Introduction to Boss 2

4000/8000

Introduction to Boss 2

A/S REGNECENTRALEN
Documentation Department

Edition 0
July 1976
RCSL 42-i 0372

Author: Rune Einersen
Text Editor: Ejvind Johansson

KEY WORDS: RC 4000, RC 8000, Basic Software, Boss 2, an introduction to.

ABSTRACT: A description of the operating system Boss 2 as seen from the programmer's point of view. The main goals of Boss 2 are to ensure a fast and reliable execution of off-line jobs (batch), while at the same time serving many terminals in restricted or full time sharing mode.
Basic information needed to run simple jobs on the system is given.

Printed by A/S Regnecentralen, Copenhagen

Table of Contents

Introduction

The main purpose of Boss is to run jobs. First, we will briefly explain the three main ways in which jobs may be executed: as off-line, on-line, or internal jobs. In all cases, the actions to be taken during the execution are specified by the user in the so-called job file. In Chapter 1 we present typical examples of job files, and in Chapters 2 through 4 we give a more detailed description of the various ways in which jobs may be executed.

The job file of an off-line job is either a paper tape (a tape job) or a deck of cards (a card job). It is the operator who determines when the job files are to be loaded into the computer and enrolled for execution.

An on-line job is composed by a user working from an on-line terminal. He has one or more job files stored on disc. He may edit a file, list it, save it for the next run, select another file, and so on. When he wants a job to be executed, he enrolls the corresponding job file.

For the sake of completeness, we mention that the job file of an internal job is on disc, and that the internal job is enrolled by the new job-command or by a job running already (see Chapter 7).

1 Examples of Job Files

In this chapter we will present some typical job files. In Chapters 2, 3, and 4 we will show how a job file is executed as, alternatively, an off-line or an on-line job - but the job file itself may be the same in both cases.

The following examples are intended as a first aid for users familiar with some other computer system. Details are given in the Boss 2 User's Manual.

1.1 Simple Translation and Execution

In this first example we show the job file needed to translate and execute an Algol program. Annotations are given in the right-hand column.

```
job btj 308                    } Job specification (read by Boss)
p = algol                      } Call of translator (read by FP)
begin real a,b;                |
      read(in,a,b);            | Algol source program (read by algol)
      write(out,a**b);         |
end                            |
p                              } Call of object program (read by FP)
2 10                           } Data for program (read by p)
finis                          } End job (read by FP)
```

The first line is the so-called job specification. It is the only part of the job file which is interpreted by Boss, and it specifies the user name (btj) and the project number to be accounted for the run (308). Boss checks against the user catalog to see that btj is allowed to use project number 308. In this example the job specification is very simple, but if the job needed special resources this would have to be stated in the job specification.

The remainder of the job file is read and interpreted by various programs in turn as the job runs. Boss assures that the first program to read and interpret from the job file is the control program FP (File Processor).

FP reads the command 'p = algol' and as a consequence calls in the Algol translator. The Algol translator goes on reading the succeeding Algol program, translates it into an object program, and stores this as a file, p. Control is then returned to FP, which reads the command 'p' and as a consequence calls in the program in p. Now, our Algol program (alias p) reads the two numbers (2 and 10). p also returns control to FP, which reads the command 'finis' and tells Boss that the job is done.

The result of the entire run appears as follows (again with annotations to the right):

```
begin                                  }
algol end 16                           } Output from the Algol translator
      10.2400'2                        } Output from the object program.
end 16                                 } The apostrophe signals the tens-exponent.

end job btj0 4 sec   log ms date 1975.08.21 12.23.30 } Output from Boss
```

The last line shows that the job ran 4 seconds, the operator was ms, the job was terminated on the date and at the time specified, and in the normal way.

The output shown above is called the primary output, to distinguish it from other output like the output file p from the Algol compiler. Translation and execution in Fortran and assembly language (Slang) may be done along the same lines.

1.2 Listings During Execution

When the Algol translator detects an error, it identifies the bad spot in the source text by means of a line number (and an operand number within the line). The user will therefore find it helpful to have a listing of the program provided with the exact line numbers. This is obtained by replacing 'p = algol' with 'p = algol list.yes'.

It is possible to roughly keep track of the job flow by letting FP print every command just before performing them. This is accomplished by inserting the line 'mode list.yes' after the job specification.

