CHAPTER 8 THE REPORT/BATCH LANGUAGE

This chapter explains the use of the report/batch language and details the functions, declarations and commands available.

Contents **Page**

Introduction to the report/batch language 8-2
Program structure 8-2
Driving logic 8-2
Field formats 8-5
Expressions and operators 8-6
Type conversion 8-7
Exception traps 8-8
Field lists 8-8
The **key=** clause 8-9
Command line syntax for **cr** 8-10
Command line syntax for **sagerep** 8-10
Command section contents 8-11

Introduction to the report/batch language

The report/batch language has been designed primarily for the production of reports and for batch processing. A powerful driving logic assists the programmer in this task.

A *driving file* is declared which will be traversed automatically by the driving logic. Cross reference files may be manually or automatically read as the main file is traversed. The main body of statements are executed once for each record in the driving file, and declarations exist to control the automatic selection or exclusion of records from the driving file.

The report/batch language elements are specifically designed for the batch processing of Sculptor keyed files. There are many powerful commands, making it possible to create sophisticated programs quickly and easily. Most of the hard work associated with page management, headers and footers, report titles, data alignment, printer control and performing complex file manipulations are handled automatically. The language has many useful default actions, but the programmer is free to override them as required.

Program structure

A report/batch program is written as a standard text file using an editor. The language is line-oriented and recognises these line types:

1. Lines commencing with a full stop are comment lines and are ignored by the compiler. Blank lines are also ignored.

2. Lines commencing with an exclamation mark "!" are declarations. Declarations are used to initialise the program and to define special actions.

4. Lines commencing with a plus sign "+" are format definitions and are used to override a field's default heading or format.

6. Other lines are program statements. If the first word on the line is not a field name or a Sculptor reserved word, and if it is in the leftmost column, then it is taken to be a line label. Multiple statements, separated by colons, may be placed on a single line.

Program statements may extend over more than one physical line by terminating each line that is to be continued with a backslash "\" character.

Driving logic

To make best use of the report/batch language, it is important to understand the driving logic. When the program is executed, the driving logic perfoms any initialisation statements then begins reading the driving file. By default, the driving file will be read from start to end, but initialisation statements may be used to specify the start and end record and also any selection or exclusion conditions.

The driving logic performs page management functions, perfoming report title, heading and footnote statements as required. Statements may also be defined which will be executed whenever a field changes, either on starting a new field value or on ending a field value. This allows grouping and sub-totalling to be performed easily.

Declarations are grouped into sets, each set being executed at the appropriate place in the driving logic. Within any one set, statements are executed in the order in which they are defined in the program.

The initialisation statements are executed first, followed by the title statements. The driving file is then read in ascending key sequence either from beginning to end or within the limits defined by **!startrec** and **!endrec** and each record is processed as follows.

Driving file

!file

Initialisation

!init
!input
!display
!constant
!read
!startrec
!endrec

Report title

!title

Control

!select if
!exclude if
!xfile (with **key**= clause)
!temp (with assignment)

Page heading

!heading

Page footnote

!footnote

On starting field value

!on starting

On ending field value

!on ending

Final

!final

Main statements

main body of statements

First, the control statements are executed. If, whilst doing so, an exclusion condition is found to be true and no prior selection condition was true, then the processing of statements terminates immediately and the next record is read.

Next, any **!on ending** statements are checked. If any require executing, sagerep backtracks to the state that existed before the current set of records was read and executes that **!on ending** statement. (Temporary data modified by the **!on ending** statements is carried forward.) If the special functions **total, min, max** and **count** are used within an **!on ending** statement, they refer only to the group of records read since the last **!on ending** statement for that field was executed or, if none, since the beginning of the file.

The **!on starting** statements are then checked and executed as necessary.

The main block of statements is then executed, following which all **total, min, max** and **count** function accumulators are updated.

If the end of the driving file has been reached, or if the key of the current record exceeds that defined with an **!endrec** declaration, or if an **exit** command is encountered, the **!on ending** statements are executed, followed by the **!final** statements. The program then exits.

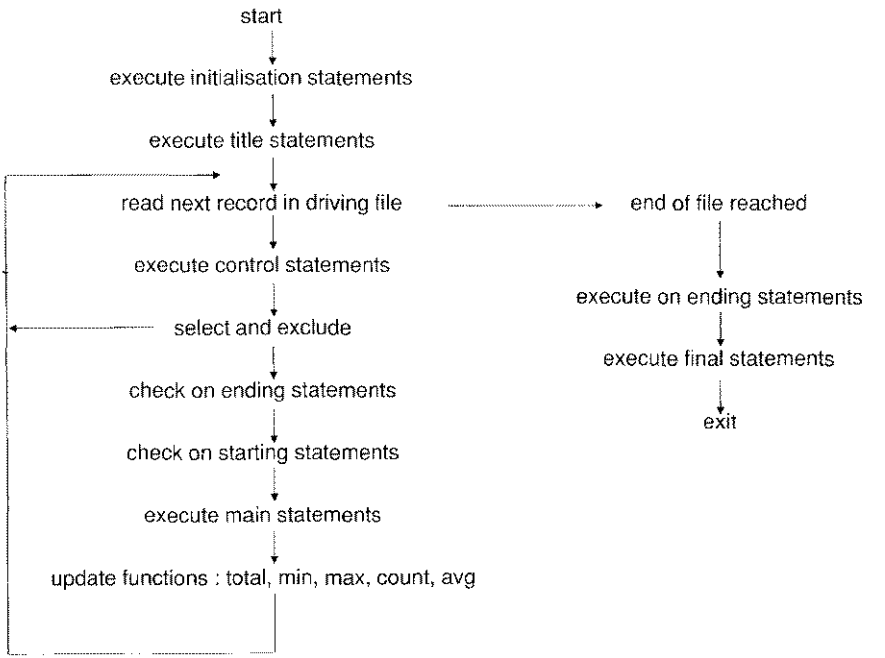
The **!heading** statements are executed at the top of each page and the **!footnote** statements are executed at the bottom of each page. Since these statement sets may be executed at any time, care should be taken not to cause unintentional changes. For example, it is wise to ensure that the special temp **scrline** is not altered. So, if the **scroll** command is used in the heading, the following code is recommended:

```
!heading scrsave = scrline
!heading  ... other heading statements
!heading scroll scrsave
```

Careful study of the report/batch language driving logic will allow your programs to take maximum advantage of the work being performed on your behalf.

The driving logic may be bypassed completely by placing all code for the program within a **!init** declaration.

The diagram below shows the processing logic applied by the report/batch language.



Field formats

Any field used in the program may have a format or a heading redefined for it through the use of the "+" declaration:

+ *field_name*, [*heading*] [, *format*]

Changing the format may affect the way in which the field is printed, input or assigned to. Alpha fields are always a fixed length (as defined by the size of the field), but their behaviour may be modified using a format modifier as defined below:

l	Force lower case on input
u	Force upper case on input
s	Remove leading spaces when printed
t	Remove trailing spaces when printed
r	Remove trailing spaces from **get** and **put**
n	Null terminated field (see **get** for details)
m+format	When a numeric value is assigned to this alpha field, treat it as a money value with two implied decimal places and use this format.
d+format	When a numeric value is assigned to this alpha field, treat it as a date (day number) and use this format.
+format	When a numeric value is assigned to this alpha field, use this format instead of the default format.

The format will only apply to fields which are input unless the format is followed with a plus sign "+" in which case the format will also apply when the field is assigned. As **sagerep** does not deal with the vdu in single character i/o mode when inputting characters, any characters entered will be echoed as typed, but the format defined will be correctly applied to the field as stored.

The format for a numeric field will affect the size of the field printed on the report, but will not affect the value that is stored for that field. Formats available for numeric fields may comprise any combination of these special characters:

#	Digit, blank if leading zero
0	Digit, shown as zero if leading zero
*	Digit, shown as asterisk if leading zero
,	Comma, grouping significant digits
.	Decimal point position

For date fields, the characters **dmy** are used to to determine the date format. A date separator character must also be defined. The examples below show alpha, date and numeric formats.

dd/mm/yyyy	Format date field with 4-digit year
dd.mm	Year is not printed
u+	All assignment and input forced to upper case
d+dd/mm/yy	Assign date to alpha using the defined format
#,###.##	Short format for an m4/m8/r8 field
#.########	Only valid for an r8 field

Expressions and operators

The report/batch language supports a comprehensive set of arithmetic and relational operators. These may be used to form expressions involving record fields, temporary fields and constants.

A table of the operators is shown below in descending order of precedence with operators having equal precedence grouped together. Parentheses may be used to force the order of evaluation.

Group	Operator	Function
1	-	Negation (unary minus)
2	*	Multiplication
	/	Division
	%	Modulus (remainder after integer division)
	/	String concatenation (trailing spaces removed)
3	+	Addition
	-	Subtraction
	+	String concatenation (trailing spaces preserved)
4	=	Equality
	<	Less than
	>	Greater than
	<=	Less than or equal to
	>=	Greater than or equal to
	<>	Not equal to (inequality)
	ct	Contains (string only)
	bw	Begins with (string only)
5	and	Logical AND
6	or	Logical OR

Numeric and alphanumeric constants may be freely used in expressions. A numeric constant is floating point if it includes a decimal point and integer otherwise. Alphanumeric constants must be enclosed in single or double quotes.

Type conversion

Expressions may include fields or constants of different types. Each operation examines the types of its operands and, if they differ, a type conversion is performed according to these rules:

1. If either operand is floating point then the other operand is converted to floating point and the result is floating point.
2. Otherwise, if either operand is integer then the other operand is converted to integer and the result is integer.
3. An operation is only alphanumeric if both its operands are alphanumeric.

When performing arithmetic operations on m4 or m8 fields remember that their data values are stored in the lower currency unit.

On conversion of real values to the integer types i1, i2, i4 or m4, the fractional part is truncated. The **rounding** command may be used to perform rounding prior to conversion.

Exception traps

Whenever an exception condition arises, Sculptor applies an appropriate default action. The exception traps allow the programmer to specify an alternative action if the default is not suitable. Trap clauses have the syntax:

trap = label

Where *trap* identifies the condition being trapped and *label* specifies the line to transfer control to if the condition occurs. A trap clause forms part of the command to which it relates and the allowed traps for a particular command are specified in the syntax of that command.

The available traps are summarised below. Each trap is explained in detail in the description of the command to which it applies.

Trap	Meaning
err	An external error occured (error code in **errno**)
nrs	No record selected
nsr	No such record
re	Record exists
riu	Record in use

Field lists

A field list is a list of field names separated by commas. Ranges are not permitted. Each field name may be from any file or alternative record layout defined in the program or may be a temporary field.

The key= clause

Commands which read records from a Sculptor keyed file according to a supplied key value have an optional **key=** clause. This clause allows the programmer to specify a field or list of fields which will be concatenated to form a key value for the operation. If the clause is omitted, the existing key data in the record buffer is used as the key value.

If the fields being used in the **key=** clause are not the same type and size as the natural key fields for the file, the following rules are applied.

1. If the number of fields in the clause is less than or equal to the number of key fields then each named field is assumed to supply data of the correct type for the corresponding natural key field and no type conversion takes place. Excess bytes are discarded and insufficient bytes are made up with nulls for numeric fields and spaces for alpha fields.

2. If the number of fields is greater than the number of natural key fields then the data from the named fields is concatenated to form a key. If the result exceeds the key length then the excess bytes are discarded. If the result is less than the key length then the remaining bytes are set to spaces.

Since the file access commands always construct keys according to the main record layout, considerable care must be exercised when reading records which have a different key structure to that of the main record (eg, when using alternate record layouts).

As the alternate record layout simply overlaps the main record layout, the recommended method is to assign values to the key fields in the alternate layout and omit the **key=** clause entirely.

Command line syntax for cr

cr [**-z**] [**-f=***funcs*] *filename*

Compiles the source file filename.**r** and produces the compiled file filename.**q** if compilation is successful.

The **-z** option forces fields which contain zero to be shown as blank. Zeros will be shown by default. This option may be overriden within the program through the use of the **!zeros** declaration.

