

This chapter describes the additional language features provided by the Sculptor pre-processor, **spp**.

Contents	Page
An overview of the pre-processor	6-2
Conditional compilation and manifest constants	6-3
Additional flow control	6-4
File inclusion	6-5
Automatic exclusion and subroutine libraries	6-5
Generated line labels	6-6
Command line syntax	6-7
Command section contents	6-9

An overview of the pre-processor

spp is a front end pre-processor for both the Sculptor screen language and the Sculptor report language. It provides additional language features such as manifest constants, the ability to include library files, conditional compilation and some new commands.

spp works by reading a source language file which contains both pre-processor statements and standard screen or report language statements. It outputs new source code which contains only standard screen or report language statements.

When the Sculptor compiler, **cf** or **cr**, reads your source file, it transforms the instructions and declarations in that source file into a form which is understood by the interpreter, **sage** or **sagerep** respectively. This *processes* the source file.

Pre-processing involves reading the source file and replacing any commands and declarations that only the pre-processor understands with commands that the compiler, **cf** or **cr**, will understand. The whole process is one of textual substitution, nothing more. The source file is read by the pre-processor, text substitution occurs and a new temporary source file is produced which is passed to the appropriate compiler for compilation. The original source file may contain commands which can only be recognised by the pre-processor, but the source file output to the compiler will contain only commands that the compiler will recognise.

Performing this extra processing has a number of advantages. During the course of reading the file, other operations may also be performed.

1. **Conditional compilation.** Sections of program code may be included or omitted from the output source file based on the contents of special constants that are defined to the pre-processor either on the command line or in the body of the program being compiled.
2. **Manifest constants.** These special constants allow the programmer to use symbolic names which will be replaced, by the pre-processor, with text as required. This aids portability of the source, simplifies reading the source code and is a very useful aid when debugging.

3. **Additional flow control.** The pre-processor provides language extensions, such as loops, which are converted to normal flow control statements in the output file.
4. **File inclusion.** A single source file may be comprised of multiple files, included as directed at any location. These included files may contain program subroutines, options, screen definitions or any other portion of the final source file.
5. **Automatic exclusion.** Sections of program code may be automatically excluded from the output source file if the code is not used by the program.

By default the source code output by **spp** is placed in a temporary file and the appropriate compiler is called to compile it (**cf** for screen form programs or **cr** for report programs). If the compilation is successful the object file produced by the compiler is renamed to have the same name as the original source file except for its extension; screen program object files have a **.g** extension; report program object files have a **.q** extension. If the compilation is unsuccessful any existing object file is left unaltered. All temporary files are automatically deleted.

Conditional compilation and manifest constants

Conditional compilation is a process whereby selected parts of the source program are included or excluded according to the value of compile-time conditions. Its main benefit is to allow one suite of source code programs to be maintained for a variety of target compilation environments. Additional uses include the ability to include debugging/trace statements during the development cycle, or to include/exclude specific functions from a compiled program according to requirements.

Compile time conditions are implemented by testing the value of certain constants. These constants fall into two basic sets; system predefined constants and programmer defined constants. Collectively these are referred to as manifest constants.

A manifest constant may be defined within the program using the **!define** declaration or on the **spp** command line using the **-D** option.

The section of source code to be included/excluded is defined by the scope of the **!ifdef/!ifndef** declaration and the corresponding **!endif** declaration.

Manifest constants may be defined without value, in which case they may be used only to control conditional compilation, or they may be given a textual value. When a manifest constant is given a value, every occurrence of that manifest constant in the output source file is replaced with the text value defined.

Important note: The pre-processor performs textual substitution only - it does not evaluate expressions.

Text enclosed within double quotes will not be replaced, but option help text and box titles enclosed in curly braces will be. The example below shows how a help message and box title can be changed according to the value of a manifest constant.

```
!define PROG          test.r

+at 8,20 : drawbox 5,40 {PROG : test screen}

*t=Test      {PROG: test option}
             message "Test PROG"
```

The PROG within curly braces will be replaced, but that within the quotes will not.

