since_forwarddecl,
(*'new' paramlist may be empty or exact the same*)

forward_not_solved,
(*forward-declared routine not followed with the real body*)
```

```
funcval_not_used,
(*function-value has not been defined at all*)

type_has_pointers,
(*locktype contains pointer-types*)

type_has_systemtypes,
(*locktype contains semaphore, reference, shadow, pool*)

operands_incompatible,
(*operands not of same typename*)

for_incompatible,
(*for-variable/startvalue/endvalue not of compatible types*)

case_incompatible,
(*case-expression/caselabels not of compatible types*)

if_type,
(*if-expression must be boolean type*)

repeat_type,
(*until-expression must be boolean type*)

while_type,
(*while-expression must be boolean type*)

with_type,
(*with-variable must be a record*)

lock_type,
(*lock-variable must be reference type*)

channel_type,
(*channel-variable must be reference type*)

not_index_type,
(*type of operand must be enumeration-type*)
```

```
not_variable,
(*operand cannot be used as variable*)

field_must_follow_recordtype,
(*<variable> in front of <.> is not a record*)

name_not_fieldname,
(*<name> after <.> is not a fieldname of <variable>*)

must_be_pointertype_before_uparrow,
(*<variable> in front of <uparrow> is not a pointer*)

mixed_type_in_setlist,
(*elements in set-value may not be mixed*)

relation_error,
(*illegal mixture of types in relation*)

arithmetic_error,
(*illegal mixture of types in term of factor*)

monadic_error,
(*illegal type for monadic operator*)

real_not_implemented,
(*real occuring in expression*)

real_expression_not_implemented,
(*real-division of integers not impl*)

illegal_in_expr,
(*illegal operand kind in expression*)

too_few_parameters,
(*too few actual parameters to routinecall (or strucrecord*)

too_many_actual_params,
(*errors in routinecall*)
```

```
too_many_values_in_record_structure,
(*error in structured-record constant*)

type_must_be_record_or_array,
(*typename in front of arglist must be ...*)

double_param_only_in_struc_const,
(*the '***' operator must only occur in structured-array constant*)

subscript_after_nonarray,
(*name in front of arglist is not of array-type*)

incompatible_index,
(*index-expression does not match array-declaration*)

assign_incompatible,
(*incompatible types in assignment*)

exchange_incompatible,
(*incompatible types in exchange*)

not_procedure_call,
(*the statement is not a procedure-call*)

variable_may_not_be_packed,
(*for-variable or actual var-param is packed*)

not_assignable,
(*operand may not be assigned: sem, pool, ref, shadow, frozen*)

not_exchangeable,
(*operand may not be exchanged: sem, pool, frozen*)

exchange_type,
(*type must be: reference or shadow*)

Illegal_var_param_substitution,
(*formal and actual type must match exactly*)
```

```
illegal_value_param_substitution,
(*actual and formal types are not compatible*)

actual_may_not_be_frozen,
(*formal is not frozen, therefore ...*)

skipparam_only_in_struc_const,
(*the '?' may only occur in structured constants*)

struc_arr_impossible,
(*incomp. types in structured array-constant*)

struc_rec_incompatible,
(*incomp. types in structured record-constant*)

var_init_incompatible,
(*incomp. types in var-initialization*)

repetition_type,
(*repetition must be integer*)
```

C.3 Messages from Pass 4 C.3

All error messages from pass 4 have the format:

xxx 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 zero or negative.
record size	Record type > 65536 bytes.
array size	Array type > 65536 bytes.
no init in environment	Initializing of variables 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 in a block occupies more than 65536 bytes.
overflow	Value of constant or constant expression outside the interval - 32768, 32767.
compiler error	Error in pass 4.

Compilation terminated by <error>

where <error> is

operand overflow	
nametable overflow	too many names in program
block level overflow	too many block levels
constant area overflow	too many or too big structured constants

C.4 Messages from Pass 5 C.5

I:

- text:

***compilation terminated after pass5

- meaning:

The compiler has stopped after (or inside) pass5 according to a "fatal error" detected in pass5 (see below).

II:

- text:

******message from pass5: <kind> at: <place>
current token/param. is: <token/param>
no. of items input: <no. of items>
line no.: <linie nummer>

- meaning:

<kind>:	meaning:
'not implemented'	The source program uses a ficility not yet implemented.
'fatal error'	A fatal error is detected in pass5. The cause of the error is described by <place>.
'warning'	Warning, pass5 continues, but the object code may by wrong.

<place>:	The cause of the error:
2305	'case labels' are not unambiguous (see RCSL No 52-AA964: PASCAL80 Report, page 14 on top [2]).
1201	too many external processes/routines (table full).
1202	too many parameters to an external process/routines (table full).
1203	too many calls of external routines (table full).
1207	inconsistency between formal parameter specification of an 'external' process/routine declaration and the object program (in a library) of the same name.
anything else	Please report to the software responsible. The reason might be a compiler error, or maybe some tablesizes are too small.

<token/param>;<no of items> is used in error detection in pass 5 (see above: <place> = 'something else').

<linie nummer> points out the number of the line in the source program in which the error was detected.

C.5 Messages from Pass 6 C.5

- Constant <name> too small

  If <name> is "maxnameix" the trouble may be avoided by increasing the codesize (option to the call of PASCAL80).

  If <name> is "max_jump_ix" the trouble may be avoided by decreasing the codesize (option to the call of PASCAL80).

- Compiler error detected

  Please inform the compiler group.

- Error in compilation

  Use option codesize, with at least <number> as page size.

D. LOAD FILE GENERATION ON RC8000 D.

D.1 CROSS-Linker D.1

The program will generate a file containing all the specified object modules and the necessary library routines. The file will be a coreimage, i.e. references between the modules are solved.

Call:

$$\text{<outfile> = crosslink} \quad \left\{ \begin{array}{l} \text{<obj file>} \\ \text{<lib file>} \\ \text{<params>} \end{array} \right\}_{0}^{\infty}$$

<outfile> contains the generated coreimage.

<obj file>::= <file name>
<file name> contains one or more object modules which (all) must be included in the coreimage.

<lib file>::= lib. <file name>
<file name> contains one or more object modules which may be included by CROSSLINK, if they are referenced from an included object module.

<obj file> and <lib file> can be output file(s) from the PASCAL80 compiler.

$$\text{<params>::=} \left\{ \begin{array}{l} \text{start.<base>.<displacement>} \\ \text{descr.} \left\{ \begin{array}{l} \text{yes} \\ \text{no} \end{array} \right\} \\ \text{map.} \quad \left\{ \begin{array}{l} \text{yes} \\ \text{no} \end{array} \right\} \\ \text{print.} \left\{ \begin{array}{l} \text{yes} \\ \text{no} \end{array} \right. \quad \left\{ \text{.<words per line>} \right\}_{0} \end{array} \right\}^{1}$$

start

<base> and <displacement> specify where the first word of the coreimage is supposed to be loaded. The start-param may only occur before the first filename.

<base>, <displ> ::= <integer> | h <hexadecimal digits>

Default is: start.0.0

Note: If the coreimage is to be autoloaded by the BOOT program, start must be: start.0.256

descr

defines whether the descriptor segments of the following modules are included in the coreimage or not.

Note: This option should not be used! (Only intended for special program generation).

Default: descr.yes

map

controls listing of the included modules. Each included module is listed with:

<modulename> <start of descriptor segment> <start of code segment>

Default: map.no

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 module).

Example

