RC9000-10/RC8000

SW8010 System Utility

Maintenance Programs

PN 99200389

RC Computer

Keywords:
RC9000-10, RC8000, System Utility, Maintenance Programs

Abstract:
This manual describes a set of programs from the System Utility package, which are used for system maintenance and administrative purposes.

Date:
February 1989

PN: 991 11265

Table of Contents

1. Basemove.....1
1.1 Introduction.....1
1.2 Call.....1
1.3 Function.....1
1.4 Resource Requirements.....2
1.5 Examples.....2

2. Checkio.....3
2.1 Introduction.....3
2.2 Call.....3
2.3 Reserving the Document.....4
2.4 Output.....4

3. Clean.....7
3.1 Introduction.....7
3.2 Example.....7
3.3 Call.....7
3.4 Function.....7
3.5 Error Messages.....7

4. Deletelink.....9
4.1 Introduction.....9
4.1 Examples.....9
4.3 Call.....10
4.4 Function.....10
4.5 Error Messages.....10

5. Disccopy.....13
5.1 Introduction.....13
5.2 Call.....14
5.3 Details.....16
5.3.1 Save.....16
5.3.2 Load.....17
5.3.3 Bin.....18
5.3.4 Packon.....20
5.3.5 Packoff.....20
5.3.6 Kiton.....21
5.3.7 Kitoff.....21
5.3.8 Kitlabel.....22
5.3.9 Kitname.....22
5.4 Error Messages and Warnings.....23
5.4.1 Logical Status.....23

5.4.2 Results from the Monitor....23
5.4.3 Warnings....24

6. Discstat....25
6.1 Introduction....25
6.2 Call....25
6.3 Function....25
6.4 Examples....26
6.5 Error Messages....26

7. Disctell....27
7.1 Introduction....27
7.2 Call....27
7.3 Examples....28

8. Do....31
8.1 Introduction....31
8.2 Call....32
8.3 Function....38
8.4 Storage Requirements....45
8.5 Messages....45
8.6 Examples....45

9. Mainstat....51
9.1 Introduction....51
9.2 Call....51
9.3 Function....51
9.4 Examples....52
9.5 Error Messages....52

10. Makelink....55
10.1 Introduction....55
10.2 Examples....55
10.3 Call....56
10.4 Function....56
10.5 Error Messages....56

11. Montest....61
11.1 Introduction....61
11.2 Examples....62
11.3 Call....64
11.4 Function....64
11.4.1 Commands....64
11.4.2 Info....64
11.4.3 typein....65
11.4.4 End....65
11.4.5 Dump....66
11.4.6 Core....66
11.4.7 Lock....66
11.4.8 Internal....66
11.4.9 External....67
11.4.10 Area....67
11.4.11 Chain....68
11.4.12 Buf....68
11.4.13 Veri....69
11.4.14 Format....69
11.4.15 Extra....69

11.4.16 Lines....70
11.4.17 O....70
11.5 Resource Requirements....71
11.6 Error Messages....71

12. Printzones....73
12.1 Introduction....73
12.2 Call....74
12.3 Function....74
12.4 Examples....74
12.5 Error Messages....75

13. Scatup....77
13.1 Introduction....77
13.2 Creating a new s-user catalog....77
13.3 Inserting entries in s-user catalog....77
13.4 Deleting entries in s-user catalog....78
13.5 Listing entries in s-user catalog....78
13.6 Example....78

14. Slicelist....81
14.1 Slicelist....81
14.2 Example....81
14.3 Syntax....81
14.4 Semantics....81
14.4 Program Output....82
14.5 Error Messages....83

Appendix A. References....85

1. Basemove

1.1 Introduction

The problem of moving one directory in the directory hierarchy of RC9000-10 from one name base to another, or in normal RC9000-10 terms of changing the entry bases of entries in one name base to entry bases in another name base, is normally solved either by

- making a backup of all files concerned by the utility program save and restoring them by the utility program load, all in processes with the proper process bases and done with the proper modifiers, or by

- using the scope program in processes with properly selected process bases.

The present program will do the job in a process with a proper standard base (the maximal standard base will always do).

The program basemove changes the entry base of all catalog entries of a specified name base to another entry base, or the entry base of entries specified by name and base to another entry base.

1.2 Call

```
{outfile =} basemove   ,
{ [name.<name>] from.<low>.<up> to.<low>.<up> }1-*
```

where

```
<name> ::= name of a catalog entry
<low>  ::=
<up>   ::= integer
```

1.3 Function

The program scans the main catalog, and whenever it finds an entry with bases equal to the specified original bases, the bases of the entry are changed to the specified target bases.

A message is displayed for each entry found. The message states the result of the change.

When the entire catalog has been scanned, a message is displayed containing the number of entries found, and - if any - the number of entries which could not be changed for various reasons.

1.4 Resource Requirements

The program needs about 15000 halfwords of memory.

1.5 Examples

Example 1

```
*basemove
***basemove call error
```

Example 2

```
*basemove from.750.800 to.700.740

c       bases moved properly  from.  750  800 to.  700  740
v       bases moved properly  from.  750  800 to.  700  740
area2   bases moved properly  from.  750  800 to.  700  740
area1   bases moved properly  from.  750  800 to.  700  740
  4 entries found and moved ok
```

Example 3

```
*basemove name.area1 from.700.740 to.790.800

area1   bases moved properly  from.  700  740 to.  790  800
  1 entry found and moved ok
```

2. Checkio

2.1 Introduction

Checkio may supervise all actions on a particular document. That is performed in the following way: Assume that checkio is executed in a job process called *dev*. Then all messages sent to dev are passed on by checkio to the document supervised. The answer from the document is passed back to the original sender process, which seems to be handling the document in the normal way.

Checkio can easily print all messages and answers as they are passed on, and you can later find out about parity errors from the document, rereading etc.

2.2 Call

The process dev must be created with a size big enough to store the data blocks passed by.

Checkio must be called with one parameter specifying the name of the document to be supervised. The messages and answers are printed on the current output of dev. A console communication to start dev may look like this:

```
to s
new dev size 5000 run; if monitor older than 8.0
                     ; mode 0 also
to dev
o lp                 ; print on the line printer
checkio t6           ; check the document t6
```

You will then see nothing from dev until some messages are sent to it, for instance from another job doing like this:

```
t-set mto dev 0 6    ; edit to the 'magnetic tape'
                     ; dev,
t-edit infile        ; which acts as the magnetic
                     ; file t6.
```

Dev will now start listing the communication on the line printer. When you have finished using dev for supervising of t6, you can proceed like this:

```
to s
proc dev break          ; the remaining output from
                        ; dev appears on the printer.
*** fp break
checkio printer         ; select a new document etc.
```

2.3 Reserving the Document

Checkio cannot simulate those actions on a document which involve reservation of the document, creation of it as an area process etc. So a process which tries to reserve dev (the document) or create it as an area process, *and* checks the result, cannot be supervised.

Processes running utility Programs and algol/fortran programs will not act this way: operations are sent and reserve process/crete area process is done if necessary after the first operation.

To remedy the problem for these processes, checkio proceeds as follows when the document does not exist: First a call of create area process is attempted. If this does not succeed, checkio outputs the console message

```
mount <document>
```

and performs the usual wait action. Then the operation is repeated before the original sender gets his answer.

If the document rejects the message, checkio reserves the document and repeats the operation before the original sender gets his answer.

If the original sender gets stopped before checkio has received/delivered the data in i/o messages, checkio retries until the sender has delivered/received the data.

To supervise actions on an area process, a file descriptor of kind=6 and document name = name of process executing checkio must be created in advance.

2.4 Output

The output from checkio consists of one line for each message:

```
message <operation> <mode> <first addr> <last addr> <segment> <sender>
```

and one line for each answer:

```
answer <bits of logical status> <hwds> <characters> <file> <block>
```

If the answer is not transmitted back to the original sender, the reason is printed as one of the lines

```
reserved by another process
not user/cannot be reserved
does not exist/not user of area process
```

3. Clean

3.1 Introduction

The program removes all catalog entries with catalog base within the specified limits.

3.2 Example

The FP call

```
clean 2800 2899
```

removes all catalog entries with base within the limits 2800 2899.

3.3 Call

```
clean <lower limit> <upper limit>
```

where

```
<lower limit> ::=
<upper limit> ::= <integer>
```

3.4 Function

The catalog is scanned and all entries eith the catalog base within the specified limits are removed.

3.5 Error Messages

*****clean call**
Left hand side in the program call.

*****clean param**
Parameter error in the program call.

*****clean low > up**
The lower limit specified is greater than the upper limit specified.

*****clean limits off stand base**
The specified limits are outside the standard base of the calling process.

*****clean area procs missing**
Create area process unsuccessful.

*****clean process too small**
The calling process is too small

In case of any error, no entries are removed.

4. Deletelink

4.1 Introduction

The program removes links, i.e. external processes with links to LAN device handlers, either on ADP (RC8000) or on LAN (RC9000-10) controllers.

4.2 Examples

Example 1.

The call :

```
deletelink printer
```

will remove the link created in example 1 under the heading of makelink and give the output

```
main : lanmain1 :

link removed : printer dev. no : 9

links removed : 1
```

Example 2.

The call :

```
deletelink 43 44
```

will remove the links created in example 2 under the heading of makelink and give the output

```
main : lanmain1 :

link removed : gin1 dev. no : 43 link removed : gout1 dev. no : 44

links removed : 2
```

4.3 Call

```
[<outfile>=]  deletelink ( [l.<lanno>] (<devspec>)1-* )1-*

                 ( <s><devname> )
<devspec> ::= (                 )
                 ( <s><devno>   )

<s>       ::= ('sp'/.)
<lanno>   ::= number of LAN controller, default : 1
<devname> ::= name of external process
<devno>   ::= device number of external process
```

4.4 Function

For every device parameter, whether it is specified by device name or by device number, the link, i.e. the external process and its link to a device handler on the LAN controller specified, is removed.

A LAN number specification is valid until another LAN number specification. If no LAN number is specified number one is assumed.

For each link, a line is produced as output :

```
link     removed : <device name>  dev. no : <devno>
```

or

```
link not removed : <device name>  dev. no : <devno> *<explanation>
```

If the link is specified by device number, <devname> will be empty.

If unsuccessfull, cf. below in 'Error Messages' concerning the *<explanation>.

The program terminates by stating the number of links removed.

If an <outfile> is specified, it is used for output, else the current output document is used.

4.5 Error Messages

Two kinds of error messages may be given, `parameter warnings` or `operational alarms`.

`Parameter warnings` come from disobeying the call syntax, and they take the form :

***** deletelink <explanation>**

where <explanation> may be one of :

```
- l.<lanno> :          <actual parameter read>
- unknown parameter :  <-      -         -   >
```

In case of parameter warning, the fp mode bit 'warning' is raised and the program continues with the next parameter.

