ISSN 0105-8525

The RIKKE BCPL System

by

Jens Kristian Kjærgaard
and
Flemming Wibroe

DAIMI MD-38
September 1980

Computer Science Department
AARHUS UNIVERSITY
Ny Munkegade – DK 8000 Aarhus C – DENMARK
Telephone: 06 – 12 83 55

ISSN 0105-8525

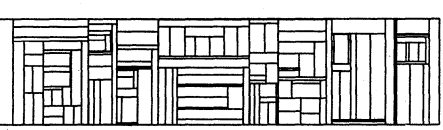
The RIKKE BCPL System

by

Jens Kristian Kjærgaard
and
Flemming Wibroe

DAIMI MD-38
September 1980

Computer Science Department
AARHUS UNIVERSITY
Ny Munkegade – DK 8000 Aarhus C – DENMARK
Telephone: 06 – 12 83 55



The RIKKE BCPL System

This document is a users manual to the BCPL operating-system on RIKKE.

The first 7 chapters of the manual are updated versions of

The RIKKE BCPL System,
A Programmer's Manual

DAIMI MD-22, Februar 1976

by Ejvind Lynning

The next 4 chapters describe the file-system, and are totally new.

The purpose of the document is to serve both as a users manual for the ordinary user, and as a reference handbook for the system programmer.
No attempt has been made to describe implementation details; however the source text of all system programs are available - and readable - and should together with this paper provide enough information for future system programmers to modify and extend the system.

DAIMI MD-38, september 1980

Jens Kristian Kjærgaard

Flemming Wibroe

1. Introduction.

The RIKKE BCPL system is an interactive single-user operating system which loads and executes BCPL programs[1] in an environment providing a range of facilities for hierarchical process organization, storage allocation, and input/output. It is a variant of the Oxford OS system[2,3].

(Througout this manual [i] will refer to reference i in the reference list, whereas {i} or {j.k} will be used for cross references to other sections within the manual itself.)

Parts of the system are implementations of OS concepts, though not generally verbatim copies from the Oxford code. It should be easy to modify programs that run under OS to run under the RIKKE BCPL system.

The system has been designed to be highly interactive; thus its basic loop is not a load-go loop, as in OS, but a command interpreter loop, and a significant part of the system has been devoted to facilities for interactive debugging and analysis of running programs.

The system runs on the RIKKE OCODE machine, which is implemented by means of a micro-programmed emulator[4], and uses the micro-programmed i/o-nucleus[5]. RIKKE itself is described in [6].

The first seven sections introduce various important concepts in the system. For most of these a familiarity with [2,3] would be helpful, although not absolutely necessary.

The next 3 sections describes the command interpreter, the file system, how to use it, and gives a list of all library routines available.

In chapter 11 all the system-programs available to the users are described, and finally the internal ASCII-character and a DeadStart guide set is given.

2. Storage utilization.

The linear store of the OCODE machine, 64K 16-bit words, is divided into the following areas: the global vector, the data area, the stack, and the code area. Each separately compiled segment of BCPL program contains a code block, which is loaded into the code area, and a data block, containing statics and strings, which is loaded into the data area. The code area is treated as a stack, whereas the data area is managed by the free store system, described below. The overall organization of storage is illustrated in fig. 2.1

Organisation of storage

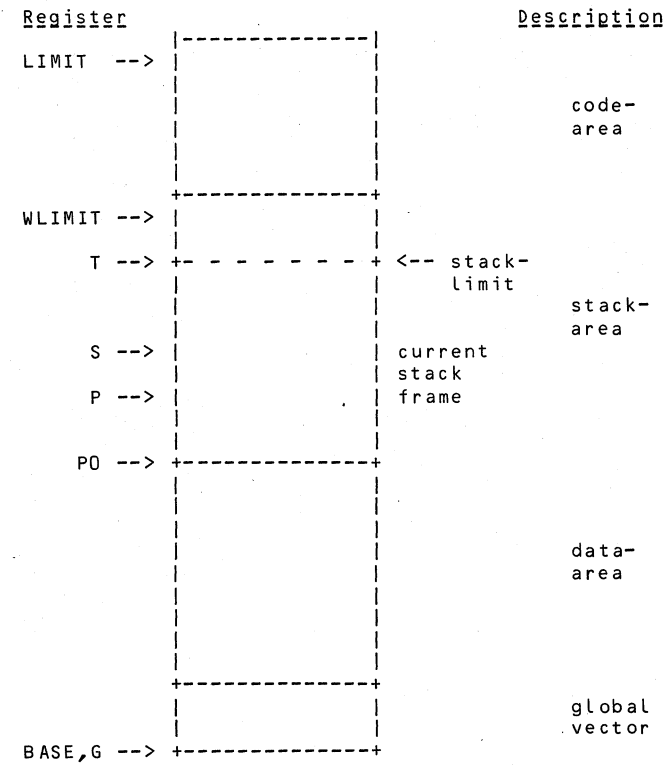


fig. 2.1.

The space between the T and WLIMIT registers is to allow the system to react nicely to a stack overflow situation. These two registers are administered by the loader and unloader, and it is seen that a trade-off exists between the amount of code that may be loaded at any given time and the available stack area. By the philosophy of the system all major blocks of storage required by a program should be claimed from the data area using the free store system. Thus the stack size is ordinarily a measure of the amount of recursion used rather than of data storage requirements.

In order to allow optimal use of the storage resources of the system, a facility (the setPO command) to change the stackbase (P0 register) has been included. This can be used to accommodate programs with heavy demands on either static data or dynamic data, but not both. The default value of P0 is 30000(decimal).

The global vector has been placed so that global numbers equal absolute machine addresses; this is useful to know if one wishes to dump the value of a particular global interactively.

The code area is write-protected to avoid accidental overwriting, particularly of system code. The registers can only be overwritten by system kernal routines. This enhances the robustness of the system; however by overwriting essential globals or system stream vectors in the data area, it is easily possible to crash the system totally.

2.1. The Free Store System.

The free store system is described in [2]. It provides a general facility for claiming and returning vectors (blocks, arrays) of storage using the routines NewVec and ReturnVec.
The lifetime of such a vector is independent of the stack discipline for procedure entry and exit, i.e. a vector claimed using NewVec lives until it is returned by ReturnVec, unless a finish occurs before this takes place {4}.

```
$(  let  v=NewVec[n]     yields in v a pointer to a vector of length
                         n+1 (v!0 through v!n).

    ReturnVec[v,n]       returns this vector.

$(  let n=MaxVecSize[]   yields in n the length of the largest
                         vector available.
```

It should be noted, that the system does no sort of Garbage Collection, so a great deal of fragmentation can arise.

3. Loading and Unloading.

There are two ways to load code, either via the command interpreter or under program control.
Code loaded under program control may be explicitly unloaded, or will be automatically unloaded when the Run{4}, in which it was loaded, terminates.
Using the command interpreter to execute a program the users code-module will be loaded and unloaded automatically. The name of the code-module will by the command interpreter be interpreted as a directive to load and execute the code{8}.

3.1. Information Blocks and Overlays.

For each segment of code loaded, an information block is kept in the data area. It describes the code and data blocks and is used for unloading. Information blocks are chained together to reflect the last-in-first-out discipline enforced on loaded code. Global 425, IBlock, always points to the most recently created, still existing information block. An IBlock value is used as a parameter to Unload to determine where to stop unloading.

Overlaying may be necessary for large programs due to the limited size of storage. It may be accomplished as follows:

```
$( let SaveIB=IBlock
   Load["overlay"]
   Proc[pvec]            // call of routine "Proc" in "overlay"
   Unload[SaveIB]
$)
```

However, it may be more natural to use the Run-mechanism, thereby avoiding explicit unloading. A small steering routine, which loads and calls the overlay, should then be used as the argument of Run:

```
Run[Steer,pvec]          // call of Run

let Steer(pvec) be
$(
   Load["overlay"]
   Proc[pvec]            // call of routine "Proc" in "overlay"
$)
```

3.2. Globals.

The discipline used for globals with respect to loading and unloading is as described in [2].
During the loading of a code segment, its global entry points are calculated and assigned to the proper globals. At the same time the old values of these globals are remembered, so that they may be restored when the segment is unloaded.

Immediately after system setup, all globals are saved in the protected code area. The saved values are written back by the reset command{8}, which also restores the code and data areas to their initial states. It may be called via the command interpreter.

4. The Run-Mechanism.

For a philosophical discussion of the Run-mechanism, consult [2]. In short, Running a routine (program) means: saving the state of the system, and then calling the routine so that when a finish occurs, its effect will be to restore the system in the saved state and continue execution from the point where Run was called.

In the hierarchical structure of a process Run may be used to mark and save general return points; it establishes Run-levels.

The notion of a state of the system to be saved and restored is explained in the following.

The Free Store System{2.1}:
When a Run is terminated (either by finish or merely returning) all storage claimed in that Run is forcibly returned. This is accomplished by initiating for each Run its own free store within the largest available vector and simply abandoning it wholesale when Run terminates. Thus after Run returns, the free store will be exactly as it was when Run was called. It is not possible inside a Run to return storage that was claimed at an outer Run-level.

PutBack{7.1}:
Items put back to streams do not survive across Run-boundaries, either into or out of a Run.

Loaded Code{3}:
All code loaded inside a Run is unloaded when the Run terminates.

4.1. The ClearUpChain.

Certain activities in the system or in a user program may require a terminating action, e.g. the writing of a file via a stream must be terminated by closing the stream. Such terminating actions may be forgotten in an error situation, or the user may even fail to include them in his program. To avoid possible nasty consequences, they may be placed in the ClearUpChain, in which case the system will see to, that they are carried out when the Run terminates. Notice that failing programs which give up cause termination of a Run.

The ClearUpChain is a list of two word blocks, each containing a

pointer to the next block, and a routine entry point:

ClearUp Chain

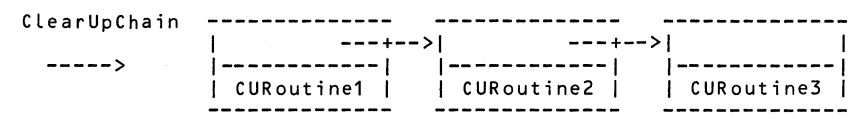


fig. 4.1.