With the above two changes the output from the job will now appear like this:

```
*p=algol list.yes
          1 begin real a,b;
          2              read(in,a,b);
          3              write(out,a**b);
          4 end

algol end 16
*p
   10.2400'2
end 16
*finis

end job btj0 4 sec   log ms date 1975.08.21 12.26.40
```

1.3 Several Translations and Executions

Assume that you want to translate two programs and execute the first one, then the second one, and finally the first one again. In that case you might use the following job file:

```
job btj 308                    } Job specification
prog1 = algol                  }
begin ...                      } Translate first program and store it as
end                            } the file prog1

prog2 = algol                  }
begin ...                      } Translate second program and store it
end                            } as the file prog2

prog1                          }
<data for prog1>               } Execute prog1
prog2                          }
<data for prog2>               } Execute prog2
prog1                          }
<data for prog1>               } Execute prog1 again
finis                          } End job
```

In this example we have also shown that longer file names may be used (like prog1 and prog2). In general, a file name consists of one small letter followed by a maximum of 10 small letters or digits.

1.4 Text Files on Backing Store

The user may store texts on the backing store, provided that the necessary backing store resources are available. Such texts may be loaded from cards or paper tape to the backing store, as explained in Chapters 2 and 3. Now, let us assume that the file tex3 contains:

```
p = algol
begin real a,b;
      read (in,a,b);
      write(out,a**b);
end
p
2 10
```

Then we may let FP and Algol read and execute this file by means of the following job:

```
job btj 308
i tex3
finis
```

The command 'i tex3' instructs FP to continue reading from the file tex3 in the usual way. Thus, the program is translated and executed just as in Section 1.1. When FP encounters the end of tex3, it returns to reading from the job file, and then the command 'finis' terminates the job as usual.

We use the term current input for the file from which FP reads for the moment. The effect of 'i tex3' is then that FP selects tex3 as current input. The selection of a new current input file may be carried on for several levels (recursively), for instance, if tex3 above contained a command like 'i tex4'.

The effect of Example 1.1 may also be achieved by means of the file tex5:

```
p = algol
begin ...
end
```

and the following job file containing the actual data:

```
job btj 308
i tex5
p
2 10
finis
```

1.5 Off-Line File Editing

A text file on the backing store may be edited and corrected in an on-line mode by Boss, or in an off-line mode by means of the program 'edit'. When working from an on-line terminal, it is usually simpler to correct the file by using Boss's on-line commands (see Chapter 4), but it is also possible to enroll a job that performs the task by means of the program 'edit'.

We are now going to show a job which transforms the previous file tex3 into a new version in newt. The changes are put in bubbles.

The new version is chosen to illustrate the editing and does not represent a useful file: The Algol program will loop endlessly trying to read beyond the last data set (2 3), because the object program - contrary to FP and the translator - does not stop automatically at the file end.

tex3:

```
p = algol
begin real a,b;
      read(in,a,b);
      write(out,a**b);
end
p
2 10
```

newt:

```
p = algol list.yes
begin real a,b;
rep: read(in,a,b);
      write(out,a**b);
     goto rep;
end
p
1 2
2 3
```

The following job performs the editing task, and translates and executes newt. It contains the command 'newt = edit tex3', which calls the edit program. Edit will copy the text in tex3 into newt, changing it at the same time according to the succeeding edit commands.

```
job btj 308
newt = edit tex3        } Call of the edit program with tex3 as source and newt as object file.

m e                     } This first edit command (marks empty) defines all characters to have their normal meaning. The command should be used as a standard.

1./algol/               } Copy from tex3 to newt until the line containing the pattern 'algol' is met.

r/ol/ol list.yes/       } In this line, replace the pattern 'ol' by 'ol list.yes'.

1./read/                } Copy on until the line with 'read'.

r/    /rep:/            } Replace the four first spaces by 'rep:'. Only the first occurrence of the pattern will be replaced, so possible blanks after the semicolon will do no harm.

1./end/                 } Copy on until the line with 'end'.

i/                      } Insert 'goto rep' before the line with 'end'.
goto rep;               } Strictly speaking, all the characters in the
/                       } bubble (including the terminating new line character) are inserted.

1./2 10/                } Copy on until the line with '2 10'. It does not matter if tex3 contains more than one space between the numbers.

d                       } Delete the line.

i/                      }
1 2                     } Insert these two new lines.
2 3                     }
/                       }

f                       } Copy on to the end of tex3, and return from the edit program to FP

i newt                  } Select newt as current input as in 1.4.

finis                   } End job.
```

This example contains the basic edit commands written in the so-called verification mode, in which the lines changed are printed out for verification. If you terminated an edit command by a comma (possibly followed by a new line) instead of by a new line alone, no verification would be printed for that correction. For instance you could write:

```
newt = edit tex3
m e                                  } This command should always be
l./algol/, r/ol/ol list.yes/,          followed by new line alone.
l./read/, r/     /rep:/
...
```

Here you would get a verification of the line with 'read', because no comma is present after the last edit command.

