IA

stichting mathematisch centrum

MC

IA

DEPARTMENT OF COMPUTER SCIENCE ID 4/75 OCTOBER

INFAL, AN INFORMATICS LIBRARY

W. BÖHM (ED.)

2e boerhaavestraat 49 amsterdam

Printed at the Mathematical Centre, 49, 2e Boerhaavestraat, Amsterdam.

The Mathematical Centre, founded the 11-th of February 1946, is a non-profit institution aiming at the promotion of pure mathematics and its applications. It is sponsored by the Netherlands Government through the Netherlands Organization for the Advancement of Pure Research (Z.W.O), by the Municipality of Amsterdam, by the University of Amsterdam, by the Free University at Amsterdam, and by industries.

AMS(MOS) subject classification scheme (1970): 68A15, 68A10

ACM -Computing Reviews- categories: 3.73, 3.74, 4.40, 4.41

Infal, an informatics library

W. Böhm (ed.)

ABSTRACT

Infal, an informatics library of programs in the field of informatics is maintained on the CONTROL DATA CYBER70 system. It contains a variety of main programs, subprograms and macros, some of which were taken over from the X8 MILLI system.

This guide contains the documentation of each program, which contains a brief description and specifies how to run the program. In case of a subprogram it states the language(s) from which it can be called. It also tells you how to get the source text or the documentation of that specific program.

KEYWORDS AND PHRASES:

Program library,informatics

Infal, an informatics library

Contents

Infal, an informatics library

Introduction.

The Infal system consists of :

1. A program library on a permanent file with permanent file name "infal" and owner identification "matcen". It is created and maintained with the EDITLIB utility [1].

2. A file containing documentations, source texts and a backup of the program library on tape. It is created and maintained with the MODLIB system [2]. The tape has tape number "ns8097t" and user identification "infalmc".

Before executing a program using infal one should:
attach,infal,id=matcen.
Put infal in a library set [1], for instance with the command:
library,infal.

Using MODLIB, the source text or documentation of a specific program can be obtained by retrieving that file from the tape. The filenames of the documentation and the source text are given in the documentation. Suppose that we would want to get the source text of the program copysfs, we would:
specify in the job command a nine track tape device
attach,modlib
(modlib will request the tape)
retrieve the file with the command:
modlib,ns8097t,infalmc,x,copysfs.

References

1. SCOPE reference manual, version 3.4.1 CONTROL DATA
2. SARA publication: Voorlopige utility publikatie

Author: H.L. Oudshoorn

Source and documentation: mccsrc, mccdoc

Revisor: P. Beertema

Institute: Mathematical Centre

Date received: 06/11/74

Brief description:
mcc contains the following procedures inherited from the X8 MILLI system:

exit :	Terminates the execution of the program
available:	Gives unused space (fl - top of runtime stack)
date :	Gives (day × 100 + month)× 100 + year - 1900
real time:	Gives (hour × 100 + minute)× 100 + second
time left:	Gives (time par in jobcard - used time) in seconds
setrandom:	Initializes random generation
random:	Gives a random number

Keywords: exit,available,date,time,random

Type: ALGOL code procedures, in compass

Calling sequence EXIT:
"procedure" exit ; "code" 11010;
Calling sequence AVAILABLE:
"integer" "procedure" available ; "code" 11011;
Calling sequence DATE:
"integer" "procedure" date ; "code" 11012;
Calling sequence REAL TIME:
"integer" "procedure" real time; "code" 11013;
Calling sequence TIME LEFT:
"real" "procedure" time left; "code" 11016;
Calling sequence SETRANDOM:
"procedure" setrandom(x);"code" 11014;
x: input parameter
initializes starting value for following calls of random.

Calling sequence RANDOM:
"real" "procedure" random; "code" 11015;

Subprograms used: None

Method RANDOM:
new value := big number × oldvalue (mod 2××48)

Author: H.L. Oudshoorn

Source and documentation: bitsrc, bitdoc

Revisor: P. Beertema

Institute: Mathematical Centre

Date received: 06/11/74

Brief description:
Bitprcs contains 5 procedures that manipulate the rightmost 48 bits of machine words. If x is a variable of type integer or real, then "integer" "array" d x[0:47] represents the 48 bits of the unpacked variable. All procedures deliver a packed and normalized value.

```
AND(A,B):
"FOR" I := 0 "STEP" 1 "UNTIL" 47 "DO"
   D AND[I] := "IF" D A[I] = 0
               "T EN" 0
               "ELSE" D B[I]
OR(A,B):
"FOR" I := 0 "STEP" 1 "UNTIL" 47 "DO"
   D OR[I] := "IF" D A[I] = 1
              "THEN" 1
              "ELSE" D B[I]
XOR(A,B):
"FOR" I := 0 "STEP" 1 "UNTIL" 47 "DO"
   D XOR[I] := "IF" D A[I] = D B[I]
               "THEN" 0
               "ELSE" 1
BITSTRING(U,L,A):
"FOR" I := 47 "STEP" -1 "UNTIL" U-L+1 "DO"
   D BITSTRING[I] := 0;
"FOR" I := U-L STEP" -1 "UNTIL" 0 "DO"
   D BITSTRING[I] := D A[I+L]
SET(C,U,L,A):
"IF" I = 47 "AND" L = 0
 "THEN" "FOR" I := 47 "STEP" -1 "UNTIL" 0 "DO"
          D SET[I] := D C[I]
 "ELSE"
    "BEGIN"
     "FOR" I := 47 "STEP" -1 "UNTIL" U+1 "DO"
         D SET[I] := D A[I];
     "FOR" I := U "STEP" -1 "UNTIL" L "DO"
         D SET[I] := D C[I-L];
     "FOR" I := L-1 "STEP" -1 "UNTIL" 0 "DO"
         D SET[I] := D A[I]
    "END"
```

Keywords: Bit manipulation

Type: ALGOL code procedures

Calling sequence AND, OR and XOR:

```
"integer" "procedure" and(a,b); "code" 12000;

"integer" "procedure" or(a,b); "code" 12001;

"integer" "procedure"xor(a,b); "code" 12002;
```

Calling sequence BITSTRING:

```
"integer" "procedure" bitstring(u,l,a); "code" 12003;

       u : upper limit
       l : lower limit
       a : source
```

Calling sequence SET:

```
"integer" "procedure" set(c,u,l,a); "code" 12004;

       c : source
       u : upper limit receiver
       l : lower limit receiver
       a : receiver
```

Subprograms used: None

Author: Ger ten Velden

Source and documentation: neat60s, neat60d

Institute: Mathematical Centre

Date received: 01/05/75

Brief Description:
This program is an ALGOL 60 program for automatic text layout of ALGOL 60 (CD-ALGOL) programs.
The requirement of well readable output, which seems to be a highly subjective matter, has been met by generating a text layout which displays the syntactical structure of the source text.
The results, obtained by this syntax oriented method, do rather conform to the usual, hand prepared, ALGOL 60 texts, used for publication purposes.

Keywords:
Text layout, ALGOL 60 programs, syntactical structure.

Type: ALGOL 60 main program.

Calling sequence:
```
attach,infal,id=matcen.
library(infal)
attach,source,id='id'.
attach,options,id='id'.  SEE OPTIONS
neata60.
```

Input Output:
Two extra channel definitions are required for:

```
  1) source text input: channel 50  (channel,50=source,p80,r)
  2) neat output:       channel 51  (channel,51=neat,p80,r).
```

The channels 60 and 61 (files INPUT and OUTPUT) are used to specify options. Options also can be read from an extra channel:

```
  3) options:           channel nn  (channel,nn=options,p80,r).
```

Options:
Each option starts at the beginning of a line and is terminated (not necessarily at the same line) by a period. All text between this period and the first following 'end of line' will be considered as comment. Options are terminated by the end option (e. or end.) or 'end of file' (actually 'end of record')

Options concern:

1) option channel: o, <unsigned integer>.
All options from the specified channel are read, starting at the second line (the first line is skipped). Afterwards, options are read again from the original channel, following the o option.

2) width of source text: k = <unsigned integer>.
k represents the number of characters per line. Exceeding characters are skipped. (standard: k = 72)

3) width of resulting text: w = <unsigned integer>.
(standard: w = 72).

4) ALGOL symbols (letters and digits excepted).
ALGOL symbols are followed by parameters e.g. "begin",$'bgn'$, t6. and (, 19.

Parameters concern:

4.1) neat representation: the desired output of the ALGOL symbol is enclosed between dollars,

4.2) tabulation value: t<digit>, this value is meaningful for some ALGOL symbols (see table below),

4.3) cohesion values: l<digit> and r<digit>, specifying the left and right hand side of the ALGOL symbol;

<digit> = 0 means: no space,
<digit> > 0 means: one space,
<digit> < 9 means: relative cohesion value of the space,
<digit> = 9 means: cohesion value infinite (no breakpoint).
These values are meaningful for (see table below):

a) operators, indicating the priority of the operator,

b) pseudo operators (, : := "step" "until" "while"), indicating the priority of the pseudo operator,

c) only a left cohesion value is meaningful for:
(, when preceded by a procedure identifier,
[, when preceded by an array identifier.

Input:
The source text must be an ALGOL 60 program or procedure declaration. The source text ends at an occurrence of the ALGOL symbol eop or at end of file (end of record). Comments before and after the program (or declaration) are skipped. The hardware representations of ALGOL 60 symbols are listed in the table below (including some alternatives).

Output:
The desired output of the ALGOL 60 symbols can be specified by means of some options. The standard representations are listed in the table below.

ALGOL SYMBOL, INPUT AND STANDARD OUTPUT REPRESENTATION	ADDITIONAL INPUT REPRESENTATIONS	TABULATION VALUE (STANDARD)	COHESION VALUES (STANDARD)	
			LEFT	RIGHT
+		-	6	6
-		-	6	6
×	"TIMES"	-	7	7
/		-	7	7
××	"POWER"	-	8	8
//	"/"	-	7	7
	"DIV"			
>	"GREATER"	-	5	5
	"GR"			
	"GT"			
>=	"NOTLESS"	-	5	5
	"GE"			
	"GQ"			
=	"EQUAL"	-	5	5
	"EQ"			
¬=	"NOTEQUAL"	-	5	5
	"NE"			
	"NQ"			
<=	"NOTGREATER"	-	5	5
	"LE"			
	"LQ"			
<	"LESS"	-	5	5
	"LS"			
	"LT"			
"AND"		-	4	4
"OR"		-	3	3
"EQUIV"	"EQV"	-	1	1
¬	"NOT"	-	-	-
"IMPL"	"IMPLIES"	-	2	2
	"IMP"			
.		-	-	-
,		-	0	1
:	..	-	3	3
;	.,	-	-	-
" (LOWER TEN)	_	-	-	-

(		1	1	-
:=	..=	-	0	2
)		-	-	-
[	(/	1	1	-
]	/)	-	-	-
"("		-	-	-
")"		-	-	-
"TRUE"		-	-	-
"FALSE"		-	-	-
"GOTO"		-	-	-
"IF"		5	-	-
"THEN"		7	-	-
"ELSE"		7	-	-
"FOR"		6	-	-
"DO"		5	-	-
"STEP"		-	3	4
"UNTIL"		-	3	4
"WHILE"		-	3	4
"COMMENT"	"CO"	4	-	-
"BEGIN"	"BGN"	4	-	-
"END"		-	-	-
"OWN"		4	-	-
"BOOLEAN"	"BOOL"	4	-	-
"INTEGER"	"INT"	4	-	-
"REAL"		4	-	-
"ARRAY"	"AR"	4	-	-
"SWITCH"		4	-	-
"PROCEDURE"	"PROC"	4	-	-
"STRING"		4	-	-
"LABEL"		4	-	-
"VALUE"	"VAL"		-	-
"CODE"		-	-	-
"ALGOL"		-	-	-
"FORTRAN"		-	-	-
"EOP"		-	-	-

Required central memory: 60000b.

Running time: 3 to 4 lines per second.

Method and performance:
Arranging the source text into neat lines is performed according to the following two main rules:

1) if some syntactical unit (think of a compound statement) doesn't fit in one line by itself, that unit will be subdivided into a number of subunits (the composing statements of the compound statement); each following line will contain as many as possible of these whole subunits.

2) if some subunit doesn't fit in one line by itself, that subunit is partitioned into several lines, according to rule 1), and the following subunit will start at the beginning of a new line (a small statement should never be hidden at the end of the last line of a preceeding large statement, nor should, as a consequence, a dummy statement).

The following actions are performed until the input has been exhausted.

The program fills a circular buffer, while parsing the ALGOL 60 text according to a simplified grammar. After the buffer has been filled the program searches the break point, which is the rightmost symbol with the lowest cohesion value. The indentation for the next line is calculated and the current line is put out.

Some constructions, viz. comments and strings, give rise to troubles because of the lack of any internal syntactical structure. Additional information should be available to determine whether existing layout characters in the source text have to be retained, or other ones may be inserted instead. These two cases are treated as follows:

<u>comments</u>:

On each occurrence of one or more consecutive spaces, one space is generated instead. If one comment doesn't fit in one line the comment is subdivided into several lines, breaking each line after the last space that fits on that line; continuation lines are indented normally,

<u>strings</u>:

All characters, including spaces are retained (the exception being spaces between the composing characters of (nested) string quotes). If one string doesn't fit in one line, the string is subdivided into several lines, breaking each line after the last character that fits in that line; continuation lines are not indented.