The clearup blocks, initialized to contain routine CURoutine, is entered in the ClearUpChain by the call EnterCUC[C], where C is the address of the clearup block, and when the corresponding activity terminates normally, it should be removed by RemoveCUC[C]. However, if this is not done before the Run terminates, the general clearup routine will call CURoutine[C].

Notice that the two word block must be allocated by the program which uses the clearup facility, not by EnterCUC, and in fact it should be placed so that its address (C) informs CURoutine of exactly what it is to do.

The exact relationship of the ClearUpChain to the Run-mechanism is the following:
When a new Run is started, the handle of the ClearUpChain is stored, and a new, empty, ClearUpChain is initiated. When a Run terminates, all orders in the current ClearUpChain are obeyed, and the saved handle is used to reestablish the old situation.

5. Traps.

A general facility exists in the system for handling traps, i.e. situations when the normal flow of execution must be interrupted for some reason.

5.1. Trap Conditions.

The following situations cause traps to occur:

1. The OCODE emulator detects
   a) an error, e.g. stack overflow, or
   b) a routine or function entry while the trace facility {6.1}
      is switched on.

2. A system or user program detects a software failure. Upon detection of irreparable failure or merely a situation which might interest the operator sufficiently to engage him in conversation, any program may call Stop[F,p1,p2,....], which generates a trap. Stop displays its parameters on the console with the same parameter convention as OutF {7.8}

   A failing program may also merely give up, i.e. call GiveUp[F,p1,p2,p3,...], which displays the message F with parameters as OutF, and finish.

3. An i/o-interrupt. This is used for the console keyboard to allow the operator free interference with program execution. Other devices do not interrupt the system.

4. Actual operator interruption. The routine which is invoked by the general trap handler in case 3 usually just appends the incoming character to a buffer, but two characters have special effects:

   a) CTRL C causes a call of Stop, cf. case 2. The same effect may be obtained by switching KA off and on again.
   b) CTRL E causes a call of global 8, Interrupt {6.2}.
   c) CRTL R displays the keyboard buffer contents on the console
   d) CTRL S, CTRL Q and CTRL O are used to control the Console output {7.3}.

Other i/o-streams may be set up to include the interrupt facilities. When a transfer of a block of information is completed by the IOnucleous, the devicedependent interrupt handler, which is inserted in the Interrupt vector, will be executed.

The system function AllocintEntry(Int-handler) inserts the routine

Int-handler in an interrupt vector an returns a value, which must be placed in an i/o-controlblock{5}.

5.2. The Trap Handler.

The task of the general trap handler is to distinguish among these cases and take proper action. In cases 1.b, 3, and 4.b it suffices to call a further routine to handle the situation; in all other cases the operator is consulted, i.e. a message is displayed on the console, and a command interpreter loop entered. When used in this manner, in a trapped context, the command interpreter accepts a set of commands which differs somewhat from that accepted in an ordinary, not-trapped context{8}.

For the curious reader we add, that to carry out its task, the trap handler needs to know the values of the OCODE machine registers, the context block, at the time of trap. When the emulator encounters an error, it places the register values on the stack, inside a frame it sets up for the Trap routine, and then generates a call to it.

Stop, not knowing the register contents, guesses them as best it can, places them appropriately, and transfers control to Trap by a goto, so that its own stack frame becomes that of Trap. For all this to work, both the emulator and Stop must know the way in which Trap references its local variables. The scheme then works in complete harmony with the stack architecture of the OCODE machine; in particular nested interrupts are merely recursive calls of Trap[9].

6. Interaction.

A major design aim of the system has been to facilitate user interaction with running programs. The tools supplied for this purpose are the following.

6.1. The Trace Facility.

All routine and function entries (with parameters), that are compiled with trace-option [1], may be traced on the line printer. This facility is controlled by the trace command, which takes a boolean parameter to switch the trace on (true) or off (false) .

6.2. The User Interrupt Routine Facility.

Along with any program a routine may be included which can be called asynchronously by typing CTRL E on the keyboard. The routine may for example be used to monitor the program by displaying important information or to control program behaviour according to further interactive commands. It must be declared as global 8 (Interrupt){5.1}.

6.3. The Command Interpreter.

If the system is trapped by an emulator or system error, a user program call of Stop, or CRTL C typed on the keyboard, a special command interpreter loop is entered{8}. In the case of emulator or system errors it is not recommended to use the cont command; rather reset or end should be given after debugging information has been obtained.

In exploring the state of a program the stack and dump commands are particularly useful. In such a situation it is often useful to be familiar with the anatomy of a stack frame, which is therefore shown in fig. 6.1 :

A stack frame

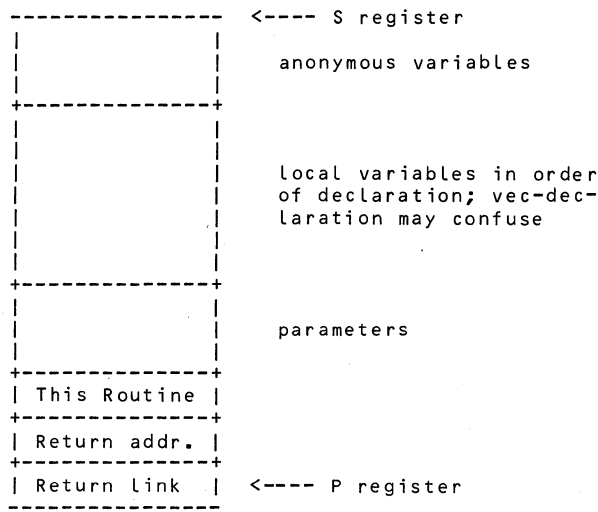


fig. 6.1.

The P register corresponds to the value of level displayed by the stack command.

7. Streams.

The apparatus for i/o in the system consists of streams [2]. A stream is either an input stream or an output stream. There exists a set of primitives applicable to all streams; these primitives are implemented as global routines.

7.1. Stream Primitives.

For input streams:

Next.
$(Let x=Next[S] yields in x the next item from stream S.

Endof.
Endof[S] has value true if the end of stream S has been reached, i.e. if no more items can be obtained from S using Next; otherwise Endof[S] is false.
For some input streams, Endof is difficult to implement sensibly. In such cases Endof always returns false and Next yields the value ENDOFSTREAMCH, when no more ordinary items can be obtained.

PutBack.
PutBack[S,x] returns the item x to stream S so that a subsequent call of Next will produce exactly x. Several items, which need not even have come from S, may be put back; they will be reproduced in a last-in-first-out fashion. See also {4}.

For output streams:

Out.
Out[S,x] results in the item x being output along stream S.

For both input and output streams:

Reset.
Reset[S] results in stream S being restored to an initial state, which depends on the particular stream in question.

Close.
Close[S] does any sort of terminating action, and relinquishes any storage associated with S. No other primitive may be applied to S after Close.

The items which are transported along a stream are typically characters (bytes) or words. But it is possible to build streams which yield or accept any sort of data structure, e.g. vectors or strings.

The system contains a number of standard streams, and some stan-

dard stream functions which produce new streams, using either an argument which may be an existing stream or file, or system information; the latter case includes stream functions for non-standard devices.

Streams may be classified into lowest level streams - those which communicate directly with the i/o-nucleus, and higher level streams - those which are built on top of lower level streams. A similar distinction applies to stream functions, since it applies to the streams they create.

A commented list of system streams and stream functions follows. The first six streams, Keyboard, Ptr, Ptp, Console, Printer, and LineBuffer are always available; they cannot be closed. Other streams must be created by the user program and should be closed when they are no longer needed.

7.2. Lowest Level Streams.

Keyboard.
The basic character input stream from the console keyboard. Endof is always false. Keyboard is unique among input streams in that PutBack is not allowed. Keyboard is intimately connected with the trap mechanism{5.1}. Characters spontaneously typed by the operator are stored in a circular buffer from which Next removes them.
Reset discards any buffer contents.
Striking CTRL R displays the buffer contents on console.

Ptr.
The basic byte stream from the RC2000 paper tape reader. Ptr uses a double buffer arrangement, as do Ptp and Printer. Endof always returns false , and ENDOFSTREAMCH are returned by Next when a tape has run out. Reset discards any buffer contents.

Ptp.
The basic byte stream to the paper tape punch. Reset outputs buffer contents.

7.3. Higher Level Streams.

The following two streams, Console and Printer, use nameless lowest level streams, which are invisible to the user. For all practical purposes they may be considered lowest level streams.

Console.
The basic character output stream for the console. It converts the internal ASCII character set{13} to console representation, and outputs one character at a time. Reset empties the Console buffer.
Striking CTRL S stops the Console output, until CTRL Q is

striked.
CTRL O inhibits Console output, until another CTRL O is striked.

Printer.
The basic character stream to the printer. It converts internal ASCII {13} to D200 line printer representation, and empties its buffers, not only when they are full, but also when a line feed is output, in order to speed up printing. Reset outputs buffer contents.

LineBuffer.
This is the commonly used stream for character input from the keyboard. It allows the operator to type and edit a whole line before its contents are taken seriously. By using Keyboard, LineBuffer reads characters up to a car return and treats them as follows:

1. ordinary characters (none of the following) are inserted in a buffer.
2. rubout causes the last character read to be deleted from the buffer and echoed to the console.
3. backspace causes the last character to be deleted from the buffer and from the line image on the console.
4. cancel, CTRL X cause the buffer to be discarded and a car return to be output on the console.
5. car return marks end of line.

The result of the call of Next, which caused the buffer filling, will be the first character in the buffer. Subsequent calls of Next will return the remaining characters in the buffer, until it is empty. Then Next will start a new buffer filling etc.

Reset for LineBuffer discards the buffer and resets Keyboard. Endof is always false.

7.4. Functions Yielding File Streams.

These functions may also be considered lowest-level. They all take a file-description (fd) {9} as an argument, and return a stream to read or write to the file.

InFromFile[fd] yields a word stream to read file fd sequentially. Reset on such a stream causes reading to continue from the beginning of the file.

BytesFromFile[fd] is similar to InFromFile, but it creates a byte stream, i.e. two subsequent calls of Next yield the two bytes of each word of the file. For compatibility Endof works the same way as for Ptr.

OutToFile[fd] returns a word output stream to write to file fd.

In order that the file description and the internal housekeeping information of the file body may be updated, it is critically important that the streams created by OutToFile or one of the three functions, which follow, are closed, when writing is complete.