The **-f** option allows adjustment of the number of fields that can be used with the special functions **total**, **min**, **max** and **count**. The default is 20 fields. Increase this number only if the "too many functions" error is encountered when compiling.

Command line syntax for sagerep

sagerep *filename* [*printername* [*arguments*]...]

Executes the compiled file *filename*.**q** and uses the printer parameter file *printername* if given. If *printername* is not given, the file named **printer** is used.

Arguments may be accessed within the program through the **arg** special temporary field.

CONTENTS

REPORT/BATCH LANGUAGE COMMAND SECTION

Contents **Page**

Language functions 8-15
String functions 8-15
Numeric functions 8-15
Special functions 8-15
asc() 8-17
centre() / center() 8-18
chdir() 8-19
chr() 8-20
dim() 8-21
inchar() 8-23
instr() 8-24
left() 8-25
power() 8-26
rand() 8-27
remove() 8-29
right() 8-30
setstr() 8-31
strlen() 8-32
sqrt() 8-33
tolower() 8-34
toupper() 8-35

Language declarations 8-37
!cfile 8-37
!constant 8-38
!depth 8-39
!display 8-40
!exclude 8-41
!file 8-43
!final 8-45
!footnote 8-46
!gap 8-47
!handles 8-48
!heading 8-50
!init 8-51
!input 8-52
!on ending 8-53
!on starting 8-54
!read 8-55

!record . . . 8-56
!select . . . 8-58
!startrec . . . 8-59
!temp . . . 8-60
!width . . . 8-65
!xfile . . . 8-66
!zeros . . . 8-67

Language commands . . . 8-68
abort . . . 8-68
chain . . . 8-69
clearbuf . . . 8-70
close . . . 8-71
close # . . . 8-72
decdate . . . 8-73
delete . . . 8-74
display . . . 8-75
encdate . . . 8-76
end . . . 8-77
exec . . . 8-78
exit . . . 8-80
find . . . 8-81
get # . . . 8-83
gosub . . . 8-85
goto . . . 8-86
hangup . . . 8-87
if . . . 8-88
input . . . 8-90
insert . . . 8-91
interrupts . . . 8-92
keep . . . 8-93
let . . . 8-94
lock . . . 8-95
match . . . 8-96
newpage . . . 8-97
next . . . 8-98
nextkey . . . 8-99
open . . . 8-100
open # . . . 8-101
pause . . . 8-102
prev . . . 8-103
prevkey . . . 8-104
print / printh . . . 8-105
put # . . . 8-108
read . . . 8-109

readkey 8-110
return 8-111
rewind 8-112
rewind # 8-113
rounding 8-114
scroll 8-115
sleep 8-116
testkey 8-117
unlock 8-118
wakeup 8-119
width 8-120
write 8-121

REPORT / BATCH LANGUAGE FUNCTIONS

Functions in Sculptor are commands that return a value. Some functions return string (alpha) values, others return integer or floating point values. Functions may be used wherever an expression of the returned type is valid. A summary of the available functions by return type is shown below and a detailed description of each function in alphabetic order follows.

STRING FUNCTIONS

The length of the alpha field returned is determined by the length of the field in which the return value will be stored (with the exception of **chr**). The **setstr** function does not directly return a value.

Function	Return type
centre()	alpha
center()	alpha
chr()	alpha
left()	alpha
right()	alpha
setstr()	no returned value
tolower()	alpha
toupper()	alpha

NUMERIC FUNCTIONS

Function	Return type
asc()	i1 integer
chdir()	i1 integer
dim()	i2 integer
inchar()	i2 integer
power()	r8 real
rand()	i2 integer
remove()	i1 integer
strlen()	i1 integer
sqrt()	r8 real

SPECIAL FUNCTIONS

These functions are automatically maintained as part of the driving logic of report/batch programs.

Function	Return Type
total()	The type of the argument
min()	The type of the argument
max()	The type of the argument
count()	i4 integer

NOTES

- By default, up to 20 fields may be the target of the special functions **total()**, **min()** and **max()**. If your usage of these functions exceeds 20 fields, the compiler will issue the error "Too many functions". If this occurs, use the **-f=***nn* switch (where *nn* is the number of functions to allow) when compiling to increase the number of functions.

SYNTAX

asc(*text_expression*)

DESCRIPTION

Returns the ASCII value (in the range 0-255) of the first character in the text expression. If the string is empty, the value 0 is returned.

RETURN TYPE

This function returns an integer in the range 0-255 which may be safely stored in an i1 type.

SYNTAX

centre | **center** (*text_expression*)

DESCRIPTION

Both **centre** and **center** function identically.

This function centres the text expression by padding with spaces within the width of that expression and returns the centred text and all padding spaces.

RETURN TYPE

This function returns an alpha type.

EXAMPLE

```
!temp fx, ,a12

      fx = "TEST"
      print fx                    /* will print "TEST        "*/
      fx = centre(fx)
      print fx                    /* will print "    TEST    "*/
```

SYNTAX

chdir(*pathname* **)**

DESCRIPTION

Change the current working directory to that specified in *pathname*. The *pathname* may be a string constant or a string expression. If an error occurs changing directory an error code is returned, otherwise 0 is returned. Your program must assign the return value to a field.

The values which may be returned are:

0	No error occurred.
1	The directory given in *pathname* could not be found.
2	Access permission to read could not be gained for the named directory.

RETURN TYPE

This function returns an integer type in the range 0-2.

NOTES

- The directory change is valid until the program exits or until a further **chdir** is succesfully executed.
- On DOS systems, the current directory will remain valid even when the current program exits.

EXAMPLE

```
!temp x, ,i1
      x = chdir("/tmp/data")
      if x <> 0 then put "CHDIR failed - error ";x : exit
      put "CHDIR succeeded"
      ...
```

SYNTAX

chr(*numeric_expression* **)**

DESCRIPTION

Returns a single character which is the ASCII representation of *numeric expression*. The expression must be in the range 0-255.

RETURN TYPE

This function returns a single character (a1) string.

NOTES

- The use of characters outside the standard ASCII range 0-127 is non-portable as some systems do not support these characters.

EXAMPLE

```
!temp crlf, ,a2
!temp init, ,a10
     crlf = chr(13)+chr(10)
     init = chr(27)+'['+chr(11)+"INIT"+chr(7)+chr(90)
     put #5,init  /* send it to channel 5 */
     ...
```

SYNTAX

dim(*field* **)**

DESCRIPTION

Returns the number of elements in the subscripted field *field*. Returns 1 if the field is not subscripted.

RETURN TYPE

This function returns an integer in the range 0-32767 which can be safely stored in an i2 field.

NOTES

- Judicious use of this function in place of constants will greatly increase the ease with which programs and data structures can be modified.

EXAMPLE

```
for( ctr=1 ; ctr<=dim(fieldname) ; ctr=ctr+1 ) {
            printh fieldname[ctr]
}
```

getstr( ) Returns a sub-string from a string

SYNTAX

getstr(*source*, *pos*, *len* **)**

DESCRIPTION

Returns the sub-string of *source* which starts at position *pos* and is *len* characters in length. The first character in *source* is always position 1. If *pos* is less than 1 or greater than the length of *source*, a null string is returned. If *len* is greater than the number of characters remaining from *pos*, only those characters remaining are returned.

RETURN TYPE

This function returns an alpha type.

EXAMPLE

```
for(ctr=1; ctr<=strlen(str); ctr=ctr+1) {
        ch = getstr(str,ctr,1)
        checksum = checksum + asc(ch)
}
```

SYNTAX

inchar(*channel* **)**

DESCRIPTION

Returns the ASCII value (in the range 0-255) of the next character from *channel*. If *channel* is zero, standard input is used. If the *channel* is not zero it must have been opened with an **open #** command. This is a low level function that does not perform any special recognition of separator characters.

RETURN TYPE

This function returns an integer which may be stored in an i2 or i4 field.

NOTES

- The channel number must be in the range 0-32.
- If an error occurs (such as end of file) **inchar()** returns -1.
- All characters are input, including separator characters.

EXAMPLE

This example is complete and demonstrates writing a sequential file with data and reading that file using **inchar()**.

```
!temp a, ,i2
!temp ctr, ,i4
!init gosub TEST

              exit
TEST          open #1,"TESTING.TST" write
              put #1,"ABCDEFGHIJK","LMNOPQRST","UVWXYZ"
              close #1
              open #1,"TESTING.TST" read
              for(ctr=1;ctr<=30;ctr=ctr+1) {
                    a=inchar(1)
                    put a,chr(a),
              }
              close #1
              return
```

SYNTAX

instr(*string, startpos, pattern* **)**

DESCRIPTION

Searches for a *pattern* in *string* starting at character *startpos.* Returns 0 if the *pattern* was not found or an integer indicating the position in *string* (starting from 1) where the match was found.

RETURN TYPE

This function returns an integer in the range 0-255 which can be stored in an i1 type.

NOTES

- Case is significant.
- If the length of *pattern* is greater than the length of *string*, no match can be found and this function will always return 0.
- Trailing spaces are removed from *pattern* for the purposes of the search.

EXAMPLE

The following example reads a file and looks for the input word in any element of an a30(10) field.

```
!file ACCIDENT ../roads/data/witness
!temp ctr, ,i1
!temp mway, ,a3          /* expecting M25, M13, M1, M6 etc */
!input "Enter Motorway", mway
      for(ctr=1;ctr<=dim(w_account);ctr=ctr+1) {
            if (instr(w_account[ctr],1,mway)<>0) then \
                  gosub OUT
      }
      end
OUT   print w_name,w_date,w_mref,w_account[ctr]
      return
```

SYNTAX

left(*text_expression* **)**

DESCRIPTION

Left justifies a string within its field width, ie, removes any leading spaces.

RETURN TYPE

This function returns an alpha type.

EXAMPLE

```
!temp title,,a30
...
     get #1,title          /* read from input file     */
     title=left(title)     /* strip leading spaces     */
```

power( ) Raise a number to a power

SYNTAX

power(*number, exponent* **)**

DESCRIPTION

Returns *number* raised to *exponent*. Both *number* and *exponent* may be numeric expressions of any type.

RETURN TYPE

This function returns a real (r8) type.

EXAMPLE

```
!temp rr, ,r8
!temp ctr, ,i2
      for( ctr=1 ; ctr<=10 ; ctr=ctr+1 ) {
             rr=power(10,ctr)
             put rr
      }
```

SYNTAX

rand(*seed* **)**

DESCRIPTION

If *seed* is zero, this function returns a pseudo random number in the range 0 to 32767. If *seed* is non-zero, the generator will use *seed* to set the start point for all following number sequences and the first number in that sequence will be returned.

To ensure that the number sequence that is returned varies, the number generator must be "seeded" with a number that is fairly random at the start of the program. The **systime** special temporary field may be used for this purpose. All further calls to **rand()** should be made with a zero *seed* to get the next random number in the sequence.

RETURN TYPE

This function returns an i2 type.

NOTES

- As with all pseudo-random number generators, the number sequence is identical for any particular *seed*.
- The actual pseudo-random number sequence for a given seed may vary on different machines.
- To return a random number within a predefined range use the modulus function, ie ((rand(0) % 100) + 1) would return an integer between 1 and 100.

EXAMPLE

In the example below, note that the seeding of the random number generator is done outside the loop.

```
.... Example of use of rand() function

!temp mutate,,r8
!temp path,,r8(100)
!temp current,,i1
!define PL_CONST          4.1093
...
      mutate=rand(systime)
CALC_PATH
      mutate=(rand(0) / 100.5) + PL_CONST
      path[current+1]=path[current]+mutate
      current=current+1
      if current>=dim(path)then \
            gosub SHOWPATH : \
            mutate=rand(path[current]) : \
            current=1
      goto CALC_PATH
...
```

SYNTAX

remove(*filename* **)**

DESCRIPTION

Delete the named file from disk storage. The *filename* may be a full pathname or just the file name and must be a string constant or a text expression. If an error occurs removing the file an error code is returned, otherwise 0 is returned. Your program must assign the return value to a field.

The values which may be returned are:

0	No error occurred. The file was removed.
1	The file could not be found.
2	Write permission could not be gained for the file or path.

RETURN TYPE

This function returns an integer type in the range 0-2.