Additional flow control

The compilers, **cf** and **cr**, support only line labels and statements that transfer control to them, ie, **goto**, **gosub** and trap clauses. Most modern languages support program flow control constructs such as **for**, **while** and **switch**. These constructs however, are usually broken down into simpler labels and jumps in the compilation or interpretation phase of these languages.

In Sculptor, the pre-processor implements the **for**, **while** and **switch** constructs, producing special line labels and the appropriate **goto** statements in the output source file.

for and while loops

The **for** loop has a construct similar to that supplied by the 'C' language. It benefits from full control of the initialisation, continuation condition and control parts of the loop. Both **for** and **while** loops execute while the loop condition remains true (non-zero) and will only execute if the condition is true when the loop is encountered.

Two additional pre-processor commands are also provided to control loop iteration. These are **break** and **continue**. The special line label versions of these (**BREAK** and **CONTINUE**) may be used within a trap clause to directly control loop iteration.

switch

The switch construct allows selective execution of one or more sections of code based on an expression evaluated at run-time.

File inclusion

As programs become larger and more complex, the ability to break a complete program down into smaller components becomes more attractive. In group programming environments, also, the ability to utilise standard subroutine libraries across applications simplifies standardisation, development and modification.

The pre-processor provides the **!include** declaration to allow the inclusion of other source files at the point of the declaration. The included files may contain both valid program code, such as declarations, options or subroutines and other **!include** declarations. Includes may be nested up to 15 levels deep.

When making extensive use of include files it is wise to adopt a naming convention for fields and labels to reduce the possibility of naming conflicts. Include files may have any name and/or extension - within operating system limits.

Automatic exclusion and subroutine libraries

This combination of include files and conditional compilation can be taken a step further. The pre-processor provides the declarations **!ifneed** and **!endneed** which allow the inclusion of code sections *only* if a named subroutine is the subject of a **gosub** statement. This allows the efficient use of a subroutine library which may be included in every program, the output code containing only those routines used by that program.

Generated line labels

When replacing flow control statements with line labels and goto statements, the pre-processor generates special line labels. Name conflicts will occur if your program contains line labels which match those generated by **spp**, but the format of the labels makes this unlikely.

The labels generated by the **for** and **while** loops have the format XXn where n is a sequential number. The first generated label will be XX0.

The **switch** statement is more complex. Line labels take the format YYn_a where n is a sequential number and where the letter a is added when the **break** statement is *not* used at the end of a preceding **case** to continue execution at the statements in the *next case*. The algorithm used to calculate the numbers ensures that no duplicate line labels will occur. The characters YY will always precede the label.

To prevent naming conflicts, do not use numeric labels in your program which begin with either the characters **XX** or **YY**.

Predefined manifest constants

The following predefined manifest constants are used to provide operating system and compiler type independence during compilation. One from each group will be defined during a compilation.

COMPILER TYPE

SAGE

Defined only when the pre-processor is producing output for the screen language compiler. This constant can be used in standard subroutine libraries to only include code which is to be compiled for the screen form language.

SAGEREP

Defined only when the pre-processor is producing output for the report/batch language compiler. This constant can be used in standard subroutine libraries to only include code which is to be compiled for the report/batch language.

OPERATING SYSTEM TYPE

MSDOS

Defined only when compiling with the MSDOS version of **spp**.

OS9

Defined only when compiling with the OS9 version of **spp**.

UNIX

Defined only when compiling with a Unix or Xenix version of **spp**. This constant will also be defined in derivatives of these operating systems, eg. SINIX, SUNOS, etc.

VMS

Defined only when compiling with the VMS version of **spp**.

Command line syntax

The command line syntax for **spp** is:

spp [-option]... *filename* [*filename*]...

where each *filename* is a text file with a **.f** or **.r** extension and *-option* may be one or more of the following:

-c

Do not compile the output from the pre-processor.
This option is useful to check for pre-processor errors or to obtain a listing only.