In order to save at least some of the file output from disastrously failing programs, a Fail-Close routine for a file output stream [8], is inserted in the ClearUpChain{4.1}, and this Fail-Close routine will close it at the point where the last Reset took place.
The function of Reset is to save enough information about the file for Fail-Close to do this job properly. However, if neither Reset nor Close is ever called, the file description will not be changed at all, and for all practical purposes the stream might as well not have existed.

BytesToFile[fd] returns a byte output stream to write to file f. Files written using streams produced by BytesToFile may be read using streams produced by BytesFromFile.

OutToFile and BytesToFile create streams which overwrite a file; if, instead, it is desired to add information to an existing file, similar streams are produced by:

AddToFile[fd], and
AddBytesToFile[fd].

All these functions are combined to 2 parameterised functions, which can be used when reading and writing files in connection with directories {10.1.1} : ReadNamedFile and WriteNamedFile.

7.5. Higher Level Stream Functions.

WordsFromBS[S] yields a stream that reads words composed of two bytes from stream S. Initially and after Reset this stream will skip zero bytes (blank tape); Reset also resets S. Endof works for streams constructed by WordsFromBS, but it is inefficient.

WordsToBS[S] takes a byte output stream as a parameter and yields a word output stream, i.e. each word output to this stream will become two bytes to be consumed by the argument stream. Reset resets the argument stream. Close also closes the argument stream.

IntcodeFromRaw[S] takes a byte input stream as a parameter and returns a stream of internal ASCII characters{13}. It works as a filter for streams containing "strange" characters. Reset resets the argument stream, which is also closed by Close.

7.6. Buffered Streams.

The Next and Out primitive routines usually work by invoking a further stream-dependent routine, but for efficiency some streams use buffer arrangements. For these streams, the primitive Next and Out routines merely consume/insert an item from/in the buffer, and only when the buffer is empty/full, a further stream-dependent routine, the HandleBuffer routine, is called.

In order to speed up block-transfers on buffered streams, which in BCPL would be written as

```
$( for i=0 to n-1 do v!i:=Next[IS] $)
$( for i=0 to n-1 do Out[OS,v!i]   $)
```

two functions NextBlock[IS,v,n] and OutBlock[OS,v,n] have been written.
IS,OS are the input/output stream, v is the address of the block containing the elements in v!0 through v!(n-1).

The result of NextBlock is DONE if the stream could deliver n elements, otherwise the result is the number of elements missing. The result of OutBlock is always DONE.

The routines are implemented by a micro-programmed copying between the stream-buffer and v, including calls of HandleBuffer, so when used on non-buffered streams nothing is gained in speed, since this is similar to the for-loops above.

NOTE: NextBlock and OutBlock works only for byte- and word-streams, but can of course be used as Next/Out functions for a block-stream.

7.7. Stream Vectors.

This subsection is intended mostly for the programmer who wishes to write his own stream functions.

Streams are implemented by means of stream vectors, whose entries contain the routines and other items of information which are necessary for the stream primitives to work.

The general format of a stream vector is as shown:

A stream vector

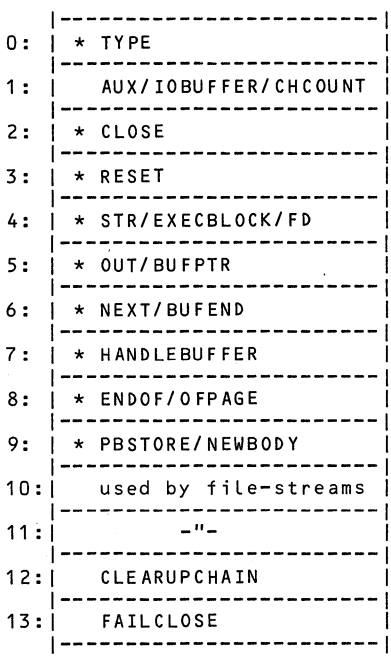


fig. 7.1.

It is possible to use stream vectors with more entries than shown, in fact only those entries marked * are used by the primitive functions. The entries Close, Reset, Out, Next, Endof must hold the routines, which implement the respective primitives for the stream in question. Where a primitive is not applicable, the global routine StreamError should be used.

The STR entry is ordinarily used to hold the value of a lower level stream on top of which the stream in question is built.

PBStore is used by PutBack, and the AUX entry has no standard interpretation.

HandleBuffer only has meaning for buffered streams, it contains the HandleBuffer routine. For buffered streams, Next/Out routines

are unnecessary, and their locations are used for buffer pointers.

In order to cooperate with the primitive Next, an input HandleBuffer routine should, after filling the buffer, adjust the pointers so that the BUFPTR entry points to the first item in the buffer, and the BUFEND entry points just after the last one, which does not necessarily have to be at the physical end of the buffer. Similarly, an output HandleBuffer routine, after outputting the buffer contents, should set the BUFPTR entry to point to the first word of the buffer, and the BUFEND entry to point just beyond the last word of the buffer. Notice that the pointers are absolute, not relative to the buffer.

The Reset routine for a buffered output stream may perform any desired action and should then set the buffer pointers exactly as described for HandleBuffer to enable a new buffer filling. For buffered input streams, Reset may sensibly set the two buffer pointers to be equal, thereby causing a subsequent call of Next to provoke a call of the buffer filling routine.

The TYPE entry of a stream vector contains the stream type, a word in which the bits are interpreted as follows.

Bit	Meaning
0	input stream
1	output stream
2	buffered stream
4	byte stream
5	word stream
10	PutBack allowed
11	lowest level stream

Only those bits which are relevant should be set in a stream type word. When resetting a buffered stream, also reset the type.

Apart from these bits, the system uses 4 more when operating on a stream:

Bit	Meaning
7	byte-stream: indicates left/rigth byte next
8	PB1MASK
9	PB2MASK, both used for PutBack
13	CTRL off/on: output mode for control characters

7.8. Formatted input/output.

Upon the Next/Out functions, there is build a set of input/output routines to help reading/writing integer, strings and characters. The parameters IS and OS are input- and output-streams respectivily:

Input:

NextCh[IS] : skips bytes that are all zeroes or all ones, and return the first byte which is neither, possibly ENDOFSTREAMCH.

NextN[IS] : reads a possibly signed integer from IS. any characters before the integer, but none after.

NextS[IS] : reads a string up to a string-delimiter: '*n','*s',',' or '.'. The delimiter is skipped.

ReadN[] : equivalent to NextN[LineBuffer] and skip the delimiter.

ReadS[] : equivalent to NextS[LineBuffer].

Output:

OutF[OS,FORMAT,p1,p2,.....p12] :
FORMAT is a BCPL-string, which is written character by character, until the escape character '%' is found. If the character following '%' is one of the following,the next parameter is written out as:

```
%%   : %
%S   : a BCPL-string
%C   : a character
%N   : a decimal number, written in minimum width
%Dd  : a decimal number, written in d places with
       zero fill
%Id  : a decimal integer, right justified in d
       places
%U   : an unsigned integer in 6 places
%Od  : an octal number in d places with zero fill.
```

Example: p1="Jens", p2='P', p3="Hansen", p4=23

```
OutF[Console,"*"%S %C. %S,  age: %N*"",p1,p2,p3,p4]
```

will output as
"Jens P. Hansen, age: 23"

OutD[OS,n,d] : write n on OS as signed decimal in d places.

OutN[OS,n] : write n on OS as signed decimal in minimum width.

OutO[OS,n] : equivalent to OutOct[OS,n,6].

OutOct[OS,n,d]: write n as octal number on OS in d places.

OutS[OS,s] : write the BCPL-string s on stream OS.

8. Command interpreter.

Commands are given to the system by typing them on the console, when the system is ready, i.e displays "..".

Commands can be divided into 2 parts:

1. commands executed by routines permanently in core, and thereby independent of the file-system.

2. commands executed by loading and running a system-program.

In this chapter we will only deal with type-1 commands, the type-2 commands will be described in chapter {11}.
When a command is typed, it is first looked for as a type-1 command, and if not found, control is given to the file-system command-interpreter, which will interpret the command as the name of a file to load and execute{10.2}.

For each type-1 command we now give its name, an indication of the kind(s) of its parameter(s), if any, its context{5.2}, and a brief description.
Parameter kinds are given by n for integer, fn for filename, and sw for a switch(on/off). Some commands may only be used, when the system is not trapped, they have context NT, others are only accepted in a trap situation, this is indicated by T. NR means no restriction.

Commands for loading and running programs, and some special facilities:

load fn NT unloads all code, except system, and loads file fn.REL.
fn=ptr loads from RC2000.

addload fn NT loads binary code from fn.REL in addition to code already loaded (for seperatly compiled segments).

go NR equivalent to Run[Start], where Start is global 1.

setP0 n NT sets stackbase register P0=n{2}

set l n NR set store location l to value n

map n NR controls load-map option:
n=0 : no load map
n=1 : a simple map of code and data segments loaded are displayed on the console.
n=2 : a full map, incl. routine entrypoints, is given on the printer.

DS sw NT turns the File-system on or off. The command "DS on" is executed automatically as part of the normal deadstart procedure{14}.

Commands for interaction with running programs:

trace sw NR controls trace facility{6.1}

dump n1,n2 NR contents of memory location n1 through n2 are displayed as unsigned decimal, signed decimal, decimal by halfwords, octal and as characters.

cont T resume trapped program{5.1}

stack T display the stack of the current (trapped) run on the console.

cb T display contents of context-block (registers).

end NR equivalent to finish

reset NR exits from all runs, unloads all code except the system, resets all globals to their values at system setup. The exit from Runs is a disorderly one, in particilar ClearUpChains are not obeyed. reset is intended to be used as a last resort to save the system, when important global values have been corrupted.

Apart from these commands, which can be used by all users, there exists a set of file- and directory managing commands, which normally only are used when bootstrapping the file-system.
They are only availble to the Master-user in a non-trapped context, and they have equivalent user-commands as described in section {11.2}:

```
Dir name          NT  walk one-step up or down the directory tree
Newdir name       NT  create a new directory 'name'
Type fn.ext       NT  type the file on console
Print fn.ext      NT  print the file
Punch fn.ext      NT  punch the file
Delete fn.ext     NT  delete the file
Files             NT  give a list of files in the directory
Readptr fn.ext NT  readin a file from RC2000, and name it fn.ext
Rename f1.e1,f2.e2 NT  rename f1.e1 to f2.e2
```

9. The RIKKE file system.

This chapter is intended to serve as an users guide to the file-system. A comprehensive description of the design and implementation of the file-system can be found in [8].