As you can imagine, the edit program will have trouble with inserting text with slashes, because the slash is taken for the 'end pattern' character in the example above. In such cases, you can simply use some other character for the slash in the 'r'-command or the 'l.'-command. For example, replacing 'a**b' by 'a/b' may look like this:

```
r-a**b-a/b-
```

where minus is the 'end pattern' character.

Another command, which uses line counting instead of pattern searching, is often useful: In order to copy one line, use the command 'l1'. Two lines are copied by 'l2', and so on. All lines until the bottom (end) of the file are copied by 'l b'. Blank lines are not included (counted) in the number of lines specified. For instance, the example above might be rewritten to this:

```
newt = edit tex3
m e
l./algol/, r/ol/ol list.yes/,
l2, r/   /rep:/,
l2, i/
      goto rep;
/,
l3, d, i/
1 2
2 3
/, f
```

In the same way, the command 'd1' will delete the current line and the next line, 'd2' will delete a total of three lines, and so on. The command 'd./pattern/' will delete all lines up to and including the first line containing 'pattern'. The command 'd b' will delete to the bottom (end) of the source file.

Finally, it is worth mentioning a command which is useful when larger portions of the text are to be moved around: The command 's1' will interrupt the present copying and continue again from the beginning of the source text (tex3 above). The object text (newt above) will continue to grow. As an example consider the contents of file t:

```
begin part a
...
end part a
begin part b
...
end part b
```

In order to exchange parts a and b, we can proceed like this:

```
nt = edit t
d./end part a/, 1 b,        } Delete part a, copy part b.
s1,                         } Start from top again.
1./begin part b/, d b,      } Copy part a, delete part b.
f
```

1.6 File Manipulation

A command like 'newt = edit tex3' creates a new file (newt) on the disc, unless you had a file of that name already. Such a new file is temporary, which means that it is cancelled as soon as your job terminates. In order to save it for later jobs, make it a permanent user file by means of this command:

```
scope user newt
```

This is not always possible, because Boss consults the user catalog to see if your project has resources available for the purpose.

The scope command may be used in a job (as an FP-command), and it may also be called directly from an on-line terminal (see Chapter 4).

Object files from translations will generally be placed on the drum, and then it is impossible to save them for later jobs. However, you may move them to the disc in this way:

```
p = algol
begin ...
end
ps = move p
scope user ps
```

In order to list a text file like tex3 on the printer, use the command 'convert tex3'. Again, this command is available as an FP-command and as an on-line terminal command (see Chapter 4).

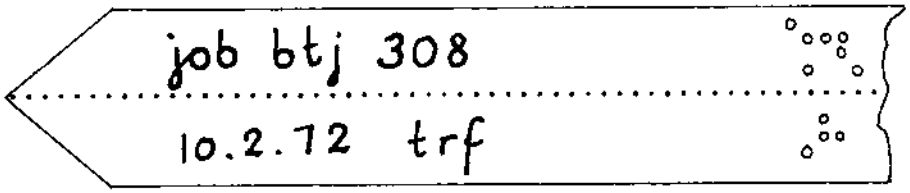
In order to get an index of all your present files, use the FP-command 'search own'.

In order to see whether you have a file named tex3, use the command 'lookup tex3' (FP-command or on-line, see Chapter 4).

In order to cancel the temporary file tex3, use the command 'clear temp tex3' (FP-command or on-line). A permanent user file is likewise cancelled by means of 'clear user tex3'.

2 Paper Tape Jobs

In case you want to submit a paper tape job, you simply punch the job file on paper tape and mark it clearly, like this:



As two different punch codes are in use, you have to specify the actual one clearly on the tape. The codes are 'tre' (which is the ISO-code with even parity) and 'trf' (which is the flexowriter code with odd parity).

The paper tape - together with possible data tapes - is forwarded to the computer room, where an operator at a suitable moment loads the job. The primary output appears on a printer and may be collected later.