NOTES

- This function will remove a single file only. Wildcard expansion is not allowed. If multiple files need to be removed, use the **exec** command and an operating system command to perform the task.
- This function provides a portable mechanism to remove named files and is commonly used to delete temporary sequential files after processing.

EXAMPLE

```
!temp x, ,i1
      x = remove("/tmp/data/junk006.dat")
      if x<>0 then put "REMOVE failed - error ";x : exit
      display "REMOVE succeeded"
```

right( ) Returns a string right justified

SYNTAX

right(*text_expression* **)**

DESCRIPTION

Right justifies a string within its field width.

RETURN TYPE

This function returns an alpha type.

EXAMPLE

```
!temp num_title,,a20
...
      get #1,num_title            /* get field title   */
      num_title=right(num_title) /* right justify     */
      ...
```

SYNTAX

setstr(*dest, pos, len, source* **)**

DESCRIPTION

Place *len* characters from the string *source* into string *dest* over-writing the characters in *dest* starting at position *pos*.

This function differs from the others in that it does not directly return a value.

The string *dest* must be an alpha field, *source* must be an alpha field or a string constant and *pos* and *len* must be either numeric constants or expressions.

RETURN TYPE

This function does not directly return a value and it is an error to attempt to use this function as if it returned a value.

NOTES

- The first character in *dest* is position 1. If *pos* is less than 1 or greater than the length of *dest* then this function is ignored.
- The string *dest* is over-written with the characters from *source* for *len* characters or until the end of *source*, whichever occurs first.

EXAMPLE

```
!temp full, ,a7
!temp code, ,a4,+0000     /* force zero padded string      */
!temp ctr, ,i2
      full="STK"
      for( ctr=1; ctr<=10 ; ctr=ctr+1) {
            code=ctr
            setstr(full,4,4,code) /* STK0001, STK0002, etc */
            put full,
      }
```

strlen( ) Return the length of a string

SYNTAX

strlen(*string* **)**

DESCRIPTION

Returns the length of the characters in *string* with trailing spaces excluded.

RETURN TYPE

This function returns an integer in the range 0-255 which may be safely stored in an i1 field.

NOTES

- This function will return 0 if the *string* does not contain any data.

EXAMPLE

```
if strlen(name)=0 then message "Name is EMPTY!!"
len=strlen(name)
```

SYNTAX

sqrt(*numeric_expression*)

DESCRIPTION

Returns the square root of the *numeric expression*. Returns zero if the expression is zero or negative.

RETURN TYPE

This function returns a real (r8) value.

tolower( ) Return a lower case version of a string

SYNTAX

tolower(*text_expression* **)**

DESCRIPTION

Returns the text with all characters in the range "A" - "Z" converted to lower case.

RETURN TYPE

This function returns an alpha type.

EXAMPLE

```
!temp a, ,a20
      a="This is a MIXTURE 12"
      a=tolower(a)
      put a                          /* "this is a mixture 12" */
```

SYNTAX

toupper(*text_expression* **)**

DESCRIPTION

Returns the text with all characters in the range "a" - "z" converted to upper case.

RETURN TYPE

This function returns an alpha type.

```
!temp a, ,a20
     a="This is a MIXTURE 12"
     a=toupper(a)
     put a                /* "THIS IS A MIXTURE 12" */
```

SYNTAX

!cfile [*file_number*] *pathname*

DESCRIPTION

Declares a Sculptor keyed file which is closed when the program starts. See **!file** to declare files which are initially open. The *file_number* is used as an identifier to refer to the file in subsequent file access commands.

The index and data files must exist when the program is run (see the program **newkf** for creating new files) and its descriptor file must exist when the program is compiled (see the program **describe**). **sagerep** will look for the file in the current working directory unless a pathname is supplied. Both the data file and its associated index file (**.k** extension) must exist in the same directory.

It is important to note that **sagerep** temporarily opens each **!cfile** when it loads the program. If it has already opened the maximum number of files permitted by the operating system then the program will abort. For this reason, **!cfile** declarations should precede **!xfile** declarations.

The maximum number of files that may be declared in one program using **!cfile**, **!xfile, !read** and **!file** is 32.

NOTES

- The **open** command is used to open files which are initially closed. If the number of open files would exceed the limit set by the operating system, the **open** will fail and the program will terminate with a "Cannot open ..." error.
- The **SAGEDATA** environment variable, if declared, is prepended, at run-time, to the name of any file declared within a report/batch language program.

EXAMPLE

```
!cfile 3 control
```

!constant Declare a temporary field and its initial value

SYNTAX

!constant *name*, [*heading*], *type&size* [, *format*] = *expression*

DESCRIPTION

Declares a temporary field and its initial value. The expression is calculated once only as part of the initialisation procedure and is useful for setting up keys for cross-reference files (see **!xfile** and **!read**). In other respects a **constant** is the same as a **temp** and its value may be altered later by direct assignment.

Special care must be taken with date constants, since the expression may be either numeric, yielding an absolute day number, or a formatted date. Mathematical symbols such as "/" and "-" are taken for their normal meaning and not as separators for day, month and year. For this reason, a comma should be used as a date separator.

See **!temp** for further details.

EXAMPLE

```
!constant ckey,,a1 = "A"
!constant status,,i1,## = 1
!constant sdate,Start Date,d4=1,1,80
!constant edate,End Date,d4=date+28
```

SYNTAX

!depth *lines*

DESCRIPTION

Defines the page length, in lines, that will be used to determine at what point **sagerep** will issue headings and footnotes. The depth is also used when calculating the number of lines left on a page (see the **lines_left** special temporary field).

It is generally preferable not to include a **!depth** statement unless it is really required, since it defines, within the program, the depth of the paper in the printer, over-riding the paper depth setting in the printer parameter file.

If a **!depth** statement is not used, the **Standard Page Depth** value set in the printer parameter file is used, allowing the report to print correctly on different paper sizes.

The declaration is useful in cases where the depth of the report is fixed (e.g. pre-printed payslips) since it allows testing on ordinary paper without adversely affecting other users of that printer.

NOTES

- Use of **!depth** will over-ride the depth read from the printer parameter file.

EXAMPLE

```
!depth 28
```

!display Display a message on the screen

SYNTAX

!display *text_expression*

DESCRIPTION

Display a message on the screen when the program starts as part of the initialisation sequence. The text is displayed on the screen even if standard output has been redirected or piped elsewhere.

NOTES

- If the standard error channel has also been redirected, the text will appear there.

EXAMPLE

```
!display "STOCK VALUATION REPORT"
!display "======================"
!display " "
!display "Please enter the report parameters below"
!display " "
!input "Warehouse Code: ",wh_code
...
```

SYNTAX

!exclude if *conditional_expression*

DESCRIPTION

Excludes records for which the specified condition is true. Refer also to **!select**. Any number of **!exclude** declarations are permitted.

If the program contains no selection or exclusion conditions then all records are selected.

If the program contains exclusion conditions only (no selection conditions) then records are selected *unless* one of the exclusion conditions is true.

If the program contains selection conditions only (no exclusion conditions) then records are selected only if one of the selection conditions is true.

If the program contains both selection and exclusion conditions then the conditions are tested in the order defined and the first one found to be true determines whether the record is selected or excluded. If none are found to be true then the record is excluded.

NOTES

- The order of definition of select and exclude declarations will determine the records that are selected from the driving file.
- The special functions **total()**, **min()**, **max()** and **count()** operate ONLY on those records which are selected, NOT on all records in the driving file. See **!temp** for details on how to perform calculations for all records in the driving file independent of selection criterea.

EXAMPLE

```
!exclude if stklev = 0 or cat <> "A"
!exclude if w_hse < "M"
!select if stk_code > "LET"
```

SYNTAX

!file [**1**] *pathname*

DESCRIPTION

Declares a Sculptor keyed file which is open when the program starts. See **!cfile** for declaring files which are initially closed. The file number must be 1 and is used to refer to the file in subsequent file access commands. All subsequent **!xfile** or **!cfile** commands must be allocated file numbers in ascending sequence.

The file must exist when the program is run (see the program **newkf** for creating new files) and its descriptor file must exist when the program is compiled (see the program **describe**). **sagerep** will look for the file in the current working directory unless a pathname is supplied. Both the data file and its associated index file (**.k** extension) must exist in the same directory.

If there are no commands in the program which can update the file and if the record locking mechanism of the operating system permits, then the file is opened in read-only mode, otherwise it is opened in update mode.

The maximum number of files that may be declared in one program using **!file**, **!cfile**, **!read** and **!xfile** is 32. The maximum number of files that may be open at the same time depends on the operating system and the way it has been configured (see the **!handles** declaration if you are using DOS).

NOTES

- Each normal Sculptor keyed file requires two operating system files - one for the data file and another for the index file. A Sculptor index only file (created using **newkf -i**) requires only one operating system file.

- If files are used in the program but a driving file is not required, one file should still be declared as the driving file using **!file**. The driving logic may then be bypassed using **!init.**

EXAMPLE

```
!file 1 stock
!xfile 2 /usr/john/income_tax
```

Declare statements to be executed at the end of the program

!final

SYNTAX

!final *statement* [: *statement*] ...

DESCRIPTION

!final statements are executed once only at the end of the report after processing of the driving file is complete or an exit comand is encountered. Any valid **sagerep** command, except **goto**, may be used in **!final** declaration.

Any number of **!final** declarations are permitted and are executed in order of their definition.

EXAMPLE

```
!final print: print "END OF REPORT";#tf
!final gosub CALCSTATS
!final gosub WRITESTATS
```

!footnote Declare a footnote statement

SYNTAX

!footnote *statement* [: *statement*] ...

DESCRIPTION

Footnote statements are printed at the bottom of each page. Any valid **sagerep** command, except **goto**, may be used in a footnote statement.

Any number **!footnote** declarations are permitted and are executed in order of definition.

NOTES

- The compiler has to be able to count the number of print commands included in **!footnote** statements so that the number of lines to be reserved at the bottom of each page is known. Since the compiler cannot follow a **gosub**, avoid putting **print** commands in a subroutine that is called from a **!footnote**.
- Since footnote statements can be invoked at any time, care should be taken not to cause unintentional changes. For example, it is normally wise to ensure that **scrline** is not altered.
- When the **sagerep** program has completed processing the driving file, or if an **exit** command is encountered, the footnote statements will also be executed.

EXAMPLE

```
!footnote print: print tab(70); "Page: ";pageno
!footnote if flag = 1 then\
            print "Continued on next page"
!footnote print Records so far : ";count()
```

SYNTAX

!gap *integer_constant*

DESCRIPTION

Declares the number of spaces to be left between print items that are separated by commas (see the **print** command). If the program does not include a **!gap** statement, a default of two spaces is set.

EXAMPLE

```
!gap 5
...
     print "AAAA","BBBB"
```

Output :

```
AAAA     BBBB
```

SYNTAX

!handles *number*

DESCRIPTION

Effective on DOS version 3.3 upwards. On earlier versions of DOS and on other systems, this declaration is ignored.

Sets the number of file handles available to the current program to *number* and thus defines the maximum number of files which the program can have open. The default number of handles per program is 20. Requesting more handles causes the program to use more memory. Requesting less than 20 handles does not save memory. To calculate the number of handles that a program needs, the following must be taken into account:

1. Four handles are required for the standard channels STDIN, STDOUT, STDERR and STDPRN. (The STDAUX channel is closed by **sage** on startup.)
2. Each open Sculptor keyed file requires two handles (one for the data file and one for the index file). An index only Sculptor file requires only one handle.
3. Each open sequential file requires one handle.

The total number of handles available in the system is set by the FILES= entry in CONFIG.SYS. This must be set high enough to support all loaded programs. For example, if a program sets **!handles** to 30 and then executes a child task which sets **!handles** to 40, the FILES= entry in CONFIG.SYS must be at least 70.

The maximum value for FILES is 99 in DOS versions 2.x and 255 in DOS versions 3.x. Prior to DOS 3.3, the maximum number of handles available to any one program is 20.