9.1. Directories.

The only way the system-routines can access a file is by a filedescription (fd), a block of words, which contains a sort of a pointer into a MasterFile on the physical disc, where the file is stored.

A directory is a list of entries, each consisting of a filename and a fd, so the purpose of directories is to enable the users to reference the files by names, instead of by fd's.
A filename on RIKKE consists of a name and an extension{9.3}. All filenames in a directory must be unique.

A special sort of entries in a directory is a directory itself, so the directories form a tree-structure. This eliminates the use of long and complicated filenames, as the number of directories and levels in the directory-tree are (almost) unlimited.
The files, whose entries are in a directory, are said to be or exist in that directory.

A directory has one more purpose, than being a collection of files. It can also be regarded as a possible environment for the user. At any time the user is in such an environment (directory), called CurrentDirectory, and will view the directory- and file-structure as seen from CurrentDirectory.

In the following figures we use rectangles for directories and circles for files:

Directory-structure:

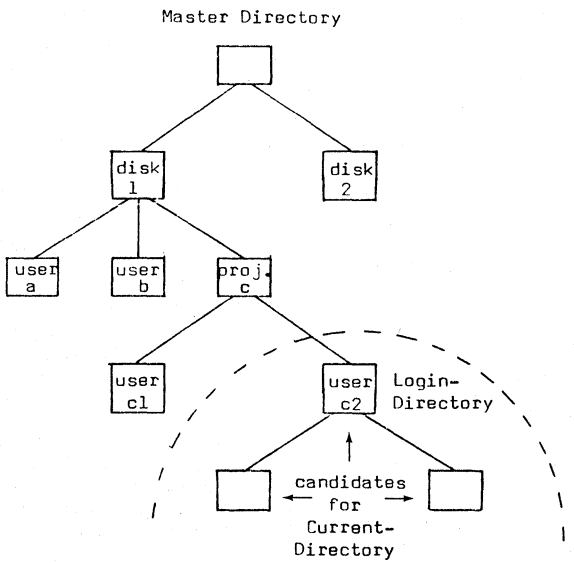


fig. 9.1.

CurrentDirectory plays a special role, when typing a name to the command-interpreter{10.2}.

At any time the user can "walk" up and down the directory-tree, i.e changing environment, by using the command 'dir'{11.2.9}.

The overall root of the directory-tree is called MasterDirectory.

9.2. Using directories.

To enter the system, a user must have a username and a password, assigned by the "Master-User". Each user-name is assigned a LoginDirectory, the users CurrentDirectory after Login.

From this directory the user can create a new directory-structure with LoginDirectory as root, and move freely around this structure, only limited by LoginDirectory at the top. It is never possible to go beyound the LoginDirectory.

CurrentDirectory also serves as a default lookup-directory, when accessing files, but it is always possible to access files in other directories, provided they are not protected.

9.3. Naming of files.

A file name in a directory consists of two parts, the name and an extension. Both name and extension can in principle be arbitrary ASCII-strings, but because the system routines interpret the characters '*s', '*n', ',' and '.' as string-delimiters, it is recommended to restrict the names to alphanumeric strings.
Note: small and capital letters are different characters.

The name is normally used to identify the file, whereas the extension is intended to tell something about the type of a file.

Default extensions are:

BCPL - a BCPL source file.
BAK - a back-up file made by the editor.
REL - a relocatable binary OCODE file.
SMB - a symbolic OCODE file.

These extensions are maintained by the system as in the following example:

1 A.BCPL exist in CurrentDirectory
2 After editing the new version is still called A.BCPL, and A.BAK is the old version.
3 The compiler compiles A.BCPL either into symbolic OCODE on A.SMB, or to relocatable binary OCODE on A.REL.
4 The assembler assembles A.SMB into A.REL.
5 Typing A on the console, the file A.REL will be loaded and executed{10.2}.

From the systems point of view directories are also files, and therefore subject to the same naming conventions. They have the special extension DIR, and it is adviced to keep this extension for directories only, and not use it for ordinary files.

9.4. Protection.

To protect files against misuse from any of the users the system provides a protection mechanism.

Allthough the users are personally identified by Login, the system uses the directory-structure to divide users into protection groups.
Therefore the access rigths doesn't depend on who the user personally is, but only in which directory he is (CurrentDirectory), so in the rest of this chapter, a user always means a directory, which of course normally can be identified with the person using that directory.

When a file is created, it is brought into the directory-structure as a son of the present CurrentDirectory. This directory will now

be regarded as the owner of that file, and will have all rights to the file. All other directories will be called users of that file, and they get the rights as decided by the owner.

Allthough directories also are created as sons of other directories, directories are users themselves, and therefore they can be made their own owners[8].

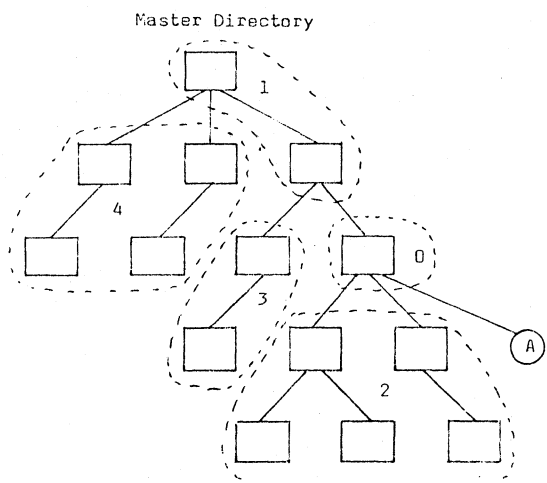
Therefore: The owner of a directory is the directory itself, the owner of any other file is the father of that file.

This user/owner division of users could be used to divide them into protection groups, but because the directory structure is static and hierarchical, it can be used to subdivide further.

In relation to each file the users are divided into five groups:

0 = the owner
1 = the ancestors of the owner
2 = the descendants of the owner
3 = those descendants of the owner's father, which are not in group 0 or 2.
4 = all other directories.

Example:



Protection groups in relation to file A

fig. 9.2.

Group 3 may not seem so relevant as the others. Its purpose is to provide the possibility to give files, common to a project group, a special access-rigth for the members of the project.

Thus the protection groups are from the system's point of view pairwise disjoint. But it will be normal to regard the protection groups hierarchical by giving group 0 at least the same rights as group 1, group 1 at least the same rights as group 2 and so on

Each of these groups is given an access-right.

There are 8 categories of access-rights:

0=CHANGE PROTECTION ACCESS:
The right to change protection of the file. (The owner has always the right to change protection, regardless of his access rights).

1=DELETE ACCESS:
The right to delete the physical file on the disc. This access-right does not imply the right to delete the entry in the directory.

2=WRITE ACCESS:
The right to change the contents of the file. The user can thus empty the file completely, but there remains an empty file. For directories this access right is equivalent to the right to remove entries .

3=UPDATE ACCESS:
This has only meaning for directories, where this access right will give the right to rename all files in that directory.

4=APPEND ACCESS:
The right to append information to the end of the file. For directories this is the right to insert new entries into the directory.

5=COPY ACCESS:
The right to copy the file to another file via the system-programs. Is not valid for directories, as these must not be copied.

6=READ ACCESS:
The right to read a file, but not to modify any of the contents of that file. For directories this is the right to look up in the directory.

7=NO ACCESS:
No rights to the file at all, except that the user may know the existence of the file, if he has READ-ACCESS to the father of the file.

The access rights are hierarchical, e.g. a user having access right 4, also has access rights 5 and 6 too.

There is one special case of protection: The owner-entry of a file cannot be deleted, without deleting the file too. This is to assure, that each file always has an owner.

As each protection group is assigned an access right, the protection key for a file is a 5 digit number.

For example default protection for a file is 00557, which means

group 0 : all rights to the file.
group 1 : all rights to the file.
group 2 : the right to copy and read the file.
group 3 : the right to copy and read the file.
group 4 : no rights at all to that file.

When a user want to access a file, the system finds out which protection group he (CurrentDirectory) belongs to. It will then look up the access right for this protection group. If the access is done in accordance to this access right, the access will be granted, otherwise an error will be the result.

The default protection for directories is 22446.

9.5. Link and copy.

If a user frequently want to access files in other directories, it can be more convenient to make an entry for those files in "his own" directory too. This can be done in 2 ways:

1. Make a copy of the file. This copy is the users your own file, and he is the one to decide anything about the file. If the original file is updated, that update will be missed.

2. Make a Link to the file. Only an entry for the file is inserted in the directory, the file is still the property of the original owner, and the user can only handle the file according to the access-rigths. The owner can delete the file any time he wants, but the users are sure always to have a link to the most recent version of the file.

Link or Copy

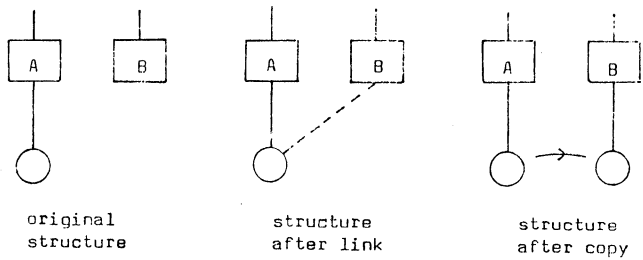


fig. 9.3.

In this sense directories are not files, and they can neither be copied nor linked.

9.6. Discs and directories.

The RIKKE disc-system consists of 2 DIABLO-drives, each consisting of 1 fixed disc and 1 exchangeable disc, giving a configuration af 2 fixed discs, and an unlimited number of exchangable discs.

Each disc consists of 408 tracks of 24 sectors of 260 bit words, 256 data-words and 4 for system-information. This gives a total data-area of 9792 pages (blocks) of 256 words on each disc.

The files in the file-system can be stored on arbitrary discs, of which up to 4 can be mounted simultaneously. These 4 discs could be identified to the system by their physical location 0-3, which is called the disc-unit.
The disc mounted on disc-unit i is called physical disc no. i.