```
coreimage=crosslink map.yes start.0.h0100 descr.yes,
bmonitor,
blinker,
ballocator,
bprintexcep (or bminiexcept),
btimer,
badam,
boperator,          } may be substituted by your own operating system
bopsys,
bconsole,
, userprocess 1 ... userprocess n
lib.stdlib,
lib.debuglib
```

will generate a coreimage in a file with the name COREIMAGE, containing (all) the object program(s) in the files explicitly mentioned, extended with the necessary object programs from the library files STDLIB, and DEBUGLIB which contains all standard runtime routines (link, create, ...).

The coreimage will start in memory module 0, displacement 256, as demanded by the BOOT program.

A supplementary loadermap is printed.

Errormessages from CROSSLINK

Errormessages are printed like:

{ fatal error / warning } at <cause>:<value>

or

***<routinename> : not defined

Error causes:

1. Illegal contents of object (or library) file, normally because the file has not been created by PASCAL80.

2. Illegal contents in outfile (error in CROSSLINK).

3. Illegal parameter. Value identifies the parameter.

4. Illegal hexadecimal digit. Value identifies the parameter.

5. Illegal hexadecimal number. Value identifies the parameter.

6. Displacement in memory too large.

7. Illegal base value.

8. Illegal displacement value.

101. No <outfile> specified.

102. No <outfile> specified.

103. No <outfile> specified.

104. No <outfile> specified.

105. Illegal parameter (value is irrelevant).

106. Illegal descr-param

107. Illegal descr-param

108. Illegal descr-param

109. Illegal start-param

110. Illegal start-param

111. Illegal start-param

112. Illegal start-param

113. Start param may only occur before object (or library) files.

114. Illegal start-param

117. Object file contains open routines.

118. Inconsistent descriptor segment.

119. Illegal start-param

7913 As 1.

7914 Illegal print-param

7915 Illegal print-param

7916 Unknown parameter option

7917 Illegal lib-param

7918 Parameter description of external routines does not match.

7919 As 1.

7920 Illegal map-param

7921 Illegal map-param

7922 Inconsistency in loader map

D.2 Use of punch16 to Generate a Load File D.2

Generating coreimages for load from papertape or via FPA.

D.2.1 Generating a Papertape D.2.1

Assume coreimage has been generated by CROSSLINK.

A job like:

```
job xx 7913 device punch
tpn = punch16 mode.8 coreimage
finis
```

will generate a papertape with the correct format for the BOOT-program.

D.2.2 Generating an FPA Bootfile D.2.2

Assume coreimage has been generated by CROSSLINK.

A job like:

```
job xx 7913
  bootfile = punch16 mode.boot coreimage
finis
```

will generate a file with the correct format for the AUTOLOAD-program.

This file may later be used like:

```
main35001 = autoload bootfile
```

E. COMPLETE LIST OF LANGUAGE SYMBOLS E.

AND	ELSE	LABEL	PROCESS
ARRAY	END	LOCK	RECORD
AS	EXPORT **)	MOD	REPEAT
BEGIN	EXTERNAL	NOT	SET
BEGINBODY *)	FOR	OF	THEN
CASE	FORWARD	OR	TO
CHANNEL	FUNCTION	OTHERWISEQ	TYPE
CONST	GOTO	PACKED	UNTIL
DIV	IF	POOL	VAR
DO	IN	PREFIX	WHILE
DOWNTO	INCLUDE	PROCEDURE	WITH

+	-	*	/	"	'	<	>
<>	<=	>=	(	)	(.	.)	
=	:=	:=:	.	,	:	;	..
***	(*	*)	!	?	<*	*>	#

RETURN LETTER

Title: RC3502 - PASCAL80 RCSL No.: 42-i1542

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-i1288

. . . . . 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