NOTES

- This declaration may be used to allow a program to open up to 32 Sculptor keyed files simultaneously. The limit of 32 is set by Sculptor. The default number of handles (20) effectively limits a program to eight Sculptor keyed files.
- If there are insufficient handles left in the system, the **!handles** declaration allocates as many as it can.
- If an **open** command fails due to insufficient handles, the program will abort with operating system error number 4.

EXAMPLE

```
/* This program opens:
     16 Sculptor keyed files (32 handles)
     5 Sculptor index only files (5 handles)
     4 sequential files (4 handles)
*/
!handles 45      /* allowing for STDIN, etc. */
```

heading

Declare statements to be executed at the start of each page

SYNTAX

!heading *statement* [: *statement*] ...

DESCRIPTION

Heading statements are executed at the start of each new page. Any valid **sagerep** command, except **goto**, may be used in a heading statement.

Any number of **!heading** declarations are permitted and are executed in the order of their definition.

Since heading statements can be invoked at any time, care should be taken not to cause unintentional changes. For example, it is normally wise to ensure that **scrline** is not altered.

EXAMPLE

```
!heading printh *p_sur,[p_fname],*p_tel
!heading print "STOCK REPORT":\
     if pageno > 1 then print " (continued)"
!heading gosub STARTPAGE: flag = 1
```

SYNTAX

!init *statement* [**:** *statement*] ...

DESCRIPTION

Declares initialisation statements which will be executed once only at the beginning of the report before any records are read from the driving file. Any valid **sagerep** command, except **goto**, may be used in an initialisation statement.

Any number of **!init** declarations are permitted and will be executed in the order of their definition.

NOTES

- Placing the main body of the program code in a subroutine called from an **!init** statement will effectively bypass the **sagerep** driving logic. If driving logic bypass is intended, the subroutine should exit without returning.

EXAMPLE

```
!init scroll 4: firstno = arg
!init scroll 5: lastno = arg
!startrec key=firstno
!endrec key=lastno
```

This example demonstrates driving logic bypass.

```
!init gosub PROCESS : exit
PROCESS     next 1 nsr=DONE
            if wf_code="A" then wf_code="B" : write 1
            if wf_type="T" then read 1 key=wf_newkey
            goto PROCESS
DONE        return
```

In the example above, the driving logic could have performed the task more efficiently, but there may be occasions when it is unsuitable for the task in hand.

SYNTAX

!input "*prompt text*", *field_name*

DESCRIPTION

Input a value into a temporary field as part of the program initialisation sequence. The prompt text is displayed on the screen with a question mark appended. The reply is validated for correct data type and stored in the designated field. Note that **sagerep** does not check validation lists. If the reply is not valid for the data type, the bell is sounded and the prompt is repeated.

Responses to **!input** may be placed in a text file by redirecting standard input. On operating systems that allow **sagerep** to detect this situation, the prompt text is suppressed and an invalid reply aborts the program.

The **!input** statement honours the **u** (upper)and **l** (lower) formats on alphanumeric fields and folds the input to upper/lower case. However, since **sagerep** does not work in single character input mode, the characters are still echoed back as typed.

NOTES

- The **sagerep** program does not use the vdu parameter file, or single character input modes, so when an **input** command or declaration is used, a line is read from standard input and the value read is stored in the field (after validation). If input is to an a3 field, the user could type a complete line of text, but only the first three characters would be stored.
- The **get** command also accepts input from channel 0 (standard input).

EXAMPLE

```
!temp stdate,Start Date,d4
!input "Start date",stdate
```

Declare a statement to be executed on ending a field value — !on ending

SYNTAX

!on ending *field_name statement* [: *statement*] ...

DESCRIPTION

Declares a statement to be executed on ending a value in the specified field. **sagerep** checks the field's value after selecting the next record from the driving file, reading automatic cross-references (see **!xfile**) and recalculating all automatic temps (see **!temp**). If a change has occurred then **sagerep** reinstates the previous record and executes the statement(s) before continuing with the next record.

The control field can be a field on the driving file, a field on an automatic cross-reference file, or an automatically recalculated temporary field.

The functions **total()**, **min()**, **max()** and **count()** when used in an **!on ending** statement are taken to refer only to the ending block of records and are therefore particularly useful for printing sub-totals and other block analysis figures. Note, however, that this does not apply if the function is in a subroutine called from an **!on ending** statement. In this case the functions return values computed from the start of the report.

Any valid **sagerep** command, except **goto**, may be used in an **!on ending** statement and any number of **!on ending** declarations are permitted and are executed in order of definition.

EXAMPLE

```
!on ending s_cat print tab(40);total(costval)
!on ending ordno gosub SHOW_TOTALS: newpage
```

!on starting — Declare a statement to be executed on starting a field value

SYNTAX

!on starting *field_name statement* [**:** *statement*] ...

DESCRIPTION

Declares a statement to be executed on starting a new value in the specified field. **sagerep** checks the field's value after selecting the next record from the driving file, reading automatic cross-references (see **!xfile**) and recalculating automatic temps (see **!temp**). If a change has occurred then **sagerep** executes the declared statements.

The control field can be a field on the driving file, a field on an automatic cross-reference file, or an automatically recalculated temporary field.

Any valid **sagerep** command, except **goto**, may be used in an **!on starting** statement and any number of **!on starting** declarations are permitted and are executed in order of definition.

EXAMPLE

```
!on starting s_cat newpage
!on starting initial print: print
!on starting t_ordno gosub START_ORDER
```

Declare a cross-reference file and initially read a record

SYNTAX

!read [*file_number*] *pathname* **key=** *field_list*

DESCRIPTION

Declares a cross-reference file and creates an initialisation statement to read the record whose key exactly matches the key defined in the **key=** clause. The values required in the key fields must have been established in earlier initialisation statements.

The file must exist when the program is run (see the program **newkf** for creating new files) and its descriptor file must exist when the program is compiled (see the program **describe**). **sagerep** will look for the file in the current working directory unless a pathname is supplied. Both the data file and its associated index file (**.k** extension) must exist in the same directory. If there are no commands in the program which can update the file, then it will be opened in read-only mode, otherwise it will be opened in update mode.

Any explicit commands in the program which access the file must use the file number. Cross-reference file numbering starts at 2 and the numbers must be allocated in ascending sequence.

EXAMPLES

```
!constant ckey,,a1 = "A"
!read control key=ckey
!temp mth,Month,i1,#
!constant ctype,a1 = "S"
!input "Which month",mth
!read /usr/sales/control key=ctype,mth
```

!record Declare an alternative record layout

SYNTAX

!record *file_number pathname*

DESCRIPTION

This command is used to declare an alternative record layout for the file referred to by the *file_number* (previously declared in a **!file, !xfile, !read** or **!cfile** statement). The *pathname* identifies an alternative descriptor(.d) file created with the program **describe**. Both sets of fieldnames may be referred to in subsequent program statements and they will both use the same record buffer. Up to eight **!record** statements may be associated with each declared file.

The use of alternative record layouts allows variable record types on a single file. The usual mechanism for an alternate record is to have the same key structure as the main record and for the key to include a record type field. The program may then use the appropriate set of fields depending on the record type. Alternative record layouts are useful not only for completely different record types contained in the same file, but also for redefining the structure of individual fields to enable access to their component parts.

Alternative record layouts must have the same key length as the main record and it is strongly recommended that the key structure is also identical to avoid ambiguity. If a different key structure is used, do not use the **!record** key fields in a **key=** clause, since **sagerep** will build the key assuming that these fields are supplying values for the main key fields. Instead, assign values to the alternative key fields, which overlay the main key fields in the record buffer, and omit the **key=** clause.

NOTES

- Using an alternate record layout does not require any additional files at run-time. The descriptor (.d) file must be present when the program is compiled.

- Accessing the component parts of numeric fields is permitted, but may not return the values expected due to differences in byte ordering between your system and the **Sculptor** standard.
- For a detailed description of alternate record layouts see the Screen Form Language, page 7-48.

SYNTAX

!select if *conditional_expression*

DESCRIPTION

Generates a control statement to select records for which the specified condition is true. Refer also to **!exclude**. Any number of **!select** declarations are permitted.

If the program contains no selection or exclusion conditions then all records are selected. If the program contains selection conditions only (no exclusion conditions) then records are selected only if one of the selection conditions is true.

If the program contains exclusion conditions only (no selection conditions) then records are selected unless one of the exclusion conditions is true.

If the program contains both selection and exclusion conditions then the conditions are tested in the order defined and the first one found to be true determines whether or not the record is selected. If none are found to be true then the record is excluded.

EXAMPLE

```
!select if acc_date >= stdate
!select if cat="B" and st_code ct "TT"
```

SYNTAX

!startrec key = *field_list*

DESCRIPTION

Defines the record on the driving file at which the report is to start. The start key value is established during the initialisation process, so the field values required must be assigned in earlier initialisation statements. The **!init** statement is the most flexible method of assigning values during the initialisation phase, but the **!constant**, **!read** and **!input** statements may also be used.

If no record having the supplied key exists on the driving file then the report starts at the record with the next higher key.

EXAMPLE

```
!read control key=ttyno
!startrec key=c_sdate
```

SEE ALSO

!endrec

!temp Declare a temporary field

SYNTAX

!temp *name*, [*heading*], *type&size* [(*dim*)], [*format*]
[= *expression*]

DESCRIPTION

Declares a temporary field for use within the program. *name* may be any valid field name. The *heading* is optional and will be the heading that is used by default if the field heading is used on the report. The *type&size* are any valid Sculptor data type and may be dimensioned as an array field. *format* may be any valid format string (see page 3-5 for details).

A temporary field may be subscripted, in which case the element accessed can either be determined by the current value of the special temporary field **scrline**, or by using standard subscripts. If either **scrline** or the subscript exceeds the field's dimension then a wrap around takes effect.

Once defined, temporary fields may be treated in the program in the same way as record fields.

If the optional *expression* is given, the expression is re-evaluated after reading each record in the driving file, but before performing any **!select** or **!exclude** declarations. This feature can be used to sum or count all fields from the driving file, in order to perform averages and comparisons to those actually selected. The example below demonstrates one **!temp** which counts all records in the main file and another which sums a particular field.

```
!temp m_count, ,i4=m_count+1
!temp m_amsum, ,m8 = m_amsum+f_amount
```

NOTES

- All !temp expressions are evaluated in the order of their declaration.
- No temporary field may exceed 32,767 bytes in size.
- The scope of a temporary field is the entire program.

- A **!temp** declaration must appear in the program before it is used in other statements.
- Field headings and formats that contain punctuation characters should be enclosed in quotes.

Special temporary fields

The following special temporary fields are available. These are automatically declared and cannot be redeclared.

!temp arg, ,a0

Accesses the command line arguments. The values in **arg** cannot be altered. Reference to a non-existent value returns an empty string.

Command line : `sagerep cust pvdu ABC`

```
var1 = arg[1]                  /*  "sagerep"       */
var2 = arg[2]                  /*  "cust"          */
scroll 3 : var3 = arg          /*  "pvdu"          */
var4 = arg[4]                  /*  "ABC"           */
```

!temp date, ,d4

The system date. If this field is directly assigned, automatic updating from the system date will cease for the duration of the program.

!temp errno, ,i2

Error number returned from last **err=** trap. The file **errors.h** located in the **Sculptor** include directory contains manifest constant definitions of the errors which may be encountered and may be included (using **!include**) in your program. The errors are listed below.

No	meaning	manifest constant	command
1	bad channel - range is 0-32	BAD_CHANNEL	get #, put #, open #, close #
2	in use - file is already open	IN_USE	open #
3	bad name - not a string	BAD_NAME	open #
4	file cannot be accessed	NO_FILE	open #
5	attempt to open chan 0	NO_ZERO	open #
6	no permission	NO_PERMS	open #
7	too many files open	TOO_MANY	open #

8	read past the end of file	FEOF	get #
9	read error on input	FGETERR	get #
10	file not open for writing or disk full	FPUTERR	put#

!temp lines_left, ,i2

Contains the number of lines left on the current page. The page length is determined by a **!depth** declaration if present, otherwise from the **Standard Page Depth** entry in the printer parameter file. The **lines_left** field is used by **sagerep** to determine whether to honour a **keep** statement and when to issue the **footnote** statements.