Because of the unlimited number of discs, that can be mounted, each disc has to be identified by its logical disc.no, also called the logical device no., to the system. Each disc is therefore given a logical device name and number, and we then have logical disc no j equal physical disc no i for 0<=i<=3 and 0<=j<=MAXDEV(20).

The files on each disc are hierarchically ordered, and the root is called the DiscDirectory. When a disc is mounted, the Disc-Directory on that disc will become a son of the MasterDirectory, and it will get the name "logdev".DIR in MasterDirectory, where logdev is the logical device name for the disc.

Thus mount means two things. One is to physically place a disc on

the disc-drive, and the other is to connect the directory structure on that disc with the MasterDirectory (and the rest of the file system).
Using the latter meaning, all discs (incl. physical discs 0 and 2) can be mounted and dismounted.

The normal disc configuration after deadstart will be
  logical disc no 0 = physical disc no 0 = disc "SysAdmin"
  logical disc no 1 = physical disc no 2 = disc "UserDisk1"
and the other two discs are to the disposal of the users.

In case of any doubt about where CurrentDirectory is situated in the directory-structure, the command "ident" {11.2.18} can be used to list the path from MasterDirectory to CurrentDirectory on the console.

9.7. A further note on files.

The first description of files was somewhat simplified, and maybe a little misleading. We will now explain some important concepts in the file-system, but for details, the reader is still referred to [8].

On each disc is a MasterFile, which contains a heading for each file. This heading holds information such as the address of the first and last page of the file, protection information, and creation- and update-time. Furthermore it contains for each file an identifier (UID), which is guaranteed to be unique among all files ever created in the system. This UID is used for control purposes.
The index of the heading in the MasterFile is called the file value (fv) of that file.

To identify a file on the disc, the logical disc no. and the fv for the file must be known. Then the heading can be found in the MasterFile, and from there the file itself. To be sure to access the right file, the UID must also be supplied. This is checked against the UID in the heading, and only if those two UID's are identical, the file can be accessed.

This set (logical disc no,fv,UID) together with some protection information is called the file description (fd), and this is precisely what is neccessary to access a file.

The normal way to find the fd for a file, is looking up in a directory for a name assigned to that file. The real contents of a directory is therefore a set of tuples (name,ext,fd).

10. Using the file system.

The user will normally be interested in using the file system at two levels, of which the upper level is the command interpreter mode, and the lower level is the user mode, i.e executing user-programs.

10.1. User mode.

The system-library contains routines available to all users to lookup a file, access a file for reading or writing.

The users are advised only to access files through streams, created by these library routines, as this is the only file-access supported by the system.

10.1.1. Reading/writing files by name.

The normal way for users to read/write files is to identify them to the system by their names in CurrentDirectory:

ReadNamedFile[fn,ext,type]:
Returns an input-streams S to read the file, identified by fn.ext in CurrentDirectory. The value of type must be BYTES or WORDS, giving a byte- or word-stream.
If the file is not found, or if the entry in the directory contains a wrong UID, a message is displayed by GiveUp{5.1}.
After reading the file it should be closed by Close[S].

WriteNamedFile[fn,ext,type,mode]:
Return an output-stream S to write the file, identified by fn.ext in CurrentDirectory. The value of type must be BYTES or WORDS, giving a byte- or word-stream.
If a wrong UID entry is found, the routine calls GiveUp.

The precise action of the routine depends of the value of mode:

OVWCR: if the file exists overwrite it, else create the file.
ADDCR: if the file exists append to it, else create the file.
OVWGU: if the file exists overwrite it, else GiveUp.
ADDGU: if the file exists append to it, else GiveUp.
CREGU: if the file exists GiveUp, else create it.

For a discussion of Reset and Close to output-streams, the reader is referred to {7.4}.

10.1.2. Reading/writing files by fd.

As the reader may have guessed, the names CurrentDirectory, MasterDirectory, LoginDirectory and SystemDirectory actually holds fd's, pointing to the directories. So in the following when we write 'dir', we mean an fd pointing to a directory.

LookUpDir[fn,ext,dir]:
Returns a fd for the file identified by fn.ext in directory dir (e.g CurrentDirectory) if it is found, otherwise the value NOTINDIRECTORY is returned.

ReturnFD(fd):
returns the fd-block to the free-store{2.1}.

The fd found by LookUpDir can now be used as argument to the functions in section {7.4} to create input- or output-streams S.

Whenever a stream is created this way, the possibility of violating the access-rigths or accessing a wrong-UID file, exist, so before using the stream S , the routine CheckNamedFile[fn,ext,S,msg] should be called, which in any of the two cases displays an error-message and gives-up.

The (system-)programmer, who wants to go beneath this level for accessing files, are referred to [8]

10.2. Command-interpreter mode.

In the command interpreter mode the system will accept commands from the users, which will then be interpreted and executed if possible.
When the system is ready to accept a command it will write two periods on the console.

When a command "com" is typed, the command-interpreter will do the following:

1 Look up in a set of commands, executed by programs always in core. If there is such a command, it will be executed{8}.
2 Else CurrentDirectory is searched for a file called com.REL. If there is such a file it will be loaded and executed.
3 Else a special directory with system programs, System-Directory, will be searched for com.REL. If there is such a file, it will be loaded and executed.
4 Else the system gives up with the message "File com.REL does not exist".

The way to pass parameters to user-programs is described in [1].

The exact format of the commands supplied by the system is given in {11.1}.

11. Commands.

This section describes all the system-programs, the users can call, via the command-interpreter.
We start by giving some general rules for the format of commands, and then we describe each of the commands in detail.

11.1. Format of commands.

The format of a command is

```
com p1 p2 p3 ... pn
```

where
com is the name of the command.
p1...pn are the parameters for the command.

com and p1, as well as each pair of parameters, are separated by " " or "," or "."

The parameter list is terminated by carriage return.

The parameters are often file names or directory names. There are some special rules for these.

If the parameter is a file name or an extension, it can normally be written with "wild characters", where

"*" means any string with length >=0
"?" means any character

Thus for example

"*" matches any name
"?a*" matches all names with "a" as second character.
"abc" matches only the name "abc"

The directory given will then be searched for all files, where both name and extension match those patterns specified in the parameters, and the command will be executed for all such files.

If the parameter is a directory, the directory must be specified by writing the directory path.

This is specified with the following syntax:

syntax for dirpath:

```
dirpath   -> ':'                  (CurrentDirectory)
          -> startdir {dir-id}* enddir

                                   start-search
startdir  -> '>'                  MasterDirectory
          -> ')'                  LoginDirectory
          -> {'<'}+               An Ancestor
          -> empty                CurrentDirectory
dir-id    -> dirname '>'
enddir    -> dir-id
          -> dirname
dirname   -> any bcpl-string
```

Directory-path example

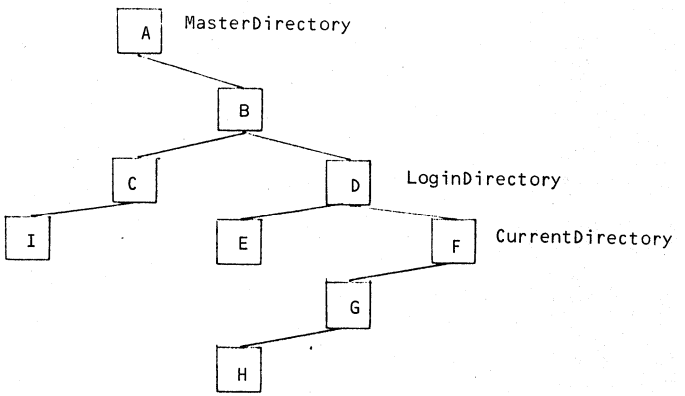


fig. 11.1.

To reach each of the directories on fig. 11.1 from Current-Directory the users must type:
(the number in brackets shows which protection group Current-Directory will be assumed to belong to)

```
A:    >         (4)    or  <<<    (2)
B:    >B        (4)    or  <<     (2)
C:    >B>C      (4)    or  <<C    (3)
D:    >B>D      (4)    or  )      (4)    or  <     (2)
E:    <E        (3)    or  )E     (4)
F:    :         (0)
G:    G         (1)
H:    G>H       (1)
I:    >B>C>I    (4)    or  <<C>I  (4)
```

Before Login, the only commands, that can be executed, are

```
DS on
DS off
mount
dismount
help
login
```

The explanation of the commands and their formats, as given in next section, are also available at RIKKE. The user can at any time issue the command "help" to get the wanted information written on the console.

In the following commands the symbols "{" and "}" mean that the parameters between the brackets are optional, and a "*" after a pair of brackets means that these parameters can be repeated zero or more times.

11.2. List of Commands.

11.2.1. append.

format: append fn.ext dir {fn-n.ext-n dir-n}*

action: If fn.ext is in dir, all files matching the fn-n.ext-n's in dir-n's are appended to file fn.ext. If fn.ext is not in directory dir, it is created before the other file(s) are appended to it.

defaults: If the last directory is omitted, CurrentDirectory is assumed.

restrictions: fn.ext in dir cannot be appended to itself.
Directories cannot be appended.
fn, ext must be simple names (no *'s or ?'s)

protection: APPEND-access is necessary to fn.ext in dir.
If fn.ext has to be created, APPEND-access is necessary to dir.
COPY-access is necessary to the fn-n.ext-n's.

11.2.2. assdr.

format: assdr fn-1 {fn-2....fn-n}.

action: All action is performed in CurrentDirectory.
One or more symbolic OCODE-files are assembled together on the file named ocode.REL, the program will ask for the name 'ocode'.
If the file ocode.REL does not exist, it will be created, else overwritten.
The files to assemble must have extension .SMB, generated by option n/N in the BCPL-compiler[1].
If {fn-2...fn-n} are omitted, fn-1 can be a 'wild'-name, else all fn's must be simple names.

protection: READ-access is neccessary to the .SMB files.
If the file ocode.REL already exist, WRITE-access is necessary, else APPEND-access is necessasy to CurrentDirectory.