Example of use:
The following procedure declaration will be used to demonstrate the use of neata60 and its options. The short alternative representations for ALGOL symbols (see table) have been used in the source.

file source:

```
"PROC"QUICK SORT(A,I,J);"VAL"I,J;"AR"A;"INT"I,J;"BGN""INT"P,Q;"REAL"X,Y,
T;"IF"J-I>1"THEN""BGN"T:=A[I];Q:=J;"FOR"P:=I+1"STEP"1"UNTIL"Q"DO""BGN"X:
=A[P];"IF"X>T"THEN""BGN""FOR"Q:=Q"STEP"-1"UNTIL"P"DO""BGN"Y:=A[Q];"IF"Y<
T"THEN""BGN"A[P]:=Y;A[Q]:=X;Q:=Q-1;"GOTO"L"END""END";Q:=P-1;"GOTO"M"END"
;L:"END";M:A[I]:=A[Q];A[Q]:=T;QUICK SORT(A,I,Q-1);QUICK SORT(A,Q+1,J)"EN
D""ELSE""IF"J-I=1"THEN""BGN"X:=A[I];Y:=A[J];"IF"X>Y"THEN""BGN"A[I]:=Y;A[
J]:=X"END""END""END"SORT;
```

First, we neat the source without specifying any option.

a)interactively:

```
attach,infal,id=matcen
attach,source,id=gtv
connect,input
xeq,libload=infal,neata60,execute
channel,50=source,p80,r
channel,51=sort,p80,r
channel,end
end.
catalog,sort,id=gtv
```

b)batch job:

```
attach,infal,id=matcen.
attach,source,id=gtv
library,infal.
neata60.
catalog,sort,id=gtv.
xend of record
channel,50=source,p80,r
channel,51=sort,p80,r
xend of file
```

The result on file output (case b) is:

```
CHANNEL,60=INPUT,P80,R
CHANNEL,61=OUTPUT,P136,PP60,R
CHANNEL,50=SOURCE,P80,R
CHANNEL,51=SORT,P80,R

ALGOL60 NEATER.
OPTION:
SOURCE DECK ENDS AT LINE     7

END OF ALGOL RUN  ×V3.1×
```

The result on file sort is:

```
"PROCEDURE" QUICK SORT (A, I, J); "VALUE" I, J; "ARRAY" A;
"INTEGER" I, J;
"BEGIN" "INTEGER" P, Q; "REAL" X, Y, T;
    "IF" J - I > 1 "THEN"
    "BEGIN" T:= A [I]; Q:= J;
        "FOR" P:= I + 1 "STEP" 1 "UNTIL" Q "DO"
        "BEGIN" X:= A [P];
            "IF" X > T "THEN"
            "BEGIN"
                "FOR" Q:= Q "STEP" - 1 "UNTIL" P "DO"
                "BEGIN" Y:= A [Q];
                    "IF" Y < T "THEN"
                    "BEGIN" A [P]:= Y; A [Q]:= X; Q:= Q - 1; "GOTO" L
                    "END"
                "END";
                Q:= P - 1; "GOTO" M
            "END";
        L:
        "END";
    M:  A [I]:= A [Q]; A [Q]:= T; QUICK SORT (A, I, Q - 1);
        QUICK SORT (A, Q + 1, J)
    "END"
    "ELSE"
    "IF" J - I = 1 "THEN"
    "BEGIN" X:= A [I]; Y:= A [J];
        "IF" X > Y "THEN" "BEGIN" A [I]:= Y; A [J]:= X "END"
    "END"
"END" SORT;
```

The same source will be neated, demonstrating the effect of some options. After the 'channel,end' of example a, we insert the following options:

(, LO.	NO SPACE BETWEEN PROC IDENTIFIER AND (
(/,LO.	NO SPACE BETWEEN ARRAY IDENTIFIER AND [
, ,RO.	NO SPACE AT THE RIGHT HAND SIDE OF ,
+ ,LO,RO.	NO SPACES AROUND +
- ,LO,RO.	NO SPACES AROUND -
:=,RO.	NO SPACE AT THE RIGHT HAND SIDE OF :=

Result on file output:

```
CHANNEL,60=INPUT,P80,R
CHANNEL,61=OUTPUT,P136,PP60,R
CHANNEL,50=SOURCE,P80,R
CHANNEL,51=SORT,P80,R
CHANNEL,END
ALGOL60 NEATER.
OPTION:
(,LO.
OPTION:
(/,LO.
OPTION:
,,RO.
OPTION:
+,LO,RO.
OPTION:
-,LO,RO.
OPTION:
:=,RO.
OPTION:
END.
SOURCE DECK ENDS AT LINE     7
END OF ALGOL RUN  ×V3.1×
```

Result on file sort:

```
 "PROCEDURE" QUICK SORT(A,I,J); "VALUE" I,J; "ARRAY" A; "INTEGER" I,J;
 "BEGIN" "INTEGER" P,Q; "REAL" X,Y,T;
     "IF" J-I > 1 "THEN"
     "BEGIN" T:=A[I]; Q:=J;
         "FOR" P:=I+1 "STEP" 1 "UNTIL" Q "DO"
         "BEGIN" X:=A[P];
             "IF" X > T "THEN"
             "BEGIN"
                 "FOR" Q:=Q "STEP" - 1 "UNTIL" P "DO"
                 "BEGIN" Y:=A[Q];
                     "IF" Y < T "THEN"
                     "BEGIN" A[P]:=Y; A[Q]:=X; Q:=Q-1; "GOTO" L "END"
                 "END";
                 Q:=P-1; "GOTO" M
             "END";
         L:
         "END";
     M:  A[I]:=A[Q]; A[Q]:=T; QUICK SORT(A,I,Q-1); QUICK SORT(A,Q+1,J)
     "END"
     "ELSE"
     "IF" J-I = 1 "THEN"
     "BEGIN" X:=A[I]; Y:=A[J];
         "IF" X > Y "THEN" "BEGIN" A[I]:=Y; A[J]:=X "END"
     "END"
 "END" SORT;
```

Now we use a separate file, called 'options', containing a number of options:

```
  THIS FIRST LINE WILL BE SKIPPED AS A COMMENT
 "PROC", $PROC$, T2.
 "VAL",  $VAL$,  T2.
 "AR",   $ARRAY$,T2.
 "INT",  $INT$,  T2.
 "REAL", $REAL$, T2.
 "BGN",  $BEGIN$,T2.
 "END",  $END$.
 "IF",   $IF$,   T3.
 "THEN", $THEN$, T5.
 "ELSE", $ELSE$, T5.
 "FOR",  $FOR$,  T4.
 "STEP", $STEP$, T5.
 "UNTIL",$UNTIL$,T6.
 "DO",   $DO$,   T3.
 "GOTO", $GOTO$.
```

The following interactive commands are used to produce a very narrow version of the sorting procedure:

```
attach,infal,id=matcen
attach,source,id=gtv
attach,options,id=gtv
connect,input
xeq,libload=infal,neata60,execute
channel,50=source,p80,r
channel,51=sort,p80,r
channel,52=options,p80,r
channel,end
(,10.
[,10.
,,r0.
,10,r0.
-,10,r0.
options,52.   Only the first letter of options is examined
w=32.         Width of result = 32
end.
```

The result on file output is:

```
CHANNEL,60=INPUT,P80,R
CHANNEL,61=OUTPUT,P136,PP60,R
CHANNEL,50=SOURCE,P80,R
CHANNEL,51=SORT,P80,R
CHANNEL,52=OPTIONS,P80,R
CHANNEL,END

ALGOL60 NEATER.
OPTION:
(,LO.
OPTION:
[,LO.
OPTION:
,,RO.
OPTION:
+,LO,RO.
OPTION:
-,LO,RO.
OPTION:
OPTIONS,52.
OPTION:
"PROC",$PROC$,T2.
OPTION:
"VAL",$VAL$,T2.
OPTION:
"AR",$ARRAY$,T2.
OPTION:
"INT",$INT$,T2.
OPTION:
"REAL",$REAL$,T2.
OPTION:
"BGN",$BEGIN$,T2.
OPTION:
"END",$END$.
OPTION:
"IF",$IF$,T3.
OPTION:
"THEN",$THEN$,T5.
OPTION:
"ELSE",$ELSE$,T5.
OPTION:
"FOR",$FOR$,T4.
OPTION:
"STEP",$STEP$,T5.
OPTION:
"UNTIL",$UNTIL$,T6.
OPTION:
"DO",$DO$,T3.
OPTION:
```

```
"GOTO",$GOTO$.
OPTION:


OPTION:
W=32.
END.
SOURCE DECK ENDS AT LINE     7

END OF ALGOL RUN  ×V3.1×
```

The result on file sort is:

```
 PROC QUICK SORT(A,I,J);
 VAL I,J; ARRAY A; INT I,J;
 BEGIN INT P,Q; REAL X,Y,T;
   IF J-I > 1 THEN
   BEGIN T:= A[I]; Q:= J;
     FOR P:= I+1 STEP 1 UNTIL Q
     DO
     BEGIN X:= A[P];
       IF X > T THEN
       BEGIN
         FOR Q:=
             Q STEP - 1 UNTIL P
         DO
         BEGIN Y:= A[Q];
           IF Y < T THEN
           BEGIN A[P]:= Y;
             A[Q]:= X; Q:= Q-1;
             GOTO L
           END
         END;
         Q:= P-1; GOTO M
       END;
     L:
     END;
   M: A[I]:= A[Q]; A[Q]:= T;
     QUICK SORT(A,I,Q-1);
     QUICK SORT(A,Q+1,J)
   END
   ELSE
   IF J-I = 1 THEN
   BEGIN X:= A[I]; Y:= A[J];
     IF X > Y THEN
     BEGIN A[I]:= Y; A[J]:= X
     END
   END
 END SORT;
```

As another example we mention the neated source text of neata60. This program has been neated without specifying any option.

Author: Dick Grune

Source and documentation: schaafs, schaafd

Revisor: Jan Karel Schreuder

Institute: Mathematical Centre

Date received: 31/01/75

Last revision: 23/06/75

Brief description:

TEXTSCH (short for "tekstschaaf", dutch for text plane) is a program that justifies and paginates texts written in a natural language. It is possible to suppress justification for certain parts of the input; pagination, however, will always take place.

The input should consist of chapters, made up of paragraphs. This structure must be indicated by special symbols on the input file. Each chapter starts at a new page. The program takes care of the pagination within the chapters. A heading will be put at the top of each page: a text supplied by the user, at the left and a chapter number and page number at the right. The chapter number must be specified at the beginning of each chapter. The page number is incremented automatically.

Two types of paragraphs exist. The first one is a sequence of words. The second one is an indivisible block, i.e. justification will be suppressed for that paragraph. A paragraph consisting of words, will be justified and reproduced as a block of text with straight left and right margins. The first line may cover the full linewidth. For the second and following lines a left margin will be kept.

The program inserts spaces to create a straight right margin. The most appropriate positions for this insertion are: immediately after a period, comma, colon, semicolon or question mark. Other appropriate positions could be found by parsing sentences. This is not within the scope of the program. However, when we observe english or dutch sentences, we discover that very often a short word and a following longer one are part of the same syntactical unit. The value that the user puts on these considerations, can be made known to the program in numerical form.

Keywords:
Justification, natural language, pagination.

Type: Main program in ALGOL 60

Calling sequence:

```
attach,infal,id=matcen.
attach,invoer,'pfn',id='id'.
library(infal)
textsch.
catalog,print,'pfn',id='id'.
catalog,pons,'pfn',id='id'.
xeor
<filespec>
xeof
```

'filespec' is a sequence of three pairs of integer values that specify the files "invoer", "print" and "pons" in that order. The first integer of each pair specifies the line width of the file. The second integer specifies the character code of the file. The value of the line width specification should be greater than or equal to the maximum linewidth of the specified file. The following table gives the range and meaning of the values that specify the character codes of the files:

type of file	read file	write file
ascii, even parity	2	18
ascii, odd parity	3	19
mc-flexowriter code	4	20
arba-flexowriter code	5	21
display code	6	22
dollar code	7	23
63 character set print file		24
95 character set print file		25

The 95 character set print file cannot be supported before the operating system SCOPE 3.4.2. is running on the CYBER installation. A dollar file is an "enriched" display file, in which a special meaning is attached to the dollar sign. For further information we refer to Chario (3.1.4)

Data and results:
1. Input file: lfn "invoer",
This file contains the text to be justified. We will describe the input in Backus-Naur-Form. Occasionaly the exact me ning of a backus-naur-formula will be slightly changed in the accompanying text, in order to keep the formal description simple.

```
<input>::= <chapter><chapter delim><input>,<chapter>
<chapter>::= <readvarblock><alinea>
<chapter delim>::= #;
```

The input consists of a sequence of chapters separated by a 'chapter delim' (a sharp sign followed by a semicolon: "#;"). A 'readvarblock' precedes each 'chapter'.

```
<readvarblock>::= (<assignments>)
<assignments>::= <variable name>:=<value>;<assignments>,<empty>
```

The 'readvarblock' contains information about the chapter that follows. The 'values' assigned to the 'variable names' are passed on to variables of the program. In this way each 'variable name' corresponds to a variable of the program. The program assigns initial values to these variables. Such a variable keeps its (initial) value until its corresponding 'variable name' is assigned to in a 'readvarblock'. The 'variable names' will be treated as ordinary ALGOL variables.

The 'variable names' are:

string kopje;

A string assigned to 'kopje' will be put as a heading at the top of each page. Self-evidently, the "string" of 'kopje' together with the chapter number and the page number should not occupy more positions than 'breedte' specifies. The string must be enclosed within quotes.

"boolean" pons;

If pons is true, a file "pons" (see beginning of this section) will be created. Otherwise only a file "print" is created.
Initial value: "true".

"integer" hoofdstuk;

Chapter number; minimum value 0, initial value 1.

"integer" bladzijde;

The number of the first page of the chapter;
minimum value 0, initi l value 1.

"integer" laatste;

The maximum value of the page number. If the page, having a page number equal to 'laatste', is followed by another page, the latter will get 'laatste' as its number followed by an "a", the next page will get 'laatste' followed by a "b" etc. This creates the possibility of inserting pages into a text that has already been numbered.
Minimum value 0, initial value 100000.

"integer" breedte;

The number of positions on a line.
Minimum value 18 (necessary to ensure sufficient space for a chapter number and a page number), initial value 75.

"integer" hoogte;

The numbers of lines per page. The first line is reserved for the heading, the next three lines are empty and the following lines may be filled with text. No page will have more than hoogte lines. Minimum value 5, initial value 56.

"integer" alinea;

A justified paragraph containing a number of lines that is less than or equal to 'alinea', will never be divided over two pages. The value of 'alinea' must be at least 4 less than the value of 'hoogte' and positive.
Initial value 10.

"integer" kantlijn;

Specifies the width of the left margin of the second and following lines of a justified paragraph. The first line will start at the same position as it did on the input file. if the left margin of the first line is smaller than the margin as specified by 'kantlijn' and the length of the first word is greater than or equal to the difference of the two margins, the first word will be put on a separate line (squeezed out). So justified paragraphs may take the following forms:

```
xxx.xx.xxxx.x
    xxx.xxxxx
 x.xxx.xxx
             or
xxxx
  xxx.xx.xxxxx
  xx..xx.xxx.x
             or
  xx..xx.xxx.x
xxx.xxx.x.xx.x
xx.xxxxx.xxxxx
```

where "x" represents letters and "." space.
impossible is:

```
xxxx.xxx.x
  xxx.xx.x
  xx.xxx.x
```

For example, instead of:

```
  10.1.3.5. The justification of nixon.
      Gods own country suffered a severe
      shock. We all know that, our dear
```

the program might produce:

```
  10.1.3.5.
      The justification of nixon.  Gods
      own  country   suffered  a severe
      shock. We all know that, our dear
```

It is possible to obtain the former layout by making the first line a separate paragraph and choosing a different indication; see <indication>.
Minimum value 0, initial value 8.