`Operational alarms` come from :

- dummy answer from the main process to the 'remove link' request

and take the form :

*** <explanation>**

terminating the line in the output concerning the unlink request (link not removed : <device name> ...).

The <explanation> is one of below :

- result 1	(e.g. the LAN numberss specified does not specify a LAN controller
- could not be reserved	(concerns the external process)
- does not identify a link - malfunction	(e.g. the LAN number specified does not specify a LAN controller)
- does not exist	(e.g. the LAN controller specified does not exist)

In case of an operational alarm, the fp modebit 'ok' is cleared and the program continues with the next parameter.

4. Deletelink

5. Disccopy

5.1 Introduction

The program discopy is primarily intended for:

- binary copying of discpacks or parts of discpacks from one discpack to anogher.
- save and load of backing storage files, using discpacks as backup medium.

In addition to this the program contains facilities, which make it possible to:

- include and exclude discs into and from the backing storage system.
- label a disc.
- rename a disc.

The program can be used in two different modes:

- as an ordinary utility program.
- as a conversational program, which resides permanently ('corelocked') in core and reads its parameters from the current input zone. This gives you complete freedom to copy discs at installations with only two discdrives.

The following subprograms are contained in the program disccopy:

save:	used to save backing storage files on a disc.
load:	used to load backing storage files from a disc.
bin:	used for binary copying from one disc to another.
packon:	includes an entire disc in the backing storage system.
packoff:	excludes an entire disc from the backing storage system.
kiton:	includes a single disc in the backing storage system.
kitoff:	excludes a single disc from the backing storage system.
kitlabel:	used to label a disc, i.e. names are assigned to 'document' and 'auxiliary catalog' and an empty auxiliary catalog is written.
kitname:	assigns new names to 'document' and 'auxiliary catalog' of a disc (to be explained later).

The program must run in an 'all' process or in a 'new'process with a function mask, which allows the process to call the privileged monitor procedures for:

- auxiliary catalog handling. - main catalog handling.
- create peripheral process.
- remove peripheral process.
- auxiliary entry handling.

When executing in the conversational mode, the program demands a process size of at least 80000 halfwords. Executing as normal utility program, the program demands a minimum size of 2000 halfwords.

5.2 Call

Utility Program Mode

When the program disccopy is used as a normal utility program, the call syntax is as follows:

```
          (save <savespec>       )
disccopy  (load <loadspec>       )
          (bin [<binary params>])

<savespec>::= <saveparams> [list. yes/no] [<criteria>] [<names>]

<loadspec>::= <loadparams> [list. yes/no] [<criteria>] [<names>]

                                        2-2
<saveparams>::= (to. <devno>       )
                ([from. <docname>])

                                      2-2
<loadparams>::= (to. <docname>   )
                (from. <docname>)

                (       ( system  )       )
                (scope.( project )       )
                (       ( user    )       )
<criteria>  ::=(                          )
                (       ( system        ))
                (       ( project       ))
                (base.  ( user          ))
                (       ( <lower>.<upper>))

                          0-*
<names>::=       (<name>)

                                          2-2
                   (to. <devno> [.all]   )
<binary params>::= (                     )
                   (from. <devno> [.all])
```

The programs packon, packoff, kiton, kitoff, kitlabel and kitname are called as utility programs as follows:

```
                          [        {yes     } ]
                          [        {no      } ]
packon devno.<devno>      [ list.  {nonsys  } ]
                          [        {error   } ]
                          [        {warning} ]

packoff devno.<devno>

                          [        {yes     } ]
                          [        {no      } ]
kiton devno.<devno>       [ list.  {nonsys  } ]
                          [        {error   } ]
                          [        {warning} ]

kitoff devno.<devno>

kitlabel <devno> <docname> <auxname> {slow/fast} <catsize>,

<slicelength> <size>

kitname <devno> <docname> <auxname>

<devno>, <lower>, <upper>, <catsize>, <slicelength>,
<size> ::= integer

<name>, <docname>, <auxname> ::= name, max. 11 chars.
```

Conversational Mode

Conversational mode is entered by the call:

```
disccopy typein
```

Now the subprograms are activated as follows:

```
save <savespec>
load <loadspec>
bin
```

The programs packon, packoff, kiton, kitoff, kitlabel, and kitname are activated as in utility program mode, i.e.

```
                          [     { yes     }]
                          [     { no      }]
packon devno.<devno>      [list.{ nonsys  }]
                          [     { warning }]
                          [     { error   }]
```

etc.

When all your tasks are completed, you leave the conversational mode by typing the command:

```
end
```

5.3 Details

In this section details are given about the subprograms and their parameters.

5.3.1 Save

This subprogram will select backing storage files as specified by the call parameters <criteria> and <names> and save these entries and their corresponding areas (if any) on the object disk.

Parameters:

to.<devno>

The parameter specifies the object disk.
Files are copied to the disk with devicenumber <devno>. The object disk must not be included in the backing storage system and the files which are copied must not exist in the auxiliary catalog of the object disk (no copying is performed if this is the case).

from.<docname>

The parameter specifies the source disk.
This parameter is optional. If source disk is specified, it must be included in the backing storage system and only files which belong to this disk are copied. If no source disk is specified, all files which satisfy the conditions stated by <criteria> and <names> are copied.

<criteria>

scope: only files with the scope specified are saved.

base: only permanent files (key > = z) with entrybases inside the interval specified are saved (base.system is equivalent to `base.-8388607.8388605)`.

If no <criteria> are specified, default is scope.system.

<names>

A list of entrynames can be used to reduce the <criteria> parameter, i.e. only files with name and scope/base as specified are saved.

list.{yes/no}

A listing of the files saved is produced on current output if list.yes is specified. list.no is default.

Examples:

ex.1. All files which belong to 'disc' and have userscope are saved on the disk mounted on device 7 (but not included in the backing storage system) by the following call:
```
save from.disc to.7 scope.user list.yes
```
A listning of the files saved is produced on current output.

Ex.2. By the following call, all backing storage files which fulfil:
`-8388607<= lower entrybase <= upper entrybase <= 8388605 and entry permkey >= 2` are saved on the disk, which is mounted on device 7:
```
save to.7 base.system
```

Ex.3. By the following call, the two backing storage files ncjfile1 and ncjfile2, which have system scope (default) are saved on the disk, mounted on device 7:
```
save to.7 ncjfile1 ncjfile2
```

5.3.2 Load

This subprogram will select files on the source disk as specified by <criteria> and <names>. The selected files are loaded into the backing storage system, i.e. non existing entries are created on object disk and existing files on object disk are overwritten.

If a file is already included from a document different from the object disk, load of the file is rejected and a message is written on current output. In this case it will be necessary to rename the entry which causes conflict and repeat the load.

Parameters:

to. <docname>

Specifies the object disk.
The object disk must be included in the backing storage system.

from. <devno>

Specifies the source disk.
The source disk must not be included in the backing storage system.

<criteria>, <names> and list

As for save.

Example:

All files of scope user on the disk, which is mounted on device 6 (but not included in the backing storage system) are loaded to 'disc1' (which must be included in the backing storage system) by the following command:

```
load from.6 to.disc1 scope.user
```

5.3.3 Bin

This subprogram is used for various kinds of binary copying between disks. The following remarks concern binary copying in general:

- For safety reasons you should writeprotect the source disk (not possible for some fixed media disks).
- The object disk must not be included in the backing storage system. You must explicitly perform kitoff on the object disk if this is the case.
- kitoff is implicitly performed on the source disk. During this possibly the text:

  notice: disc with maincatalog is removed.

  will appear. If this is the case, it will be necessary to perform one of the following actions (depending on the "programmode") in order to connect the maincatalog when the copying is terminated:

  utility mode:

  switch off the writeprotection and autoload the system.

  conversational mode:

  switch off the writeprotection and perform kiton on disk with main catalog.

  The same procedure must be used if the object disk during the copying has replaced the disk containing the main catalog. Remember to reinstall the main catalog disk first.

- When the copying is terminated, the text:

  ```
  copying terminated
  number of segments copied: <segments>
  ```

  is written on current output.

Bin with <binary params>.

When the subprogram is called with <binary params>, it is used for copying from one disk to another, as described below:

Parameters:

```
from/to. <devno> [.all]
```

'all'-parameter present:
copying is performed from/to the physical disk, which contains the logical disk with devicenumber <devno>.

no 'all'-parameter:
copying is performed from/to the disk (physical or logical) with devicenumber <devno>.

Examples:

```
bin to.7 from.6          (logical disk -> logical
                         disk)
bin to.7.all from.6.all  (physical disk -> physical
                         disk)
bin to.7 from.6.all      (physical disk -> logical
                         disk)
```

Bin Without Parameters

When called without parameters, the subprogram bin is used for copying of specified parts of disks or for copying between disks of unequal size (examples of this are given below). If the subprogram bin is called without parameters, it will ask for:

```
to device:
start segment:
from device:
start segment:
number of segments:
```

To every equestion, you must type in a number, terminated by 'NL' (<devicenumber> respectively <segmentnumber>). In case you do not know the size of the disk, just type in a too big integer.

Example 1:

In this example a 33 Mb disk is copied from device 6 to device 7.

```
to device:           7
start segment:       0
from device:         6
start segment:       0
number of segments:  43155
```

Example 2:

In this example is shown how to copy a 66 Mb disk on two 33 Mb disks.

```
to device:           6 (33 Mb)
start segment:       0
from device:         7 (66 Mb)
start segment:       0
number of segments:  43155
to device:           6
start segment:       0
from device:         7
```

```
start segment:          43155
number of segments:     43155
```

It is impossible to copy the entrie disk, but all slices containing data are copied (the last 105 segments are not copied).

In the conversational mode, the subprogram must be explicitly activated for each copying by typing the text:

```
bin
```

Sizes of Disks and Copytime

size	size in segments	copytime in minutes	
33 Mb	43155	1 1/2	RC8000
66 Mb	86415	3	-
124 Mb	163989	6	-
133 Mb	174293	6 1/2	-
248 Mb	328377	12	-
850 Mb	951510	35	RC9000-10
1.2 Gb	1600452	59	-

5.3.4 Packon

The program is used to include into the backing storage system an entire disk (on RC8000 only RC83xx disk), i.e. all the logical disks it contains. The remarks below for kiton are valid for each disk included by packon.

Parameters:

devno. <devno>

The device specified must be the device number of the physical disk to be included.

list

As for kiton.

5.3.5 Packoff

The program is used to exclude from the backing storage system an entire disk (on RC8000 only RC83xx disk), i.e. all the logical disks it contains. The remarks below for kitoff are valid for each disk excluded by packoff.

Parameters:

devno. <devno>

The device specified must be the device number of the physcial disk to be excluded.