-Dsymbol[=text]	Define the manifest constant <i>symbol</i> . If the optional <i>text</i> is specified the manifest constant is given that value, otherwise it is given a null value. See !define for further information on the use of manifest constants.
-f=funcs	When calling the report/batch language compiler, set the number of fields that can be the target of the special functions total , min , max and count to <i>funcs</i> . The default is 20.
-l	Produce a listing of the new source code to standard output.
-L=lines	Set the maximum number of source lines that the pre-processor can handle to <i>lines</i> . The default is 4000.
-m	List the output in 24-line pages with a prompt to continue or terminate at the end of each page.
-n	Provide overall line numbers on the output listing.
-N	Provide overall line numbers and also file numbers plus line numbers within file for the main file and for each include file.
-q	Suppress header messages. By default the pre-processor identifies itself and indicates the filename and the language type of each file that it compiles.
-o=filename	Place the output source code in <i>filename</i> .
-x	List all subroutine names and the number of times that each one is called.
-z	Display or print blank if a field is zero. This may be over-ridden in a program with the !zeros declaration or with a field format that forces zeros to be shown.

Example

```
spp -l stock
spp -lN custs.r custs.s
spp -DUNIX -DMAXLINE=6 -qo=addr.out addr.f
spp -x *.f *.r >subs.out
```


Contents	Page
Pre-processor declarations	6-11
#define	6-11
#undef	6-12
#ifndef	6-13
#ifdef	6-14
#include	6-15
Language extensions	6-16
for	6-16
switch/case	6-18
while	6-20
break	6-22
continue	6-23
Special labels	6-24
BREAK	6-24
CONTINUE	6-25

SYNTAX

`!define symbol [text]`

DESCRIPTION

Declares the named *symbol* and assigns *text* to that symbol. Every occurrence of *symbol* from the point of the definition onwards, except within quoted text, is replaced with *text*. If *text* is omitted, the symbol is defined but has a null value. The *text* may contain other manifest constants and, provided the definition of the constant appears *before* this use, it will be replaced correctly.

If a manifest constant is redefined, the redefinition will be ignored.

It is convenient to adopt a naming convention for manifest constants. The examples shown in this manual use upper case for all manifest constants.

The pre-processor automatically defines certain manifest constants for your use (see page 6-6).

NOTES

- The *symbol* cannot be a reserved word.
- An occurrence of *symbol* inside quotes will not be replaced.
- An occurrence of *symbol* inside curly braces will be replaced.

EXAMPLE

```
!define DEBUG
!define AT          at 23,1
!define WARN        AT : put "WARNING : ";
!define CONFIRM     AT : put "CONFIRM : ";

*d=Delete
  WARN : put "Delete this record " ; : prompt "OK" no=D1
  CONFIRM : prompt "Are you sure" no=D1
  delete NAME : clear : end
D1  message "Record was NOT deleted" : end
```

SYNTAX

```
!ifdef symbol
statements
!else
statements ]
!endif
```

DESCRIPTION

The statements between the **!ifdef** and its corresponding **!endif** are included in the output file only if the named *symbol* has been previously defined. Each **!ifdef** must have a matching **!endif** which terminates its scope.

If *symbol* is defined the statements which follow the **!ifdef** are included and if there is a matching **!else** declaration the statements which follow the **!else** are excluded.

If *symbol* is undefined the statements which follow the **!ifdef** are excluded and if there is a matching **!else** declaration the statements which follow the **!else** are included.

A symbol may be defined with the **!define** declaration or by using the **-D** option on the command line.

The statements which follow a **!ifdef** or a **!else** may contain other complete **!ifdef** and **!ifndef** constructs.

The pre-processor automatically defines certain manifest constants for your use (see page 6-6).

EXAMPLE

```
!ifdef DEBUG
!ifdef MSDOS
    gosub MSDOS_DEBUG
!else
    gosub NON_MSDOS_DEBUG
!endif
!endif
```

SYNTAX

```
!ifndef symbol
    statements
[!else
    statements]
!endif
```

DESCRIPTION

The statements between the **!ifndef** and its corresponding **!endif** are compiled only if the named symbol has not been previously defined. Each **!ifndef** must have a matching **!endif** which terminates its scope.