"integer" diepte;
The number of lines of a paragraph (counted after justification) that are used by the process of justification, in order to calculate the number of words on the lines (see Method and performance).
Minimum value 1, initial value 3.

"integer" zwak, middel, sterk, lzwak, lmiddel, lsterk;
These variables regulate the protection of certain word pairs against insertion of spaces or linebounderies. Zwak, middel, sterk determine the amount of protection enjoyed by weakly, mildly and strongly bonded words against the insertion of spaces. Lzwak, lmiddel and lstrong define the amount of protection against the insertion of linebounderies. A large value of lsterk, for instance, implies that two strongly bonded words will never be separated by a lineboundery.
Minimum value 1.
Initial values:

zwak 1	lzwak 1
middel 2	lmiddel 1
sterk 4	lsterk 1

"integer" rafels;
Indicates the importance the user attaches to an exactly straight right margin. A value much smaller than the value of zwak, middel etc. destroys completely the effect of justification; in that case each line will be filled with as many words as possible.
Minimum value 1, initial value 100.

<u>Examples</u>:
(hoofdstuk :=3; bladzijde := 1)
Pages will be numbered 3-1, 3-2, 3-3, etc.

(hoofdstuk:= 3; bladzijde:= 13; laatste:= 13)
Pages will be numbered: 3-13, 3-13a, 3-13b, etc

(hoogte:= 10000000; kantlijn := 0; diepte:= 1; rafels:= 1;
zwak:=1000; middel:=1000; sterk:= 1000; lzwak:= 1000;

```
 lmiddel:= 1000; lsterk:= 1000)
The text will be packed as closely as possible:
no pagination, no left margin, and no straight right margin.

  <alineas>::= <alinea><alineas>,<alinea>
  <alinea>::= <layout ind option><quoted block><alinea delim>,
              <layout ind option><words><alinea delim>
  <layout ind option>::= <layout><layout ind option>,
                         <indication><layout ind option>,<empty>
  <indication>::=<indication delim><spec><indication delim>
  <spec>::= h<spec>,k<integer>spec,<empty>
  <alinea delim>::= ‡"
  <indication delim>::= ‡=
  <quoted block>::= <quote delim><block><quote delim>
  <quote delim>::= ‡?
```

The 'indication' is enclosed within 'indication delim's and consists of the character "h" and/or the character "k" followed by an integer. The integer following "k" specifies the margin for the paragraph that follows (in other words, 'kantlijn' will be turned off temporarily). "h", if present, causes the boolean variable "hard" of textsch to take the value true; if not present,"hard" will have false as its value. the trueness of hard will ensure that the paragraph that follows, is put on the current page if and only if it fits completely on that page. If it is too long it will start at a new page.

Tabs, spaces, and line transitions constitute the layout. This layout will be copied and determines the starting point of the justified paragraph. If the paragraph starts at a new page preceding line transitions will be ignored. A 'quoted block' starts and ends with a 'quote delim'.

All symbols but 'quote delim's are allowed in a 'quoted block'. A 'quoted block' without its enclosing 'quote delim's will be copied and no justification will take place.

```
  <words>::= <word>,<word><separator><words>
```

Tabs, spaces and/or line transitions constitute a 'separator'. One space, tab or line transition suffices to specify a 'separator'; the exact composition of a 'separator' is of no importance.

```
  <word>::= <normal word>,<quoted word><mark option>
```

A 'normal word' may contain all available symbols with the exception of spaces, tabs, line bounderies and 'alinea delim's. A 'normal word' must not begin with a 'quote delim'. Its position in the paragraph will be determined by the process of justification. Like a 'quoted block', a 'quoted word' is enclose within 'quote delim's. However, tabs and line transitions are not allowed in a 'quoted word'. The program removes the enclosing 'quote delim's and the process of justification determines the position of the 'quoted word' in the paragraph (the uncertainty of its future place makes it impossible to allow tabs and linebounderies in a 'quoted word').

```
        <mark option>::= <empty>,<comma>,<period>,<colon>,<semicolon>,
                         <question mark>
```

The 'mark option' forms an indivisible unit with the 'quoted word'. For example:

```
                 ╫=n:= n + 1;╫=
        has the same meaning as:
                 ╫=n:= n + 1╫=;
```

2. Output files.
  2.1. lfn "print", containing the justified text plus error messages.
  2.2. lfn "pons", containing the justified text.

General remarks pertaining to input/output
The internal code is ascii. The read and write procedures return and accept ascii. So, capitalisation and underlining are possible. The program was written originally for the x8 and this imposes the following restrictions on the input. Barred and underlined symbols must always appear in the order:

bar/underline backspace symbol

The reverse order is not interpreted according to the intentions of the user. If the text does not contain capitals or backspace the input file can be in display code and specified accordingly. A richer code is necessary to represent capitals, backspace, underlining. This can be provided for by using the dollar convention. (see Chario) If the input file code is "richer" than the output file code as specified by 'filespec' the extra symbols of input are replaced by question marks on the output file.
The sharp symbol(╫) indicates a delimiter (chapter delimiter, alinea delimiter, etc.). Sharps that are part of the text to be justified, should be represented by pairs of sharps.
Errormessages are self-explaining and interspersed in the justified text on the file "print". The file "pons" contains the justified text without additional remarks.

Method and performance:
The program processes paragraphs. One can divide the processing of each paragraph into an input phase, a justifying phase and an output phase.

<u>The input phase</u>:

The words of a paragraph are stored in a doubly linked list, one integer per symbol. In the links of the list, space is reserved for justification data. The strength of the bond between two successive words is specified in the links, according to the criteria as explained in "brief descripton".

<u>The justifying phase</u>:

The thus created list of words is passed on to an algorithm, which determines the line division as follows. The first line is filled with as many words as possible. Then the degree of its justifiability is determined. The remaining part of the list is regarded as a new paragraph, the justifiability of which is determined recursively by the same process. These two justifiabilities are added up and this result constitutes the justifiability of this paragraph when having this particular line division. Then the last word of the first line is removed and added to the remainder of the paragraph. Again the justifiability is determined. Clearly the line has become less justifiable; the justifiability of the rest of the paragraph, however, may have increased sufficiently to have improved the overall picture. The distribution of spaces over the first line and its division do not become final, before the optimal number of words on the line has been determined in this way. Then this process is repeated iteratively for the remainder of the paragraph. The execution time of this algorithm is roughly proportional to 2↑r, where r is the number of lines in the justified paragraph. This implies that the justification of a paragraph of 15 lines takes 1000 times as much time as the justification of a paragraph of 5 lines. This is intolerable. Moreover, looking beyond more than a few lines does not heighten the esthetic effect. The depth of the recursion in the above algorithm is limited, therefore, by the variable diepte. The time required to process a paragraph is now:

```
   if r > diepte:          (r - diepte + 1) × 2 ×× diepte
   if r <= diepte:         2 ×× r
```

If 'diepte' = 6, the ratio of the executiontimes of two paragraphs of 15 and 5 lines is now: 20 : 1.

The above algorithm makes use of a numerical value that indicates the degree of justifiability. This value is defined as follows. The length of the words and the linewidth determine the number of spaces to be inserted,n, and the number of groups of spaces,ng. Then:

```
                        n = sum(i, 1, ng, s[i]),
```

where s[i] is the length of the i-th group of spaces. Justifiability is defined by:

```
                        -sum(i, 1, ng, k[i]  s[i] < 2)
```

where k[i] is the strength of the bond between two words. The degree of justifiability decreases quickly if one of the groups of spaces grows too big. If we do not limit the values of s to integers the justifiability has a maximum value if:

```
        s[j] = (n / sum(i, 1, ng, 1 / k[i])) / k[j];
```

this maximum value is:
 -n ×× 2 / sum(i, 1, ng, 1 / k[i].

This value is used as an indication of justifiability in order to determine the line division. Inserting the correct number of spaces however, necessitates a solution with integers. This is found by trial and error: starting with an initial guess of the distribution of spaces, the program shifts words and word groups until no no improvement is possible.

The output phase:

The precise length of the paragraph is known at this moment. If the current page cannot contain the entire paragraph, the value of "hard" is inspected; if it is found to be true, the paragraph starts at a new page; if "hard" is false, the length of the paragraph is compared with the value of "alinea" (see data and results) and then it is decided whether or not to start at a new page. If the paragraph starts a new page, the line transitions in layout of layout ind option are thrown away. The 'layout' is executed, followed by the text of the paragraph. New pages are inserted where necessary.

Subprograms used:
rdchar, wrchar, open, close (Chario)
available, exit (Mcc)

Required Central Memory: 70000b.

Example of Use:
In this example a dollar file is read in and an arba-flexowriter file is put out. The file specifications are:
80 7 80 22 80 21

The input file contains:

```
(kopie:= "brief van meester $fok"; hoofdstuk:= 123; bladzijde:= 13;
laatste:= 13; breedte:= 50; hoogte:= 40; alinea:= 20; kantlijn:= 2)
                    $<waarde vriend$>
$ik heb over veertien dagen geschreven met een $hollandsch schip, doch
alzo het zelve nooh eerst een reis moet gaan doen naar $angola, zo
vertrouw ik wel, dat die brief eenige maanden na deze zal arriveren. $
ik
leer hier van alle slag van ambachten; alzo ik buiten $fiscaal, voor
$secretaris, voor $raad, voor $notaris, voor $ambassadeur, voor $kaper
en voor den eenen drommel met den anderen moet spelen: zo dat gy wel
kund denken, dat ik niet veel tyd heb om spelen te loopen; daar ook ni
et
veel occasie toe is in dit barbaarsche, melancholique, en verbaasde
dorre land, 't welk ik niet gezind ben heel net af te schilderen, uit
vrees, dat gy schreien zoud als een kind, en de arme $fok beklagen, om
dat hem het noodlot in zo verdoemde plek gebracht heeft. $want beeld u
zelfs eens in te zien een zwaarmoedig kasteel, gesitueert op een schra
```

```
le
en dorre rots, daar de zee, met een eeuwig geruisch, op leid te gnorre
n;
figureert u vorders aan de rechterhand van 't voorschreve kasteel te
zien een langwerpig dorp, bestaande in hutten, gedekt met zwart verbra
nd
hooi, en strooi, of riet(want de duivel zelfs zou niet kunnen raden we
lk
van drien het is) waar in het zwermt van half naakte, en koolverwige
schimmen, die u den ganschen dag de ooren warm maken met een eeuwig
getoet van loejende hoorens, daar zy haar $artem $musicam met het
abominabelste geschal des weerelds op exerceren, 't geen u wel een
baal kattoen in 't jaar zou kosten, om uw geluitvangers daar mede toe
te stoppen, $aan de slinker zyde van 't kasteel zwalpt een droevig
riviertje, 't geen al 't zout van de zee in zyn boezem schynt
ingezogen te hebben, alzo 't zelve tienmaal zouter is dan het
alderziltste pekelnat.#"$;
   $bedenk nu vorders by u zelven omtrent twee mylen in 't rond
te zien een barre en schrale woestyn, waar op noch telg,
noch lover te vinden is, die u voor een straal van de zon kan
beschutten, die hier zo schrikkelyk steil boven onze kruin, in 't
$zenith staat, dat men op 't midden ven den dag, zelfs ontrent de
hoogste tooren des werelds, geen duimbreet schaduw zou kunnen vinden,
$denk nu vorders, of ik geen reden heb van somtyds in drie weken niet
buiten het kasteel te komen, en in myn sel te blyven; alwaar gy my
zoud zien zitten, in 't compagnie van myn twee zwarte jongens, al
dampende dat het zyn oogen verdraaid, en dat zy met hun beide
eeuwig werk hebben met toebak te kerven, en te stoppen; dit gaat zo
zyn gang al schryvende, of iets vermakelyks lezende, of met een
eerlyke ziel of twee by my, onder de beneficie van een glaasje, om de
geest te verfraaijen, en de melancholie te diverteren. $wat aangaat
myn muzyk, die is, door het afsterven van myn kouzyn van $heden, die
met my overgekomen, en hier zedert eenige weken overleden is,
zodanig verstorven, dat gy myn violon met droefheid aan de wand zoud
zien hangen, zodanig gediscordeert, dat gy daar niet dan een enkele
bas op zoud vinden; terwyl in de holte van dat droevig instrument
de spinnekoppen zodanig haar logement hebben verkozen, dat ik geloof,
dat zy van sins zyn van hun eigen weefzel nieuwe snaren daar op te
maken. $in 't end, ik vind, dat ik met recht mag zingen, pas als de
kinderen $israels in een der $psalmen doen: $-$super flumina $babylone
,
illic sedimus, ∧ flevimus, ∧ suspendimus $organa nostra.$-#"$;
#?                     $dat is:$;
$;
            $aan de $babylonsche stroomen$;
               $hingen wy met naar gesteen,$;
               $en met jammerlyk geween,$;
            $al ons speeltuig aan de boomen.$;
$;
#?#"$;
   $doch echter patientie, is 't land slecht, het goud is goet, en
```

dat is het alleen, 't geen my veel ongenuchten, die my hier
voorkomen, doet dirigeren; want daar is geen cardiacum in de weereld,
dat zo krachtig is, als dat; dieshalven is het, dat ik geresolveert
ben in alles geduld te nemen, en ondertuschen, terwyl ik hier ben,
myn naad te naaijen zo veel ik kan, en de plaizieren van de weereld
voor een jaar of zes te vergeten, als of ik dood was. $want hier is
geen vermaak ter weereld, als alleen dat in uw eigen gemoed, en by
u zelfs bestaat; want de wyn in overdaad, en de zwarte vrouwen
haat ik dapper: en ik geloof niet, dat ik tot een van beiden
heel licht zal vervallen, alzo ik het egaal voor beestachtigheid,
en een doodelyke coyonnerie hou. $alleen heb ik myn meeste
vermaak in een kleine zwarte jongen, die ik heb, die van zeer grooten
huize, en van zeer treffelyke luiden is; want ik,verklaar u, dat ik
nooit schoonder, noch heroiquer wezen gezien heb, vermengd met een
groots, doch eenigzins stuurs opslag van oogen, 't geen my vaak op
hem doen appliceren de woorden van $seneca in $hippolytus:#"$;
#?$;

    $quam $grata est $facies torva $viriliter,$;
    $et $pondus $veteris triste $supercilii.$;

$;

                 $dat is:$;

$;

    $hoe heerlyk, en voortreffelyk staat$;
    $een fier en mannelyk gelaat,$;
    't $geen, door den opslag zyner blikken,$;
    $een ieder vol ontzag doet schrikken.$;

$;
#?#"$;
   $want inderdaad, dat wezen is in die jongen zo heerlyk te zien,
dat ik my dikwils inbeeld in hem te zien een schets van dien ouden
$afrikaanschen $hannibal; ook zyn al zyn inclinatien groots, en
moedig, ja zo, dat hy met jongens van zyn jaren(die ontrent 12. zyn)
niet zal omgaan, maar altyd met zyn ouder, waar boven hy noch altyd
wil de preferentie hebben, 't zy in den dans, of andere speelen,
daar hy altyd de eerste wil zyn; of zo iemand hem die rang
bedisputeert, zo ontziet hy zelfs geen volwassen jongens voor de
kop te slaan. $en by al deze barsheid is hy weer by my zo vriendelyk,
beleeft, en trouw, dat ik die jongen lief heb in myn hart, en zou(zo h
y
een slaaf was) niet weigeren een pond goud voor hem te geven, enz,#"$;
#?$;
   $op 't kasteel $st. $george da $mina,$;
      $den 10 $february, 1669.$;
$;

                                          $focquenbroch.#?#"$;#;

   The arba-flexowriter file is displayed on the next pages without
the usual infal headings.