If you have data tapes for the job (i.e., paper tapes which should be loaded during execution of the job), you have to ask Boss to load these tapes to the backing store. In case you have a data tape in ISO-code (tre) and one in flexo-code (trf), the job specification may look like this:

```
job btj 308    load tre tex2    load trf pap
```

The data tapes must be marked carefully as number 1 and number 2, with file names tex2 and pap, so that the operator can load them in the proper sequence. During the run the two tapes will be accessible as files 'tex2' and 'pap'. The coding of both files is now done in ISO-code, which is used as a standard within the system. The two files will be cancelled after the run, unless you make them permanent by means of the scope command (see Section 1.6).

3 Card Jobs

In case you want to submit a card job, you prepare a card deck consisting of a so-called job separation card followed by the job file (punched with one line to a card). The job separation card looks like this:

```
---job
```

where '---job' is in columns 1 through 6. Job file and job separation card must always be punched in EBCDIC code. The job separation card should be in a distinctive colour.

The job is now forwarded to the computer room, where an operator at a suitable moment stacks it with other card jobs and loads it. The job separation card ensures that Boss can separate the preceding job from yours, and its distinctive colour ensures that the operator also can. The primary output from the job appears on a printer and may be collected later.

If you want to have cards loaded to the backing store (possibly with other punch codes), this must be stated in the job specification. Assume, for instance, that you have a card file in EBCDIC code (crc below) and one to be loaded in binary (crb below). Binary means that all 12*80 bits of the card are to be stored - contrary to the normal way where each column is converted into a seven-bit character. Then the job specification should be:

```
job btj 308    load crc tex2    load crb binfile
```

The card files in question are now stacked after the job file, each of them preceded by a so-called file separation card, containing

```
---file
```

in columns 1 to 7.

During the run, the two files will be accessible as the backing store files 'tex2' and 'binfile'. The files will be cancelled after the run, unless you make them permanent by means of the scope command (see Section 1.6).

The entire card deck for the job will now look like this:

```
---job
job btj 308    load crc tex2    load crb binfile
<remainder of job file>
---file
<the file in EBCDIC code>
---file
<the file in binary>
```

4 On-Line Jobs

We are now going to demonstrate how the example of Section 1.1 may be run from an on-line terminal. The Boss 2 User's Manual tells more about terminals and their operation; here it suffices to know that a terminal can transmit an attention signal when we want the attention of the operating system.

Below we indicate by ■ that we have pushed the attention button and by -> that we type in a line terminated by a new line character.

Assuming that the terminal is connected to RC 4000 and that Boss is in the computer, the conversation may look like this:

```
■ att -> boss                          } Activate Boss. This is the
  type user name and project number      login procedure. Boss ac-
-> btj 308                               knowledges the login by re-
  in: 1975.08.21 15.46                   turning the time. We are
                                         now connected to an empty
                                         job file.
-> 10 p = algol                        }
-> 20 begin real a, b;                   Here we compose the job
-> 30       write(out,a**b);             file. Each line of the file
-> 40 end                                is preceded by an identifi-
-> 50 p                                  cation number.
-> 60 2  10                            }
-> 1000 finis                          } At this point, the file is
                                         incomplete.
-> 25       read(in,a,b);               } Here we correct it by in-
                                         serting a line.
-> go                                  } This on-line command en-
  finis btj0 at  15  52                  rolls the job. Boss returns
                                         the estimated finishing time.
  begin                                }
  algol end 16                           Output from the algol translator.
    10.2400'2                          } Output from the object program.
  end 16                               }
  end job btj 4 sec log ms date 1975.08.21 15.51.00 } Output from Boss
```

The system is now ready for new corrections to the job file and new runs. For instance, we could replace line 60 by three lines in order to get more results:

```
->55 3 10
->60 p
->65 5 10
->go
```

When we want to terminate the session, we type 'logout'.

The job file we have composed above does not include a job specification, which would be unnecessary since the information is available to Boss from the login procedure. But if we want to use special resources in the job, we may extend the job file with a job specification and enroll the job by means of the 'run' command:

```
->1 job btj 308 load trf tex2
->run
```

(This example presupposes that the operator has a paper tape for use.)

When we have enrolled a job, we cannot just go on typing new commands until the run is over (we use the phrase that the terminal is passive during the run). If we want to tell Boss something in this case, we must push the attention button, after which Boss reads one command and returns to the passive state. For instance, we can ask for immediate termination of the job in this way:

```
■ >>
->kill
```

We can also get a list of the jobs waiting for execution if we type 'display' instead of 'kill'. However, we cannot change the job file or enroll new jobs while the terminal is passive.