!temp pageno, ,i2

Stores the current page number. This number is incremented every time the driving logic outputs a new page or every time a **newpage** command is issued. The **#tf** print item will issue a top of form character but will not increment the page counter.

!temp printer, ,a12

The name of the printer currently in use. This name is read from the printer parameter file when the program starts. It could be used for information purposes or to configure specialist reports to particular printers, giving more control than is directly available through the printer parameter file.

!temp scrline, ,i2

The current scroll line number. Set by the **scroll** command and cannot be altered by direct assignment.

!temp separator, ,a1

Separator character for use with **get** and **put** commands. This field may be directly assigned. It may not be redefined.

NOTE: The **get** and **put** commands always use an end of line character to denote the end of a record. The **separator** field defines the field separator only.

!temp systime, ,i4

The system time in seconds. The base value of **systime** is arbitrary and it should, therefore, only be used to calculate time differences or to seed the random number generator.

!temp task, ,a5

The current task number. Blank if the system is not multi-tasking.

!temp time, ,m4

The current time from the system. An **m4** type is used so that the time may be formatted as hours.minutes. If calculations are required on time values, the hour and minute portions must be separated and the calculations performed accordingly.

EXAMPLE

```
hour=time/100 : min=time%100
min=min+30 : if min>59 then hour=hour+1 : min=min-60
if hour>24 then hour=0
newtime=(hour*100)+min
```

!temp tstat, ,i1

Child task termination status. The return value may be defined in a screen form or report program with the **exit** command.

!temp ttyno, ,i2

The terminal number. On Unix® systems, this is a number based on the major and minor device numbers. Its purpose is to provide a unique identifier for each terminal.

!temp user, ,a9

The user log on name. Blank on single-user systems.

!temp userid, ,i2

The user identification number for this user. Zero on single-user systems.

!temp day, ,i1

The day field used in encdate/decdate commands. May be directly assigned.

!temp month, ,i1

The month field used in encdate/decdate commands. May be directly assigned.

!temp year, ,i2

The year field used in encdate/decdate commands. May be directly assigned.

SYNTAX

!width *numeric_constant*

DESCRIPTION

Declares the number of columns required for the report. The run-time interpreter, **sagerep**, uses this value to determine the initial character compression to set on the printer.

If there is no **!width** statement in the program, the compiler estimates the required output width based on the largest line required by a single **print** or **printh** statement.

If there is no width in the printer parameter file which exactly matches the one required, the next higher width is set, if any. If there is no higher width, the largest width is set.

The **width** command (see page 120) may be used within the program to set any other width which the printer supports, as defined in the printer parameter file.

EXAMPLE

```
!width 96
```

SYNTAX

!xfile [*file_number*] *pathname* [**key=** *field_list*]

DESCRIPTION

Declares a cross reference file which is initially open. See **!cfile** (page 8-37) for details on declaring a file which is initially closed.

The file must exist when the program is run and its descriptor file must exist when the program is compiled. The *pathname* may be any valid file or path name. Both the data file and its associated index file must exist in the same directory.

If there are no commands in the program which update the file, it is opened in read-only mode, otherwise it is opened in update mode.

Any explicit commands in the program which access the file must use the file number. Cross reference file numbering starts at 2 and the numbers must be allocated in ascending sequence.

If a **key=**clause is included, the cross reference file is automatically read after reading each record from the driving file. The fields in the *field_list* may be temps, fields from the driving file, or fields from a preceding cross-reference file as required. If the record is not present on the cross-reference file, a blank record is returned.

A maximum of 32 files may be declared with **!file, !read, !cfile** or **!xfile** declarations.

EXAMPLE

This example produces a report from a stock file which contains a field `stk_wh`. This field is the key value for the file `ware`. The correct record in `ware` will be looked up automatically for every record in `stock`.

```
!file 1 stk                          /* stock file */
!xfile 2 ware key=stk_wh             /* warehouse names  */
!heading printh *stk_code, *stk_qty, *wh_name
     print stk_code, stk_qty, wh_name
```

SYNTAX

!zeros on | off

DESCRIPTION

When a field with a zero value is printed using **print** or **printh**, it is normally shown containing zero even if the format for the field does not contain any explicit zeros. This declaration allows zero value numeric fields to be printed as blanks unless explicitly formatted to show as zero.

The default is **zeros on**. Setting **zeros off** will *only* suppress the printing of zeros if the field format does not explicitly force their use.

The compiler, **cr**, or the pre-processor, **spp**, may also be used to force suppression of zeros through the use of the **-z** option. The **!zeros** declaration will always over-ride any command line compilation switch.

EXAMPLE

```
!file 1 stock
!zeros off
!heading printh *stk_code,*stklev1,*stklev2,*stklev3,*stktot
     printh stk_code,stklev1,stklev2,stklev3,stktot
```

This example will show:

```
Stock Code  Level 1      Level 2      Level 3      Total
STK0001           13           15                    28
STK0002                        18           20       38
STK0003           12           12           12       36
...
```

SYNTAX

abort [*numeric_expression*]

DESCRIPTION

Immediately terminates the program and returns control to the calling task. The optional *numeric_expression* may be used to pass back a termination code. If the termination code is omitted a default of zero is returned (except on VMS where 1 is returned by default).

abort is similar to **exit** except that it skips the execution of any **!final** statements.

EXAMPLE

```
abort 1
abort
```

chain — Chain a new program

SYNTAX

chain *text_expression*

DESCRIPTION

Terminates **sagerep** and replaces it with a new program. The text expression may be a string constant, an alphanumeric field or a concatenation of several such items and specifies the program to be called and its arguments. When the called program exits, return is direct to the parent of the current process.

WARNING: The **chain** command is available only if supported by the operating system in use and is not guaranteed to operate in an identical way on all systems that do support it.

The **chain** statement does NOT call a new shell (command processor) to process the specified command. It merely replaces the current process in memory. A shell is required, however, if the command line involves I/O redirection, pipes or shell expansion and in this case the new shell or command processor must be explicitly chained, with the command to execute passed to it as a parameter.

EXAMPLE

```
chain "sage ordlines " + ordno
chain "menu main"

. following called with a  shell as redirection required
chain "sh -c sagerep calc pvdu >/dev/tty"
chain "command /C sagerep calc pvdu >com1"
```

SEE ALSO

exec

SYNTAX

clearbuf *file_number*

DESCRIPTION

The specifed file buffer is cleared and the currently selected record, if any, is unlocked. The buffer is initialised according to the main record layout defined by a **!file**, **!cfile, !read** or **!xfile** declaration, alphanumeric fields being set to spaces and other fields to zero.

NOTES

- When using alternate record layouts, ensure that any alternate record fields have been initialised as required.

EXAMPLE

```
clearbuf 3
```

close — Close a keyed file

SYNTAX

close *file_number*

DESCRIPTION

Closes the specified file and unlocks the current record. The content of the file's record buffer remains unaltered.

If the file is later reopened, the file position is unchanged but any selected record has been unlocked, so a write will not be permitted unless a record is first read.

NOTES

- The **clearbuf** command will still operate on a closed file's record buffer.
- An attempt to close a file that is already closed is ignored.

EXAMPLE

```
close 1 : close 2 : close 3
```

SYNTAX

close # *channel*

DESCRIPTION

Closes the sequential file which was opened on *channel* using the **open #** command.

channel is a number in the range 1-32 which corresponds to the channel number used when the file was opened. It may be a constant, field name or expression.

NOTES

- Channel 0 (zero) is the standard I/O channel and cannot be closed.
- It is not an error to close a channel which is already closed. In this case the command is simply ignored.

EXAMPLE

```
close #1
close #barcode
```

decdate Decode a date

SYNTAX

decdate *expression*

DESCRIPTION

Decodes a date field into day, month and year components. The expression must yield a valid day number in the range 0 to 3,652,059 and will normally be a simple date field, but may be any integer expression.

The decoded values are placed in the predefined special temporary fields **day**, **month** and **year**.

EXAMPLE

```
decdate datedue
tempmth = month
decdate date
if tempmth = month then message "Delivery this month"
```

This example demonstrates taking a date and producing a string like "Sun Oct 15 1989" from it. This example, as it stands, also provides the day and month components separately.

```
!temp monthtext, ,a36     /* storage for month lookup data */
!temp daytext, ,a21       /* storage for day lookup data */
!temp tdate,Test Date,d4
!temp final, ,a15         /* output string */
!temp dow,Day of week,a3  /* day of week - Mon, Tue, etc */
!temp mon,Month,a3        /* month - Jan, Feb, etc  */
!temp dt,Day text,a2,+00  /* zero padded day number as text */
!temp yt,Year text,a4     /* year as text   */
      monthtext="JanFebMarAprMayJunJulAugSepOctNovDec"
      daytext="SunMonTueWedThuFriSat"
            tdate = "15/10/89"
            decdate tdate
            dow=getstr(daytext,(((tdate%7)+1)*3)-2,3)
            mon=getstr(monthtext,(month*3)-2,3)
            dt=day : yt=year
            final=dow+" "+mon+" "+dt+" "+yt
            print final
```

SYNTAX

delete *file_number*

DESCRIPTION

Deletes the currently selected record from the specified file. If no record is currently selected, then the command is ignored.

A record is selected when it is read from the file, either explicitly or by an automatic read. A record is no longer selected after it has been written back, deleted, unlocked or an attempt has been made to read a new record.

NOTES

- Deleted records do not return space on the file back to the operating system, rather the space is marked for re-use on subsequent insertions.
- A **kfcopy** operation may be performed to return space to the operating system if there has been a substantial and permanent reduction in the number of records.
- The number of deleted records on a file may be determined using the **kfcheck** utility with the **-d** switch.
- Deletion is permanent and is carried out immediately. There is no way to recover a deleted record.

EXAMPLE

```
delete 3
```

display — Display text on the vdu

SYNTAX

display *text_expression*

DESCRIPTION

Displays the text expression on the screen. The text is always output to the standard error output channel and therefore will be displayed on the screen even if output is redirected elsewhere (e.g, to the printer).

If standard input has been redirected and the operating system permits **sagerep** to detect this, the display is suppressed.

Use of the **put** command to channel 0 also outputs to the screen, but only if output is not redirected. When output is redirected, **put** places its output as redirected.

EXAMPLE

```
display "Scanning serial file ..."
```

Encode a date from the temps day, month and year

SYNTAX

encdate *date_field*

DESCRIPTION

Encodes the current values in the special temps **day**, **month** and **year** into a day number and stores the result in the designated date field. The temps **day**, **month** and **year** are predefined and need not be declared (see **!temp**).

NOTES

- If the date to be encoded is not valid, the designated field is set to zero.

EXAMPLE

```
. encoding the last day of the current year

    decdate date
    day = 31: month = 12
    encdate eoy
```

SYNTAX

end

DESCRIPTION

Terminates the processing of the current set of statements and moves on to the next set as determined by the driving logic. For example, if **sagerep** encounters an **end** while processing an **!on ending** statement, no further **!on ending** statements are checked for the current set of records and **sagerep** continues with the **!on starting** statements.

If the main block of statements is followed by subroutines, the main statements must terminate with an **end** to avoid execution falling through to the first subroutine.

NOTE

- An **end** statement clears all pending subroutine **return**s.

EXAMPLE

This example is complete and represents one of the simplest forms that a sagerep program can take. It declares a file and prints all the records from it, with page headings as required.

```
!file 1 service
!heading gosub HEADING
     printh s_code,s_name,s_amount,s_rol
     end
HEADING
     print : print
     print "FULL REPORT",,date,time,,,"Page:",pageno
     print : print
     printh *s_code,*s_name,*s_amount,*s_rol
     print
     return
```

SYNTAX

exec *text_expression*

DESCRIPTION

Executes the text expression as a system command line. The expression may be a string constant, an alphanumeric field or a concatenation of several such items using the + and / operators. When the child task completes, control is returned to the statement following the **exec**. The special temporary field **tstat** contains the child tasks' termination code.

WARNING: The **exec** command is available only if supported by the operating system in use and is not guaranteed to operate in an identical way on all systems that do support it.