WAARDE VRIEND Ik heb over veertien dagen geschreven met een Hollandsch schip, doch alzo het zelve noch eerst een reis moet gaan doen naar Angola, zo vertrouw ik wel, dat die brief eenige maanden na deze zal arriveren. Ik leer hier van alle slag van ambachten; alzo ik buiten Fiscaal, voorSecretaris, voor Raad, voor Notaris, voor Ambassadeur, voor Kaper en voor den eenen drommel met den anderen moet spelen: zo dat gy welkund denken, dat ik niet veel tyd heb om spelen te loopen; daar ook niet veel occasie toe is in dit barbaarsche, melancholique, en verbaasde dorre land, 't welk ik niet gezind ben heel net af te schilderen, uit vrees, dat gy schreien zoud als een kind, en de arme Fok beklagen, om dat hem het noodlot in zo verdoemde plek gebracht heeft. Want beeld u zelfs eens in te zien een zwaarmoedig kasteel, gesitueert op een schrale en dorre rots, daar de zee, met een eeuwig geruisch, op leid te gnorren; figureert u vorders aan de rechterhand van 't voorschreve kasteel tezien een langwerpig dorp, bestaande in hutten, gedekt met zwart verbrand hooi, en strooi, of riet(want de duivel zelfs zou niet kunnen raden welk van drien het is) waar in het zwermt van half naakte, en koolverwigeschimmen, die u den ganschen dag de oore warm maken met een eeuwig getoet van loeje de hoorens, daar zy haar Artem Musicam met het abominabelste geschal des weerelds op exerceren, 't geen u wel een baal kattoen in 't jaar zou kosten, om uw geluitvangers daar mede toe te stoppen, Aan de slinker zyde van 't kasteel zwalpt een droevig riviertje, 't geen al 't zout van de zee in zyn boezem schynt ingezogen te hebben, alzo 't zelve tienmaal zouter is dan het alderziltste

pekelnat.
Bedenk nu vorders by u zelven omtrent twee mylen in 't rond te zien een barre en schrale woestyn waar op noch telg, noch lover te vinden is, die u voor een straal van de zon kan beschutten, die hier zo schrikkelyk steil boven onze kruin, in 't Zenith s aat, dat men op 't midden ven den dag, zelfs ontrent de hoogste tooren des werelds, geen duimbreet schaduw zou kunnen vinden, Denk nu vorders, of ik geen reden heb van zomtyds in drie weken niet buiten het kasteel te komen, en in myn sel te blyven; alwaar gy my zoud zien zitten, in 't compagnie van myn twee zwarte jongens, al dampende dat het zyn oogen verdraaid, en dat zy met hun beide eeuwig werk hebben met toebak te kerven, en te stoppen; dit gaat zo zyn gang al schryvende, of iets vermakelyks lezende, of met een eerlyke ziel of twee by my, onder de beneficie van een glaasje, om de geest te verfraaijen, en de melancholie te diverteren. Wat aangaat myn muzyk, die is, door het afsterve van myn kouzyn van Heden, diemet my overgekomen, en hier zedert eenige weken overleden is, zodanig verstorven, dat gy myn violon met droefheid aan de wand zoudzien hangen, zodanig gediscordeert, dat gy daar niet dan een enkele bas op zoud vinden; terwyl in de holte van dat droevig instrument de spinnekoppen zodanig haar logement hebben verkozen, dat ik geloof, dat zy van sins zyn van hun eigen weefzel nieuwe snaren daar op te maken. In 't end, ik vind, dat ik met recht mag zingen, pas als de kinderen Israels in een der Psalmen doen:
<u>Super flumina Babylone, illic sedimus, ^ flevimus, ^ suspendimus Organa nostra.</u>

Dat is:

Aan de Babylonsche stroomen
Hingen wy met naar gesteen,
En met jammerlyk geween,
Al ons speeltuig aan de boomen.

Doch echter patientie, is 't land slecht, het goud is goet, en dat is het alleen, 't geen my veel ongenuchten, die my hier voorkomen, doet dirigeren; want daar is geen cardiacum in de weereld, dat zo krachtig is, als dat; dieshalven is het, dat ik geresolveert ben in alles geduld te nemen, en ondertuschen, terwyl ik hier ben, myn naad te naaijen zo veel ik kan, en de plaizieren van de weereld voor een jaar of zes te vergeten, als of ik dood was. Want hier is geen vermaak ter weereld, als alleen dat in uw eigen gemoed, en by u zelfs bestaat; want de wyn in overdaad, en de zwarte vrouwen haat ik dapper: en ik geloof niet, dat ik tot een van beiden heel licht zal vervallen, alzo ik het egaal voor beestachtigheid, en een doodelyke coyonnerie hou. Alleen heb ik myn meestevermaak in een kleine zwarte jongen, die ik heb, die van zeer grooten huize, en van zeer treffelyke luiden is; want ik,verklaar u, dat ik nooit schoonder, noch heroiquer wezen gezien heb, vermengd met een groots, doch eenigzins stuurs opslag van oogen, 't geen my vaak op hem doen appliceren de woorden van Seneca in Hippolytus:

Quam Grata est Facies torva Viriliter,
Et Pondus Veteris triste Supercilii.

Dat is:

Hoe heerlyk, en voortreffelyk staat
Een fier en mannelyk gelaat,
't Geen, door den opslag zyner blikken,
Een ieder vol ontzag doet schrikken.

Want inderdaad, dat wezen is in die jongen zo heerlyk te zien, dat ik my dikwils inbeeld in hem te zien een schets van dien ouden Afrikaanschen Hannibal; ook zyn al zyn inclinatien groots, en moedig, ja zo, dat hy met jongens van zyn jaren(die ontrent 12. zyn)niet zal omgaan, maar altyd met zyn ouder, waar boven hy noch altyd wil de preferentie hebben, 't zy in den dans, of andere speelen, daar hy altyd de eerste wil zyn; of zo iemand hem die rangbedisputeert, zo ontziet hy zelfs geen volwassen jongens voor de kop te slaan. En by al deze barsheid is hy weer by my zo vriendelyk, beleeft, en trouw, dat ik die jongen lief heb in myn hart, en zou(zo hy een slaaf was) niet weigeren een pond goud voor hem te geven, enz,

Op 't kasteel St. George da Mina,
Den 10 February, 1669.

Focquenbroch.

Author: G.H.A. Kok

Source and documentaion: omnisrc, omnidoc

Institute: Mathematical Centre

Date received: 28/02/75

Brief description:
Omnisort is a procedure written in ALGOL 60.
It is designed to do all kinds of sorting or merging lists of units, each consisting of two lists of words, containing respectively sort and nonsort information - in real or integer format -.
The units are either offered to the procedure in two linear arrays, containing the sort and nonsort information, by procedure calls from the calling program or read in from file(s) created by former calls to omnisort.
It returns to the calling program - and if the caller specifies so, it writes to a file in a special internal form as well - a list of units, ordered ascendingly after the numeric value of the elements of the first array, the significance of which is from 1 upward. In case of character-type information - integers in the range [0 : 255] - there is a facility to have omnisort pack the data very efficiently with respect to memory usage.

Keywords: ALGOL 60, sorting, merging.

Type: ALGOL code procedure, in ALGOL 60.

Calling sequence:
The heading of the declaration of omnisort reads:

```
 "code" 14001;
 "integer" "procedure" omnisort(throughput,ls,ln,lt,store module,
     fetch module,i,sym i);
     "value" ls,ln,lt; "string" throughput;
     "integer" ls,ln,lt,i,sym i;
     "procedure" store module, fetch module;
```

To understand the meaning of the parameters to be supplied to omnisort some knowledge is required of the way omnisort functions. The next steps may be distinguished:

i. Creation of space for the arrays sort, nonsort and work, with lower bound equal to zero and upper bound to resp. ls, ln, and abs(lt).
ii. Analysis of the string throughput.
In this string from zero up to three filenames may be specified, each obeying the rules for SCOPE logical file names and each corresponding to a SCOPE logical file.
The type of the file (input file with sorted data or output file,

to receive sorted data) depends upon the separator before each name: a comma or no separator denotes an input file, a slash one for output.
At most 2 inputfiles and 1 outputfile can be specified for one call to omnisort, resulting from the next 6 possible string forms - fn denotes a filename, the pai (n,m) the number of in- and outputfiles -:
f1,f2/f3 => (2,1); f1/f3 => (1,1); f1,f2 => (2,0);
f1 => (1,0); /f3 => (0,1); empty => (0,0).

iii. If input file(s) are specified reading of their contents.
Call to store module, which should provide the elements to be sorted.
Omnisort supplies to store module resp. the arrays sort and nonsort and the no-type procedures store and comp. (comp is to be explained in section data and results a). within store module for each unit to be sorted the arrays sort and nonsort should be filled - from 1 upward -, after which store should be called with two integer parameters containing resp. the number of elements filled of sort and nonsort. store will copy those elements to work.

iv. Completion of the sorting process.

v. If an output file is specified writing the ordered units to this file.
Call to fetch module, which hands the sorted units one by one to the calling program in much the same way wav as store module puts units into omnisort.
Fetch module receives the arrays sort and nonsort from omnisort as the first two parameters and moreover a boolean procedure fetch and a no-type procedure decomp.
The latter is to be explained in section data and results a. Each call to fetch, with parameters by name ls and ln, fills sort and nonsort with the data of the next unit and returns "true" as long as there are units left and "false" otherwise.
The number of elements used of sort and nonsort is returned resp. in ls and ln.

vi. A value is returned to the calling program.
It specifies the number of units sorted if it is nonnegative or else signals an error-situation after the list given in section data and results d.
The parameters i and sym i are explained in section data and results a.

<u>Data and results</u>:

a. Compression/decompression/recoding:
Omnisort handles all data to be represented by real or integer values. A special case arises when the data consist of integer values in the interval [0 : 255], e.g. representing a character

set. In this case a call to comp (ref. to calling sequence) with parameters s, a linear array, and l, an integer variable, compresses the contents of the elements of s (1 : l) and stores the compressed data back into s; the number of elements used is returned in the variable l. Along with the compressing a recoding will be done. The recoding must be specified by the parameters i and sym i by means of jensens' device: let n be the number of integers to be recoded; sym i should specify, for i from 0 to n-1, the value, which should be sorted with ordinal i+1. For i equal to n sym i should take a negative value, to denote the end of the recoding list. The list specified in this way may be empty, but it shouldn't contain exactly one element.
During compression the order relation is preserved:
recoded compressed and recoded uncompressed data will be sorted in the same way.
Decompression can be done within fetch module by a call to decomp with parameters a linear array s and an integer variable l. l should contain before the call the number of elements of s containing data; after execution of decomp each element of s contains one decoded integer up to a number which is returned in l.

b. Input sorted, lt - parameter.
If the first units offered by store module are already in order omnisort can operate more efficiently. Information about this situation can be conveyed to it by entering lt negatively. In that case omnisort checks the order of the units; after detecting the first unit out-of-order omnisort goes on in sorting mode.

c. Value of sort[0] and nonsort[0], equal keys.
After a call to fetch, sort[0] contains the number of units with sortkey given by sort[1 : lsort].
If these unit(s) do not have nonsort information, the next call to fetch will deliver a unit, if any, with the next sort information. In case there are units with nonsort information - and maybe without it as well -, each call to fetch will fill the array nonsort from 1 to lnsort with the pertinent information and assign to nsort[0] the ordinal of that unit, beginning at 1 after the ordening:

1. units read from file1 (from throughput), if any;
2. units read from file2, if any;
3. units entered by store module in 'first-in-first-out' order.

The number of units without nonsort information can be found by counting the number of units, that do have nonsort information returned by fetch in successsive calls until a change of sort information, and subtracting that number from the total number of units containing the current sort information, as given by sort[0].

d. Value returned by omnisort, errors.

The value returned by omnisort specifies an errormessage if it is negative, else the number of units handled - in case there are units without nonsort information all are taken account of -.
If an error is detected omnisort sometimes writes a message to channel 61. Often omnisort won't stop immediately, but it will go on until the present phase is finished or the number of errors detected till then exceeds some limit, which depends upon the number of units handled till then.
4 categories of errormessages may be distinguished, depending upon the phase of the process:

i. Initialisation phase:

1. ls < 0
2. ln < 0
3. abs(lt) < max (ls + ln + 10, 1024)
10. sym i > 255
11. sym i produces some value more than once
12. length of recodinglist equals exactly one
20. filename incorrect
21. filename takes more than 7 characters
23. inputfile incorrect
24. inputfile not created by omnisort
25. inputfile created with ls and/or ln exceeding the present value
26. inputfile created with another recoding
29. omnisort stops because of errors in intialisation phase

ii. Input- and sorting phase:

(lsort and lnsort denote the number of elements offered to omnisort by a call to store, ls and ln identify resp. the second and third parameter to omnisort)
30. lsort < 0
31. lsort > ls
32. lnsort < 0
33. lnsort > ln
41. comp is called without a recoding having been specified
42. integer to be compressed out-of-range
43. integer to be recoded not specified in recodinglist
49. omnisort stops because of too many errors in sorting phase

iii. Output phase:

51. decomp is called without a recoding having been specified
52. the array offered to decomp does not contain data compressed by comp
53. the size of the array offered to decomp is too small to contain the decompressed data
59. omnisort stops because of too many errors in the output phase
99. during execution of fetch module errors are detected
(the number of units skipped is written to channel 61)

iv. Errors by omnisort:

101, 102, 103. units are lost
141, 151, 152. errors in the filesystem

If omnisort returns -59, units are offered to fetch module; in case of returning -99 moreover units are written to the output file, if any. In both cases the results should not be relied upon.