5.3.6 Kiton

The program is used to include a logical disk in the backing storage system. If the maincatalog has been removed and the auxiliary catalog of the disk involved contains an entry with the same name <:catalog:>, this entry is selected as maincatalog and the maincatalog is connected. This is very convenient, as you need not autoload the system if the maincatalog has been removed and you want to switch to a task which demands the presence of the maincatalog (when in the conversational mode).

Parameters:

devno. <devno>

The logical disk with devicenumber <devno> is included in the backing storage system.

list

yes: all entries in the auxiliary catalog are listed on current output during insertion in the maincatalog. If an insertion causes trouble, the monitor result is listed, too.

no: no listing.

nonsys: listing of all non system entries, which have been inserted in the main catalog.

warning: listing of entries, which have not been inserted due to nameoverlap.

error: listing of entries, which caused trouble during insertion. The monitor result is listed too.

Example:

```
kiton devno.7 list.yes
```

5.3.7 Kitoff

The program excludes a logical disk from the backing storage system. If the disk involved contains the maincatalog, the maincatalog is removed and the text:

```
notice: disc with maincatalog is removed
```

is written on current output.

If the program itself resides on the disk involved, the text:

```
notice: disc with program file is removed
```

is written on current output. This causes no trouble if you run in the conversational mode. If in normal utility mode, the program must be reinstalled before further use.

If the file processor is found on the disk, the text:

```
notice: disc with fp is removed
```

is written on current output. In conversational mode it represents no problem until the end command is given.

When the program terminates, it sends a finis message to the parent.

Parameters:

devno. <devno>

The logical disk with devicenumber <devno> is excluded from the backing storage system.

Example:

```
kitoff devno.6
```

5.3.8 Kitlabel

The program assigns names to 'document name' and name of 'auxiliary catalog' of a logical disk and writes an empty auxiliary catalog on the disk.

Parameters:

<devno>:	devicenumber
<docname>:	documentname
<auxname>:	name of auxiliary catalog
slow/fast:	disk/drum, respectively (obsolete)
<catsize>:	size of auxiliary catalog in segments
<slicelength>:	segments/slice
<size>:	disksize in slices

Example:

```
kitlabel 7 disc1 catdisc1 slow 50 21 2045
```

5.3.9 Kitname

The program is introduced as an 'ad hoc' solution to a problem, which arises if the program disccopy terminates during a save. During a save, the object disk is included in the backing storage system with

workingnames for "document" and "auxiliary catalog" in order to avoid confusion if nameoverlap occurs. These workingnames are dumped in the chainhead of the object disk each time an entry is created on the object disk. Normally, this causes no trouble as the original names are rewritten when the program terminates properly. If, however, the program terminates during the save, you will have to use the program kitname if you want to reestablish the original names without erasing the files saved on the disk.

Example:

```
kitname 7 discl catdiscl
```

5.4 Error Messages and Warnings

5.4.1 Logical Status

If the program disccopy finds it impossible to read or write a segment, one of the following errormessages are written:

```
input  from <name> segm:     <segnumber> status
                             <bitpattern>
output to   <name> segm:     <segnumber> status
                             <bitpattern>
```

Where <bitpattern> is a logical statusword, describing the read/write-operation, cf. [1], chapter 6.

The copying will ignore the segment and continue, though, with fp mode bits warning.yes and ok.no, with following exceptions:

```
a.  <bitpattern> = .....1.............1.
b.  <bitpattern> = .................1.....
```

a. means end of document, i.e. the object disk is too small.

b. means receiver does not exist, i.e. the diskdrive is switched off, and the like.

5.4.2 Results from the Monitor

The program responds to errors returned from monitorcalls by writing a text describing the monitor procedure and the actual result value, e.g.

```
create peripheral process <name> result <number>
```

When executing in utility program mode, the program will terminate and return to fp with modebits ok.no, warning.yes.

When executing in conversational mode, the program will continue reading commands without touching the mode bits.

The list below describes all errormessages of this type. In parenthesis the numbers of the corresponding monitorprocedures are given.

```
create peripheral process     (54)
delete bs                     (108)
delete entries                (110)
prepare bs                    (102)
set catalog base              (72)
create entry                  (40)
create aux entry              (120)
insert entry                  (104)
insert bs                     (106)
```

When one of these errors occurs, please consult ref. [2] to get the details.

5.4.3 Warnings

When running in utility program mode, the program will return to FP after normal termination with modebits ok.yes, warning.yes in case of warnings.

Save

If you attempt to save an entry which already exists in the auxiliary catalog of the object disk the following warning is written:

```
entry already exists in auxcat: <entryname>
```

and the entry is skipped.

Load

If an entry to be loaded is included in the backing storage system from a document different from the object disk, the following warning is written:

```
entry already included from another document: <entryname>
```

and the entry is skipped.

Both

If one or more entries specified cannot be found in the catalog/auxiliary catalog, the warning

***** entries not found**

succeeded by a list of the entries concerned is written.

6. Discstat

6.1 Introduction

The program is a diagnostic tool for printing statistical information maintained by a physical disk driver, serving a disk storage module connected to an RC9000-10 computer.

The program may as well print directly from the core as from a file containing a core picture.

6.2 Call

```
(outfile) = discstat {[clear.{<yesno>}] ,
     [dump.<dumpfile>] {<disc>}1-*}1-*

<disc>      ::= disc.<discname>/
                addr.<discaddr>/
                devno.<devno>  /

<discname>  ::= name of disc process

<discaddr>  ::= address of disc process

<devno>     ::= device number of disc process

<dumpfile>  ::= name of system dump file

<yesno>     ::= yes / no,  default: no
```

6.3 Function

The program sends a message to the autoload disk on the same physical disk as the specified disk to collect the statistics. The disk may be either a logical or a physical disk - in either case the statistics concern the information created by a physical disk.

Please note that the disk will reset its statistics when the information has been collected. Please note that a logical disk holding a logical backing

storage during normal use will be reserved by the anonymous process executing process functions - therefore the program uses the autoload disk on the same physical disk.

If dump.<dumpfile> is specified it is supposed that the backing storage area <dumpfile> contains a memory picture with memory address 0 equivalent to area address 0.

The disk addressed may be either a logical or a physical disk - in either case the statistics concern the information created by a physical disk.

6.4 Examples

The following command extracts statistics about the disk to which the disk processes named 'disc2' and 'disc3' are connected:

```
discstat disc.disc2 disc.disc3
```

The following command extracts statistics about the disk with the process description address 24678 from the memory dump in the file named core8000:

```
discstat dump.core800024678
```

6.5 Error Messages

*****discstat, syntax**
syntax error in call

*****discstat, buffer claim exceeded**
no message buffer available

*****discstat, monitor result <result>**
a normal answer was not received from the disk driver. <result> is the result delivered in the answer to the message.

*****discstat, create <result>**
not possible to create a peripheral process for a reason given by <result>.

*****discstat, status error <status>**
<status> is the (decimal) statusword received from the disk driver.

*****doscstat disc missing <param>**
no disk was specified in the call.

7. Disctell

7.1 Introduction

When copying disks by the utility program disccopy, hard errors reading the disk will be identified by the program as errors reading disk segments given by the program as physical segment numbers on the given physical disk.

The present program solves the problem of identifying the logical context in which a physically numbered segment occurs.

The program may

- list the physical disks described in the monitor
- list the logical disks located on a given physical disk
- list the logical disk and the logical slice and segment number of a given segment number of a given physical disk together with a description of a possible area in which the segment belongs (name, base, namekey, permkey, slicelength and slicelist of the area).

7.2 Call

```
{outfile =} disctell [<param>]

                                                          0-*
<param>  ::= {physical                                   }
             {<physical disk no> {.<segment no>}0-*}
```

Explanation

`disctell`

gives a list of the possible parameters.

`disctell physical`

gives a list of the physical disks described in the monitor.

disctell {<physical disk no>}1-*

gives a list of the logical disks located on the disks <physical disk no>, one by one.

disctell {<physical disk no>{.<segment no>}0-*}1-*}

gives the logical disk and slice number of the given segments, possibly including name of area, size, bases, permkey, slicelength and slicelist of the area with an indication of whether the segment is inside or outside the used part of the slices of the area.

7.3 Examples

Example 1

```
*disctell physical
physical disc : device no. 27  kind: 6            process descr. addr. 30650
physical disc : device no. 29  kind: 6            process descr. addr. 30766
```

Example 2

```
*disctell 27 29
logical disc :  device no. 28  kind : 6  wrk017035 process descr. addr. 30708
logical disc :  device no. 54  kind : 6  disc      process descr. addr. 34846

logical disc :  device no. 30  kind : 6  wrk017035 process descr. addr. 30824
logical disc :  device no. 55  kind : 6  disc1     process descr. addr. 34904
logical disc :  device no. 60  kind : 6  disc2     process descr. addr. 35282
logical disc :  device no. 62  kind : 6  disc3     process descr. addr. 35420
logical disc :  device no. 63  kind : 6  disc4     process descr. addr. 35478
logical disc :  device no. 64  kind : 6  disc5     process descr. addr. 35536
```

Example 3

```
*disctell 27.1000 27.86000
disc reserved for system purpose: wrk017035
device number                       28
process descr. addr.              30708
segment number      1000 of        2000

segment no.  86000 is located on device  54   disc
                              on logical slice no. 999


entryname    rtp35022p5
size         200
bases        -8388607  8388605
namekey      119 on catdisk
permkey      3
```

```
slicelength  84
slicelist
     998     999     1000
number of lices      3
the segment is inside the area  (segment no. 168 of the area
                                 segment no. 84 of the slice)
```

Example 4

```
*disctell  86000

***disctell illegal device number
```

Example 5

```
*disctell  30

***disctell device is not a physical disc
```

8. Do

8.1 Introduction

The do language is intended for monitor and hardware testing and supervising, but it is designed so that it can be used as a general programming language, in which the user can set register and memory contents, execute monitor and FP procedures as well as slang instructions, output any information about the memory in an easy readable form etc.

Examples

The command:

```
do write 102.word.4 116
```

will output the contents of words 102, 104 and 116, i.e. the values of max time slice, time slice and number of memory halfwords.

The command:

```
do w1.w2 w2.w3.in.20 140.w3
```

will execute

```
w1:= w2, w2:= w3 + 20, word 140:= w3.
```

The command:

```
outfile=do w0.74,                ; w0:= 74
          w0.x0.0,               ; w0:= word(w0)
          w1.76,                 ; w1:= 76
          z1.x1.0,               ; z1:= word(w1)
          do w0.in.2,            ; w0:= w0 + 2
          while w0.ne.z1,        ; while w0 <> z1
          w2.x0.0,               ; w2:= word(w0)
          write x2.peripheral.26 end,
       ,                         ; write 26 hwds of x2 with
       ,                         ; layout peripheral
       od                        ; end inner do
```

will output the first 26 halfwords of the descriptions of all external devices on outfile.

See also further examples section 8.6.