11.2.3. bcpl.

format: bcpl fn-1 {fn-2....fn-n}.

action: The files to be compiled must be in Current-Directory.
The action depends on the options given when prompted:
if option n/N is specified, each file is compiled into symbolic (not readable) OCODE on a file named fn-i.SMB, else each file is compiled into a relocatable binary file named fn-i.REL.
Options:
t,T : generate trace information.
x,X : system words are in CAPITAL-letters.
r,R : repeat-option, each file can consist of one or more segments, seperated by '.', and they are compiled into one file named fn.REL or fn.SMB.
d,D : If the compiler error "TOO MANY NAMES DECLARED" occur, you may increase the size of the internal name-tabel by specifying e.g. d 2500.

For further options see [1].
If {fn-2...fn-n} are omitted, fn-1 can be a 'wild'-name, otherwise all fn-i's must be simple names.

defaults: Default options: d 2000.
Generate .REL file without trace information.
If a get-file isn't found in CurrentDirectory, the compiler will look for it in the system-get directory, named >SysUser>Header.

restrictions: The files to be compiled must have extension .BCPL

protection: READ-access is necessary to all .BCPL and .GET files used.
WRITE-access is necessary to .SMB and .REL files, if they exists.
If a new file is created, APPEND-access to Current-Directory is neccessary.

11.2.4. combine.

format: combine ToFile file-1 file-2 ... file-j

action: The file ToFile.REL may contain several segments of relocatable binary code, or it may not exist.
Each segment is identified by the '$ident "identifier"' construction in the source-text[1] The command will substitute those segments on ToFile.REL, which are present in the files file-1.REL .. file-j.REL, with these latter versions.
example:

file-1 .. file-j:	ToFile.REL:
f1.REL	f1.REL :
f3.REL	f2.REL :
	f3.REL :
	f4.REL :

The segments f1.REL and f3.REL on ToFile.REL are exchanged with the new versions.
If a file-i is not found, it can be appended.

protection: COPY-access is necessary to file-1 .. file-j.
WRITE-access is necessary to ToFile.REL.
If Tofile.REL has to be created, APPEND-access is neccessary to CurrentDirectory.

11.2.5. commands.

format: commands {fn}

action: Gives a list of all commands available to the users as system-programs in SystemDirectory, either on the console or on the file fn.COMM.

protection: WRITE-access is neccessary to fn.COMM, if it exists.

11.2.6. copy.

format: copy fn1.ext1 dir1 {fn2{.ext2 {dir2}}}

action: fn1.ext1 in dir1 will be copied to dir2 and given the name fn2.ext2.
If fn2.ext2 already exists in dir2, you will be asked whether to unlink,delete or rename the old fn2.ext2, or give up the copying; if it does not exist, it will be created.

defaults: If fn2, ext2 or dir2 is omitted, fn1, ext1 or CurrentDirectory is assumed.

restrictions: Directories cannot be copied.

protection: COPY-access is necessary to fn1.ext1.
If fn2.ext2 has to be created, APPEND-access is necessary to dir2.
Access-rigths needed to unlink,delete or rename fn2,ext2 - please look under the commands unlink,delete, or rename.

11.2.7. count.

format: count {device}

action: Counts the number of free pages on the specified device, e.g "UserDisk1".

defaults: If device is omitted, all mounted devices will be counted.

11.2.8. delete.

format: delete fn.ext {dir}

action: fn.ext will be deleted (that is: the physical file will be destroyed, and the entry in dir will be removed), or unlinked (that is the entry in dir will be removed), depending on the protection.

defaults: If dir is omitted, CurrentDirectory is assumed.

restrictions: Directories can only be deleted, if they are empty. A file cannot be unlinked from the OWNER-directory.

protection: The precise action that will be performed depends strongly of whether dir is user or owner of the file, and which access-right dir has:

	dir is owner of fn.ext	dir is user of fn.ext
at least DELETE-access	delete the file	give a warning before delete/unlink
not DELETE-access	NOTHING	unlink the file

11.2.9. dir.

format: dir path

action: The directory found by 'path' becomes CurrentDirectory.
The syntax for 'path' can be found in {11.1}.

protection: 'path' must be a directory path, and in that path LoginDirectory must not be exceeded

11.2.10. disass.

format: disass fn

action: Makes an OCODE disassembling of the file fn.REL in CurrentDirectory on the file fn.DISASS.

protection: READ-access is neccessary to fn.REL.
WRITE-access is neccessary to fn.DISASS, if it exists.

11.2.11. dismount.

format: dismount {name}

action: The discpack named 'name' is dismounted, i.e. the files on this device are removed from the file-system.

restrictions: The device holding the MasterDisk cannot be dismounted.
The device holding CurrentDirectory cannot be dismounted.

11.2.12. do.

format: do fn p1,p2,....,pn

action: The file fn.CMD is taken to be a command file with a simple parameter substitution.
In fn.CMD each occurence of #i is replaced by pi, if i<=n.

Example:
The file modify.CMD is:
```
edit #1
bcpl #1
```
The command 'do modify prog' will edit and compile the file prog.BCPL.

default: If fn.CMD isn't found in CurrentDirectory, it will be looked for in SystemDirectory.

11.2.13. edit.

format: edit {fn{.ext}}

action: The basic function of the editor is described in [7], so for a tutorial description of the editor, and a list of commands, see that description.

The editor can be run in two modes:

1. if no parameters are specified, no input-output files exists, and they must be specified by commands like:
   'gr','gf' for input and 'gt','gw' for output

2. the normal way:
   if the editor is called like 'edit fn.ext', the editor will create a file named fn.BAK,which is a copy of the file fn.ext, when editing is over, and the file fn.ext will be the edited file.
   If the file fn.ext does not exist, it will be created and a notice is given.
   When working in mode 2, all the g-commands from 1 are illegal.

If you type 'CTRL C, end' when editing, you will return to the command interpreter, and nothing has happened to your files, except when an output file has been closed by a new 'gw'.

defaults: default extension for mode 2 is BCPL

restrictions: the call 'edit fn.BAK' is illegal.

protection: To enter a file into the editor, COPY-access is required.
When working in mode 2, the old BAK-file will be deleted regardless of its protection upon completion of editing.
The output-file for the editor must be APPEND/WRITE - permitted, depending of how it is specified:
APPEND-permitted when specified by 'gt',
WRITE-permitted else: in mode 2 and by 'gw'.

11.2.14. filcom.

format: filcom

action: The program asks for the 2 files to compare.
In case of a difference, the next 60 characters of each file will be listed on console.

protection: at least READ-access is needed to the files to compare.

11.2.15. filedump.

format: filedump fn.ext {dir {from {to}}}{$l}

action: The file fn.ext in dir will be dumped:
Each word from word no. "from" to word no. "to" will be listed as unsigned decimal, as signed decimal, as two decimal bytes, octal, and as two characters.
If '$l' is specified, the dump of each file will be made on the file fn.FILDMP, otherwise on the console.

default: if dir is omitted, CurrentDirectory will be assumed.
If from or to is omitted start-of-file or end-of-file will be assumed.

restrictions: directories cannot be dumped.

protection: at least READ-access is needed to fn.ext.
WRITE-access is necessary to the file fn.FILDMP if it exists.

11.2.16. files.

format: files {fn{.ext{dir{searchdir}}}}{opt1}{opt2}{opt3}.
The options can be:
$a/$all/$A/$ALL/$All
$l/$list/$L/$LIST/$List
$s/$sort/$S/$SORT/$Sort

action: Makes a list of information on all files, whose names matches the pattern fn.ext.
If searchdir is specified, the directory-subtree, which has dir as root, will be traversed, and for all directories, whose name matches searchdir, a list will be given too.
If option all is set, the list will contain all information of that file, else a shorter list will be given.
If option list is set, the list will be listed on the file "files.LPT" in CurrentDirectory, else it will be listed on the console.
If option sort is set, the files will be sorted alfabetically before listing.

defaults: If fn, ext, dir or searchdir are omitted, "*","*", CurrentDirectory or NONE will be assumed.
NB! in the last three parameters written, the options will override fn, ext, dir or searchdir.

protection: READ-access is necessary to 'dir'.
WRITE-access is necessary to 'files.LPT', if it exists.

11.2.17. help.

format: help {item} {$l}

action: If 'item' is specified, all help-texts for items, matching 'item' (wild-characters '*' and '?' allowed) are listed, else a list of all items, on which help is available, is given.

The list is given on the console, or if '$l' is specified, on the file 'help.LPT'

11.2.18. ident.

format: ident

action: the path from MasterDirectory to CurrentDirectory and from MasterDirectory to LoginDirectory are listed on the console.

11.2.19. info.

format: info {all} {$l}

action: The latest 'message of the day' is typed.
If 'all' is specified all the accumulated messages will be typed too.
It '$l' is specified the information will be typed on the file named 'info.LPT', else on the Console.

11.2.20. link.

format: link fn1.ext1 dir1 {fn2{.ext2 {dir2}}}

action: A link will be made in dir2 to fn1.ext1 in dir1, giving the entry in dir2 the new name fn2.ext2.
If fn2.ext2 already is in dir2, you are asked whether to unlink, delete, or rename the old fn2.ext2, or to give up the linking.
If fn2.ext2 is not in dir2, it will be created.

defaults: If fn2, ext2, or dir2 is omitted, fn1, ext1, or CurrentDirectory will be assumed.

restrictions: Directories cannot be linked.
dir1 and dir2 must not be the same directory.

protection: READ-access is necessary to fn1.ext1.
If fn2.ext2 must be created, APPEND-access is necessary to dir2.
Access-rights needed to unlink,delete, or rename - please look under the commands unlink, delete, or rename.

11.2.21. login.

format: login username,password

action: The pair username,password is checked to be a valid user, and if so, CurrentDirectory is set to the users LoginDirectory.
If there is any mail to you, you are advised, see 'help mail'.

11.2.22. logout.

format: logout

action: The user is logged out, and the command interpreter enters a special mode, where the only legal commands are:
login,help,mount and dismount.

11.2.23. mail.

format: mail 'command'

action: The mail-system allows users to send messages to each other, i.e. to their directories.
The following 6 commands are legal:

accept	: accept mail to CurrentDirectory, e.g. create a file named MAIL.BOX, and allow all users to send mail to you.
type	: the mailboxes in the directories from MasterDirectory to CurrentDirectory are typed on Console.
send us	: send a message to the user named 'us'. The message is sent to that users LoginDirectory if he has a mailbox, else nothing will be done.
reject	: leave the mail-system, i.e. delete the file MAIL.BOX.
archive	: the current mailbox is appended to the file MAIL.ARC, and then emptied. Hence MAIL.ARC contains all messages sent to you, and it can be deleted at any time you want.
empty	: the mailbox is emptied without archiving.