4.1 Correction of Typing Errors

On all terminals, two keys are reserved for the purpose of character cancellation and line cancellation. On most terminals, the characters are ampersand (&) and percent (%). Thus, in order to cancel the latest character typed, type &. In order to cancel several of the latest characters (in the current line only), type & the corresponding number of times. In order to cancel the entire line, type %.

If you do not type anything for a certain period (time out period, installation dependent), Boss will get the portion typed in and will continue reading a new portion. This time out is clearly audible on most terminals. However, line cancellation and character cancellation works only within a portion at present. This may cause you correction trouble in installations with a short time out period.

In all cases - even after time out periods - an attention signal will cancel the current line.

4.2 Get, Save, List, Verify, Lookup

When we have composed a job file, we can store it on the backing store for later use. For instance, it will be saved with the name pr3 when we type this line:

```
save pr3
```

A file saved in this way disappears when we 'log out', unless we make it permanent, for instance by typing

```
scope user pr3
```

This is not always possible, because Boss consults the user catalog to see if the project has resources left for the purpose.

In order to get the file tex4 from the backing store into the job file, type

```
get tex4
```

This will also cause the lines of the file to be identified by their numbers: 10, 20, ... It is now possible to modify the job file by typing line number and line contents, just as above.

In order to get an empty job file, type

```
clear
```

In order to list the current job file, type

```
list
or      list 140
or      list 140 200
```

The second command lists from line 140 on. The last command lists from line 140 to line 200. The listing may always be terminated by a push on the attention key.

In order to verify a correction made or a line in the job file, type

```
verify
or      verify 40
or      verify 40 3
```

The first command lists the line last used, e.g., edited or typed in. The second command lists line 40 and the third command lists 3 lines starting with line 40.

Suppose we have a permanent file named 'prog' on the backing storage, which another user in our project wants to utilize. First we want to check that this file exists, and we type

```
lookup prog
or      lookup user prog
```

and we get the answer:

```
prog=set 36 disc 1975.821 16.15 0 0 0; user
; 336 46 3 100 199
```

provided no login file with the name 'prog' exists.

In order to make the file visible to all the users in our project we type

```
scope project prog
```

and in order to check what has happened to the file we type

```
lookup prog
```

or

```
lookup project prog
```

to which we get the answer

```
prog=set 36 disc 1975.821 16.15 0 0 0; project
; 336 46 3 100 199
```

If we this time had typed the command

```
lookup user prog
```

we would have got the answer

```
file does not exist
```

The output from our latest job is always stored in a file named primout. In most cases this is not interesting, but if the job was 'killed' while producing a lot of output on the terminal, primout usually contains the entire output file, because the terminal works much slower than the job execution process. By means of 'get' and 'list' we may then pick out parts of the output for closer examination.

5 User Catalog

As mentioned in the preceding chapters, Boss checks the user catalog to see if the user is allowed to run with the project number he specifies. However, the user catalog may contain a lot more information, for instance, which resources his job will need as a standard, and which resources his job may request at most.

In order to be allowed to use Boss, you will normally have to fill in a form, so that the computer department can enter you into the user catalog.

6 Job Scheduling

All the jobs enrolled at a given moment are in one stage or another of their execution. Most jobs will spend some time in waiting for resources (core store, tape stations, etc.). As a general rule, a job will have to wait longer the more resources and the more run time it demands. In the simple examples presented above, we did not specify the estimated run time, because it was assumed to be standard, but if a short turn-around time is desired, an accurate time specification may be important.

A job with short run time and a modest demand of resources will be able to bypass a large job, even if the large job has begun execution in the core store. On the other hand, small jobs may be delayed by other small jobs. However, large jobs cannot be delayed indefinitely, not even by a steady stream of small jobs.

From a terminal it is always possible to get a list of the jobs enrolled at the present moment by means of the display-command. This list will contain Boss's latest estimate of the finishing times.

7 Internal Job

It is possible to enroll a job for off-line execution, either from a terminal by means of an on-line command, or from another job by means of an FP-command like this:

```
newjob file7
```

Here, file7 must be a permanent file holding the job file of the new job. The new job specification will normally look like this:

```
job btj 1 308
```

A job index (1) is added between the user name and the project number in order for Boss to be able to tell the new job from the old job, which carried the job specification 'job btj 308' with an implicit job index of 0.

As an example, let us assume that we work from a terminal and want to execute a long job while we use the terminal for something else. First, we generate the job file of the long job as a permanent file named 'long', and then we enroll it as an off-line job by means of the newjob-command, for instance in this way:

```
10 job btj 1 308
20 p = algol
30 ...
save long
scope user long
newjob long
```

It may happen that the newjob-command is rejected by Boss because the job queue is completely filled up already. We will then have to try again later, or run the job as an on-line job instead.