The **exec** statement normally calls a new shell (command processor) to process the specified command. However, if the command is a simple program call (with or without arguments), then a shell is not required. This can be indicated to **Sculptor** by preceding the command with a - as in the example below. A shell is required if the command involves I/O redirection, pipes, shell expansion or multiple commands.

NOTES

- When constructing command line parameters, please ensure that quotes are used to surround parameters that may contain spaces, otherwise these spaces will be taken as parameter delimiters. (see example below.)
- If **exec** is used to call the Sculptor program **newkf** in order to re-initialise files used in the program, these files should be closed before the **exec** and re-opened afterwards. Failure to do so can cause file corruption on some operating systems.

EXAMPLE

```
exec "kfcheck *.k"
exec "sagerep printinv " + ptr + "|" + spooler
exec "-/bin/sage stock"          /* shell not used */
```

In the example below care is taken to ensure that the date passed on the command line is quote delimited as the date itself may contain spaces.

```
!temp txt,Text of date,d4,d+"dd/mm/yyyy"
...
      txt=date
      exec "sagerep putdate pvdu "+'"'+txt+'"'
      if tstat<>0 then put "PUTDATE FAILED" : prompt : exit
```

Date fields are, however, best passed as day numbers by assigning to an i4 field and then to an alpha field as shown below.

```
!temp dayno,,i4
!temp txt,,a7

      dayno=date
      txt=dayno
      exec "sagerep putdate pvdu " + txt
```

SYNTAX

exit [*numeric_expression*]

DESCRIPTION

Terminates the program and returns control to the parent task. The optional numeric expression may be used to pass back a termination code (the default is zero on most systems and one on VMS).

NOTES

- **!final** statements are still executed after the **exit** command is given. The **abort** command may be used if this is not required.
- The termination code of a child process is contained in the **tstat** special temp.
- If a shell (or command processor) is used to execute a child process, the value returned will be that returned by the shell. Normally, this is the value returned from the last child task the shell executed, but some operating systems do not pass this termination code on to the calling program.
- If a non-zero termination code is returned to **menu**, the code will be displayed and a keypress will be required to return to the menu.
- The **sagerep** program will exit automatically with a termination code of zero when all records in the driving file have been processed.
- If an operating system error occurs within the program the error is displayed and the program exits with the code as its exit status.

EXAMPLE

```
exit errctr
```

SYNTAX

find *file_number* [**key** = *field_list*] [**nsr** = *label*] [**riu** = *label*]

DESCRIPTION

Unlocks any existing locked record on the file then searches for the first record on the file whose key matches the supplied key. The file number may refer to the driving file or a cross-reference file, but note that a find on the driving file may alter the file position and the report sequence.

If the file is open for update, and if the record is currently locked by another user, **sagerep** waits until it becomes free. This condition may be trapped with the **riu=** clause.

If no matching key is found then a blank record is returned. This condition may be trapped with the **nsr=** clause, in which case control passes to the label indicated.

If the **key=** clause is omitted, the key data in the record buffer is used as the key. If the **key=** clause is present, a key is constructed using data from the named fields.

The **find** command differs from **read** by not requiring an exact key. The rules are:

1. If the natural key field is alphanumeric, then trailing spaces in the supplied data are ignored and only the leading characters must match the corresponding characters in that key field, e.g. if "Smith" is supplied then "Smithson" will match but "Smythe" will not.
2. If the natural key field is numeric (including dates) and the supplied data is non-zero, then that key field must match exactly.
3. If the natural key field is numeric (including dates) and the supplied data is zero, then any value in that key field matches.

NOTES

- Further details may be found in the Screen Form Language chapter on page 7-89.

EXAMPLE

```
!init gosub LOOKUP
              ...
LOOKUP        find 2 key=c_code nsr=LU_DONE
LU_LOOP       print l_name,l_date
              match 2 nsr=LU_DONE
              goto LU_LOOP
LU_DONE       return
```

SEE ALSO

match, read, readkey

get # Read data from a sequential file

SYNTAX

get [**#** *channel* ,] *fieldname* [, *fieldname*] ... [**err** = *label*]

DESCRIPTION

Reads data from the sequential file which is open on *channel* into the specified fields. If *channel* is 0 (zero) or omitted, data is read from the standard input (normally the keyboard).

Each **get** command reads in one record terminated by the system end of line character(s). A record consists of data items, each terminated by either a valid separator or end of line. The entire record must be ASCII text.

The data items read are assigned to the corresponding fields. If there are too few data items in the record, the extra fields are given null values. If there are two many data items in the record, the extra items are ignored. The fields may be of any type. The input data is converted accordingly. The separator and the end of line character(s) are not stored in the fields.

Input from the keyboard is a special case. Characters are read until a RETURN character is typed. The entire line is then stored in the first field. The value in the special temp **separator** is ignored. It is not possible to enter more characters than the width of the field. If more than one field is specified, the extra fields are ignored.

If the optional **err** trap is present and an error occurs, the system error number is stored in the special temp **errno** and control passes to the specified label. If an error occurs and there is no **err** trap, an error message is output and the program aborts. See **!temp errno** for a complete list of returned errors.

The separator is currently defined in the special temp **separator** which is an **a1** field and has the default value ",".

NOTES

- If a field has the **n** (null terminated) format defined, the field is stored as read. It is not padded with spaces to the field width, so the field length is preserved.

EXAMPLE

```
      open #1,"ADDRESS.TXT" read
LOOP  get #1,name,add1,add2,add3,postcode err=DONE
      insert 2
      goto LOOP
DONE  close #1 : exit
```

gosub — Call a subroutine

SYNTAX

gosub *label*

DESCRIPTION

Transfers control to a subroutine at the label indicated. When a return statement is encountered, control is returned to the statement following the **gosub** command.

Subroutines may be nested to a maximum limit of 90 levels deep. Each subroutine must always be exited eventually using a **return** statement. Repeated use of **goto** commands to exit a subroutine will either cause a stack overflow error or a "too many nested gosubs" error.

NOTES

- An **end** command will clear the internal subroutine stack.
- Subroutines may call themselves (recurse) but it should be remembered that all variables within the subroutine are global and are not local to that recursion.

EXAMPLE

```
!init gosub SETUP
...
SETUP gosub GETSYS
            gosub CALCHEADER
            return
GETSYS      read 2 key=syskey nsr=SYSERR
            unlock 2 : return
CALCHEADER
            if s_user=0 then h_text="SUPER USER PERMISSIONS"
            if s_user >0 and s_user <10 then \
                 h_text="NORMAL PERMISSIONS"
            if s_user >=10 then h_text="GUEST PERMISSIONS"
            return
```

SYNTAX

goto *label*

DESCRIPTION

Transfers control to the statement at the line indicated. It is permissible to jump to any label in the program, but it is not advisable to jump into or out of subroutines. The compiler will not complain but your program is unlikely to work as intended.

NOTES

- The **goto** comand may not be used in control statements such as **!heading**, **!final**, etc.
- The pre-processor, **spp**, uses the special line labels **XXnn** and **YYnn_nn** where **n** may be any digit, so it is recommended that you do not use these labels in your programs.

EXAMPLE

```
goto DISP
```

hangup — Enable or disable hangup interrupt.

SYNTAX

hangup on | off

DESCRIPTION

If **hangup** is **off**, hangup interrupts are ignored.

If **hangup** is **on** and **sagerep** receives a hangup interrupt, it completes any file access being processed and then terminates. This is the default state.

The operating system usually sends a hangup interrupt to indicate that a modem connection has failed or, in the case of a multi-user system, that the system is about to close down.

NOTES

- On Unix systems, switching the terminal off or temporarily removing the serial cable may cause a hangup interrupt. To prevent this, see the **clocal** option of the Unix command **stty**.

EXAMPLE

```
hangup off
```

SYNTAX

if *expression* **then** *statement* [**else** *statement*]

DESCRIPTION

The statement which follows **then** is executed only if *expression* is true. If the optional **else** clause is included, the statement which follows **else** is executed only if *expression* is false. Both the statement which follows **then** and the statement which follows **else** may be multiple statements separated by colons.

The statement which follows **else** may be another **if** statement with an optional **else** clause, and so on.

The statement which follows **then** may also be another **if** statement with an optional **else** clause, but in this case, the first **if** statement cannot have an **else** clause.

The *expression* may include all supported arithmetic, relational and logical operators. Parentheses may be used to force a particular order of evaluation. An arithmetic statement which evaluates non-zero is true and one which evaluates to zero is false.

The folowing relational and logical operators are available:

=	equal to
<	less than
<=	less than or equal to
>	greater than
>=	greater than or equal to
<>	not equal to
ct	contains (alpha only)
bw	begins with (alpha only)
and	logical and
or	logical or

EXAMPLE

```
if st_rol  0 and st_stklev < st_rol then \
      print "Item below re-order level"; : \
      if st_eoq  0 then \     ,
            print ".  EOQ = "; st_eoq \
      else \
            print

if vcode = "A" then \
      acnt = acnt + 1 : gosub CALCA \
      else if vcode = "B" then \
            bcnt = bcnt + 1 : gosub CALCB \
            else if vcode = "C" then \
                  ccnt = ccnt + 1 : gosub CALCC
```

SYNTAX

input "*prompt text*", *field_name*

DESCRIPTION

Inputs a value into a field. The prompt text is displayed on the screen with a question mark appended. The reply is validated for correct data type and stored in the designated field. Note that **sagerep** does not check validation lists. If the reply is not valid for the data type then the bell is sounded and the prompt is repeated.

Responses to **input** may be read from a text file by redirecting standard input. On operating systems that allow **sagerep** to detect this situation, the prompt text is suppressed and an invalid reply aborts the program.

input honours the **u** and **l** formats on alphanumeric fields. However, since **sagerep** does not work in single character input mode, the characters are echoed back as typed.

EXAMPLE

```
input "Is the paper correctly aligned",reply
if toupper(reply)="Y" then ...
```

insert — Insert a new record

SYNTAX

insert *file_number* [**key** = *field_list*] [**re** = *label*]

DESCRIPTION

Inserts a new record on the specified file. The index is immediately reorganised so that the record appears in its correct location in the file. If a record is inserted on the driving file, the file position may be altered and the report sequence affected.

The key must be unique. If a record having the supplied key already exists, then the insert does not take place and the command is ignored. This condition may be trapped by using the **re** trap, in which case control passes to the line indicated.

Normally, the **key=** clause is omitted and the key data in the record buffer is used as the key. If the **key=** clause is present, a key is constructed using data from the named fields and the natural key fields are updated accordingly.

NOTES

- Use of the **insert** command will cause the keyed file to be opened in update mode for the duration of the program.
- If there is insufficient space on the disk for the new record and its index entry, an operating system error will occur and the file may become damaged (see the **kfcheck** and **kfri** utilities).
- As the **insert** command does not show any error if a record with the same key exists, it is recommended that the programmer always trap this condition with the **re** trap.

EXAMPLE

```
     insert 3 re=I1
     display "New customer recorded" : end
I1   error "Customer already on file" : end
```

SYNTAX

interrupts on | off

DESCRIPTION

If interrupts are on and **sagerep** receives a standard keyboard interrupt, it will abort the program. If interrupts are off, keyboard interrupts are ignored.

Whatever state is set with this command, **sagerep** does not respond to interrupts while it is updating a disk file. This prevents the index from becoming damaged.

The default state for interrupts is **on** if there are no file update commands in a program and **off** if there are file update commands.

EXAMPLE

```
interrupts on
```

SYNTAX

keep *numeric_expression*

DESCRIPTION

Checks the number of lines left on the current page (excluding footnote lines) and if there are less lines left than the value of the expression, starts a new page, issuing footnotes and headers accordingly.

The command is useful to ensure that a block of lines can all be printed on the current page. To avoid printing over the perforations on continuous stationery, a minimum **keep 2** is recommended at the start of the main statements, with a complementary **!heading** statement that prints two blank lines.

Before starting a new page, all **!footnote** statements are executed and at the start of the new page, all **!heading** statements are executed.

EXAMPLE

```
!on starting ordno keep nlines + 4
```

SYNTAX

[**let**] *field_name* = *expression*

DESCRIPTION

The expression is evaluated and the result stored in the designated field. If the type of the result does not match the type of field then an appropriate conversion takes place (see page 8-7). The expression may include all supported arithmetic, relational and logical operators and all functions. Parentheses may be used to force a particular order of evaluation. A relational expression yields zero if false and non-zero if true.