When a user log's in, he will be advised whether there is any mail to him or his fathers. This mail can then be read by 'mail type'

comment: you have to type 'mail accept' to join the mail-system.

11.2.24. mount.

format: mount {unit {name}}

action: A named device is mounted on a disk-unit, i.e. the files on that device are entered into the file-system.
The program will ask for the missing parameters.

comment: After normal system-setup, the only devices mounted are the System-Disk and the UserDisk1.
The System-Disk is mounted on unit 0 and the UserDisk1 on unit 2.

11.2.25. move.

format: move fn1.ext1 dir1 {dir2{.fn2 {ext2}}}

action: fn1.ext1 will be copied to dir2 with the new name fn2.ext2, and then deleted in dir1.
If fn2.ext2 is already in dir2, you will be asked whether to unlink, delete, or rename the file, or to give up moving.
If fn2.ext2 is not in dir2, it will be created.

defaults: If dir2, fn2 or ext2 is omitted, CurrentDirectory, fn1 or ext1 is assumed.

restrictions: Directories cannot be moved.
dir1 and dir2 must not be the same directory (then move would be meaningless).
Link-files cannot be moved.

protection: WRITE-access is necessary to dir1 and APPEND-access to dir2, and you must have DELETE-access to the file, else it cannot be deleted in dir1, and nothing will happen.
Access-right needed to unlink/delete/rename the old fn2.ext2 - please look under the commands unlink/delete/rename.

11.2.26. newdir.

format: newdir dirname

action: a new directory called dirname will be created as a son of CurrentDirectory. This new directory will become the new CurrentDirectory.

restrictions: There must not exist a file (directory) in CurrentDirectory called 'dirname.DIR'.

11.2.27. newuser.

format: newuser {dir}

action: A new pair of username/password is accepted as login to directory 'dir'.

defaults: dir =CurrentDirectory

restrictions: dir must identify a directory in the tree, having current LoginDirectory as root.
Usernames must be unique.

11.2.28. oclist.

format: oclist fn

action: The symbolic OCODE on file fn.SMB (not readable) in CurrentDirectory is listed on file fn.OCLIST.

protection: READ-access is neccessary to fn.SMB.
WRITE-access is neccessary to fn.OCLIST, if it exists.

11.2.29. print.

format: print {fn{.ext {dir}}} {op1...opt3}.
The options can be:
$s/$sort/$S/$SORT/$Sort
$nh
$ni

action: The files in dir, whose filenames matches fn.ext, will be printed on the lineprinter.
If the option sort is set, the files will be sorted by name before printing.
If option ni is set, no index of the printed files will be given.
If option nh is set, no header will be printed.

defaults: If fn, ext, or dir is omitted, "*", BCPL, or CurrentDirectory is assumed.

restrictions: Directories cannot be printed (use the command files).
Files with extension REL cannot be printed.

protection: READ-access is necessary to fn.ext.

11.2.30. protect.

format: protect fn.ext {prot {dir}}

action: the protection of the file fn.ext in dir will be set to prot.
prot must be a 5-digit number, with each digit between 0 and 7 (incl.).
The first digit means the access rights for group 0 (the owner), the second for protection group 1 and so on.

access-rigths:
0=CHANGE-PROTECTION ACCESS.
1=DELETEACCESS.
2=WRITEACCESS.
3=UPDATEACCESS.
4=APPENDACCESS.
5=COPYACCESS.
6=READACCESS.
7=NO ACCESS.

protection-groups:
0=the owner-directory.
1=the ancestors of the owner directory.
2=the descendants of the owner directory.
3=those desc. of the owner's father, which are not in group 2.
4=other directories.

default: If directory is omitted, CurrentDirectory is assumed.
If protection is omitted, the files will be protected to 00???, where '?' means unchanged.

protection: CHANGE-PROTECTION access is neccessary to fn.ext, or directory must be owner of fn.ext.

11.2.31. ptdump.

format: ptdump {fn{.ext {dir {dumpfile}}}}

action: The files matching fn.ext in dir are dumped on the file dumpfile.MAS in a special format, which can be read by 'ptload'.
If dumpfile = "ptp", the files will be dumped on papertape.

defaults: If fn, ext, dir, or dumpfile are omitted, "*", "*", CurrentDirectory or the name of the directory are assumed.

restrictions: Directories cannot be dumped.

protection: Only files of which dir are the owner are dumped, and COPY-access is neccessary to these.

11.2.32. ptload.

format: ptload {file {device}}

action: The directory, specified by 'device' and the directory path from 'ptdump', will be loaded with the files on the file 'file.MAS'.
If the files already exists, they will not be overwritten, but the file will be read to the file 'file.NEW'.
If file = "ptr", the files will be loaded from papertape reader.

defaults: file = "ptr".
device = device for CurrentDirectory.

11.2.33. punch.

format: punch fn.ext {dir}

action: all files matching fn.ext in 'dir' are punched on the paper-tape.

restrictions: directories are not punched

defaults: If ext is omitted, '*' is assumed, if dir is omitted CurrentDirectory is assumed

protection: only files with COPY-access are dumped.

11.2.34. readdec.

format: readdec fn.ext

action: Transfers a file from DEC-10 to Rikke, and gives it the name fn.ext in CurrentDirectory.
The transport must be initiated on DEC-10 by 'copy RIKOUT: = fn.ext (/I if not an ASCII-file)'.
If the file exists on Rikke, it is overwritten.

default: ext = "BCPL"

protection: WRITE-access is neccessary to fn.ext if it exists.

11.2.35. readptr.

format: readptr fn.ext {dir}

action: A file is read from papertape-reader, and given the name 'fn.ext' in directory 'dir'.
If fn.ext already exists in dir, you will be asked whether to unlink,delete or rename the old fn.ext, or give up the reading; if it does not exist, it will be created.

defaults: dir = CurrentDirectory.

restrictions: ext = 'DIR' illegal.

protection: If you want to create fn.ext, APPEND-access is necessary to dir.
Access needed to unlink,delete or rename fn.ext- please look under the commands unlink, delete, or rename.

11.2.36. remwuid.

format: remwuid {fn{.ext {dir}}}

action: the entry fn.ext is removed from dir, if the entry is Wrong-UID.

defaults: Remove *.* in CurrentDirectory with wrong UID.

protection: WRITE-access is necessary to 'dir'.

11.2.37. rename.

format: rename fn1.ext1 fn2{.ext2 {dir}}

action: the entries matching fn1.ext1 in dir are renamed to fn2.ext2.

default: if ext2 or dir is omitted, ext1 or CurrentDirectory will be assumed.

restrictions: either fn1 or ext1 must be a simple name (no *'s or ?'s).
If fn1 is extended, fn2 has to be "*" (meaning the same name).
If ext1 is extended, ext2 has to be "*".

protection: UPDATE-access is necessary to 'dir' to rename entries.

11.2.38. senddec.

format: senddec fn{.ext}

action: The file fn.ext in CurrentDirectory is send from Rikke to DEC-10.
The transport must be initiated on DEC-10 by
'copy fn.ext = RIKIN: (/I if not an ASCII-file)'.
If the file exists on DEC-10, it is overwritten.

default: ext = "BCPL"

protection: READ-access is neccessary to fn.ext

11.2.39. sort.

format: sort {dir}

action: The entries in the directory will be sorted alphabetically.

defaults: dir = CurrentDirectory

protection: UPDATE-access is necessary to 'dir'

11.2.40. time.

format: time

action: current date and time is typed on Console

11.2.41. tty.

format: tty command

action: the commands specifies working-mode for the console. Legal command are:

ctrlon : control-characters are output as control-characters.
ctrloff: control-characters are typed as ^CH, where CH=the character+64.

comment: the normal working-mode is ctrlon

11.2.42. type.

format: type {fn{.ext {dir}}}

action: The files matching fn.ext in dir will be typed on the console.
Striking CTRL E while listing, will skip to next file.

default: if fn is omitted, you will be asked to prompt the filename.
if ext or dir is omitted, BCPL or CurrentDirectory will be assumed.

restrictions: Directories cannot be typed. Files with extension REL cannot be typed.

protection: At least READ-access is needed to fn.ext

11.2.43. unlink.

format: unlink fn.ext {dir}

action: The entries matching fn.ext is removed from dir, but the file itself will not be deleted.

defaults: If dir is omitted, CurrentDirectory is assumed

restrictions: directories cannot be unlinked.
OWNER-files cannot be unlinked.

protection: WRITE-access is necessary to 'dir'.

11.2.44. users.

format: users

action: List all usernames on the console.

12. Library routines.

This chapter contains an alphabetical list of all global routines and variables, which may be neccessary or just usefull for user-programs. Each item is briefly described, or an reference is given to a description elsewhere in this or other manuals.

The library routines are included in user-programs by using the directive get "SysHdr" in the program.

Globals 100-399 are for user-programs.