Notice, that the primary output from the long job will appear on printer. If we want to see some output on the terminal, we may let the long job switch to producing output in a permanent file 'longout' by means of this contents of 'long':

```
job btj 1 308
o longout
scope user longout
p = algol
...
o c
finis
```

When the job is completed, we may list the output on the terminal in this way:

```
get longout
list
```

We may check from time to time whether the job is completed by means of this on-line command:

```
display btj1
```

The parameter btj1 identifies the job with user name btj and job index 1. When display prints 'no such job', the job is completed.

From an off-line job it is also possible to make an internal job as a so-called replace job, in which case the termination of the old job leaves the new job in its place.

8 Files on Magnetic Tape

Often the backing storage is too small to hold all user's files, but may still be large enough to hold all files for running jobs and users logged in at a terminal. In such cases, the users should store their files on magnetic tape, load them from the tape to the disc at the beginning of a session, and save them on the tape after the session (if they have been changed).

In order to avoid the superfluous saving of scratch files produced during the job, the convention is adopted that only login files are saved on the tape in the normal case. Files produced by means of the on-line command 'save' will have login scope; files created inside a job may be given login scope by means of the scope command, i.e., 'scope login file1'.

Assume that you have had permission from the computer department to use the magnetic tape mt285032. Then the FP-command

```
save mt285032.1.label.btj
```

will ask the operator for the tape. The tape is given a user label 'btj', and all login files of the user are saved on file 1 of the tape. The program prints a log of the label and the files saved.

In a later job you may use the FP-command

```
load mt285032.1
```

to reestablish the files. The next version of the files must be saved on file 2 (the next again on file 3, and so on) by means of this FP-command:

```
save mt285032.2
```

You should only use the parameter 'label.btj' the first time you save on the tape.

In jobs using magnetic tapes, the job specification should state the number of tape stations needed:

```
job btj 308 stations 1
```

READER'S COMMENTS

Introduction to Boss 2
RCSL 42-i 0372

A/S Regnecentralen maintains a continuous effort to improve the quality and usefulness of its publications. To do this effectively we need user feedback - your critical evaluation of this manual.

Please comment on this manual's completeness, accuracy, organization, usability, and readability:

Do you find errors in this manual? If so, specify by page.

How can this manual be improved?

Other comments?

Please state your position: _____

Name: _____ Organization: _____

Address: _____ Department: _____

Date: _____

Thank you!

- Fold here -

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

Affix
postage
here

A/S REGNECENTRALEN
Marketing Department
Falkoner Allé 1
2000 Copenhagen F
Denmark

INTERNATIONAL

EASTERN EUROPE
A/S REGNECENTRALEN
Glostrup, Denmark, (02) 96 53 66

SUBSIDIARIES

AUSTRIA
RC – SCANIPS COMPUTER
HANDELSGESELLSCHAFT mbH
Vienna, (0222) 36 21 41

FINLAND
OY RC – SCANIPS AB
Helsinki, (90) 31 64 00

HOLLAND
REGNECENTRALEN (NEDERLAND) B.V.
Rotterdam, (010) 21 62 44

NORWAY
A/S RC – SCANIPS
Oslo, (02) 35 75 80

SWEDEN
RC – SCANIPS AB
Stockholm, (08) 34 91 55

SWITZERLAND
RC – SCANIPS (SCHWEIZ) AG
Basel, (061) 22 90 71

UNITED KINGDOM
REGNECENTRALEN LTD.
London, (01) 439 9346

WEST GERMANY
RC – GIER ELECTRONICS G.m.b.H.
Hannover, (0511) 6 79 71

REPRESENTATIVES

FRANCE
SORED S.a.r.l.
Nanterre, (1) 204 2800

HUNGARY
HUNGAGENT AG
Budapest, 88 61 80

TECHNICAL ADVISORY REPRESENTATIVES

POLAND
ZETO
Wroclaw, 45 431

RUMANIA
I.I.R.U.C.
Bucharest, 33 21 57

HUNGARY
NOTO-OSZV
Budapest, 66 84 11

A/S REGNECENTRALEN

HEADQUARTERS: FALKONER ALLE 1; DK-2000 COPENHAGEN F · DENMARK
Phone: (01) 10 53 66 · Telex: 16282 rc hq dk · Cables: regnecentralen