The word **let** is optional and is normally omitted.

EXAMPLE

```
let fullname = firstname / " " + surname
total = qty * price * (1 + vatrate)
roflag = stklev < rol
if (roflag) then ...
if (qty*price) then ...
```

lock Place a read lock on a file

SYNTAX

lock *file_number* [**riu**=*label*]

DESCRIPTION

Any existing record on the file which is locked by this program is first unlocked. The command then places a read lock on the entire file.

If any record on the file is currently locked by another process, the command waits until the file can be locked. This condition may be trapped using the **riu**=*label* clause, in which case control passes to the line indicated.

A file lock prevents the file from being updated by any program until the file is unlocked using the **unlock** command, closed using the **close** command or the program exits.

NOTES

- Index only files cannot be locked.
- On systems which permit it, several programs may have the same file read locked at the same time.
- If a program which holds a file lock inserts or deletes a record on the file, the read lock is released. If another program holds a file lock, the insert or delete will wait until the file is unlocked.
- If a program which holds a file lock attempts to write a record back to the file, the result is undefined. On some systems the write will succeed, on others the write will be ignored.
- On VMS, this command is ignored,

EXAMPLE

```
lock stock
```

SYNTAX

match *file_number* [**nsr** = *label*] [**riu** = *label*]

DESCRIPTION

Unlocks any existing locked record on the file then returns the next record whose key matches the key supplied to the previous **find** command on the indicated file. The **match** command starts its search at the current file position. Refer to **find** for full details of key matching.

If no matching key is found a blank record is returned. This error may be trapped by using the **nsr** trap, in which case control passes immediately to the line indicated.

If the located record is currently locked by another user, a read is retried every three seconds until successful. This status may be trapped by using the **riu** trap, in which case control passes immediately to the line indicated. On most systems, the **record in use** status can only occur if the file is open in update mode.

NOTES

- A **match** operates on the key supplied to the last **find** command on the relevant file.
- As the **match** command does not report an error if no match was found, it is recommended that the programmer always trap this condition using the **nsr** trap.

EXAMPLE

```
            find 2 nsr=FC2
FC1         printh m_refid,c_name,c_refid,c_status
            match 2 nsr=FC3
            goto FCi
FC2         print "Cannot find customer",c_refid
FC3         end
```

SYNTAX

newpage

DESCRIPTION

Since **sagerep** knows the page depth (either from a !**depth** declaration or from the printer parameter file), it starts new pages automatically. The **newpage** command may be used to force a new page at a particular point in the report.

Before starting a new page, all !**footnote** statements are executed and at the start of the new page, all !**heading** statements are executed.

The **pageno** special temp is incremented when the **newpage** command is issued.

EXAMPLE

```
!on starting s_cat newpage
```

SYNTAX

next *file_number* [**nsr** = *label*] [**riu** = *label*]

DESCRIPTION

Unlocks any existing locked record on the file then reads the next record in ascending key sequence from the specified file. The next record is the one whose key immediately follows the last key referenced by any file access command except **testkey**, even if that key does not actually exist on the file. Note that **next** never returns the first record on a file if its key is completely null (all bytes binary zero).

If end of file has been reached, a blank record is returned. This error may be trapped by using the **nsr** = *label* clause, in which case control passes to the line indicated.

If the next record is currently locked by another user, the read is retried every three seconds until successful. This status may be trapped by using the **riu** = *label* clause, in which case control passes immediately to the line indicated. In this case the file position is not changed, so another **next** will try to read the same record. The **nextkey** command may be used to skip a busy record. On most systems, the **record in use** status can only occur if the file is open in update mode.

EXAMPLE

```
            next 2 riu=N1 nsr=NOREC
            printh m_refid,c_name,c_status
            end
N1          print "Record in use : ",m_refid :end
NOREC print "All records read for this pass"
            end
```

SYNTAX

nextkey *file_number* [**nsr** = *label*]

DESCRIPTION

Unlocks any existing locked record on the file then reads key data only for the next record in ascending key sequence. No attempt is made to read the data record, so a **record in use** status cannot occur and the file's data fields remain unaltered, although the key fields in the record buffer will be updated.

Since **nextkey** is faster than the **next** command, it is useful when searching keys for particular values. It may also be used to skip a locked record whilst reading a file sequentially.

The next key is the one which immediately follows the last key referenced on the file by any file access command except **testkey**, even if that key does not actually exist. Note that **nextkey** never returns the first key on a file if that key is completely null (all bytes binary zero).

If end of file has been reached, blank key values are returned. This error may be trapped by using the **nsr** = *label* clause, in which case control passes to the line indicated.

EXAMPLE

```
A1    nextkey 2 nsr=END
      if f2_type=intype then inctr=inctr+1
      goto A1
END   end
```

SYNTAX

open *file_number*

DESCRIPTION

This command opens a Sculptor keyed file (i.e. one with records to be accessed in the normal indexed-sequential way according to key values). For sequential files see the **open #** command.

open should be used before accessing any of the current programs standard data files which were initially declared as closed (see the **!cfile** declaration), or have been closed with the **close** command.

Closing and re-opening a file does not alter the current file position and does not clear the file's record buffer.

If there are no commands in the program which can update the file then it is opened in read-only mode, otherwise it is opened in update mode.

If the maximum number of open files allowed by the operating system in use is exceeded, the program will abort with an operating system error. An attempt to open a file which is already open is ignored.

EXAMPLE

```
open 3
```

SEE ALSO

!handles, close, open #

open # Open a sequential file

SYNTAX

open #*channel*, *"pathname"* **read** | **write** | **append** [**err** = *label*]

DESCRIPTION

Opens a sequential file on *channel*, which must be a number in the range 1-32. It may be a constant, field name or expression. *pathname* is the name of the file to be opened. This may be a string constant (in quotes) or a field.

If the file is opened for reading, an error will occur if the file does not exist. If the file is opened for writing, it will be created if it does not exist or will be truncated to zero length if it does exist. If the file is opened for appending, it will be created if it does not exist. If it does exist, the file pointer will be positioned at the end of the file.

If a file needs to be opened for both reading and writing, it may be opened twice on different channels.

A sequential file may be read using the **get #** command and written using the **put #** command. The **rewind #** command may be used to reposition the file pointer to the beginning of the file. When it is no longer required, the file should be closed using the **close #** command.

If the optional **err** trap is present and an error occurs, the system error number is stored in the special temp **errno** and control passes to the specified label. If an error occurs and there is no **err** trap, an error message is displayed and the program aborts. See **!temp** on page 8-60 for a complete list of error codes returned.

EXAMPLE

```
open #1, seqfile write
open #barcode, "/dev/barcode" read err=OPENERR
```

SYNTAX

pause

DESCRIPTION

The program sleeps until an alarm interrupt is sent to the process. On receiving an alarm interrupt, processing continues with the statement which follows the pause.

NOTES

- This command is available only On Unix and certain similar operating systems.
- A good understanding of the equivalent operating system function is recommended before using the **pause** command. For example, it is wise on Unix to ensure that at least a few seconds elapse before a program which has paused can receive an alarm interrupt. Otherwise, because of task switching, it is possible for the program which is pausing to receive and ignore the alarm before it has completed the **pause** operation, with the result that it sleeps forever.

SEE ALSO

wakeup

prev — Read the previous record

SYNTAX

prev *file_number* [**nsr** = *label*] [**riu** = *label*]

DESCRIPTION

Unlocks any existing locked record on the file then reads the previous record from the specified file. The previous record is the one whose key immediately precedes the last key referenced by any file access command except **testkey**, even if that key does not actually exist on the file. Note that **prev** cannot return the last record on a file if its key contains the highest possible value.

If beginning of file has been reached, a blank record is returned. This error may be trapped by using the **nsr = label** clause, in which case control passes to the line indicated.

If the previous record is currently locked by another user, the read is retried every three seconds until successful. This status may be trapped by using the **riu = label** clause, in which case control passes to the line indicated; in this case the file position is not changed, so another **prev** will try to read the same record. The prevkey command may be used to skip a busy record. On most systems, the **record in use** status can only occur if the file is open in update mode.

NOTES

- As no error is shown when there is no previous record, it is strongly recommended that the programmer always trap this condition with the **nsr** trap.
- On VMS, **prev** only works on Sculptor format files. On RMS indexed files (.v extension), **prev** always returns a **nsr** condition.

EXAMPLE

```
prev 5 riu=N1 nsr=N2
print c_name
```

SYNTAX

prevkey *file_number* [**nsr** = *label*]

DESCRIPTION

Unlocks any existing locked record on the file then reads key data for the previous record in descending key sequence. No attempt is made to read the data record, so a **record in use** status cannot occur and the file's data fields remain unaltered. **prevkey** is useful for skipping locked records when reading through the file in descending key sequence. It is also faster than the **prev** command when only key data is required.

The previous key is the one which immediately precedes the last key referenced on the file by any file access command except **testkey**, even if that key does not actually exist. By definition, **prevkey** cannot return the last key on a file if that key contains the maximum possible data value.

If beginning of file has been reached, blank key values are returned. This error may be trapped by using the **nsr** = *label* clause, in which case control passes to the line indicated.

NOTE

- On VMS, **prevkey** only works on Sculptor format files. On RMS indexed files (.v extension), **prevkey** always returns a **nsr** condition.

EXAMPLE

```
prevkey 5 nsr=BOF
```

SYNTAX

print[h] [*print_item*] [, *print_item*] ...

DESCRIPTION

Prints the items listed to standard output. Items may be separated by either commas or semicolons. A comma prints the number of spaces specified in the **!gap** declaration (the default is 2 if not declared). A semicolon causes no spaces to be printed. The end of the print list generates a new line unless it is terminated by a comma or a semi-colon.

If an item or series of items in the print list is enclosed in square brackets "[]", then the printing of the items is suppressed and an equivalent number of spaces is printed instead. This facility makes it easy to align continuation print lines and total lines.

If a field name in the print list is preceded with an asterisk "*" , the field heading is printed instead of the data.

Using the **printh** format of the command will pad the field according to the size of the heading or data that would be printed for this field (whichever is the larger). So, if the field was an a3 with a heading "TEST HEADING", and the field was printed using **print**, the 3 characters would be printed then the next field would follow. Using **printh**, however, the field would be printed, followed by an extra 9 characters to make up the width of the heading. If the data width is greater than the heading, using **printh** to display the heading will correctly pad it to match the field size.

Print items may be any of the following:

Item	Interpretation
field name	Current data value.
*****field name*	The field's heading.
total(*field name* **)**	Current total.
min(*field name* **)**	Current minimum value.
max(*field name* **)**	Current maximum value.
count()	Selected record count.

Item	Interpretation
constant	Value of that constant.
spc(*constant* **)**	Specified number of spaces.
spc(*field name* **)**	Spaces according to field value.
tab(*constant* **)**	Tab to specified column (from 0).
tab(*field name* **)**	Tab according to field value.
#tf	Top of form character.
#dw	Start double width characters.
#sw	Start single width characters.
#su	Start underline.
#eu	End underline.
#ec	Start enhanced characters.
#oc	Start ordinary characters.
#ac	Select alternate character set.
#sc	Select standard character set.
#c0...#c28	User defined sequences from the printer parameter file.

Field data values are printed according to the field's format. This is the format declared in the data dictionary or, in the case of a temporay field, the format in the **!temp** declaration. In the absence of a declared format, a default is used. A pre-declared field format may be changed with a format declaration (+). For further details see page 8-5.

The **total()**, **min()**, **max()** and **count()** special functions are updated after each cycle through the main block of statements. If used in an **!on ending** statement they return values for the ending block only, which makes for easy sub-totalling. Otherwise they return values for the report so far, but since the update takes place (necessarily) at the end of the main statement cycle, the functions are in a sense one step behind. Correct *running* values for the report are obtained by using the special functions in **!on starting** and **!final** statements. These functions may be applied both to record fields and to temporary fields.