If *symbol* is undefined the statements which follow the **!ifndef** are included and if there is a matching **!else** declaration the statements which follow the **!else** are excluded.

If *symbol* is defined the statements which follow the **!ifndef** are excluded and if there is a matching **!else** declaration the statements which follow the **!else** are included.

A symbol may be defined with the **!define** declaration or by using the **-D** option on the command line.

The statements which follow a **!ifndef** or a **!else** may contain other complete **!ifdef** and **!ifndef** constructs.

The pre-processor automatically defines certain manifest constants for your use (see page 6-6).

EXAMPLE

```
!ifndef DEBUG
    input password eoi=KWIT bs=KWIT
    gosub PASS_CHECK
    if user_level<>0 then goto ACCESS
!else
    message "Password bypassed"
    user_level=9
!endif
```

!ifneed

Include statements if named subroutine is called

SYNTAX

```
!ifneed name  
statements  
!endneed
```

DESCRIPTION

The *statements* are included only if there is a **gosub** to the subroutine *name* somewhere in the source code.

Typically **!ifneed** is used in an include file which contains a library of subroutines. If the statements for each subroutine are bracketed by **!ifneed** and **!endneed**, the subroutine is included only if needed.

The source code within the scope of this declaration may contain any valid program code, but will usually contain the subroutine named *label* and any subroutines associated with that subroutine. Judicious use of **!ifneed** will allow large subroutine libraries to be used without sacrificing program size.

NOTES

- As the complete source file has to be checked for the named subroutine, using **!ifneed** will cause a second pass through the source file to resolve the outstanding needs.
- When including multiple subroutines within a **!ifneed**, bear in mind that the decision to include or exclude the complete section of code will depend on only the named subroutine being called.

EXAMPLE

```
!ifneed USER_CHECK  
USER_CHECK      ... code for the subroutine  
                  return  
!endneed
```

SYNTAX

!include "*filename*" | < *filename* >

DESCRIPTION

An **!include** declaration is replaced by the entire content of the named file. The included file is compiled as if it were physically part of the current source file.

If *filename* is enclosed in angle brackets (<>) the **!include** sub-directory of the central SCULPTOR directory is searched for the named file. (By default the central SCULPTOR directory is **/usr/SCULPTOR** on Unix and **\SCULPTOR** on DOS. This default may be changed by setting the SCULPTOR environment variable to another path.)

If *filename* is enclosed in double quotes ("") the normal path searching rules apply.

NOTES

- If the named file could not be found, an operating system error is returned and the compilation is aborted.
- The contents of the included file should be relevant to the position of the declaration.

EXAMPLE

```
!include <errors.h>

!ifdef DEBUG_SIGNAL
!include "debug.scr"
!endif
```

SYNTAX

```
for ( [ initialisation ] ; condition ; [ control ] ) {  
    statements  
}
```

DESCRIPTION

Performs an *initialisation* statement at the start of the loop and then repeats *statements* while *condition* is true. The *control* statement is performed at the end of each iteration of the loop. Left and right braces are used to delimit the statements which are controlled by a **for** command.

condition is tested before the start of each iteration. If false, the loop terminates immediately and control passes to the statement which follows the closing brace.

There are two special commands which may only be used inside a loop. These are **break** and **continue**.

A **break** command causes the loop to terminate and passes control to the statement which follows the right brace. A **continue** command causes the current iteration to end. Statements between the **continue** command and the right brace are skipped but the *control* statement is still executed. The loop then continues or terminates depending on the value of *condition*.

An expression is true if it is non-zero and false if it is zero. The simple expression **1** may be used as a condition and will cause a loop to repeat indefinitely. Such a loop can only be terminated by a **break** command.

The statements between the braces may contain other loop and **switch** constructs.

Both the *initialisation* and the *control* statements may be multiple statements separated by colons in the usual manner.

The pre-processor will expand the **for** statement to standard **Sculptor** labels and **goto** statements. The pre-processed output can be listed with the -l command line option.