8.2 Call

The program is called as follows:

```
<output file> = do <do command>

                 {<assignment>     }
                 {<write command>  }
                 {<monitor call>   }
                 {<fp call>        }
                 {<slang command>  }
                 {jump <value>     }
                 {clear            }
<do command> ::= {<wait command>  }
                 {if <value>       }
                 {fi               }
                 {do               }
                 {while <value>    }
                 {od               }
                 {go <value>       }
                 {og               }
                 {exit             }

<wait command> ::=  wait  {<integer>    }
                          {<name>       }

<value ::= <expression> {.<expression>} 0-*

<assignment> ::=
                                                    0-*
{<simple variable> <expression tail>} {.<name>      }
{<array base>                       } {.<expression>}
```

```
                                {0}
<simple variable> ::= {w}       {1}
                      {z}       {2}
                                {3}

<array base> ::=  {<integer>    }
                  {<array name>}

                       {0}
<array name> ::=  {x}  {1}
                  {y}  {2}
                       {3}

<expression tail> ::=  {.<operator> > <operand>}0-*

<expression> ::= <operand> <expression tail>

                {<integer>          }
                {<simple variable>  }
                {<indexed variable>}
<operand> ::=   {ba                 }
                {cc                 }
                {fp                 }
                {hn                 }

<indexed varable> ::= <array name> . <integer>

                { in }    +
                { de }    -
                { le }    shift left
                { ri }    shift right
                { or }    logical or
                { an }    logical and
                { eq }    =
<operator> ::=  { ne }    <>
                { gr }    >
                { ls }    <
                { ng }    <=
                { nl }    >=
                { mu }    *
                { di }    //
                { mo }    modulo
                { po }    **

<write command> ::= write <write action> {end              }
                                         {<end of do call>}

                    {<simple variable>  {.<format>}0-*}
<write action> ::=  {<base> {.<hwds>  }0-*            }
                    {       {.<format>}               }

              { <array base> }
              { ba           }
<base> ::=    { cc           }
              { fp           }
              { hn           }
              { bittable     }
```

```
            ( <integer>          )
<halfwords> ::= ( <simple variable>  )
            ( <indexed variable> )

<format> ::= ( <simple format>     )
             ( <structured format> )

                      (empty     ) nothing is output
                      (word      ) word as signed integer
                      (halfwords    ) 2 halfwords
                      (octets    ) 3 8-bit groups
                      (sextets   ) 4 6-bit groups
                      (octal     ) positive octal number
                      (binary    ) bit pattern
<simple format> ::=   (text      ) 3 iso-characters
                      (code      ) slang instruction
                      (double    ) 2 words as double word
                      (groups    ) word,halfwords,octets,sixtets,octal
                      (all       ) code,octal,word,halfwords,octets,text
                      (binword   ) word and binary
                      (words5    ) 5 words/line
                      (halfwords10  ) 10 halfwords/line
                      (name      ) 4 words as text
                      (procname  )
                      (procnames)

                         (buffer          )
                         (area            )
                         (peripheral      )
<structured format>::=   (internal        )
                         (tail            )
                         (zone            )
                         (share           )
                         (<mon. 2 format>)
                         (<mon. 3 format>)

<mon. 2 format> ::= note

                      ( answer     )
<mon. 3 format> ::=   ( chaintable )
                      ( entry      )

<slang command>::= slang <value> (<instruction>)0-*
                                 (w <word>     )     ,

                    (end             )
                    (<end of do call>)

                           (           (0) )0-*
                           (       (w) (1) )
<instruction> ::=  <code>  ( ( . ) (x) (2) )
                           (           (3) )
                           ( <word>        )
```

```
                 {           (0) }0-*
                 {       (w) (1) }
<word> ::=       { ( . ) (x) (2) }
                 {           (3) }
                 { <integer>     }

<code> ::= <two letter mnemonic slang instruction>

                        { monitor procedure <integer>  }
                        { set interrupt                }
                        { process description          }
                        { initialize process           }
                        { reserve process              }
                        { release process              }
                        { include user                 }
                        { exclude user                 }
                        { send message                 }
                        { wait answer                  }
                        { wait message                 }
                        { send message                 }
                        { send answer                  }
                        { wait event                   }
                        { get event                    }
                        { get clock                    }
<monitor call> ::=      { set clock                    }
                        { look up entry                }
                        { change entry                 }
                        { rename entry                 }
                        { remove entry                 }
                        { permanent entry              }
                        { create area process          }
                        { create peripheral process    }
                        { create internal process      }
                        { start internal process       }
                        { stop internal process        }
                        { modify internal process      }
                        { remove process               }
                        { generate name                }
                        { copy                         }
                        { <mon. 2 call>                }
                        { <mon. 3 call>                }

                        { modify backing store }
                        { select backing store }
<mon. 2 call> ::=       { select mask          }
                        { test log             }
                        { return status        }
```

```
                        { set catalog base           }
                        { set entry base             }
                        { lookup head and tail       }
                        { set backing storage claims }
                        { create pseudo process      }
<mon. 3 call> ::=       { regret message             }
                        { create backing storage     }
                        { insert entry               }
                        { remove backing storage     }
                        { permanent entry in,        }
                        { auxiliary catalog          }
                        { create entry look process  }

                        { fp procedure <integer> }
                        { finis message          }
                        { inblock current        }
                        { inblock                }
                        { outblock current       }
                        { outblock               }
                        { wait ready input       }
                        { wait ready output      }
                        { wait ready             }
                        { inchar current         }
                        { inchar                 }
                        { outchar current        }
                        { outchar                }
                        { connect current input  }
                        { connect input          }
                        { connect current output }
                        { connect output         }
                        { stack input            }
<fp call> ::=           { stack zone             }
                        { unstack input          }
                        { unstack zone           }
                        { outtext current        }
                        { outinteger             }
                        { outend current         }
                        { outend                 }
                        { closeup current        }
                        { closeup                }
                        { parent message         }
                        { wait free input        }
                        { wait free output       }
                        { wait free              }
                        { break message          }
                        { terminate input        }
                        { terminate output       }
                        { terminate zone         }
```

8.3 Function

The program processes the list of commands given in the program call. All variables are placed so that they remain unchanged by successive calls. A sequence of commands can therefore be executed in one or

several do calls, provided these calls are of same kind (all with or without specified output file).

(Numbers in [ ] refer to the examples in section 8.6.)

w0, w1, w2, w3, z0, z1, z2, and z3 are names of simple variables [1,3].

x0, x1, x2, x3, y0, y1, y2, and y3 are names of arrays. An array name followed by a point and an integer acts as an indexed variable [2,3,5,7]. The start address of an array depends on the context.

An expression is a list of alternating operands and operators separated by points [1,2]. The list must begin and end with an operand. An expression is interpreted from left to right. The result is an integer word. The operands and their values are:

<integer>	the integer value.
w0-w3,z0-z3	the value of the variable.
x0-x3, indexed	the word addressed by the value of the corresponding w-variable increased by the index.
y0-y3, indexed	the word addressed by the value of the corresponding z-variable increased by the index.
ba	the buffer address (first free hwd).
cc	the current command address (the hwd after the last free hwd).
fp	the fp base.
hn	the address of a word array containing the fp h-names (h0-h99).

All operators are dyadic and have the same priority [24]. The notation, the meaning, and the effect of the operators are (left and right refer to the operands):

in	increase	left + right
de	decrease	left - right
le	left shift (logical)	left shift right
ri	right shift (logical)	left shift (-right)
or	or (logical)	left or right
an	and (logical)	left and right
eq	equal	if left=right then -1 else 0
ne	not equal	if left<>right then - 1 else 0
gr	greater	if left>right then -1 else 0
ls	less	if left<right then -1 else 0
ng	not greater	if left<=right then - 1 else 0
nl	not less	if left>=right then - 1 else 0
mu	multiply	left * right
di	divide	left / right
mo	modulo	left mod right
po	power	left ** right

8.3.1 Assignments to Simple Variables

An expression where first operand is a name of a simple variable assigns the result to the variable [38]. If the expression is followed by one or more other expressions or names, the result of the last expression or the first word of the last name is assigned to the variable [2,4].

8.3.2 Assignments to Arrays

An array assignment consists of a start address (an integer or an array name) followed by a list of expressions and names stored in consecutive words. Expression values are stored as integer words, names as four text words [5,7].
Four fields corresponding to the names x0-x3 are reserved for building up arrays. Each field has a size of 34 halfwords. An assignment to one of the names x0-x3 will place the array in the corresponding field, and transfer the address of this field to the w-variable of same number. When assigning to y-names the value of the corresponding w- variable will be used as start address. (About the use of x- and y-arrays in expressions, see 8.3, in write actions, 8.3.3, and in slang, 10.3.6). An integer used as start address is interpreted as an absolute address in memory.

8.3.3 Write Command

Output from the do-program is controlled by write commands. Current output is used unless the call specifies another file, in which case current program zone (h19) is used.

Notation:

```
write <write actions separated by spaces> end
```

The terminating word (end) may be omitted at the end of the parameter list.

All output concerns memory contents. There are three kinds of write actions:

1. simple variable write action
2. array write action
3. special write action

1. simple variable write action:
The write action consists of the name of the simple variable [9]. Initially the format is integer word. This can be changed by one or more formats (see later) [10].

2. array write action:
The write action consists of a write base defining the start address [13]. The write base can be an integer using this as the start address, an x-name using the value of the corresponding w- variable as start address, an y-name using the value of the corresponding z-variable as start address, or one of the constant operands, ba, cc, fp and hn, using this value as start address. Initially the number of halfwords to be output is two and the format is integer word. Halfwords and format can be changed by placing parameters after the write base [13]. Halfwords can be defined by an integer, a simple variable, or an indexed variable. For each definition the number of halfwords will be output using the current format. The format can be changed by one or more formats (see later).

Simple formats define how words or small groups of words should be output. Each word or group of words is output in the same way [15]. A structured format defines a relative start address and a structure of several simple formats. Further it defines the number of halfwords to be output [37], unless this is specified after the format.

3. special write action:
This acts as an array write action except for the write base which is a name defining a special set of: start address, initial number of halfwords, and initial format [16].

The special write actions are:

bittable	the bittable placed at the top of the core store contains one bit for each backing store segment (only used in monitor 2). Format: binary.

The simple formats are:

empty	nothing is output
word	signed integer
bytes	two halfwords as positive integers
octets	three 8-bit groups as positive integers
sixtets	four 6-bit groups as positive integers
octal	a positive octal number
binary	binary number with points instead of zeroes
text	three 8-bit groups as iso characters. Character values less than 32 are output as spaces.
code	slang instructions
double	two words as a double word integer
groups	a combination of word, halfwords, octets, sixtets, and octal
binword	a combination of word and binary
all	a combinationof code, octal, word, halfwords, octets, and text
words5	five words on one line
bytes10	ten halfwords on one line
name	four words as a text
procname	a signed integer forllowed by a point and the name of the process having the word value as process description address
procnames	a binary word followed by the names of the internal processes, the identification bits of which match the ones in the word value.