505	AddBytesToFile[fd]	{7.4}
435	AddToFile[fd]	{7.4}
480	AllocIntEntry[]	{5.1}
418	BytesFromFile[fd]	{7.4}
419	BytesToFile[fd]	{7.4}
511	CheckNamedFile[f,e,fd,text]	{10.1.2}
497	Clock[S]	Outputs current date and time to character stream S as a text.
20	Close[S]	{7.1}
524	Concatenate[s1,s2]	Returns the concatenation of string s1 to s2, neither s1 or s2 is returned.
44	Console	{7.3}
529	CopyFS[IS,OS]	Equivalent to $(while not Endof[S] do Out[OS,Next[IS]] $)
50	CopyString[s1,s2]	Copies string s1 to s2. s2 must be allocated before the call.
500	CopyVec[v1,v2,n]	Equivalent to $(for i=0 to n-1 do v2!i:=v1!i $)
69	CurrentDirectory	{9.1}
53	Discard[fd]	[8]
21	Endof[S]	{7.1}
72	EqS[s1,s2]	Compares BCPL-strings s1 and s2 and returns true/false
441	FindHeading[....]	[8]
75	GiveUp[F,p1,..p12]	{5.1}
416	InFromFile[fd]	{7.4}
71	InitVec[n,a0,...an]	Calls NewVec(n), and initialises the vector to the following n+1 parameters, n<=30.
445	InsertCUC[C]	{4.1}
432	IntCodeFromRaw[S]	{7.3}
8	Interrupt[]	{6.2}
41	Keyboard	{7.2}
31	Level[]	Yields current stack-frame pointer (P register), used by LongJump e.g
43	LineBuffer	{7.3}
39	Load[fn,dir]	The file fn.REL in directory dir is loaded, dir=fd of a directory.

```
32   LongJump[p,i]              Jumps to label i at level p. In con-
                                junction with Level, LongJump may be
                                used to return from nests of routine
                                calls, without invoking the Run-
                                mechanisme.
59   LookUpDir[fn,ext,dir]       {10.1.2}
58   MakeNewFile[.....]         [8]
27   Map[n]                     Same effect as the map command{8}
438  MaxVecSize[]               {2.1}
36   NewVec[n]                  {2.1}
17   Next[S]                    {7.1}
499  NextBlock[S,v,n]           {7.6}
80   NextCh[S]                  {7.8}
87   NextN[S]                   {7.8}
525  NextS[S]                   {7.8}
450  OpenReadFile[fn,ext,type,dir]
                                ReadNamedFile[fn,ext,type]{10.1.1} =
                                OpenReadFile[fn,ext,type,CurrentDirectory]
451  OpenWriteFile[fn,ext,type,mode,dir]
                                WriteNamedFile[fn,ext,type,mode] =
                                OpenWriteFile[fn,ext,type,mode,CurrentDirector
18   Out[S,x]                   {7.1}
520  OutBlock[S,v,n]            {7.6}
84   OutD[S,n,d]                {7.8}
83   OutF[S,F,p1...p12]         {7.8}
81   OutLine[S]                 Equivalent to Out[S,'*n']
82   OutN[S,n]                  {7.8}
85   OutO[S,n]                  {7.8}
86   OutOct[S,n,d]              {7.8}
80   OutS[S,s]                  {7.8}
417  OutToFile[fd]              {7.4}
530  Pack[From,To,n]            The vector From contains one byte in
                                each rigth-halfword of From!0 to
                                From!(2*n-1). These bytes are packed in
                                vector To!0 to To!(n-1) with first byte
                                in the left byte, second byte in the
                                rigth byte, and so on.
                                Pack does not allocate vector To.
66   PackString[v,s]            The vector v contains characters in v!1
                                through v!(v!0)). These characters are
                                packed as a BCPL-string in vector s.
                                Notice that PackString does not al-
                                locate vector s.
46   Printer                    {7.3}
92   Prompt[S]                  Equivalent to
                                $(
                                  OutS[Console,S]
                                  Reset[LineBuffer]
                                  resultis ReadS[]
                                $)
95   PromptN[S]                 Equivalent to
                                $(
                                  OutS[Console,S]
```

```
                              Reset[LineBuffer]
                              resultis ReadN[]
                            $)
45    Ptp                   {7.2}
42    Ptr                   {7.2}
405   PutBack[S,x]          {7.1}
94    ReadN[]               {7.8}
431   ReadNamedFile[fn,ext,type]   {10.1.1}
433   ReadS[]               {7.8}
444   RemoveCUC[C]          {4.1}
19    Reset[S]              {7.1}
473   ReturnFD[fd]          {10.1.2}
447   ReturnHeading[h]      [8]
454   ReturnString[s]       Equivalent to
                              ReturnVec[s,((s!0) rshift 8)/2]
35    ReturnVec[v,n]        {2.1}
24    Run[routine,pvec]     {3.1}
1     Start[...]            [1]
30    Stop[F,p1,...p12]     {5.1}
25    StopGiveUp[F,p1..p12]   Equivalent to
                              $(
                                Stop[F,p1,..p12]
                                GiveUp["cannot continue"]
                              $)
74    StreamError[]         {7.7}
498   Time[d,m,y,h,m,s,t]   Return day, month, year,  hour,  second
                            and tenth-seconds in the lv-parameters.
33    Unload[iblock]        {3.1}
531   Unpack[From,To,n]     Reverse  of Pack: The n words in vector
                            From!0 to From!(n-1) is  unpacked  with
                            one  byte  rigth justified in each word
                            of To!0 to To!(2*n-1).
                            Unpack does not allocate vector To.
67    UnpackString[s,v]     Reverse of PackString
410   WordsFromBS[S]        {7.3}
437   WordsToBS[S]          {7.3}
429   WriteNamedFile[fn,ext,type,mode]   {10.1.1}
```

13. The Character Set.

The only devices attached to the system which provide information in legible characters are the console and the line printer. Both of these have an ASCII character set (the printer with some minor oddities). The internal character code used in the system is ASCII, in the following sense:

1. The codes 32-127 have standard ASCII interpretations as given in {13.1}.
2. The codes 9-13 are format codes which are output as follows (by Console and Printer{7.3}):
   9 : ('*t') tabulator, so many spaces that the number of characters on current line is divisible by 10.
   10: ('*n') car return followed by line feed.
   11: empty
   12 :('*p') car return followed by form feed, on the console this is ten line feeds.
   13: car return only. It is not possible to input this character from the keyboard, since car return is immediately converted to '*n'. In this fashion the return key receives its most natural interpretation, and texts in internal ASCII only contain one character per newline.
3. Other codes have no meaning (are illegal).

Texts created in the editor[7] will be in internal ASCII. Existing ASCII text, e.g. on imported paper tape, read into the editor, may be converted to internal ASCII by using the gf command to select input.

The Keyboard stream can deliver any value in the range 0-127 (except 13 as described above , and the control characters described in {5.1}). This is utilized by LineBuffer, and user programs may also use the control characters (refer to Delta Data manuals).

The Console and Printer streams both mask off parity bits, treats codes 32-127 as ordinary characters, and treat codes 9-13 according to the above description. Printer ignores any other codes, whereas Console passes them on. Thus the control characters for the Delta Data screen may also be used on output, but a switch can cause the them to be output as legible characters on the screen.

The stream function IntcodeFromRaw{7.3} is used to convert ASCII texts which contain parity bits or characters unknown in the internal code to internal ASCII. It creates streams which mask off parity bits and then filters illegal characters.

13.1. The ASCII character code.

The following table gives the decimal values and graphics of the characters available on the RIKKE implementation of BCPL.

0:		32:	SPACE	64:	@	96:	`
1:		33:	!	65:	A	97:	a
2:		34:	"	66:	B	98:	b
3:		35:	#	67:	C	99:	c
4:		36:	$	68:	D	100:	d
5:		37:	%	69:	E	101:	e
6:		38:	&	70:	F	102:	f
7:	BELL	39:	'	71:	G	103:	g
8:	BACKSPACE	40:	(	72:	H	104:	h
9:	TAB	41:	)	73:	I	105:	i
10:	NEWLINE	42:	*	74:	J	106:	j
11:		43:	+	75:	K	107:	k
12:	NEWPAGE	44:	,	76:	L	108:	l
13:	CR	45:	-	77:	M	109:	m
14:	SO	46:	.	78:	N	110:	n
15:	SI	47:	/	79:	O	111:	o
16:		48:	0	80:	P	112:	p
17:		49:	1	81:	Q	113:	q
18:		50:	2	82:	R	114:	r
19:		51:	3	83:	S	115:	s
20:		52:	4	84:	T	116:	t
21:		53:	5	85:	U	117:	u
22:		54:	6	86:	V	118:	v
23:		55:	7	87:	W	119:	w
24:		56:	8	88:	X	120:	x
25:		57:	9	89:	Y	121:	y
26:		58:	:	90:	Z	122:	z
27:	ESC	59:	;	91:	[	123:	{
28:		60:	<	92:	\	124:	|
29:		61:	=	93:	]	125:	}
30:		62:	>	94:	^	126:	~
31:		63:	?	95:	_	127:	DEL

14. DeadStart of RIKKE/Mathilda.

The procedure to Deadstart both RIKKE and Mathilda is given here. It is assumed, that power is on, on both RIKKE, WideStore, Mathilda and the Disc-drives.

RIKKE-DeadStart:

1) press STEP on RIKKE control panel.

2) insert the small deadstart tape (usually blue) in RC2000, and press RESET.

3) press DEADSTART button on RIKKE control panel.
Now the tape will be read in.

4) if you want to use default deadstart, type anything but 'no' when promted on TTY, else contact system-programmers.

5) login

Mathilda DeadStart:

1) press STEP on Mathilda control panel.

2) press DEADSTART-button.

If the machines won't deadstart, it may help to power them down and up again.
NOTE: This must only be done to Mathilda, WideStore and RIKKE. Powering the disks down and up must be left to system-programmers or technicians.

Mathilda must not be powered up after RIKKE is deadstarted, as this will cause WideStore to start strange block-tranfers.

15. List of References.

[1]: Jens Kristian Kjærgaard and Ib Holm Sørensen:
The RIKKE-BCPL compiler
DAIMI MD-36, August 1980

[2]: J.E.Stoy and C.Strachey:
An experimental Operating System for a Small Computer.
part 1: General Principles and Structure.
Computer Journal, Vol 15 (1972), Number 2, pp.117-124

[3]: Part 2 of [2]: Input/Output and File System.
Computer Journal, Vol 15 (1972), Number 3, pp.195-203

[4]: Ole Sørensen:
The emulated OCODE Machine for support of BCPL.
DAIMI PB-45, April 1975

[5]: Eric Kressel and Ejvind Lynning:
The I/O-nucleus on RIKKE-1.
DAIMI MD-21, October 1975

[6]: Jørgen Staunstrup:
A description of the RIKKE-1 system.
DAIMI PB-29, May 1974

[7]: Jens Kristian Kjærgaard and Ib Holm Sørensen:
The RIKKE editor
DAIMI MD-37, August 1980

[8]: Flemming Wibroe og Peter Schou:
Design og implementation af et filsystem for RIKKE.
DAIMI internal paper, May 1979

[9]: E.I.Organick:
Computer System Organisation.
Academic Press, New York, 1973

Wibroe, Flemming.
The RIKKE BCPL system / by Jens Kristian Kjoergard and Flemming Wibroe.-- Aarhus, Denmark: Computer Science Department, Aarhus University, 1980.
(DAIMI; MD-38)

I. Joint author. II. Title.