NOTES

- The opening brace must be on the same line as the **for** command, but line continuation characters are permitted.
- **for** loops may be nested up to 95 levels deep.
- The semi-colons are always required.
- The special labels **BREAK** and **CONTINUE** can be used on the trap clause of Sculptor commands to perform the same functions as the **break** and **continue** commands.

EXAMPLE

```
for (scroll 1; scrline < 10; scroll) {  
    input name eoi=BREAK  
    if (name = "") then break  
    input surname, phone  
}
```


SYNTAX

```
switch ( field ) {  
    case condition :      statements  
                           [ break ]  
    default :             statements  
                           [ break ]  
}
```

DESCRIPTION

A **switch** command selects a set of statements to be executed according to the value of a record or temporary field. Left and right braces are used to delimit the statements which are controlled by a **switch** command.

The concatenation of *field* and *condition* must form a relational expression which yields true or false. The conditions are tested in order and if an expression yields true the statements for that case are executed.

Normally the statements for a case should be terminated by a **break** command. This causes statement execution to continue with the statement which follows the closing brace, regardless of the condition attached to any subsequent case. If the **break** command is omitted execution continues with the first statement of the next case, regardless of the condition attached to that case.

The statements for the optional **default** are executed only if no other case is true except that it is still possible to fall through to the default case if the preceding case is not terminated by a **break** command. The position of the default case may be anywhere in the list of cases.

If no case is true and if there is no default case then no statements within the switch are executed.

The pre-processor will expand the **switch** statement to standard **Sculptor** labels and **goto** statements. The pre-processed output can be listed with the -l command line option.

NOTES

- The conditions are evaluated in the order defined.
- If no conditions are true and there is no **default**, no statements are executed.
- The special label **BREAK** may be used on the trap clause of **Sculptor** commands and will perform the same function as the **break** command.

EXAMPLE

```
switch (value) {  
    case = 10:  
        message "Value is 10"          /* fall through */  
    case < 20:  
        message "Value is less than 20"  
        break  
    case > 50:  
        message "Value is greater than 50"  
        break  
    default:  
        message "Value is between 20 and 50"  
        break  
}
```

SYNTAX

```
while ( condition ) {  
    statements  
}
```

DESCRIPTION

Repeats the *statements* while the *condition* evaluates true. Left and right braces are used to delimit the statements which are controlled by a while statement. The *condition* is evaluated at the start of each iteration of the loop and if it is false, the loop terminates immediately and control is passed to the statement following the closing brace.

If *condition* evaluates false at the start of the first iteration, the loop is not entered.

There are two special commands which may only be used inside a loop. These are **break** and **continue**.

A **break** command causes the loop to terminate and passes control to the statement which follows the right brace. A **continue** command causes the current iteration to end. Statements between the **continue** command and the right brace are skipped. The loop then continues or terminates depending on the value of *condition*.

A relational expression is true if it is non-zero and false if it is zero. The simple expression **1** may be used as a condition and will cause a loop to repeat indefinitely. Such a loop can only be terminated by a **break** command.

The statements between the braces may contain other loop and switch constructs.

The pre-processor will expand the **while** statement to standard **Sculptor** labels and **goto** statements. The pre-processed output can be listed with the **-l** command line option.

NOTES

- The special labels **BREAK** and **CONTINUE** can be used on the trap clauses of **Sculptor** commands and they perform the same functions as **break** and **continue**.

EXAMPLE

```
while (1) {  
    input value  
    if (value < 0) then break  
    total = total + value  
}
```

SYNTAX

break

DESCRIPTION

May be used only within a **for** or **while** loop, or the case section of a **switch** statement. In loops, a **break** command interrupts the current iteration of the loop and passes control to the statement which follows the loop's terminating right brace.

In a **switch** statement, a **break** command terminates the current case and passes control to the statement which follows the switch's terminating right brace.

NOTES

- This command is ignored if encountered outside a **for**, **while** or **switch** statement.
- The special label **BREAK** may be used within the trap clause of a **Sculptor** command to perform the same function.