The structured formats are:
(the words are output using simple formats corresponding to their contents). In the paranthesis is specified the standardlength in halfwords for monitor 2 and monitor3.

buffer	a message buffer (24, 24)
area	an area process description (20, 24)
peripheral	a external process description (90, 100)
internal	an internal process description (74, 92)
tail	a catalog entry tail (20, 20)
zone	a zone descriptor (50, 50)
share	a share descriptor (24, 24)
note	an fp note (22, -)

answer	an answer from external process (10, 10)
entry	an area entry (-, 36)
chaintable	a chaintable head and a part of the table (-, 66)

Only the last format of a sequence of formats is used. For all write actions output takes place only after an explicit definition of halfwords or at the end of the write action.

8.3.4 Call of Monitor Procedures

A monitor procedure will be called when its name appears in the parameter list [35]. The call uses the variables w0-w3 as register values. On return the registers are stored in these variables. The names of the procedures can be found in the syntax description. New procedures will be included when appearing. The first procedure has the following effect:

```
monitor <s> procedure <s> <integer>: jd 1<11+<integer>
```

8.3.5 Call of File Processor Procedures

A file processor procedure will be called when its name appears in the parameter list [18]. The call uses the variables w0-w3 as register values. On return the registers are stored in these variables. The names of the procedures can be found in the syntax description. New procedures will be included when appearing. The first procedure has the following effect:

```
fp <s> procedure <s> <integer>: j1 w3 <fpbase>+<integer>
```

8.3.6 Slang Command

Notation:

```
slang <values> <instructions> end
```

Function:

Stores a list of instructions and word values in consecutive words starting with the address defined by <value> [50]. An instruction starts with a mnemonic code. Modifications are determined thus:

`<s> w-name`	working register
`. w-name`	relative mark and working register
`<s> x-name`	index register
`. x-name`	indirect mark and index register.

Only the last modification of each kind is used. A word value starts with the letter, w. The word value and the instruction discplacement are determined as the sum of a zero and possible appearings of the values:

```
<s> integer  + <integer
 . integer   - <integer>
```

```
<s> y-name    + <value of w-variable>
  .  y-name   - <value of w-variable>
<s> z-name    + <value of z-variable>
  .  z-name   - <value of z-variable>
```

8.3.7 Jump Command

Notation:

`jump <value>`

Function:

jumps to the address defined by <value> with link in w3. The variables w0, w1, and w2 are used as register values. On return w0, w1, and w2 are stored in these variables [58].

8.3.8 Clear Command

Notation:

`clear`

Function:

clears all variables and the buffer area by setting zeroes in the process area not occupied by the file processor and the do-program itself.

8.3.9 Wait Command

Notation:

`wait <s> <integer>`

or

wait <s> <name>

Function:

An integer denotes the number of seconds (CPU=time) in which dummy instructions are executed. A name is assumed to be a process name and the do-program tries to reserve the process until it succeeds doing this. If the name is not a process name it is looked up in the catalog and its possible document name is used as process name [8].

8.3.10 If Command

Notation:

```
if <s> <expression>
```

Function:

If the expression is negative (true), the command has no effect. If it is positive (false), commands up to and including the corresponding fi command are skipped. This means that nesting of conditions is possible [26].

8.3.11 Fi Command

Notation:

```
fi
```

Function:

No effect [26], but see the if command.

8.3.12 Do Command

Notation:

```
do
```

Function:

stacks a return point for the corresponding od command [39].

8.3.13 While Command

Notation:

```
while <s> <expression>
```

Function:

If the expression is negative (true), the command has no effect. If it is positive (false), commands up to and including the corresponding od command are skipped and the command pointer is unstacked. Unstacking the outermost command pointer has no effect [28].

8.3.14 Od Command

Notation:

```
od
```

Function:

Interpretation continues at the stacked command pointer [29], see do command.

8.3.15 Go Command

Notation:

```
go <value>
```

Function:

The command is a procedure declaration head or a procedure call. <value> defines the procedure number which must not be negative or above a certain limit which for the moment is 30. The first appearance of a command with a given number acts as the head of a procedure declaration [43]. The command pointer is saved and the procedure body terminated by the corresponding og- command is skipped. Later appearances of a go-command act as procedure calls [48]. The command pointer is stacked and interpretation continues at the saved procedure command pointer.

8.3.16 Og Command

Notation:

```
og
```

Function:

The command pointer is unstacked and interpretation continues at this command pointer [46].

8.3.17 Exit Command

Notation:

```
exit
```

Function:

Termination of the program [33].

8.4 Storage Requirements

5730 halfwords plus space for FP.

8.5 Messages

Appearing on current output.

*****do param <illegal parameter>**
parameter in illegal syntactical position. The parameter is ignored.

*****do connect <i> <outfile>**
<outfile> could not be connected for output because of a hard error. The value of <i> determines the error:

1 no resources
2 malfunctioning
3 not user, non-exist
4 convention error
5 not allowed
6 name format error

The ok-bit is set to false and the program is terminated.

*****do no core**
the memory area is too small. The ok bit is set to false and the program is terminated.

*****do core addr**
attempt to use a storage word outside the memory area of the process. The ok bit is set to false and the program is terminated.

*****do format**
error in the format table (error in the do-program). The ok bit is set to false and the program is terminated.

*****do niveau**
the number of format niveaus is exceeded. The number is an assembly option. Indicates possibly an error in the format table (error in the do-program). The ok bit is set to false and the program is terminated.

8.6 Examples

(numbers in [ ] used for references in section 10.3).

```
[1]  do w0. 17              ; w0:= 17;

[2] do w1.cc.x1.0           ; w1:= current command;
                            ; w1:= word(w1);
[3] do z2.cc.y2.2           ; z2:= current command;
                            ; z2:= word(z2+2);
[4] do z3.longname          ; z3:= <:lon:>;
[5] do x2.0.de.1.test.1.2 ; w2:= address of w2-field;
                            ; word(w2):= 0-1;
                            ; words(w2+2:w2+8):= <:test:>;
                            ; word(w2+10):= 1;
                            ; word(w2+12):= 2;
[6] do w3.w2.in.10          ; w3:= w2+10;
[7] do y3.w0                ; word(w3):= w0;
[8] do wait lp              ; wait until lp is ready;
[9] do write w0,            ; write (out,w0,
[10]   w1.bytes,            ;     <<halfwords>,w1,
[11]   z2.octets,           ;     <<octets>,z2,
[12]   z3.text,             ;     <<text>,z3,
[13]   x2.code.2,           ; <<code>,word[w2),
[14]     .name.8,           ; <<name>,words(w2+2:w2+8),
[15]     .word.4,           ;     <<word>,words(w2+10:w2+12),
[16]   bittable.2           ;     <<binary>,word(bittable));
```

The following example reads characters from current input and determines the two types: digit and other.

```
[17] do,                                ; begin
[18] inchar current,                    ; start:
[19] z0.w2 w2.10,                       ;   inchar(i);
[20] outchar current,                   ;   outchar(10);
[21] w2.z0 outchar,                     ;   outchar(i);
[22] w2.32 outchar,                     ;   outchar(32);
[23] z1.z0.ng.57,                       ;   digit:= i<=57;
[24] z2.z0.nl.48.an.z1,                 ;   digit:= digit and
                                            i>=48;
[25] x0.other,                          ;   text:= <:other:>;
[26] if z2 x0.digit fi,                 ;   if digit then text:=
                                            <:digit:>;
[27] outtext,                           ;   outtext(text);
[28] while z0.ls.122,                   ;   if i<122 then
[29] od,                                ;   goto start;
[30] w2.10.outend,                      ;   outend(10);
[31] w1.hn.x1.40.in.fp,                 ;   in:= fpbase+h20;
[32] write x1.zone                      ;   write(<<zone>,in)
     a1p3z                              ; end a1p3z

[33] do exit so this is not exeucted
```

The following example prints the process description and the event queue for the internal process, s.

```
[34] do x3.s,                    ; begin
[35]    process description,     ;   process description(<:s:>,proc);
[36]    if w0.gr.0,              ;   if proc>0 then
[37]      write x0.internal end, ;     begin write(<<internal>,proc);
[38]      w0.in.14 w2.w0,        ;       head:= buf:= proc+14;
[39]      do w2.x2.0,            ;       for buf:= word(buf)
[40]        while w2.ne.w0,      ;           while buf<>head do
[41]        write x2.buffer end, ;         write(<<buffer>,buf)
[42]    od                       ; end end;

[43] do go 0,                    ; procedure go 0;
[44]     w0.in.1,                ; begin w0:= w0+1;
[45]     write w0 end,           ;   write(w0)
[46]    og,                      ; end;
[47]    clear,                   ; clear;
[48]    go 0 go 0                ; go 0; go 0;
```

The following example assembles, prints, and executes a piece of code.

```
[49] do w2.fp,                   ; w2:= fp base;
[50]  slang ba,                  ; slang(ba);
[51]    rs.w3   0,               ; begin save link;
[52]    al w0   11,              ;   w0:= 11;
[53]    al.w1  .4,               ;   w1:= ba;
[54]      ac w2 x2 0,            ;   w2:= -w2;
[55]    jl.w0.x0.8,              ;   goto saved link;
[56]  end,                       ; end;
[57]  write ba.code.10 end,      ; write(<<code>,words(ba:ba+8]);
[58]    jump ba,                 ; jump(ba);
[59]    write ba w0 w1 w2 fp.0   ; write(ba,word(ba),w0,w1,w2,fp);
```

Output from examples:

```
w0 =     17
w1 =   2   10
z2 = 120 100 111
z3 = lon
x2.234598
  +0  63.w3(x3-1)
  +2  test
 +10       17
 +12        2

bittable.262138
  +0  ......111111111111111.1.

a other
1 digit
p other
3 digit
z other
```

```
x1.232350
 -36   244789
 -34   245301
 -32   233474
 -30   233474
 -28   233474

 -26  2048     0
 -24  boss
 -16     8900
 -14        0
 -12        0
 -10       69

  -8  ........................1
  -6   232034
  -4   10   1   0
  -2        0    0    0    0    0    0    0    0    0  00000000

  +0   244791
  +2   244793
  +4        0
  +6        0    0    0    0    0    0    0    0    0  000000000

  +8        0    0    0    0    0    0    0    0    0  000000000
 +10        0    0    0    0    0    0    0    0    0  000000000
 +12        0    0    0    0    0    0    0    0    0  000000000
```

```
x0.16804
  -4 -8388607
  -2  8388606
  +0        0
  +2  s
 +10  .........1......1...1111       4  143
 +12  .1.......................
 +14    16818
 +16    16818
 +18    16822
 +20    16822
 +22    32304
 +24   262144
 +26     3    1
 +28  ............111111111111       0 4095
 +30        0
 +32  ....1...................     128    0
 +34  1..1111111111111111111
 +36    32304
 +38    37080      9  216    0 144 216    0  9  3 24  00110330
 +40    35992      8 3224    0 140 152    0  8 50 24  00106230
 +42        0      0    0    0   0   0    0  0  0  0  00000000
 +44        3      0    3    0   0   3    0  0  0  3  00000003
 +46        0
 +48    34090
 +50        0.
 +52      116
 +56              40448
 +60                  0
 +64     1767260780000
 +66    16818
 +68 -8388607
 +70  8388605
 +72 -8388607
 +74  8388605
 +76 -8388607
 +78  8388605
 +80     0    0
 +82     0    0
 +84     0    0
 +86     0    0

w0 =        1

w0 =        2

ba.239824
  +0  rs.w3 0     239824
  +2  al w0 11
  +4  al.w1 -4    239824
  +6  ac w2 x2+0
  +8  jl.  (-8)   239824

ba.239824
  +0   236310
```

```
w0 =        11
w1 =    239824
w2 =   -231990
fp.231990
```

9. Mainstat

9.1 Introduction

The program is a diagnostic tool for printing statistical information and testoutput from an RC8000 or an RC9000-10 main process.