Subprograms used:

Word-adressable file procedures by D. Winter; these procedures will become part of the infal-library.

Required central memory - octal -:

			source
omnisort:		15000	infal
filesystem:		7450	infal
record manager routines, used by filesystem:		4404	sysio
blank common, used by record manager:		2200	
algorun,alglib00,alglib01, alglib02,alglib06:	(15756)	p.m.	sysmisc
dynamic storage:	$\pm$ 1500 + ls + ln + abs(lt)		

Execution time:
The order of the process is n × log(n).
Compared to the CD Sort/Merge system it is slow.

Method and performance:
The sorting process consists of repeatedly building a binary tree as large as the size of the working store 'work' allows and dumping the ordered subset to a backing store, followed by merging the subsets.

References:
G.H.A. Kok, Alfabetiseren en andere sorteerwerkzaamheden, MC Publication MR 120 , october 1970.

Example of use:
In this example the text of the sample program is read in and decomposed into words.
A word is defined as a sequence of letters, separated by a start-of-line or a non-letter. The whole sequence of letters should lay before position 73.
The words are offered to omnisort by the procedure in, sorted and returned to the procedure out, which prints the words in alphabetical order together with the number of occurrences of each word in front of it.
A recoding is applied merely to signal to omnisort, that only 25 characters are to be cared about (this invokes an errormessage, because the letter z occurs also)

```
"BEGIN"
    "COMMENT" SAMPLE PROGRAM DEMONSTRATING THE USE OF OMNISORT;

    "INTEGER" "PROCEDURE" OMNISORT
    (THROUGHPUT, LS, LN, LT, STORE MODULE, FETCH MODULE, I, SYM I);
    "CODE" 14001;

    "INTEGER" I, RESULT; "REAL" TO;

    "PROCEDURE" IN (SO, NSO, STORE, COMP); "ARRAY" SO, NSO;
    "PROCEDURE" STORE, COMP;
    "BEGIN" "INTEGER" "ARRAY" BUFFER [1 : 80]; "INTEGER" I, J, T42, LS;
        EOF (1, EINDE INVOER); T42:= 2 xx 42;
    AGAIN1: INPUT (1, "("80(A)")", BUFFER); J:= 0;
    AGAIN2:
        "FOR" I:= J,
              I + 1
              "WHILE"
              I < 73 "AND"
              (BUFFER [I] > EQUIV ("("Z")") "OR" BUFFER [I] = 0)
        "DO" J:= I;
        "COMMENT" J POINTS TO LAST NON-LETTER;
        "IF" J = 72 "THEN" "GOTO" AGAIN1; LS:= 0;
        "FOR" I:= J + 1,
              I + 1
              "WHILE"
              I < 73 "AND" BUFFER [I] <= E UIV ("("Z")") "AND"
              BUFFER [I] > 0
        "DO"
        "BEGIN" LS:= LS + 1; SO [LS]:= BUFFER [I] / T42; J:= I "END";
        "COMMENT" J POINTS TO LAST LET ER;
        COMP (SO, LS); STORE (LS, 0);
        "GOTO" "IF" J = 72 "THEN" AGAIN1 "ELSE" AGAIN2;
    EINDE INVOER:
    "END" IN;

    "PROCEDURE" OUT (SO, NSO, FETCH, DECOMP); "ARRAY" SO, NSO;
    "PROCEDURE" DECOMP; "BOOLEAN" "PROCEDURE" FETCH;
    "BEGIN" "INTEGER" LS, LNS, J, T42;
        T42:= 2 xx 42;
        "FOR" J:= 0 "WHILE" FETCH (LS, LNS) "DO"
        "BEGIN" DECOMP (SO, LS);
            OUTPUT
            (61, "("/9ZDB,50(A)")", SO [0], (SO [I] × T42, I:= 1 : LS))
        "END"
    "END";
    TO:= CLOCK;
    RESULT:=
    OMNISORT
    ("(""")", 100, 0, 3072, IN, OUT, I,
     "IF" I < 25 "THEN" I + 1 "ELSE" - 1);
    OUTPUT
    (61,
     "("×"("WAARDE VAN OMNISORT NA AANROEP IS:")",+5ZD///,             "(
"BENODIGDE TIJDSDUUR:")", 3ZD.3D,"(" SEC")"/")",
     RESULT, CLOCK - TO)
"END"
```

Results:
 - only part of the output of the sample program is shown, the remaining part is indicated by periods -:

```
CHANNEL,60=INPUT,P80,R
CHANNEL,61=OUTPUT,P136,PP60,R
CHANNEL,1=TE TOMN,P80,R
 26 NOT IN ALPHABET
ERROR  43
 26 NOT IN ALPHABET
ERROR  43
 26 NOT IN ALPHABET
ERROR 43
 26 NOT IN ALPHABET
ERROR  43
 26 NOT IN ALPHABET
ERROR  43
         2 A
         1 AANROEP
         5 AGAIN
         3 ARRAY
         5 BEGIN
         1 BENODIGDE
         1 BOOLEAN
         ....
         ....
         1 THROUGHPUT
         1 TIJDSDUUR
         3 TO
         1 USE
         1 VAN
         1 WAARDE
         3 WHILE
 5 UNITS SKIPPED
ERROR  99
WAARDE VAN OMNISORT NA AANROEP IS:    -99
BENODIGDE TIJDSDUUR:   8.829 SEC
     END OF ALGOL RUN  ×V3.1×
```

Author: Han Noot

Source and documentation: kwicins, kwicind

Contributor: Dik Winter

Institute: Mathematical Centre

Date received: 24/06/75

Brief description:
The program produces a keyword in context index (kwic-index) and an index, from input consisting of text-units and optionally together with the output of previous runs of the program. Each text-unit consists of a codefield, followed by a text body.
In the text body, subsections preceded by a marker symbol, as well as the subsection that starts with the first character of the text-body, are used as sorting-keys for the kwic-index.
The index produced, consists of the complete text-units, sorted alphabeticaly, where the code contained in a codefield acts as the sorting-key.
The kwic-index consists of output, sorted according to the keys marked in the text-input. This output is made up from one line for each key. The key is printed in its textual context. Each output-line terminates with the codefield of the text-unit from which it is part.
Apart from processing text-units, the output of previous runs (stored in an intermediate form) can be used as input too. In this case, the old output (at most two files) is merged with the new output, or alternatively, two old output-files are just merged.
The input and output-files can be coded in varicus character sets, while the layout of the input and output is , to a certain extend, variable too.
The significant characters that make up keys are in sorted order:

```
                             a,b,c,d,..............z
"blank",0,1,2,3,4,5,6,7,8,9,v v v v                v
                             A,B,C,D,..............Z
```

Other characters are just ignored, when keys of a specified number of characters are read in from the input.

Keywords: Keyword, sort, merge, index, kwic-index.

Type: Main program in ALGOL 60.

Calling sequence:

```
attach,bin1k, .....
attach,bin2k, .....
attach,bin1i, .....
attach,bin2i, .....
```

```
 attach,text, ......
 attach,control, ...
 attach,infal,id=matcen.
 library,infal.
 kwicind.
 print and/or catalog charkw
 print and/or catalog charid
 catalog,binkw, .....
 catalog,binid, .....
```

In this calling-sequence, the first 5 attach-commands are optional. Whether they are necessary or not, depends on the input used and the sort/merge operation to be performed. The same holds for the last two catalog-commands. For details, the next section should be consulted.

Data and results:

Input-files:
As input, files with the following 'logical file names' can be used by the program:
'control', 'text', 'bin1k', 'bin2k', 'bin1i', 'bin2i':

The file 'control' contains parameters that determine the sort/merge operation to be performed and the input/output format, while file 'text' contains text-units in a format consistent with the one, specified on file 'control' (see below). The files 'binxk', respectively 'binxi', contain a compressed form of a kwic-index respectively index produced in a previous run of kwicind. The contents of these files can be merged with each other and/or with the output. Only certain combinations of these input-files are meaningful, depending on the sort/merge operation that must be performed. They are:

1) 'control', 'text'.
2) 'control', 'text', 'bin1k', 'bin2k', 'bin1i', 'bin2i'.
3) 'control', 'text', 'bin1k', 'bin1i'.
4) 'control', 'bin1k', 'bin2k', 'bin1i', 'bin2i'.

For a discussion of the meaning of these input-file combinations, see the subsection "control" below.

Output-files:
There are four output-files generated, namely 'charkw', 'charid', 'binkw' and 'binid'. Files 'charkw' and 'charid' are character-files, that contain the kwic-index respectively index produced (they should explicitly be printed out or cataloged, at least if the user is interested in his output). Files 'binkw' and 'binid' are binary files, that contain the kwic-index respectively index in compressed form. They can be used in subsequent merging operations using kwicind, in which case they must then be attached with logical file-names 'bin1k' or 'bin2k', respectively 'bin1i' or 'bin2i' (see above). Furthermore, the standard

output-file is used for error-messages and for a printout of the specifications, contained in file control.

Text:
The file 'text' consists of a number of text-units, eventually separated by blanks, "new line" characters and "carriage return" characters (or there analogon in non-ascii code).
Text-units are sequences of characters belonging to the character set specified for the input (see control, int1). A text-unit starts with a codefield (of a length specified by int6 on control) and is followed by a text-body. In this text-body, the first character of keys (for the kwic-index), is indicated by preceding it with a marker symbol (specified on file control, the length of the keys is specified by int5 on this file). The first character of the text-body is always the beginning of a key, it must not be preceded by the marker symbol, otherwise the first text-body characters are treated as a key twice.
If the marker-symbol is separated from the first character in the key, which is not a blank , "carriage return" or "new line" by one or more of these characters, they are skipped. The same applies to "carriage return" and "new line" characters anywhere in the text-body; they are ignored. Strings of more then one blank are reduced to one blank, however.
Finally, the end of a text-body is indicated by two marker symbols in succession.

Note: file 'text' should be presented to kwicind without sequence numbers from the editor, because these numbers will be treated as part of the text-units.

Control:

The contents of this file are:

int1,int2,int3,char,int4,int5,int6,int7,<a>, where
<a>::=svm1svm2svm2
Int1 is the codenumber which determines the code (display, ascii etc.) in which file text is coded. Int2 is the codenumber which defines the character-set in which the output is produced. Int1 and int2 can have the following values and meanings:

int1	int2	code
0	16	8-bit binary
1	17	12-bit binary
2	18	ascii even parity
3	19	ascii odd parity
4	20	mc-flexowriter code
5	21	arba-flexowriter code
6	22	display code
7	23	dollar code
	24	63 char set printfile
	25	95 char set printfile

For more information on this subject, see the documentation on Chario, contained in this library.

Int3 must be set to the maximum number of errors that may occur in the input, before the program stops. Input errors are occurences of characters in the input, that are not part of the character set specified for it.

Char is the character, that is used to mark keys on file text. It must belong to the character set defined by int1. It is considered an error, if the marker character is a digit, a letter or a 'blank .

Int4 must be set equal to the maximal number of characters that may occur in an input text-unit.

Int5 defines the number of characters in the keys used for the kwic-index. Hence, keys are completely defined by int5, the marker symbol and the set of significant characters for keys (see brief description).

Int6 is set equal to the number of characters in the codefield of the text unit.

Int7 defines the position of the first character of the key in a kwic-index output-line. This key is printed, surrounded by its context in the text-body, where it belongs to. The last parameter on control, specifies the input-files present and the sort/merge operations that must take place, in the following way:

s : sort only: produce a kwic-index and an index from the text-units on file 'text'.

m1s : sort/merge: produce a kwic-index and an index from the input on 'text' and merge the result with the result of previous runs, stored in files 'bin1k' and 'bin1i'.

m2s : sort/merge: like m1s, with the exeption that the results of two previous runs are used, which are stored on files 'bin1k', 'bin2k', 'bin1i', 'bin2i'.

m2 : merge: produces an index and a kwic-index by merging the results of two previous runs, stored on files 'bin1k', 'bin2k', 'bin1i', 'bin2i'.

Remark: all parameters on file 'control' are checked first; as well by them selves as for mutual consistency. For instance: a value of int7 larger then the maximum number of characters on a printed output-line (135) is considered an error. If an error occurs, a message is printed and program-execution stops.

Relation between logical file-names and the program:
File 'control' is used as a scope-file (channel, 50=control, p80, r) and as a file processed by chario (filenr. 4).
File 'text' is solely read from or written upon by procedures from chario. It is used with file number 1. Files 'charkw' and 'charid' are solely processed by chario too. Their filenumbers are 2 and 3 respectively.
The files 'bin1k', 'bin2k', 'bin1i', 'bin2i' are input-files for the sorting procedure omnisort, files 'binkw' and 'binid' are output-files for this procedure. Their names are passed as parameters to omnisort, where they are opened. For details the reader should consult the documentation on omnisort, contained in this library.

Subprograms used:

Omnisort
Chario
Word-addressable file procedures by D. Winter; these procedures will become part of infal.

Required central memory: 125000b

Running time:
Because of the long running time, needed for the production of substantial output, no details are yet available on this subject.
We kindly ask users to communicate their experience in this respect to the infal redaction or to the author so that their information can be incorporated in this documentation.

Method:
This program is strongly dependent on three other elements of infal. The central element in the sorting process is procedure omnisort which takes care of all the sorting- and merging operations to be performed. It is called twice during the execution of kwicind, once for the production of the index and once for the kwic-index. In the first case, procedures storeindex and fetchindex are passed as parameters to omnisort , in the other procedures storekwic and fetchkwic. These procedures pass, respectively accept input-, respectively output units to, respectively from omnisort, one by one. The contents of file 'text' are read once during the execution of kwicind, namely by procedure storekwic. This procedure reads in one text-unit, each time it is called by omnisort. Next, it tabulates keypositions and strips the text-unit from marker-symbols, and superfluous "blanks", "new line" and "carriage return" characters. It is at this point that parameter int4 from file 'control' is of consequence, because text-units are kept temporaray in an array for this and other processing. The stripped units are written to a scratch-file for later use by storeindex. Next, for every tabulated keyposition, one output-line (key in context) and one key are generated and stored in arrays, which are passed to omnisort. Leading and trailing blanks in output-lines are not stored, however. The first two elements of the array containing the output-

line, are used to store the numbers of these blanks instead.
Procedures storeindex, fetchkwic and fetchindex are straightforeward and will not be discussed here.
The flexibility of input- and output code is optained, by using the procedures from chario. They are used in the fetch- and store procedures just mentioned, as well as for the processing of file control, in order to be able to read in a marker-symbol, not beeing part of display code.