SYNTAX

continue

DESCRIPTION

May be used only inside a **for** or **while** loop to end the current iteration of the loop. Statements between the **continue** command and the loop's closing brace are skipped. The loop then continues only if its continuation condition is still true.

This command is used by the pre-processor when generating line labels and **goto** statements for the output source file. It is not passed on to the compilers.

NOTES

- This command is ignored if encountered outside a **for** or **while** loop.
- The special label **CONTINUE** may be used within the trap clause of a **Sculptor** command to perform the same function.

BREAK

Special label used to terminate a loop

SYNTAX

trap = **BREAK**

DESCRIPTION

Used only within the *trap* clause of a Sculptor command to immediately terminate a **for** or **while** loop or the **case** section of a **switch** statement and transfer control to the statement following the closing brace.

trap may be any valid *trap* for the Sculptor command being used.

NOTES

- Functionally equivalent to the **break** command.

Special label used to end the current iteration of
a loop

CONTINUE

SYNTAX

trap = **CONTINUE**

DESCRIPTION

Used only within the trap clause of a Sculptor command to end the current iteration of a **for** or **while** loop. The loop will continue only if its condition evaluates true.

trap may be any valid trap for the Sculptor command being used.

NOTES

- Functionally equivalent to the **continue** command.

This chapter explains the use of the screen form language and details the functions, declarations and commands available.

Contents	Page
Introduction to the screen form language	7-2
Program structure	7-2
Screen overlays	7-3
Screen fields	7-3
The scroll area	7-4
Field formats	7-4
Program options	7-5
Expressions and operators	7-6
Type conversion	7-7
Exception traps	7-8
Field lists	7-8
The key= clause	7-9
Command line syntax for cf	7-10
Command line syntax for sage	7-10
Command section contents	7-11

Introduction to the screen form language

The screen form language has been designed primarily for processing data interactively on a screen form. Although a screen form program may be written to perform without operator intervention, most screen form programs display data and allow data input.

A screen form is defined containing screen fields in which data may be input and displayed. Program options are defined which are presented on a menu line and the operator selects an option from those shown. When an option is selected by the operator, the screen form language code for that option is executed. When the code terminates, the menu line is redisplayed and the operator is again able to select an option.

The screen form language elements are specifically designed for the screen based 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 placing a form on the screen, inputting valid data, displaying formatted data 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 screen form program is written as a standard text file using an editor. The language is line-oriented and recognises these line types:

1. The first line in the program is considered the title line and will be displayed, centred, on the top line of the screen. Leave this line blank if a title is not required.
2. Lines commencing with a full stop are comment lines and are ignored by the compiler. Blank lines are also ignored (except for the title line).
3. 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 screen field definitions and declare for each screen overlay used, the fields that will be placed on that screen overlay.

5. Lines commencing with an asterisk introduce a program option and are followed by the program statements for that option.
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.

Screen overlays

Up to eight screen overlays may be defined within a single screen form program. Each screen overlay may contain screen fields and/or graphics and each may be turned on or off using the **screen** command. When the program starts, screen one is on and all others are off. When a screen overlay is turned on all fields and/or graphics associated with that screen are displayed and when it is turned off they are removed and any underlying fields or graphics are redisplayed.

The **!screen** declaration defines the screen or screens on which the fields and/or graphics which follow are to be placed.

Screen fields

All fields that are to be displayed or input must be given a screen field definition which takes the format:

```
+field_name, [ heading ] , row, column [ , format ] [ { help text } ]
```

The *field_name* may be a field from a declared **Sculptor** keyed file or a temporary field (defined with **!temp**). A field may only be defined once, but may be placed on multiple screen overlays. If the *heading* is not given, the default heading for the field is used. The heading is automatically displayed to the left of the field.

The *row* and *column* are numeric constants which define the location on the screen where the first character of the data is shown. The optional *format*, if declared, will over-ride the default format for the field.

The *help text*, if defined, will over-ride any help text previously defined for the field.

Screen fields are automatically delimited with the characters "[" and "]". This default may be over-ridden with the **!box** declaration, and the default may be changed within the **sage** program using the **lcf** program.