The program may print directly from the memory as well as from a file containing a memory dump. When printing directly from memory the generation of testoutput is disabled for a moment.

9.2 Call

```
[<outfile> =] mainstat ,
{[test.<yesno>] [dump.<dumpfile>]{<main>}1-*}1-*

<main>        ::= main.<mainname>/
                  addr.<mainaddr>/
                  devno.<devno>/

<mainname>    ::= name of main process

<mainaddr>    ::= address of main process

<devno>       ::= device number of main process

<dumpfile>    ::= name of system dump file

<yesno>       ::= (yes / no)
```

9.3 Function

If an outfile is specified, the program writes on outfile, else on current output document.

If called without parameters, the program displays its own call syntax.

The main process, specified either by name, address otr device number, are handled one by one.

If dump.<dumpfile> is specified, it is supposed to contain a memory dump with memory address 0 in area address 0, and the main processes are searched in the dumpfile.

If test.yes is specified, testrecords are displayed for all the main processes.

For RC8000/FPA main processes, the program finds the receiver and transmitter processes associated with the main process specified.

The program extracts and prints on current output the statistics, first from the receiver process, then from the transmitter process description.

For other RC8000 main processes and for RC9000-10 main processes the program acquires and prints the statistics from the main process.

If the call specifies output of the testrecords, the generation of testoutput to the test buffer is disabled while the test mask and the test buffer are inspected and enabled again with the same testmask (in case of testrecords from the memory), even in case of abnormal program termination.

The testrecords of the buffer are printed in FIFO order as indicated by the buffer pointers.

9.4 Examples

Example 1

The following command extracts the statistics and the testrecords from the RC9000-10 main processes named 'iocmain1' and 'lanmain1' and prints it in the file 'outfile':

```
outfile = mainstat test.yes main.iocmain1 main.lanmain1
```

Example 2

The following command extracts the statistics and the testbuffer from the main process with the address 20786 in the memory dump contained in the file 'coredump':

```
mainstat test.yes dump.coredump addr.20786
```

9.5 Error Messages

*****mainstat, error in call**
the call is not syntactically correct

*****mainstat, no testbuffer**
the main process specified has no testbuffer

*****mainstat, mainprocess unknown**
the main process specified does not exist

*****mainstat, main process not found**
- the kind of main process specified and found is not correct (neither 80 nor 20)
- the memory dump area could not be found or the address points outside the memory dump area
- the address of the main process specified was not found in the name table in the memory dump.

*****mainstat main missing <param>**
no main process was specified

10. Makelink

10.1 Introduction

The program creates links, i.e. external processes with links to LAN device handlers, either on ADP(RC8000) or on LAN (RC9000-10). The link either allows access by many users (static) or access by one user only, the calling process (dynamic).

10.2 Examples

Example 1

The call :

```
makelink printer.printer.rc93100200
```

will create a link, i.e. an external process, `printer` with a link to a LAN device named `printer` on LAN controller no. 1. The link will be open for access by all internal processes that are users of the main process on an exclusive basis. The external process will be selected from the first free ones, and the corresponding device number, as well as the name of the main process will be displayed in the output from the program :

```
main    : lanmain1 :

    link : printer dev.no :  9 ---> printer lan : rc93100200 connected

links created      : 1
```

Example 2

The call

```
makelink 1.2 users.one 3270in.gin1.43 3270out.gout1.44
```

will create two links, i.e. external processes, `gin1` and `gout1`, with links to 3270 input and 3270 output device handlers, respectively, on the LAN controller no. 2. The links will only be open for access by the calling process itself only, and they will vanish with the calling process, if not

removed before. The external processes chosen will be numbers 43 and 44, supposing they are free :

```
main    : lanmain2 :

    link : gin1        dev.no : 43 ---> 3270in     index :  0
    link : gout1       dev.no : 44 ---> 3270out    index :  0

links created      : 2
```

10.3 Call

```
[<outfile>=]   makelink ([1.<lanno>] ([users.(all/one)]   ,

(<type>(<devspec>)1-*)1-*)1-*)1-*

<devspec> ::= (.<name>.<devno>)
              (.<name>         )
              (buf.<buffers>   )
              (.<devno>.<name>)
              (.<devno>        )

<type>    ::= console/printer/3270in/3270out/
              floppy/imc/streamer

<buffers> ::= integer, if 0 then default,
              default : buffer claim

all/one   ::= default : all

<lanno>   ::= number of LAN controller, default : 1
<name>    ::= (<devname>/<devname>.<landev>) (.log)

<devname> ::= name   of external process, default : wrk-name
<devno>   ::= number of external process, default : first free
<landev>  ::= name   of LAN      device
```

10.4 Function

For every <type> specification, one or more device specifications follow, either in pairs (<name>.<devno> or <devno>.<name>) or in singles (.<name>.<name>. or .<devno>.<devno>). For the types console and printer, the <name> part must consist of the pair : <devname>.<cspname>. Whenever a pair is met, it specifies both a device name and a device number to be used, whenever a single is met, it specifies either a device name or a device number to be used.

For each device specification, a link is created, using the main process specified (a main process specification is valid until the next one) and the 'users' specification (which also is valid until the next one).

If the device specification is without a device number, the first free external process will be used. If it is without a device name, a wrk-name will be generated and used.

If a buf.<buffers> specification is met in an IMC type specification, the value of <buffers> will be used as number of buffers to be allocated to the port. The buffer specification is valid until the next buffer specification, or until a new IMC type specification.

If no buffer specification is made, or <buffers> is not positive, the number of free message buffers of the calling process is used as <buffers>.

If no LAN number is specified, number one is supposed, i.e. the main process 'lanmain1' is supposed.

In case of 'users.all', the link will be generally available to internal processes, in case of 'users.one' only to the calling process.

Two kinds of <type> parameters, the 'console' and the 'printer' types must have an extra specification, the <landev> specification, right after the <name> part of the <devnane> specification. The <landev> parameter is used as name of the LAN device to be used by the device handler. If the link is created, the program tries each three seconds in approx. 1 minute to send a sense operation to the link. If it returns ok, the link is considered connected, if not it is not connected, but the link remains.

For the 'console' type, an optional parameter '.log' may follow the device specification. Its function is: if the link is connected, and if a file 'slogarea' exists, its contents are output to the device just connected.

The <type>'s '3270in' and '3270out' need a special commment : they should be given in pairs, first the 3270 input and then, at the latest before another 3270 input specification, the 3270 output specification should be given. The reason is, that the two of them must use the same device index of the device handler to which the link is requested, and the program sees to it, if the above rule is obeyed.

For each link requested, a line is produced on current output, stating the result of the request, either :

```
link : <device name> dev.no : <devno> ---> <type> <inf1> <inf2>
```

or :

```
no link : <device name> dev.no : <devno> +++> <type> *<explanation>
```

If successfull, <inf1> and <inf2> will be :

console/printer

<inf1> LAN device name as specifed in the call
<inf2> the text 'connected : +' or 'connected : -'

imc

<inf1> maxsendsize, i.e. the maximum number of characters to go into each block to send or receive, incl. a possible header

character. Exceeding characters will be lost.
<inf2> number of buffers allocated to the port

3270in/3270out
<inf1> device index on the device handler
<inf2> nothing

others
<inf1> nothing
<inf2> nothing

The program terminates, giving the number of links created.

If unsuccessfull, cf. below in 'Error Messages' concerning the ***<explanation>**.

If an <outfile> is specified, it is used as for the output, else the current output document is used.

10.5 Error Messages

Two kinds of error messages may be given, **parameter warnings** or **operational alarms.**

Parameter warnings come from disobeying the call syntax, and they take the form :

***** makelink <explanation>**

where **<explanation>** may be one of :

```
- outfile param connect impossible <actual parameter read>
- l.<lanno>:                       <-      -         -   >
- users.<name>:                    <-      -         -   >
- users.(all/one) :                <-      -         -   >
- <type>.(<name>/<integer>)        <-      -         -   >
- lan name missing :               <-      -         -   >
- unknown parameter :              <-      -         -   >
```

In case of parameter warning, the fp mode bit 'warning' is raised and the program continues with the next parameter.

Operational alarms come from :

- problems indicated in the answer to the 'create link' request
- dummy answer from the main process to the 'create link' request
- dummy answer from the imc process to the 'alloc descr' request
- problems creating the external process

and they take the form :

***<explanation>**

terminating the line in the output concerning the link request (no link : <device name> ...).

<explanation> is one of the below : **comment :**

Problems indicated in the answer to the request :

- no free external process	(or the one specified is not free)
- no free device handler **- unknown status**	(the main process specified is not a main process capable of creating a LAN link)

Dummy answer to link request :

- lan/ext. not user/not res.	(the calling process is not a user or cannot reserve the process representing the LAN controller (only imc))
- lan/ext. unintelligible	(e.g.the device number specified does not exist)
- lan/ext. malfuntion **- lan/ext. does not exist**	(as 'unknown status') (e.g. the main process specified does not exist)

Problems creating the external process :

- function forbidden	(the calling process must have function bit 4 = 1 shift 7 set)
- calling process is not user **- name conflict**	(e.g. the device name exists with the same name base)
- no such device number	(prevented by 'unintelligible')
- reserved by another process	(prevented by 'no free external')
- name format illegal	(cannot occur)

In case of operational alarm, the fp modebit 'ok' is cleared and the program continues with the next parameter.

11. Montest

11.1 Introduction

This chapter describes the use and function of the program montest, used to display monitor tables and other data structures in the monitor.