Example of use:

Input-files:

File 'control':

```
6,22,0,$,72,4,4,10,s,
```

File 'text':

```
030 this is a small $example.$$
345 hopefull it $illustrates the use of $kwicind.$$
009 if not,the $author will try to $help.$$
541 good $luck.$$
```

Files 'bin1i', 'bin2i', 'bin1k', 'bin2k': not present.

Output-files:

File 'charid':

```
009 if not,the author will try to help.
030 this is a small example.
345 hopefully it illustrates the use of kwicind.
541 good luck.
```

file 'charkw'

Because of the length (135) of the lines on file 'charkw' we shorten these. This is indicated by "><".

```
ot,the   author will try to help.                  ><   009
 small   example.                                  ><   030
         good luck.                                ><   541
try to   help.                                     ><   009
         hopefully it illustrates the use of kwicind. ><   345
         if not,the author will try to help.       ><   009
lly it   illustrates the use of kwicind.           ><   345
use of   kwicind.                                  ><   345
  good   luck.                                     ><   541
         this is a small example.                  ><   030
```

Part of the standard output-file:

```
index and kwicindex

inputcode:  display

outputcode:  display

maxerrnumber:  0

ascii-representation marker:  36

maxstringlength:  72

keylength:  4

codefieldlength:  4

tabposition:  10

input:  new text-units
```

3.1 File handling

Author: Dick Grune

Source and documentation: copysfs, copysfd

Institute: Mathematical Centre

Date received: 11/11/74.

Brief description:
The program 'copysfs' accepts a (multi)file, possibly composed of more than one file, and formats it for display on the line-printer. Control characters are added and each record is preceded by a title line giving the file number, record number (both counting from 0), date, time and logical name of the input file.
If the contents of a record cannot be interpreted as characters, the rest of the record will be skipped, to prevent the printing of binary records.
It is possible to restrict the printing to the first 'n' lines of each record. If this option is not chosen, each record will start on a new page and will be printed in full.

Keywords: Output formatting.

Type: Main program, in compass.

Calling sequence:
From the batch:
```
copysfs,par1,par2,par3.
```
From intercom:
```
xeq,libload=infal,copysfs,execute=,par1,par2,par3
```

The program has three parameters:
1. The name of the input file. The file will not be rewound. (default is 'input')
2. The name of the output file. (default is 'output')
3. The maximum number of printed lines per record.

If this parameter is present, the record will not start at a new page. If the parameter is empty or zero, records will still be printed in full; otherwise the parameter must be a decimal integer giving the maximum number of printed lines per record.

Data and results: See brief description.

Subprograms used: None.

Required central memory: 10000b.

Method:
The program reads and writes through internal routines relying on cpc, which do not use automatic recall.

Example of use:
```
copysfs,bigfile,,3.
```
The eor/eof structure of bigfile will be displayed, together with the first 3 lines of every record.

Author: Dick Grune.

Source and documentation: simpios, simpiod

Institute: Mathematical Centre.

Date received: 07/02/75.

Last revision: 03/04/75.

Brief Description:

Simple i/o routines in compass.
The package contains the macros GETWORD, PUTWORD, WORDFILE, OPENGET, and OPENPUT.
GETWORD generates a routine that will read 60-bits words from a file. PUTWORD generates likewise a routine that will write 60-bits words to a file. These routines are activated through rj-calls. They can also handle end-of-record conditions. WORDFILE generates the administration and buffer for a specific file.
Files must be opened by OPENGET and OPENPUT, respectively. The routines can service connected files.

Keywords: I/O, Compass.

Type: Compass macros in systext form.

Calling sequence and results for GETWORD and OPENGET:

```
     NAME  GETWORD
```

A routine with label 'name' is generated. This routine is called as follows:

```
          SX1     FILEADDR     SEE MACRO 'WORDFILE'
          RJ      NAME
```

If now X3 = 0, then X2 contains the next word from the file the address of which was (and still is) in X1. Otherwise X3 > 0 and an end-of-record/file has been read; the routine will not automatically start reading the next record/file (but the next call to 'name' will). The level of the 'eor' is X3-1; end-of-file is yielded as X3 = 16; end-of-information is yielded as double end-of-file.
The routine expects that B1=1 and affects X2...X7 and A1...A7. It is 24 words long and uses SYS=.

All input files must be opened by calls of the macro OPENGET:

```
          SX1    FILEADDR
          OPENGET
```

If an input file is not opened, bad service will result.

<u>Calling sequence and results for PUTWORD and OPENPUT</u>:

```
      NAME  PUTWORD
```

A routine with label 'name' is generated.
To write a word to a file the calling sequence of this routine is:

```
          SX1    FILEADDR     SEE MACRO 'WORDFILE'
          SX2    WORD
          SX3    0            OR MX3 0
          RJ     NAME
```

To write an end-of-record to a file, code:

```
          SX1    FILEADDR
          SX3    LEVEL+1
          RJ     NAME
```

The routine expects that B1=1 and affects X4...X7 and A1...A7. It is 23 words long and uses SYS=.

All output files must be opened by calls of the macro OPENPUT:

```
          SX1    FILEADDR
          OPENPUT
```

If an output file is not opened, bad service will result. All output files must be closed by writing an end-of-record on them. The routines do not have any resistance to bad input parameters: anything may happen.

<u>Calling sequence for WORDFILE</u>:

```
      FILEADDR WORDFILE LFN
```

It generates the fet (see SCOPE RM, ch 11) and buffer for a file with logical file name 'lfn'. The fet is labeled 'fileaddr'; it is this value that is passed to the routines in X1.

For a file on magnetic tape the call is:

```
      FILEADDR WORDFILE LFN,TAPE
```

A call of 'WORDFILE' without tape-parameter occupies 199 words, the tape-version occupies 1543 words. A non-tape file can be handled by the tape-version but a tape-file cannot be handled by the non-tape version.

Calling sequence of compilation of a program using simpio:

The systext 'simpio' is stored in 'infal' and must be made known to the compass-assembler, e.g. in the following way:

```
attach,infal,id=matcen.
compass,s,s=infal/simpio.
```

The single s in the above call preserves the system systext 'systext'.

Example of use:

The following program will copy one file from the file 'tape' to the file 'disk'. The file 'tape' may be on tape.

```
         IDENT   COPY
         ENTRY   COPY

 IN      WORDFILE TAPE,TAPE
 OUT     WORDFILE DISK

 READ    GETWORD
 WRITE   PUTWORD

 COPY    SB1     1
         SX1     IN
         OPENGET
         SX1     OUT
         OPENPUT
 LOOP    SX1     IN
         RJ      READ          SET X2 AND X3
         SX1     OUT
         RJ      WRITE         INPUT TO 'WRITE' MIRRORS OUTPUT OF 'READ'
                               WILL ALSO CLOSE FILES
         AX3     4             DIVIDE BY 16
         ZR,X3   LOOP          OTHERWISE END OF FILE

         ENDRUN
         END     COPY
```

Author: Dick Grune

Source and documentation: ptchsrc, ptchdoc

Institute: Mathematical Centre

Date received: 11/11/74

Brief description:
This program makes the following changes to an ALGOL 3 object module:

1. 00b in the program name are replaced by spaces and spaces in the entry point are replaced by 00b. This is to make the module retrievable from a library.
2. Date and time are added in the prefix table.
3. The right-most 70 spaces in the prefix table are replaced by 00b

2. and 3. are to get a better layout of the loader map and the listing of editlib.

Keywords:
Library, layout, ALGOL 3 object module.

Type: Main program, in compass.

Calling sequence:
From the batch:
ptcha60,par1,par2.
From intercom:
xeq,libload=infal,ptcha60,execute=,par1,par2

The program has two parameters:

1. The name of the input file (default is 'lgo').
2. The name of the output file (default is 'mgo').

Subprograms used: None

Required central memory: 20000b

Method:
The program reads and writes through internal routines relying on cpc, which do not use automatic recall.

Example of use:
The following job compiles an algol program,names it 'name' and puts it in a library:

```
algol.
rewind,lgo.
ptcha60.
editlib.
xeor
'name'
"begin"
   .
   .
"end"
xeor
library(lib1,old)
rewind,mgo.
add(x,mgo,al=1)
finish.
xeof
```

Author: Paul Klint

Sources: chrios1, ... ,chrios5

Institute: Mathematical Centre

Date received: 15/03/75

Brief description:

A filesystem for the processing of binary and character data is described in this section. Some characteristics of this filesystem are:

- ALGOL-60 callable.
- implicit conversions to and from ASCII, display and flexowriter code. The user communicates by means of ASCII characters and the conversions are transparant to him.
- transmission on character or array basis.
- limited string manipulation facilities.

Keywords: Filesystem, character manipulation

TABLE OF CONTENTS

4. APPENDIX

1. INTRODUCTION

A filesystem for the processing of binary and character data is the subject of this section.

Design and implementation of this filesystem were initiated by the desire to alleviate the severe restrictions imposed by the small (63) character set offered by the computer manufacturers system ware. Some characteristics of this filesystem are:

- ALGOL-60 callable (many existing programs should run on the CYBER).
- flexibility:
  - . no fixed line sizes
  - . user controlled end-of-record and error processing
  - . operating system interface through logical filename
- a facility to transmit complete array's, thus eliminating many (time consuming) procedure calls. A stop character may be defined by the user to terminate array reading.
- a facility to write strings.
- 18 different filetypes: 8 read only, 10 write only.
- transparency of different filetypes: only ASCII characters are written to or read from a file. Only binary files form an exception.
- efficiency: all programs are written partly in FORTRAN IV, partly in COMPASS. In this way execution speed compares favourable with the available character processing facilities.
- automatic reprieve at (normal or abnormal) jobtermination. There is no need to explicitly close a file at the end of a program. In case of error termination (time limit, overflow etc.) all data written until the moment the error occurred are actually written.

This description is subdivided in 3 chapters. Chapter 2 describes the basic concepts underlying the structure of the filesystem. A survey of the available filetypes is given. In chapter 3 the ALGOL-60 interface is treated in detail.

2. A FUNCTIONAL DESCRIPTION OF THE FILESYSTEM

2.1 A survey of filetypes

In this section we survey the filetypes the filesystem can process. The types will be discussed in the following order:

types 0 and 16:	8-bit binary	(see 2.1.1.1)
types 1 and 17:	12-bit binary	(see 2.1.1.2)
types 2 and 18:	ASCII, even parity	(see 2.1.2)
types 3 and 19:	ASCII, odd parity	(see 2.1.2)
types 4 and 20:	MC-flexowriter code	(see 2.1.3.1)
types 5 and 21:	ARBA-flexowriter code	(see 2.1.3.2)
types 6 and 22:	display code	(see 2.1.4)
types 7 and 23:	dollar code	(see 2.1.5)
type 24:	63 character set print file	(see 2.1.6.1)
type 25:	95 character set print file	(see 2.1.6.2)

The types 0-7 are read only, 16-25 write only.

2.1.1 Binary files

Files of type binary (0,16,1,17) are not really character files. Binary numbers in the ranges 0-(2**8-1) and 0-(2**12-1) are written to or can be read from these files. (note: we use "**" to denote exponentiation) There exists no character representation for these values but we will often call them characters. A more appropriate name is frame. The concepts line and linewidth (see 2.2) will also be applied to these files.

2.1.1.1 8-bit binary files (readtype 0, writetype 16)

Values in the range 0-(2**8-1) may be written on or can be read from a file of this type. Internally 7 8-bit frames are stored in each 60 bits memory word. The frames are left justified with zero fill. The layout of one memory word looks like:

```
 -------- -------- -------- -------- -------- -------- -------- ----
|        |        |        |        |        |        |        |    |
| frame1 | frame2 | frame3 | frame4 | frame5 | frame6 | frame7 |    |
|        |        |        |        |        |        |        |    |
 -------- -------- -------- -------- -------- -------- -------- ----
                                                                 |
                                                           4 zero bits
```

2.1.1.2 12-bit binary files (readtype 1, writetype 17)

Values in the range 0-(2**12-1) may be written on or can be read from a file of this type. Internally 5 12-bit frames are stored in each 60 bits memory word. The layout of one word is:

```
 ------------ ------------ ------------ ------------ ------------
|            |            |            |            |            |
|   frame1   |   frame2   |   frame3   |   frame4   |   frame5   |
|            |            |            |            |            |
 ------------ ------------ ------------ ------------ ------------
```

2.1.2 ASCII files

Transput via ASCII papertapes for example may be achieved by means of the ASCII filetypes (2, 18, 3, 19). Types 2 and 3 resp. 18 and 19 differ only in the treatment of the paritybit. Values in the range 0-127 may be written on or can be read from a file of these types. Internally each character is treated as a 12-bit frame, the upper 4 bits equal to zero. One word looks like:

```
 ---- -------- ---- -------- ---- -------- ---- -------- ---- --------
|    |        |    |        |    |        |    |        |    |        |
|    | char1  |    | char2  |    | char3  |    | char4  |    | char5  |
|    |        |    |        |    |        |    |        |    |        |
 ---- -------- ---- -------- ---- -------- ---- -------- ---- --------
  |
  | 4 zero bits
```

For read files the even (odd) parity of each frame is checked and the paritybit is stripped off. If a parity error occurs a user defined error procedure is called (see 3.3). For write files a paritybit is attached to the value of the ASCII character to obtain a frame of even (odd) parity.

2.1.3 Flexowriter files

Transput via MC-flexowriter or ARBA-flexowriter papertapes for example may be achieved by means of the flexowriter filetypes. To obtain a transparent behaviour of the different filetypes as seen by the user, ASCII characters must be written to flexowriter files. The conversion from ASCII-code to flexowriter-code is invisible. Reading from a flexowriter file delivers ASCII values. The internal representation is equal to the one used for ASCII files.

2.1.3.1 MC-flexowriter code (readtype 4, writetype 20)

This type is used to write ASCII characters on or read ASCII characters from a file in MC-flexowriter code.
note: The symbols bar (|) and underline (_) are implicitly followed by a backspace symbol, in other words bar and underline do not influence the position in the current line.

2.1.3.2 ARBA-flexowriter code (readtype 5, writetype 21)

Apart from one exception this type is identical to the MC-flexowriter type. The exception concernes the symbols bar and underline: in an ARBA-flexowriter file the symbols bar and underline are not implicitly followed by a backspace symbol and increment the line position.