For alpha fields, the width of the data displayed is determined by the size of the field. For numeric fields, the width displayed is determined by the format applied.

The scroll area

If the row number for a field equals the row number in a **!scroll** declaration, the field is displayed as a column of fields equal to the depth of the scroll area, starting at row+1. The heading is displayed on row, centred above the field.

The scroll area is global to all screen overlays and provides a mechanism to display multiple copies of a field on the screen. The **scrline** special temporary field holds the line number of the current line of the scroll area. The current line may be changed using the **scroll** command.

Field formats

A field may have a format defined for it which will affect the way in which the field is displayed, 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:

e	Suppress echo of input characters
l	Force lower case on input
u	Force upper case on input
r	Remove trailing spaces on 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 applies only to fields which are input unless it is followed by a plus sign "+" in which case the format also applies when the field is assigned.

The format for a numeric field defines the size of the field shown on the screen, but does not affect the value that is stored for that field. Formats used 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 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	Valid for display only, no year shown
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

Program options

A screen form program usually includes a number of options from which the operator is invited to choose. Each program option has an associated section of program statements. An option is introduced in the program by an option header line:

```
*keys=description [ { help text } ]
```

Where *keys* may be one or two printable characters, and defines the characters required to select the option (except when **!option** is used - see page 7-47).

Each option will be shown on the option line of the screen in the order of definition in the program. The option line may be changed with the **!option** declaration.

Following an option line are the statements to be executed when that option is selected by the operator. The processing of a selected option continues until:

1. An **end** statement is encountered. This returns control to the option prompt.
2. An **exit** statement is encountered. This terminates the program.
3. An untrapped error condition occurs. A relevant message is shown and control returns to the option prompt.
4. The operator cancels the option during an input operation by pressing the CANCEL key as defined in the vdu parameter file. This returns control to the option prompt (see the **cancel** command for further details).

If the first character of *keys* is an asterisk then that option is not shown on the screen, but may still be selected by the user by typing the asterisk and the key that follows. For example,

```
**d=Delete
```

In this instance the option will not be shown, but may be selected by typing "**d" at the option prompt.

The program statements for an option must terminate with an **end** command if execution is not to fall through into the code for the next option.

A program may be created which contains no program options, all processing being performed in the initialisation code. Ensure that such a program terminates with an **exit** command.

Expressions and operators

The screen form 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
	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
bs	Backspace past first field in input command
eoi	The "End of Input" key was pressed
err	An external error occurred (error code in errno)
ni	No change to displayed data on input
no	A "no" reply to a prompt command
nrs	No record selected
nsr	No such record
re	Record exists
riu	Record in use
yes	A "yes" reply to a prompt command

Field lists

Several commands require a field list. There are two types, a screen field list and a field list. The type to use will be obvious from the context.

Screen field lists

This is a list of screen field names separated by commas and may also include ranges of screen fields indicated by a hyphen between the start and end field names of the range. A range implies all the screen fields defined in the program between (and including) the named screen fields in order of their definition. A screen field is defined by a line commencing with a plus sign "+" (see above).

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 cf

cf [-z] *filename*

Compiles the source file *filename.f* and produces the file *filename.g* if compilation is successful.

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

Command line syntax for sage

sage *filename* [*arguments*]...

Executes the compiled file *filename.g*. The arguments may be accessed within the program through the **arg** special temporary field.

CONTENTS

SCREEN FORM LANGUAGE COMMAND SECTION

Contents	Page
Language functions	7-15
String functions	7-15
Numeric functions	7-15
asc()	7-16
centre() / center()	7-17
chdir()	7-18
chr()	7-19
dim()	7-20
inchar()	7-22
instr()	7-23
keycode()	7-24
left()	7-26
power()	7-27
rand()	7-28
remove()	7-30
right()	7-31
setstr()	7-32
strlen()	7-33
sqrt()	7-34
tolower()	7-35
toupper()	7-36
Language declarations	7-37
!at	7-37
!box	7-38
!cfile	7-39
!depth	7-41
!file	7-42
!handles	7-44
!highlight	7-46
!option	7-47
!record	7-48
!screen	7-51
!scroll	7-52
!temp	7-55
!width	7-60
!zeros	7-61