Section 11.1 is a short introduction to the program. Section 11.2 gives the call conventions and Section 11.3 gives the function and the detailed conventions for the various commands to the program. Section 11.4 gives the minimal resource requirements to run the program.

The program may execute in two different modes, either as

- a normal utility program taking its parameters from the call, or as
- a conversational program taking its parameter from current input zone.

The program may switch from one mode to the other any number of times, governed by commands in the call and in the current input zone. The program may at any time transfer all of its segments to memory and stay memory resident during further execution, thereby being independent of the backing storage system and segment i/o.

The program may at any time switch to take the monitor information from memory directly or from any backing storage file containing a memory dump.

During display in the conversational mode of certain data structures (internal, external and area process descriptions, message buffers and chaintables) it is possible to

- stop further execution while the latest data structure is inspected.
- repeat the display of the latest structure
- direct current output to maybe continue writing in any file before the next structure or before a repeat of the latest one
- redirect current output to the primary output to continue inspection in the conversational mode
- leave the data structure to accept further commands.

The program is able to help the user by displaying the repertoire of commands or by displaying information concerning the single commands on request.

The program will interprete the commands, one by one, until the end of the parameter list in the call is reached.

11.2 Examples

Example 1

The call

```
montest area user. myself
```

will display on current output all the area processes to which the internal process 'myself' is a user.

Example 2

The call

```
montest buf sender. myself
```

will display all the message buffers in the message buffer pool in which the internal process 'myself' is recorded as sender.

Example 3

The call

```
montest typein
```

will make the program enter the conversational mode and the program will take input from the terminal.

Now any command may be typed, e.g.

```
external devno.10.used buf sender. myself
```

The program will now display the process description for the external process with the device number 10. Following the display will appear the prompt character:

```
>
```

telling that the program is ready for a directive. Now every empty directive (RETURN) will make the next external process appear on the screen followed by the prompt character, until all processes actually in use have been displayed.

The directive

```
> o outfile
```

will make the program write on the terminal:

```
* o outfile
```

connect current output to the file 'outfile' and another prompt character will appear.

The directive

```
> r
```

will cause the last external process description seen on the screen be repeated, this time in the file 'outfile'. When done the prompt character will appear on the screen again, ready for another directive.

The directive

```
> o c
```

will make the output reappear on the screen. If the file just used should be connected again, writing in the file will be continued, even in backing storage files.

The directive

```
> f
```

will make the program quit the command 'external' yet before all processes have been displayed.
When the command 'external' is left, either because all have been displayed or because of the directive 'f', another command is read from current input, here the command

```
buf sender. myself
```

and the sequence starts all over until the command 'buf' is left.

When no more commands are found in the line, another line is read from current input. This time type:

```
end
```

and the program will leave the conversational mode and take the next parameter in the call.
Since no parameter appear after the 'typein', the program will terminate.

Example 4

The command

```
commands
```

whether in conversational mode or not, will display the repertoire of commands on current output, and the command

```
info <command>
```

<command> being any of these, will display a short description of its use.

11.3 Call

```
[<outfile> -]  -  montest (<command>) 0-*

<command>- { commands           }
           { info       <param> }
           { typein             }
           { end                }
           { dump       <param> }
           { core lock          }
           { internal   <param> }
           { external   <param> }
           { area       <param> }
           { chain      <param> }
           { buf        <param> }
           { veri       <param> }
           { format     <param> }
           { extra      <param> }
           { lines      <param> }
           { 0          <param> }
```

11.4 Function

If <outfile> is specified, current output is connected to <outfile>.

Now the program gets the next command with optional parameters and executes it, until the end of the parameter list in the call is met.

11.4.1 Commands

The command is

```
commands
```

with no parameters.

The program lists on current output the repertoire of commands.

11.4.2 Info

The command is

```
info <command>
```

where <command> is the name of one of the commands listed in 11.3.

The program displays on current output a short explanation of the parameter syntax and the function of the command.

11.4.3 typein

The command is

```
typein
```

with no parameters.

If the program is in the conversational mode, the command has no effect.

If the program is not in the conversational mode, it enters the conversational mode, i.e. reads the next line from current input zone and gets the first command from the line. The program prepares a return to the next command in the call to be executed when conversational mode is left again.

In conversational mode, the display of records from each of the data structures internal, external and area process descriptions, message buffers and chaintables will be followed by the output of a prompt character (the character >) on current input.

The first letter of the first parameter in next line in current input is interpreted as one of the following directives:

r Skip the rest of the line in input and repeat the latest display unchanged.

o The next parameter in current input will be handled as a file name, the rest of the line after the filename will be skipped. Current output will be stacked for later continuation and connected to the file specified. If the file does not exist, a backing storage area of that name will be created. If the file has been connected before, it is reconnected for continued writing. (a maximum of 10 different files ready for reconnection is allowed).
Now a prompt is output and the next directive may be given.

f The rest of the input line is skipped and the display of records stops. The next command will be executed.

anything else or empty line:
The rest of the input line is skipped. The next record is displayed. If the data structure is emptied, the next command is executed.

11.4.4 End

The command is

```
end
```

with no parameters.

If the program is not in conversational mode, the command has no effect.

If the program is in conversational mode, the mode is left and the next command in the call after the command 'typein' is executed.

11.4.5 Dump

The command is

```
dump <dumparea>
```

where <dumparea> is the name of a backing storage file containing a memory dump.

The program sets the memory dump mode, disconnects from a possible file, connects to the file specified and reads the monitor key variables from the file. Any succeeding command concerning data structures in the monitor will concern the data structures in the memory dump contained in the file until the memory dump mode is left again or another file connected.

11.4.6 Core

The command is

```
core
```

with no parameters.

The program will set the core mode, and get the monitor key variables from locations in the memory.

Any succeeding command concerning data structures in the monitor will concern the data structures in the memory until the memory dump mode is entered.

11.4.7 Lock

The command is

```
lock
```

with no parameters.

The program transfers all of its segments to memory and locks them, i.e. the entire program stays memory resident until it terminates.

11.4.8 Internal

The command is

```
internal all/used/free/name.<name>
```

where

```
<name> ::= name of an internal process.
```

The program will display a number of lines from all internal process descriptions, all used internal process descriptions, all free ones, or the one specified by name.

The number of lines is by default all, but may be set otherwise by the command 'lines' (11.4).

11.4.9 External

The command is

```
external ( all / used / free / kind.<kind> /
main.<mainname> / user.<user> / reserver.<reserver> /
name.<name> / devno.<devno> / devno.<devno>.all )

<user>    ::=
<reserver>::=
<name>    ::= name of an internal process
<devno>   ::= positive integer
```

The program will display the external process descriptions specified.

`all`	all external processes
`used`	all external process descriptions in use
`free`	all not in use
`kind.<kind>`	all external processes of the specified kind
`user.<user>`	all external processes with <user> as user of the process
`reserver<reserver>:`	all with <reserver> as reserver of the process
`main.<mainname>`	all external processes with the specified main process as main
`name.<name>`	the one with that name
`devno.<devno>`	the one corresponding to the device number
`devno.<devno>.all`	the same and all succeeding

A number of lines in extension to the process descriptions may be displayed, the number controlled by the command 'extra' (13.4) and the display format controlled by the command 'format' (13.4).

11.4.10 Area

The command is

```
area all / used / free / kind.<kind> / / user.<user> /
reserver.<reserver> / main.<main process / name.<name>
```

The program will display the area processes specified. The specifications equal the specifications for the command 'external'. A number of lines in extension to the process descriptions may be displayed as for external processes.

11.4.11 Chain

The command is

```
chain (all / docname.<name>)

<name> ::= name of a logical disk
```

The program will display the chaintables specified.

all	all chaintables
docname.<name>	the one with the document name specified.

11.4.12 Buf

The command is

```
buf (all / used / free / / sender.<name> /
receiver.<name> / sender.<name1>.receiver.<name2> /
receiver.<name2>.sender.<name1> / addr.<integer> /
addr.<integer>.all)

<name>    ::=
<name>    ::=
<name>    ::= name of a process
<integer ::= positive integer
```

The program will scan the message buffer pool and display the message buffers specified.

all	all message buffers in the entire pool
used	all message buffers of the pool which are in use
free	all message buffers of the pool which are not in use
sender.<name>	all the ones with <name> as sender
receiver.<name>	all the ones with <name> as receiver
sender.<name1>.receiver.<name2> receiver.<name2>.sender.<name1>	all the ones with <name1> as sender and <name2> as receiver

addr.<integer>	the one with the address specified
addr.<integer>.all	the same and all succeeding

11.4.13 Veri

The command is

```
veri <first addr> [.<no of halfs>]

<first addr>   ::=
<no of halfs>  ::= positive integer
```

The program will display the contents of the words specified in a format controlled by the command 'format' (11.4.14).

<first addr>	the word addressed by <first addr>, i.e. if odd then <first addr>-1
<no of halfs>	: as many words as given by <no of halfs>//2

The words specified should be kept below 'first free', i.e. the first address of the first process created by s and certainly cpa of the process must be above the words to be printed.

11.4.14 Format

The command is

```
format <format> { .<format> }0-*
```

where

<format> ::= {integer/ octal / half / byte / bit / text / all / code}

The programs sets a format used in the display by the commands 'veri' (11.4.13) and 'external' (11.4.9) or 'area' (11.4.10) in connection with 'extra' (11.4.15).

The format is valid until changed by another format command. Default is all except code and bit. 'All' means all except code.

11.4.15 Extra

The command is

```
extra <integer>
```

where

```
<integer> ::= positive integer
```

The program sets an internal value controlling the number of lines displayed in the extension of external and area process descriptions (11.4.9 and 11.4.10).

The value is valid until changed by another 'extra' command. Default is zero.

11.4.16 Lines

The command is

```
lines <first line> [.<last line>]
```

where

```
<first line>  ::=
<last line>   ::= positive integer
```

The program will set the line interval displayed in internal process descriptions (11.4.8). The lines are numbered 1,2,3..., line no of last line of bs claims.

First line will be set no lower than 1.

Last line will be set no higher than line no of last line of bs claims.

The values of first and last line are valid until changed by another 'lines' command.

Default is (1, max integer).

11.4.17 O

The command is

```
o <name>
```

where

```
<name>   ::= the name of a file descriptor
```

The program will stack current output for later reconnection and connect it to the file with the specified name.

If the file does not exist, a backing storage area of that name will be created.

If the file has been connected before, it will be reconnected for continued writing (a maximum of 10 different files ready for reconnection may exist).

The command is a command equivalent to the directive of the same name described in 11.4.3, typein.

11.5 Resource Requirements

The program may run in a process of only 11000 (RC9000-10: 12288) halfwords, conversational mode or not.

In case the lock command (11.4.7) is used, the size must be at least 88900 (RC9000-10: 90112) halfwords.

The process will need only 3 message buffers and only 3 area processes, unless the dump command (11.4.5) is used, in which case the process needs 4 area processes.