2.1.4 Display code (readtype 6, writetype 22)

Display code is the favourite 63-character set code of the CDC CYBER- system.

To obtain the transparency already mentioned in 2.1.3 reading from a display code file delivers values in a 65-character ASCII subset. This subset contains all characters available in display code plus the symbols carriage return (CR) and line feed (LF). Every character outside this 65-character ASCII subset written to a display code file is replaced by a question mark (?). Internally each character is treated as a 6-bit frame. Ten characters are stored in one memory word:

```
 ------ ------ ------ ------ ------ ------ ------ ------ ------ ------
|      |      |      |      |      |      |      |      |      |      |
| ch 1 | ch 2 | ch 3 | ch 4 | ch 5 | ch 6 | ch 7 | ch 8 | ch 9 | ch 10|
|      |      |      |      |      |      |      |      |      |      |
 ------ ------ ------ ------ ------ ------ ------ ------ ------ ------
```

A line terminator (hereafter referred to as physical line terminator see 2.2) is represented by zeros in the bits 0 through 12. A line terminator read is replaced by the symbols CR and LF. A carriage return symbol (CR) written to a display code file is replaced by a line terminator. In fact a line feed symbol (LF) written to a display code file has no effect at all. To maintain compatability with other filetypes it is advisable, however, to write the characters CR and LF to generate a newline.

2.1.5 dollar code (readtype 7, writetype 23)

Files in dollar code are introduced to supply the full set of 128 ASCII characters within the restrictions of the 63 character set of the CYBER-system. A "dollar" file derives its name from the fact that a $-sign is used as escape character. Dollar files must be formatted according to the dollar convention described in the following paragraph.

2.1.5.1 The dollar convention

The dollar convention can be subdivided in two parts:

a. case definitions
b. special symbols

Three case definitions are used:

```
lower case       (representation: $>)
upper case       (representation: $<)
underline switch(representation: $- or $_)
```

Upper and lower case affect letters only. For this reason it is not necessary to insert a lower case symbol before certain non-letter symbols.
The following sequence of character is a dollar file

```
$<ex.1 use of upp$>ercase
```

is interpreted as:

```
EX.1 USE OF UPPercase
```

It is cumbersome to write five symbols to get one uppercase letter, for example:

```
$<e$>x. 2
```

with interpretation:

```
Ex. 2
```

In this special case the "less than" symbol (<) of the uppercase symbol and the corresponding lower case symbol may ne omitted. The rule is:

dollarsign followed by a letter is interpreted as one letter uppercase.

Therefore the preceeding example may be written:

    $ex. 2

with the same interpretation.

The underline switch turns underlining on and off. After the occurrence of an underline switch symbol which turned the underlining on, every following (ACSII) character is prefixed by the symbols

    _ (underline) and
    BS (backspace)

until the next underline switch symbol turns the underlining off. Note that certain sequences of symbols line uppercase, lowercase symbol and some others (which we will mention shortly) are not prefixed by the symbols underline and backspace if underlining is on. The sequence of characters

    so$_me u$-nderlinings

is interpreted as

    some underlinings

or more precisely as the sequence of ASCII characters:

```
"s","o","_",BS,"m","_",BS,"e",
"_",BS,SP,"_",BS,"u","n","d","e","r","l","i","n","i","n","g","s".
```

where SP stands for a space symbol.
A less trivial example that covers all rules introduces so far is:

    mi$rabil$<e $-dic$>tu$-.

with interpretation:

    miRabilE DICtu.

The special symbols are subdivided in printable and not printable symbols. The printable symbols are:

ACSII character	dollar representation
BS	$(
HT	$)
LF	$,
VT	$+
CR	$.
$	$$
(percent sign)	$/
(accent grave)	$'
(accolade open)	$[
(accolade close)	$]
(bar)	$!
thilde	$~

notes:

1. The sequence of symbols CR, LF (in dollar notation $.$,) may be abbreviated to $;
2. After the symbols $. and $; a physical end-of-line is inserted to improve the readability of dollar files. This extra end-of-line has no significance (see next section).
3. When reading from a dollar file all symbols between $. or $; and the next physical end-of-line are skipped. Because of this rule copying from one dollar file to another one does not destroy the layout.

The not printable ASCII characters are mostly communication and synchronization symbols. These symbols are represented by:

dollar symbol, ASCII value (octal) plus or minus symbol.

A complete list of these symbols appears in the appendices. One example will illustrate the rule. The ASCII character NAK (negative acknowledge) with value 25 (octal) is represented by:

$25+ or $25-

Though it seems very useless this mechanism may be applied to every ASCII character. In this way the letter-a-symbol (value 97 = 141 (octal)) can be represented by

$141+

2.1.5.2 Dollar convention and dollar files

A dollar file is nothing mcre than a display code file in which a special meaning is attached to the dollarsign. This view on dollarfiles is usefull to discover some problems. By definition a line of ASCII characters is terminated by a carriage return symbol. An arbitrary number of characters can preceede this carriage return symbol. Now observe that

- a display code file and therefore a dollarfile has a fixed linesize, and that
- the representation of ASCII characters according to the dollar convention occupies on the average more than one character position.

An obvious conclusion is that the carriage return symbol only (represented by $. or combined with LF by $;) terminates a line. There is no meaning attached to possible physical end of lines in the dollar file. One line of ASCII characters may be represented in a dollar file by several physical lines and vice versa.

This property of dollar files motivates the introduction of two line positions in a file:

- the logical line position (defined by the stream of ACSII characters) and
- the physical line position (defined by the representation of the stream of ASCII characters).

see 2.2 for more details on this subject.

2.1.6 Print files

Print files are specially formatted for reproduction on a lineprinter. Before each line a printer controlcharacter is inserted to allow line feed, page eject and overprinting.

2.1.6.1 63-character set print file (writetype 24)

A 63 character ASCII subset may be represented on these files. Chararacters outside this subset are replaced by a resembling character.

2.1.6.2 95-character set point file (writetype 25)

The 95 printable ASCII characters can be represented on these

files.
note: this filetype cannot be supported before the operating system
SCOPE 3.4.2 is running on the CYBER installation.

2.2 Logical and physical line position

As pointed out in section 2.1.5 it is necessary to make a clear distinction between a stream of ASCII characters and their representation. We repeat that the logical line position depends on a stream of ASCII characters only. The physical line position depends on the representation of this same stream of ASCII characters.

The properties of logical and physical line position are discussed in three subsections: one for write files, one for read files and a separate section for binary files.

2.2.1 Write files

Files may be regarded as a collection of lines. ASCII characters written to a file can influence the character position in the line which is currently under construction. Each file has three properties concerning the logical line position:

logical line width	(logw)
logical line position	(logpos)
left margin	(lm)

at most logw ASCII characters, which increment the logical line position may be written before the symbols CR, LF and a number of space symbols (SP) equal to the current value of the left margin are inserted automatically.

Note that logical line width and left margin can be controlled directly by the user (see 3.). Each file has two other properties concerning the physical line position:

physical line width	(physw)
physical line position	(physpos)

Not more representations of ASCII characters, or parts thereof, that increment the physical line position from 0 to physw can be written before a physical end of line mark is inserted. This process is independent from the influence of the ASCII characters on the logical line position.

An example will probably clarify these definitions and rules.

Suppose the ASCII characters

aBCDefgHIJKl

are written to a dollar file with specifications:

logical line width = 5
physical line width = 6

These write operations have the effect:

	a	$	<	b	c	d	***
physpos	1	2	3	4	5	6	
logpos	1	1	1	2	3	4	

	$	>	e	$	;	***
physpos	1	2	3	4	5	
logpos	4	4	5			

	f	g	$	<	h	i	***
physpos	1	2	3	4	5	6	
logpos	1	2	2	2	3	4	

	j	$	;
physpos	1	2	3
logpos	5		

	k	$	>	l
physpos	1	2	3	4
logpos	1	1	1	2

note:

***	physical end-of-line inserted
$; ***	CR, LF anf physical end-of-line inserted

2.2.2 Read files

Read files resemble write files in almost every respect. Only one exception exists: when reading from a file the symbols CR and LF are never inserted, in other words the value of the logical line width is unreachable and input lines of indefinite length can occur. The logical lise position in only reset to zero if a CR symbol is encountered. The physical line position is restricted to the values $0 \leq physpos \leq physw$.

Outside this range symbols are skipped until the next physical end of line mark. For read files the values of left margin and logical line width have no meaning.

2.2.3 Binary files

Binary files follow the rules: When exactly logw frames have been written a number of zero frames equal to the current value of leftmargin are inserted. Each frame read from a binary file increments the logical line position by one.

2.2.4 summary

A summary of section 2.2 is given in the following table:

file description	type	chars/ log.line	chars/ phys.line	type	chars/ log.line	chars/ phys.line
8-bit bin.	16	lw	lw	0	or	or
12-bit bin.	17	lw	lw	1	or	or
ASCII even	18	lw	lw	2	or	or
ASCII odd	19	lw	lw	3	or	or
MC flex.	20	lw	lw	4	or	or
ARBA flex.	21	lw	lw	5	or	or
display	22	lw	phw	6	or	phw
dollar	23	lw	phw	7	or	phw
63 print	24	lw	phw	-	-	-
95 print	25	lw	phw	-	-	-

lw:	logical line width
phw:	physical line width
or:	indefinite sequence of characters terminated by a CR symbol.

3. ALGOL-60 interface with the filesystem

This chapter contains a detailed description of a set of procedures which can be used to communicate with the filesystem. The behaviour of these procedures and the meaning of their parameters is explained fully.

The maximum number of files which can be processed simultaneously is an assembly parameter of the filesystem itself. In the sequal we refer to this value minus one as "maxfiles". The value of maxfiles in the current version is seven.

3.1 Open

BRIEF DESCRIPTION:

Before a file can be accessed the procedure open must be called to specify certain properties like filetype, filename etc. Every attempt to access file before it is opened results in a fatal error.

CALLING SEQUENCE:

```
    procedure open(n,lfn,w,t,rew);
    value n,w,t,rew;
    integer n,w,t,rew; strint lfn;
    code 13000;
```

note: the not existing specifier strint is used to indicate a type that may be either string or integer.

The meaning of the parameters is:

n	: filenumber, which will identify all following accesses for this file. The filenumber must obey the restriction $0 \leq n \leq$ maxfiles.
lfn	: legal SCOPE logical filename. Two types are allowed for lfn: either a string ("("fileone")") or an integer (the integer equivalent of a string which does not exceed eight characters; this integer equivalent can be obtained by means of the ALGOL 60 library function equiv or by means of an explicit computation.)
w	: logical and physical line width. The value of w is assigned to both system variables. The value of the logical line width can be changed afterwards by means of a call to the procedure rmarge (see 3.11), the value of the physi al line width, however cannot be changed without first closing (see 3.2) and then reopening the file with a different value for w.
t	: filetype in one of the ranges $0 \leq t \leq 7$ for readfiles and $16 \leq t \leq 25$ for writefiles. Note that an attempt to read from a write file and vice versa causes a fatal error.
rew	: rewind option. File will be rewind if the value of rew is not

equal to zero.

EXAMPLE OF USE:

example:

```
        open(3,"("file3")",72,6,1); meaning:
        open a file with filenumber 3, logical filename "file3",
        logical and physical line width 72. The filetype is 6 (display
        code, reading) and rewind file before processing.
```

3.2 Close

BRIEF DESCRIPTION:

Close completes the processing of a certain file. After a call of close the filenumber which identified all accesses for this file is released and may be used again in a call of open to identify accesses to another file. At job termination (normal or abnormal) the filesystem calls close for every file not yet closed. For this reason the programmer does not have to close all files at the end of his program.

CALLING SEQUENCE:

```
    procedure close(n);
    value n;
    integer n;
    code 13001;
```

The meaning of the parameter is:

n	: filenumber of the file for which all accesses are terminated.

EXAMPLE OF USE:

example: close(3)
meaning: evident.

3.3 Rdchar

BRIEF DESCRIPTION:

Rdchar reads the next character from a file. If the character is not present (end-of-file or end-of-record encountered) a user supplied end-of-record procedure is executed. If the next character is erroneous (parity-error, violation of dollarconvention) a user supplied error procedure is executed. In the normal case, however, the value of the character is delivered as value of the function identifier. Reading from a file with type outside the range 0-7 is fatal error.

CALLING SEQUENCE:

```
    integer procedure rdchar(n,eor,error);
    value n;
    integer n;
    procedure eor,error;
    code 13002;
```

The meaning of the parameters is:

n : filenumber.

eor : end-of-record procedure. This procedure will be executed when during a read operation an end-of-record is encountered. The procedure eor must be declared as:

```
        procedure eor(n,eorlev,eorsym);
        value n;
        integer n,eorlev,eorsym;
```

The parameters of the procedure eor have the meaning:

n	: filenumber of file in which end-of-record was encountered.
eorlev	: end-of-record level. An end-of-file is equivalent with eorlev = 15.
eorsym	: end-of-record symbol. eorsym defines the value rdchar must deliver for the current read operation. A meaningful value must be assigned to eorsym in the body of the procedure eor.

error : errorprocedure, to be executed when an errorcondition occurs during a read operation. The procedure error must be declared as

```
        procedure error(n,er);
        value n;
        integer n,er;
```

The parameters have the following meaning:

```
n       : filenumber of the file in which the error occurred.
er      : errorcode and errorcharacter.
          Possible values of er are:
          -1, ..., -255 parity error: the value read inverted
                                      from sign is passed as
                                      value of er.
          -299                        (in dollar file) exceed
                                      digits following a dollar
                                      sign the value 127.
          -300                        (in dollar file) is a
                                      dollar sign followed by
                                      digits not closed by a
                                      plus or a minus sign.
          -302                        (in dollar file) is a
                                      dollar sign followed by
                                      an illegal symbol.
          -303                        ring the bell! a system
                                      inconsistency occurred.
          To er a meaningful value must be assigned in the
          body of the procedure error: this value replaces the
          erroneous character and is delivered as value of
          rdchar.
```

EXAMPLE OF USE:

example: after the declarations:

```
        procedure eor(n,eorlev,eorsym);
        value n;
        integer n,eorlev,eorsym;
        if eorlev = 15 then goto eof reached else eorsym:= -1;
        procedure error(n,er);
        value n;
        integer n,er;
        begin errorcount:= errorcount+1;
              er:= -2
        end error;
```

integer errorcount;

and a proper call of open:

open(3,"("inpfile")",80,4,1);

rdchar could be called as:

n:= rdchar(3,eor,error);

In this example "eof reached" is supposed to be a global label.

meaning: evident.