The special **#** items print the control codes defined in the selected printer parameter file, allowing simple selection of double width characters, underlining, etc. Note that some printers automatically revert to single width at the start of each new line whereas others don't. To be safe, the code to return to single width should always be given and not assumed. Take care also when selecting different character sets, since this can send the printer back to a default character compression. The character compression may be reset with the **width** command. Note that the **newpage** command, which executes the **!heading** statements, is the normal way to start a new page; **#tf** throws a page without executing the **!footnote** or **!heading** statements.

NOTES

- By default, up to 20 fields may be the target of the special functions **total()**, **min()** and **max()**. Using more than 20 fields with these functions will result in the compiler error message " Too many functions ". Use the **-f=***nn* switch (where *nn* is an integer defining the number of fields) when compiling to increase the number of fields that can be used with the special functions.
- Use of **spc()** and **tab()** functions will prevent correct alignment using **printh** unless the same function is used in both data and header print statements.

EXAMPLE

```
!heading print #dw;#su;"SALES ANALYSIS",date;#eu;#sw
!heading print: print
!heading printh *c_name,*c_date,*c_amount
!heading printh ["Item: ";itemno],*i_code,*i_bin
!on ending cat print [i_code,i_costpr],total(costval)
     printh "Item: ";itemno,i_code,i_bin
     print c_name,c_initial;".",c_date,c_amount,c_status
     print s_code,s_desc,s_bin;spc(20);s_st
     printh c_name,c_date,c_amount
```

SYNTAX

put [**#** *channel*,] *expression* [**format** *format*]
[, *expression* [**format** *format*]] ... [**err** = *label*]

DESCRIPTION

Writes data to the file which is open on *channel*. If *channel* is 0 (zero) or omitted, data is written to the standard output.

Each *expression* is evaluated and output as ASCII text. If a **format** keyword is not present, a default format to suit the type of the expression is used. If the **format** keyword is present, *format* must be a valid Sculptor field format. It may be an alpha field or a string constant (in quotes).

A comma between items causes the value in the special temp **separator** to be output. **separator** is an **a1** field and has the value ",". It may be assigned a new value. A semi-colon may be used in place of the comma in which case a separator is not output. A newline is appended at the end of the data unless the final item is terminated by a comma or a semi-colon.

If the optional **err** trap is present and an error occurs, the system error number is stored in the special temp **errno** and control passes to the specified label. If an error occurs and there is no **err** trap, an error message is displayed and the program aborts. See **!temp errno** (page 8-60) for a complete list of error codes returned.

NOTES

- If the **n** (null-terminated) format is defined for a field, the field is written as stored, so the assigned field length is preserved.

See the Screen Form Language section, page 7-128 for a full example of the use of this command.

SYNTAX

read *file_number* [**key** = *field_list*] [**nsr** = *label*] [**riu** = *label*]

DESCRIPTION

Unlocks any existing locked record on the file then reads the record whose key exactly matches the supplied key. If no matching key is found, a blank record is returned. This status may be trapped by using the **nsr** = *label* clause, in which case control passes to the line indicated. If a read is performed on the driving file, the file position may be altered and the report sequence affected.

If the requested record exists but is currently locked by another user, the read is retried every three seconds until successful. This status may be trapped by using the **riu** = *label* clause, in which case control passes to the line indicated. On most systems, the record in use status can only occur if the file is open in update mode.

If the **key=** clause is omitted, the key data in the record buffer is used as the key. If the **key=** clause is present, a key is constructed using data from the named fields.

EXAMPLE

```
read 3 key=df_key1,df_key2 nsr=READERR
f2_key=f4_data1  : read 2 nsr=READERR
read 7 key=f3_data1,f4_data3 nsr=READERR riu=LOCKED
```

SYNTAX

readkey *file_number* [**key** = *field_list*] [**nsr** = *label*]

DESCRIPTION

Unlocks any existing locked record on the file then reads key data only from the designated file. If a record is located whose key exactly matches the supplied key, then the file's natural key fields are updated and control passes to the next statement. No attempt is made to read the data record, so a **record in use** status cannot occur and the file's data fields remain unaltered.

If no matching key is found and the error is not trapped, the key fields are cleared. Data fields are always left unaltered. This error may be trapped by using the **nsr = label** clause, in which case control passes to the line indicated and no fields are cleared.

Whether or not a matching key is found, the current file position is changed for the purpose of the **next** and **nextkey** commands. In this respect, **readkey** differs from **testkey** which does not alter the file position.

If the **key=** clause is omitted, the key data in the record buffer is used as the key. If the **key=** clause is present, a key is constructed using data from the named fields.

EXAMPLE

```
.Position driving file to read from first "TT" item

      if st_code<"TT" then \
            clearbuf 1 : \                /* empty fields     */
            st_code = "TT" : \            /* set start value */
            readkey 1 nsr=M1              /* OK if not there */

M1    ...     rest of main statements

      end
```

SYNTAX

return

DESCRIPTION

Returns control from a subroutine to the statement following the calling **gosub**.

NOTES

- All pending **return**s are maintained on an internal stack and a " **Stack overflow** " error will occur if subroutines are nested more than 90 levels deep. Each **return** removes an entry from the stack.
- A " **Return without gosub** " error will occur if a **return** is encountered and the return stack is empty (no previous **gosub**).
- An **end** command clears the internal stack and thus clears any pending **return** statements.

EXAMPLE

```
...
!init       gosub SETUP : gosub SCAN : gosub REPORT
            exit
SETUP       gosub READ_SYS
            sys_rec=sys_rec+1
            gosub WRITE_SYS
            return
SCAN        next 1 nsr=DONE
            frn_rad=frn_rad+1.04
            goto SCAN
DONE        return
REPORT      display "Update complete." : return
READ_SYS    rewind 2 : next 2 nsr=NOSYS : return
WRITE_SYS   write 2 : close 2 : return
NO_SYS      display "System record missing!"
            sleep 3 : exit 9
```

SYNTAX

rewind *file_number*

DESCRIPTION

Repositions the specified file at its start so that the **next** command will return the first record in the file. The content of the file's record buffer is not affected.

NOTES

- The **next** command cannot return a record whose key is completely null (i.e. all binary zeros).

EXAMPLE

```
rewind 2
```

SEE ALSO

open, close

SYNTAX

rewind # *channel*

DESCRIPTION

Rewinds the sequential file open on *channel.*

channel is a number in the range 1-32 which corresponds to the channel number used when the file was opened. It may be a constant, field name or expression.

If the file is open in **read** mode, the next **get** will read from the beginning of the file.

If the file is open in **write** or **append** mode, the next **put** will overwrite the beginning of the file. Rewinding a file does not truncate it.

EXAMPLE

```
rewind #1
```

SYNTAX

rounding on | off

DESCRIPTION

When floating point values are assigned to integer fields, the fractional part is truncated before assignment to the integer. The fractional part may be rounded to the nearest integer by turning rounding on. By default, **rounding** is **off**.

For example, if the value 5.6 is assigned to an integer field, the result is 5 if rounding is off (the default) or 6 if rounding is on. If the value 5.4 is assigned, the result is always 5. The value 5.5 normally becomes 6 but since floating point formats cannot always exactly represent a number, this is not guaranteed.

EXAMPLE

```
rounding on
```

SYNTAX

scroll [*expression*]

DESCRIPTION

Resets the special temp **scrline** (the scroll line number), according to the value of *expression* as follows:

expression = 0 (or omitted)	Increments **scrline** by 1.
expression > 0	Sets **scrline** to the value of *expression*
expression < 0	Reduces **scrline** by the value of *expression* (but not below the value 1).

The value in **scrline** is the default index value for all subscripted fields which are not explicitly indexed when referenced.

If the value in **scrline** exceeds the dimension of a subscripted field, the field is indexed on a wrap around basis.

NOTES

- The value held in **scrline** cannot be altered by direct assignment.

EXAMPLE

```
scroll ordline
```

SYNTAX

sleep *numeric_expression*

DESCRIPTION

Suspends program execution for the number of seconds specified in the expression. When the time has elapsed, execution resumes at the statement following the **sleep** command.

Since many operating system clocks are only accurate to the nearest second, an error of up to one second is possible.

EXAMPLE

```
display "Sleeping..." : sleep 5 : display "Awake now!"
```

testkey — Check for record with specified key

SYNTAX

testkey *file_number* [**key** = *field_list*] [**nsr** = *label*]

DESCRIPTION

Tests the specified file for a record whose key exactly matches the supplied key. If the record exists, then the file's natural key fields are updated and control passes to the next statement. No attempt is made to read the data record, so a **record in use** status cannot occur and the file's data fields remain unaltered.

If no matching key is found, and the error is not trapped, the appropriate key fields are cleared. Data fields are always left unaltered. This error may be trapped by using the **nsr** = **label** clause, in which case control passes to the line indicated and no fields are cleared.

Whether or not a matching key is found, the current file position remains unchanged for the purpose of the **next** and **nextkey** commands. In this respect **testkey** differs from **readkey** which alters the file position.

If the **key=** clause is omitted, the key data in the record buffer is used as the key. If the **key=** clause is present, a key is constructed using data from the named fields.

EXAMPLE

```
testkey 3 key=temp_ref,temp_name nsr=E5
```

SYNTAX

unlock *file_number*

DESCRIPTION

Unlocks the currently selected record from the specified file to allow access by other users. The data in the file's record buffer is not affected but the record may no longer be written back or deleted.

A locked record is automatically unlocked if it is written back or deleted or if an attempt is made to read another record from the same file. Records are only locked if the file is open in update mode (see !**file** on page 8-43).

If a **lock** command has been executed on the named file by this program, **unlock** will remove that lock.

NOTES

- On single user operating systems which do not support record locking, the **unlock** command is ignored.

wakeup — Send an alarm interrupt to a process

SYNTAX

wakeup *task_id*

DESCRIPTION

An alarm interrupt is sent to the specified process. The alarm call can be sent to any process whose id is known and which is capable of accepting the interrupt. For example, it could be sent to a **sage** or **sagerep** program which has used the **pause** command. The suspended program will then restart from the point at which it paused.

NOTES

- A simple way of determining the task id of another program is for each participating program to write its id into a shared file.
- A good understanding of the equivalent operating system function is recommended before using the **wakeup** command. For example, it is wise on Unix to ensure that at least a few seconds elapse before a program which has paused can receive an alarm interrupt. Otherwise, because of task switching, it is possible for the program which is pausing to receive and ignore the alarm before it has completed the **pause** operation, with the result that it sleeps forever.
- This command is available only on Unix and certain similar operating systems.

EXAMPLE

```
wakeup re_id
```

SYNTAX

width *numeric_expression*

DESCRIPTION

Changes the line width to the specified number of columns on devices that support this feature. This has the effect of selecting a new character compression by issuing the required code sequence from the printer parameter file. It useful for highlighting parts of the report and for compressing lines that would otherwise be too wide for the output device.

If *numeric_expression* does not exactly match a width in the printer parameter file, the next higher value is used.

The new value remains in force until a subsequent **width** command is encountered. The initial line width may be set with a **!width** declaration (see page 8-65). If there is no **!width** declaration, the initial width set will be that defined as the standard width in the printer paarmeter file.

EXAMPLE

```
...
     width 120
     print t_var1,t_var2,t_var3,t_var4,t_var5,t_totline
     width 90 : print "End of report",,date,time
...
```

SYNTAX

write *file_number* [**re** = *label*]

DESCRIPTION

Writes back and unlocks the record last read from the specified file. Note that a record must be written back if amendments made to its key or data fields are to be permanently recorded.

If no record is currently selected, then the command is ignored.

If any key data has been altered since the record was read, a new record is inserted and then the old record is deleted. In this case, the file is positioned at the old key value for the purpose of the **next** and **nextkey** commands.

NOTES

If key data has been altered but a record with the new key value already exists on the file, then the command is ignored. This condition may be trapped by using the **re = label** clause in which case control passes to the line indicated.

EXAMPLE

This example performs a batch update on the driving file and demonstrates the ease and simplicity with which batch updates can be performed.

```
!file 1 ../acc/data/costytd

      cost_ytd=0 : cost_last=cost_current
      cost_mtd=0 : cost_rise=750
      write 1
```