11.6 Error Messages

The messages from the program should be self explaining.

12. Printzones

12.1 Introduction

In the permanent part of the runtime system of ALGOL and FORTRAN programs are placed a number of key variables to the system, cf. ref. [1].

In the stack of running ALGOL and FORTRAN programs are placed all the active zones of the program, chained together down the stack in the reverse order of which they were allocated in the stack.

If context blocks or activities are allocated in the stack, descriptor records for each context block or activity are allocated as well.

The present program prints values in these data structures either from a program residing in memory or from a dumped memory image in a file, provided the process has been stopped or the file contains an image of the memory of a stopped process running an ALGOL/FORTRAN program.

The program is intended for inspection of runtime system variables and/or zone and share descriptors, context block descriptors and activity descriptors in ALGOL and FORTRAN programs for debugging purposes.

The program being inspected must have been translated by:

- the ALGOL compiler in SW8500/1, release 11.0, 1979.11.12 or newer, or
- the FORTRAN compiler in SW8501/2, release 1.0, 1983.09.01 or newer.

If the program to be inspected executes in an internal process, the following must be fulfilled:

- the program must have been stopped by its parent
- the CPA of the calling process must permit read access to the internal process to be inspected

- the address base of the calling process must permit read access to the internal process to be inspected
- the internal process to be inspected must not be a process swopped out.

If the program to be inspected is found in a memory picture in a file, the file must contain the entire process dumped.

12.2 Call

The program is called:

```
[<outfile] - printzones [<procname>]
```

where

```
<outfile>  ::=
<procname> ::= a name of at most 11 characters
```

12.3 Function

If an <outfile> name is specified in the call, the program connects its output zone to a file of that name, i.e. if no such file exists, a backing storage area is created and the zone is connected.

If no <procname> is specified the program uses the default name 'image'.

If an internal process exists of the name specified in <procname>, the program connects its input to the process.

If no process exists, a file descriptor of the name is looked up in the catalog. If it exists, the input zone is connected, otherwise the program terminates with an alarm.

Now the key variables and the zone chains/descriptors are input and the values printed.

12.4 Examples

Example 1

The call

```
printzones
```

will try to print from a file named 'image'.

Example 2

The call

```
outfile = printzones dumpfile
```

will try to print on the file 'ourfile' from a file named 'dumpfile'.

Example 3

The call

```
printzones fgs274001
```

will try to print from a process named 'fgs274001'. If no such process exists, it will try to print from a file of that name, and if that fails it will terminate.

12.5 Error Messages

*****printzones connect <outfile> <cause>**
The program could not connect current output zone to <outfile> for the reason stated in <cause>. The program continues with current output connected as before the call.

*****printzones call error as <explanation>**
no run 0.....
called from...

<explanation> = 'no file or process name'
The program was called with an integer as parameter.

<explanation> = 'internal process invisible'
The address base of calling process does not permit read access to the internal process to be inspected, or the processes have some primary store in common. In this case the process to be inspected is either an ancestor not allowing read access through address base of calling process, or it is a process swopped out of memory at present.

<explanation> = 'internal process unreadable'
The CPA of calling process does not permit read access to the internal process to be inspected.

<explanation> = 'no coredump area exists'
No file descriptor with the name given in <procname> exists.

In all cases the program terminates with the fp mode bits: warning.yes, ok.no

give up 0 algolcheck
called from ...
***** device status <file>**
stopped
process does not exist
The executing process has too few area processes or the file descriptor given in <procname> describes a non existing area. The program terminates with fp mode bits set: warning.yes, ok.no

block 2 recprocs6
called from ...
The file given by <procname> does not contain a dump of the entire process to be inspected.
The program terminates with fp mode bits set: warning.yes, ok.no

13. Scatup

13.1 Introduction

The program `scatup` handles the s-user catalog `susercat`.
The format of `susercat` is shown in ref [3].
The program may be used to :

- create a new s-user catalog
- insert entries in the s-user catalog
- list the contents of entries or all of the s-user catalog

The output from the program when listing entries from the catalog may be used directly or after modification as parameters to a new call of `scatup` to insert entries.

13.2 Creating a new s-user catalog

Syntax of call:

```
scatup newcat.(size) (disk name)1-*
```

13.3 Inserting entries in s-user catalog

Syntax of call:

```
                        (      <disk name>. <slices>.<entries>)0-*
                        (perm.[<disk name>].<slices>.<entries>)
                        (temp.[<disk name>].<slices>.<entries>)
                        (prio. <integer>                       )
                        (comm. <integer>                       )
                        (buf . <integer>                       )
                        (area. <integer>                       )
                        (inter.<integer>                       )
scatup insert.<name>    (func. <integer>                       )
                        (std.  <lower>.<upper>                 )
                        (max.  <lower>.<upper>                 )
                        (user. <lower>.<upper>                 )
                        (addr. <integer>                       )
                        (size. <integer>                       )
                        (prog. <prog>                          )
```

Negative bases may be represented as n.<integer> as well as -<integer>.

If no disk name is specified, the first disk will be used.

The default values for the parameters are:

```
<first disk>.5.5
buf. 4 area.6
std. 8388605.8388605
max. 8388605.8388605
user.8388605.8388605
size.12800 prog.fp
```

All others fields are set to zero.

13.4 Deleting entries in s-user catalog

Syntax of call:

```
scatup delete.(<name>)
```

13.5 Listing entries in s-user catalog

Syntax of call :

```
            (cat   )
scatup list.(<name>)
            (all   )
            (names )
```

The options will list :

- entry zero
- the entry given
- all entries
- the disk names and the names of all entries

from the s-user catalog.

The following example will create and initialize an s-user catalog, insert two entries and list them :

13.6 Example

```
susercat = set 21 0 d.0 0 0 11.0 0
scope user    susercat

scatup newcat.       50,    catsize
disc         ,
disc1        ,
disc2        ,
disc3

scatup insert.adm,
,*****************
prio. 0 comm. 0,
buf. 12 area. 10,
inter. 0 func. 1776,
std .        0.        9,
user.        0.        9,
max . -8388607.  8388605,
addr. 0 size.  80000 prog.fp,
,resource         slices   entr              slices   entr,
temp.disc        .     0.     0   perm.disc  .     0.    10,
temp.disc1       .     0.     0   perm.disc1 .     1.    10,
temp.disc2       .     0.     0   perm.disc2 .     1.    10,
temp.disc3       .     0.     0   perm.disc3 .     1.    10

scatup insert.maxbase,
,*********************
prio. 0 comm. 0,
buf. 12 area. 10,
inter. 0 func. 1776,
std . -8388607.  8388605,
user. -8388607.  8388605,
max . -8388607.  8388605,
addr. 0 size. 200000 prog.fp,
,resource         slices   entr              slices   entr,
temp.disc        .     0.     0   perm.disc  .     4.    22,
temp.disc1       .     0.     0   perm.disc1 .     1.    10,
temp.disc2       .     0.     0   perm.disc2 .     3.     2,
temp.disc3       .     0.     0   perm.disc3 .     5.     3

scatup list.names
end
```

14. Slicelist

14.1 Slicelist

The utility program slicelist can provide you with a survey showing the slices constituting a backing storage file and, for RC82xx disks on RC8000), how these are distributed on cylinders.

14.2 Example

If a user wants a survey of the slicenumber of his file 'myfile', then the FP command

```
slicelist myfile
```

is sufficient. If he also wishes to know how the slices are distributed on the cylinders, he will have to use the FP command

```
slicelist cylinder.yes myfile
```

(only for RC82xx disks on RC8000)

14.3 Syntax

```
                                                              1-*
[<outfile> =] slicelist ( {<modifier>}1-* <filename> )

<modifier> ::= ( cylinder )
               ( segment  )  . (yes/no)
               ( slice    )
```

14.4 Semantics

<filename> The name of a file stored on an RC9000 disk or on an RC82xx/RC83xx disk on RC8000.

<modifier> The <modifier>s are used to determine the surveys to be output. An occurence of a <modifier> applies to all the following occurences of <filename>. Their initial values and their associated surveys are described below.

cylinder (Only RC82xx disks). Default is cylinder.no. If cylinder.yes is specified, a survey will be output showing how the slices are distributed on cylinders. The cylinders are numbered from zero, cylinder number zero being the first cylinder of the logical disk on which the file is stored.

segment Default is segment.no. If segment.yes is specified, a survey of the numbers of the first segments in each of the slices constituting the file will be output. The segments are numbered from zero, segment number zero being the first segment of the logical disk on which athe file is stored. The possible difference between the numbering of segments on odd numbered cylinders and even numbered cylinders is not taken into account.

slice Default is slice.yes. A survey of the numbers of the slices constituting the file is output. If slice.no is specified, the survey is suppressed.

<outfile> The name of a file. If an <outfile> is present in the program call, this file is used for program output (including possible error messages) - otherwise the current output file is used. If <outfile> is not found in the catalog, a backing storage area of that name with scope temp is created on the main disk.

14.4 Program Output

For each filename the corresponding catalog entry is output using the same layout as the one used by the program lookup. Furthermore, some characteristics for the logical disk and for the physical disk, on which the file is stored, are output:

```
logical disk
      slicelength
      the number of the first segment

physical disk
      the address (in memory) of the process description
      the number of segments per cylinder (only RC82xx)
      the odd cylinder shift (only RC82xx)
```

Finally, depending on <modifier>, one or more of the following three surveys are output:

- slicenumber
- the number of the first segment of each of the slices constituting the file

- the distribution of the slices on cylinders (only RC82xx)

14.5 Error Messages

*****slicelist <outfile> cannot ne connected**
the chosen output file could not be connected, the current output file is used instead.

*****slicelist <filename> cannot be looked up**
no entry in the main catalog corresponding to <filename> can be found.

*****slicelist <filename> is not a bs-area**
the catalog entry corresponding to <filename> does not describe a backing storage area.

*****slicelist <filename>, <devicename> intervention**
the device <devicename>, on which the file <filename> is stored, is not currently accessible due to some error.

*****slicelist <filename>, <devicename> is not an RC92xx/ RC82xx/-rc83xx disk**
the device <devicename> is not an RC92xx/ RC82xx/RC83xx disk, and no slicelist can be produced for the file <filename>.

*****slicelist param: <erroneous parameter> illegal**
parameter error in the call of slicelist. The <erroneous parameter> is printed.

An error message does not cause the program to terminate, the program continues with the next parameter in the list.

Appendix A. References

Disccopy

1)	*System 3 Utility Program, Part one*	991 02567
2)	*Monitor Reference Manual* Part of SW9890I-D, Monitor Manual Set	991 11259
3)	*Operating System s, Reference Manual* Part of SW9890I-D, Monitor Manual Set	991 11260

Printzones

1)	*Code Procedures and Run Time Organization of ALGOL Programs.*	991 11296