3.4 Rdarray

BRIEF DESCRIPTION:

Rdarray reads characters from a file and assigns the values of these characters to consequtive elements of a given array. The number of characters read is delivered as value of the function identifier. End-of-record and error processing are identical to the method described in the preceeding section. One additional feature of rdarray is the definition of a stopcharacter. Characters are read from the file until either the array is completely filled or the stopcharacter is encountered.

CALLING SEQUENCE:

```
integer procedure rdarray(n,a,eor,error,stop);
value n,stop;
integer n,stop;
integer array a;
procedure eor,error;
code 13003;
```

The meaning of the parameters is:

n	: filenumber.
a	: array to be filled with character values. If a is more dimensional the array is filled rowwise.
eor	: end-of-record procedure. see 3.3 for a complete description. When an end-of-record or end-of-file is encountered rdarray returns immediately with a partially filled array a. When rdarray is called the nexttime and no calls of rdchar for the same file occurred in the meantime, the eor procedure is executed before any array element is assigned a value. In this case the user defined value of "eorsym" will be assigned to the first array element.
error	: error procedure. see 3.3 for a complete description. The position of the errorneous character is marked in array a by the value -1 before procedure error is called. The user defined value of this marked array element.
stop	: stopcharacter. When a character with value "stop" is encountered in the file

during a call of rdarray, reading is terminated and the last array element defined contains the value of "stop". Rdarray behaves as a readline procedure if the value of stop is equal to carriage return or line feed.

EXAMPLES OF USE:

example: after declaration of eor, error, an array a[1:100] and a proper call of open, rdarray could be called as:

```
a n:= rdarray(3,a,eor,error,13)
```

or as

```
b n:= rdarray(3,a,eor,error,-1)
```

meaning: a fill all 100 elements of array a with characters from a file with filenumber 3 unless a carriage return (ASCII value 13) occurs. Return in that case with array a filled with all characters upto and including this carriage return.

b fill all 100 elements of array a. The value -1 implies that the stopcharacter will never be encountered. (any value outside the range 0-127 can be used instead of -1).

3.5 Wrchar

BRIEF DESCRIPTION:

Wrchar writes one character to a file. Writing to a file with type outside the range 16-25 results in a fatal error.

CALLING SEQUENCE:

```
    procedure wrchar(n,char);
    value n,char;
    integer n,char;
    code 13004:
```

The meaning of the parameters is:

n	: filenumber.
char	: character to be written. The legal values for char differ for binary and character files.

```
         8-bit binary : 0 ≤ char ≤ 2**8  -1
        12-bit binary : 0 ≤ char ≤ 2**12 -1
        other type    : 0 ≤ char ≤ 127 (ASCII character)
```

EXAMPLE OF USE:

```
example: open(6,"("file")",80,22,1);
         wrchar(6,65);
```

meaning: the letter "a" (ASCII value 65) is written to the file with logical filename "file".

3.6. Wrarray

BRIEF DESCRIPTION:

Wrarray writes a specified number of character values, which are stored in an array, to a file.

CALLING SEQUENCE:

```
    procedure wrarray(n,a,l);
    value n,l;
    integer n,l;
    integer array a;
    code 13005;
```

The meaning of the parameters is:

n : filenumber.

a : array containing characterdata.
for legal values see 3.5.2.

l : length of traject.
Suppose array a is declared as

```
        array a[begin:end];
```

now distinguish the two cases:

```
begin+l < end: the values in
               a[begin], a[begin+1], ... , a[begin+l-1]
               are written.

begin+l > end: the values in a[begin], ... a[end]
               are written.
```

The index of the last array element to be written is determined by

```
        minimum(begin+l,end).
```

EXAMPLE OF USE:

```
example: integer array a[1:3];
         open(6,"("file")",80,22,1);
         a[1]:= 65; a[2]:= 66; a[3]:= 67;
         wrarray(6,a,2); wrarray(6,a,10);
```

meaning: a the first call of wrarray writes the letters "a" and "b" to file "file".

b the second call of wrarray writes the letters "a", "b" and "c" to file "file".

3.7 Wrtext

BRIEF DESCRIPTION:

Wrtext writes a string in dollarformat to a file.

CALLING SEQUENCE:

```
    procedure wrtext(n,s);
    value n;
    integer n;
    string s;
    code 13006;
```

The meaning of the parameter is:

n	: filenumber.
s	: string according to dollar convention. Any violation of the dollar convention results in a fatal error.

EXAMPLE OF USE:

example1:

```
        open(1,"("file")",80,22,1);
        wrtext(1,"("ab$/c$;e")");
```

meaning1:

the symbols: "a", "b", "?", "c", CR, LF, "e" are written to file "file1".
Note the replacement of the percent sign (represented by $/) by a questionmark because percent sign is not available in display code.
If "file1" was empty before the call of wrtext it contains now:

```
                ab?c
                e
```

example2:

```
        open(1,"("file2")",80,23,1);
        wrtext(1,"("ab$/c$;e")");
```

meaning2:

the symbols: "a", "b", (percent sign), CR, LF, "e" are written

to file "file2".
If file2 was empty before the call of wrtext it contains now:

```
ab$/c$;
e
```

3.8 Pos

BRIEF DESCRIPTION:

Pos delivers the current value of the logical position in a file. Pos gives then the value of the last used position.

CALLING SEQUENCE:

```
    integer procedure pos(n);
    value n;
    integer n;
    code 13007;
```

The meaning of the parameter is:

n	: filenumber.

EXAMPLE OF USE:

example1:
suppose the display code file we are reading from contains as current line:

abcdefg

and we have already read the characters "a" and "b". A call of pos delivers the value 2.

example2:
suppose that two characters have been written on the current line of a display code file. A call of pos delivers the value 2.

3.9 Type

BRIEF DESCRIPTION:

Typ delivers the filetype of a given file.

CALLING SEQUENCE:

```
integer procedure type(n);
value n;
integer n;
code 13008;
```

The meaning of the parameter is:

n : filenumber.

EXAMPLE OF USE:

example1:

```
open(3,"("f")",80,25,1);
n:= type(3);
```

meaning1:

the value 25 is assigned to n.

example2:

```
n:= type(4);
```

meaning2:

Suppose a file with filenumber4 is not opened. The value -1 is assigned to n.
This is the only operation possible on a not opened file.

3.10 Lmargin

BRIEF DESCRIPTION:

Lmargin is used to redefine or inspect the value of the left margin of a file.

CALLING SEQUENCE:

```
    procedure lmargin(n,x,q);
    value n,q;
    integer n,x,q;
    code 13010;
```

The meaning of the parameters is:

n : filenumber.

x : variable or expression.
distinguish the two cases:

a inspection of left margin (q=0).
The value of left margin is assigned to the variable x.

b redefinition of left margin ($q \neq 0$).

The value of the variable or expression x is assigned to left margin. The value of x must lie in the rang $0 \leq x <$ logical line width. The effect of redefinition of the left margin is not visible until the next line. In other words redefinition does not influence the number of spaces or binary zeros written at the beginning of the current line.

q : question or redefine.
$q = 0$ inspect.
$q \neq 0$ redefine.

EXAMPLE OF USE:

example:
```
        open(1,"("file")",6,22,1);
        wrtext(1,"("abcdefg")");
        lmargin(1,3,1);
        wrtext(1,"("hijklmnop")");
```

meaning:
"file" contains the characters:

```
abcdef
ghijkl
   mno
   p
```

3.11 Rmargin

BRIEF DESCRIPTION:

Rmargin is used to redefine or inspect the value of the logical line width of a file.

CALLING SEQUENCE:

```
    procedure rmargin(n,x,q);
    value n,q;
    integer n,x,q;
    code 13009;
```

The meaning of the parameters is:

n	: filenumber.
x	: variable or expression (see 3.10.2)
q	: question or redefine (see 3.10.2).

Redefinition of the logical line width influences the number of characters written on the next line. The value of logical line width at the moment that a line is started determines the maximum number of position incrementing characters on that line.

EXAMPLE OF USE:

example:
```
         open(1,"("file")",6,22,1);
         wrtext(1,"("abcdefg")");
         rmargin(1,3,1);
         wrtext(1,"("hijklmnop")");
```

meaning:
"file" contains the characters:

```
                        abcdef
                        ghijkl
                        mno
                        p
```

	0	1	2	3	4	5	6	7	8	9
0	NULL	SOH	STX	ETX	EOT	ENQ	ACK	BEL	BS	HT
									BS	TAB
10	LF	VT	FF	CR	SO	SI	DLE	DC1	DC2	DC3
								STOP	PUOF	
20	DC4	NAK	SYN	ETB	CAN	EM	SUB	ESC	FS	GS
30	RS	US	SP	!	"	#	$	%	∧	'
			SP	?	"	?	?	?	⋏	'
40	(	)	*	+	,	-	.	/	0	1
	(	)	*	+	,	-	.	/	0	1
50	2	3	4	5	6	7	8	9	:	;
	2	3	4	5	6	7	8	9	:	;
60	<	=	>	?	@	A	B	C	D	E
	<	=	>	?	TEN	A	B	C	D	E
70	F	G	H	I	J	K	L	M	N	O
	F	G	H	I	J	K	L	M	N	O
80	P	Q	R	S	T	U	V	W	X	Y
	P	Q	R	S	T	U	V	W	X	Y
90	Z	[	\	]	⋏	_	`	a	b	c
	Z	[	\	]	NOT	_	?	a	b	c
100	d	e	f	g	h	i	j	k	l	m
	d	e	f	g	h	i	j	k	l	m
110	n	o	p	q	r	s	t	u	v	w
	n	o	p	q	r	s	t	u	v	w
120	x	y	z	{	|	}	~	DEL		
	x	y	z	?	|	?	?			

notes:

STOP:	stopcode
PUOF:	punch off
NOT:	logical not sign
TEN:	low ten symbol
SP:	space symbol

	0	1	2	3	4	5	6	7	8	9
0	NULL	SOH	STX	ETX	EOT	ENQ	ACK	BEL	BS	HT
		$1+	$2+	$3+	$4+	$5+	$6+	$7+	$(	$)
10	LF	VT	FF	CR	SO	SI	DLE	DC1	DC2	DC3
	$,	$+	$14+	$.	$16+	$17+	$20+	$21+	$22+	$23+
20	DC4	NAK	SYN	ETB	CAN	EM	SUB	ESC	FS	GS
	$24+	$25+	$26+	$27+	$30+	$31+	$32+	$33+	$34+	$35+
30	RS	US	SP	!	"	#	$	%	^	'
	$36+	$37+	SP	!	"	#	$$	$/	^	'
40	(	)	*	+	,	-	.	/	0	1
	(	)	*	+	,	-	.	/	0	1
50	2	3	4	5	6	7	8	9	:	;
	2	3	4	5	6	7	8	9	:	;
60	<	=	>	?	@	A	B	C	D	E
	<	=	>	?	@	A	B	C	D	E
70	F	G	H	I	J	K	L	M	N	O
	F	G	H	I	J	K	L	M	N	O
80	P	Q	R	S	T	U	V	W	X	Y
	P	Q	R	S	T	U	V	W	X	Y
90	Z	[	\	]	↑	_	`	a	b	c
	Z	[	\	]	↑	_	$'	A	B	C
100	d	e	f	g	h	i	j	k	l	m
	D	E	F	G	H	I	J	K	L	M
110	n	o	p	q	r	s	t	u	v	w
	N	O	P	Q	R	S	T	U	V	W
120	x	y	z	{	|	}	~	DEL		
	X	Y	Z	$[	$!	$]	$↑			

TABLE C: ascii -> 63 char print file

	0	1	2	3	4	5	6	7	8	9
0	NULL	SOH	STX	ETX	EOT	ENQ	ACK	BEL	BS	HT
									BS	TAB
10	LF	VT	FF	CR	SO	SI	DLE	DC1	DC2	DC3
	LF	LF	FF	CR						
20	DC4	NAK	SYN	ETB	CAN	EM	SUB	ESC	FS	GS
30	RS	US	SP	!	"	#	$	/	^	'
			SP	!	"	#	$	/	^	'
40	(	)	*	+	,	-	.	/	0	1
	(	)	*	+	,	-	.	/	0	1
50	2	3	4	5	6	7	8	9	:	;
	2	3	4	5	6	7	8	9	:	;
60	<	=	>	?	@	A	B	C	D	E
	<	=	>	?	@	A	B	C	D	E
70	F	G	H	I	J	K	L	M	N	O
	F	G	H	I	J	K	L	M	N	O
80	P	Q	R	S	T	U	V	W	X	Y
	P	Q	R	S	T	U	V	W	X	Y
90	Z	[	\	]	↑	_	`	a	b	c
	Z	[	\	]	↑	_	'	A	B	C
100	d	e	f	g	h	i	j	k	l	m
	D	E	F	G	H	I	J	K	L	M
110	n	o	p	q	r	s	t	u	v	w
	N	O	P	Q	R	S	T	U	V	W
120	x	y	z	{	|	}	~	DEL		
	X	Y	Z	[	!	]	↑			

	0	1	2	3	4	5	6	7	8	9
0	NULL	SOH	STX	ETX	EOT	ENQ	ACK	BEL	BS	HT
	?	?	?	?	?	?	?	?	?	?
10	LF	VT	FF	CR	SO	SI	DLE	DC1	DC2	DC3
		?	NL	NL	?	?	?	?	?	?
20	DC4	NAK	SYN	ETB	CAN	EM	SUB	ESC	FS	GS
	?	?	?	?	?	?	?	?	?	?
30	RS	US	SP	!	"	#	$	%	∧	'
	?	?	SP	!	"	#	$	?	∧	'
40	(	)	*	+	,	-	.	/	0	1
	(	)	*	+	,	-	.	/	0	1
50	2	3	4	5	6	7	8	9	:	;
	2	3	4	5	6	7	8	9	:	;
60	<	=	>	?	@	A	B	C	D	E
	<	=	>	?	@	A	B	C	D	E
70	F	G	H	I	J	K	L	M	N	O
	F	G	H	I	J	K	L	M	N	O
80	P	Q	R	S	T	U	V	W	X	Y
	P	Q	R	S	T	U	V	W	X	Y
90	Z	[	\	]	↑	_	`	a	b	c
	Z	[	\	]	↑	_	?	A	B	C
100	d	e	f	g	h	i	j	k	l	m
	D	E	F	G	H	I	J	K	L	M
110	n	o	p	q	r	s	t	u	v	w
	N	O	P	Q	R	S	T	U	V	W
120	x	y	z	{	|	}	~	DEL		
	X	Y	Z	?	?	?	?			