Language commands	7-62
at	7-62
autocr	7-63
autogoi	7-64
autohelp	7-65
bell	7-66
cancel	7-67
chain	7-68
check	7-69
clear	7-70
clearbuf	7-72
clearbox	7-73
close	7-75
close #	7-76
decdate	7-77
delete	7-78
display	7-79
drawbox	7-80
editmode	7-82
encdate	7-83
end	7-84
error	7-85
exec	7-86
exit	7-88
find	7-89
get #	7-91
gosub	7-93
goto	7-94
hangup	7-95
highlight	7-96
hline	7-97
if	7-98
input	7-100
insert	7-106
interrupts	7-107
inputerr	7-108
let	7-109
lock	7-110
match	7-111
message	7-112
newform	7-113
next	7-114
nextkey	7-115
on	7-116
on global	7-117

on INTERRUPT	7-118
on local	7-119
open	7-120
open #	7-121
opthelp	7-122
pause	7-123
preserve	7-124
prev	7-125
prevkey	7-126
prompt	7-127
put #	7-128
read	7-130
readkey	7-131
redraw	7-132
return	7-133
rewind	7-134
rewind #	7-135
rounding	7-136
screen	7-137
scroll	7-138
skip	7-139
sleep	7-140
testkey	7-141
unlock	7-142
validhelp	7-143
vdu	7-144
vline	7-145
wakeup	7-146
write	7-147

Screen form 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
keycode()	i2 integer
power()	r8 real
rand()	i2 integer
remove()	i1 integer
strlen()	i1 integer
sqrt()	r8 real

SYNTAX

asc(*text_expression*)

DESCRIPTION

Returns the ASCII value (in the range 0-255) of the first character in the *text_expression*. If the *text_expression* 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,fx,10,35
    fx = "TEST"
    display fx           /* will display "TEST"      " */
    fx = centre(fx)
    display fx           /* will display "      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 successfully 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
  message "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 */
    ...
```

dim()

Return the number of elements in a subscripted field

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.

EXAMPLE

```
for( ctr=1 ; ctr<=dim(fieldname) ; ctr=ctr+1 ) {  
    .      scroll ctr : input fieldname  
}
```

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

```
/* this example determines the date of birth from a */
/* DVLC driver number in format SMITH611270D67FR */

!temp dvlc,Driver number,a16
!temp dob,Date of birth,d4
    year=(getstr(dvlc,6,1)+getstr(dvlc,11,1))
    month=getstr(dvlc,9,2)
    day=getstr(dvlc,7,2)
    encdate(dob)
    if dob=0 then message "Error in driver number" : exit
    ...
```

inchar()

Return the ASCII value of the next character from a sequential input channel

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 *channel* is not zero, the channel must have been opened with the **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 an 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()**.

Demonstration of inchar

```
!temp a, ,i2
!temp ctr, ,i4
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
exit
```

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* where the match was found. Character positions start at 1.

RETURN TYPE

This function returns an integer in the range 0-255.

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

This example is complete and demonstrates string searching.

```
Demonstration of Instr()
!temp a, ,a20
!temp b, ,a20
!temp s, ,i2
!temp x, ,i2
+a,Source String,10,35
+b,Search for,11,35
+s,Start at,12,35
+x,Found starting at,14,35
LOOP input a,b,s bs=QUIT
      if s=0 then s=1
      x=instr(a,s,b)
      display x
      goto LOOP
QUIT exit
```

keycode()

Await a keypress and return a code

SYNTAX

keycode(*mode*)

DESCRIPTION

The modes currently available are **0** and **1**.

keycode(0) awaits a single key press, attempts to recognise the key and returns a code according to these rules.

- If the key pressed corresponds to one of the "Sent by" sequences defined in the vdu parameter file, **keycode()** returns a negative number which is the complement of the entry number from the vdu parameter file corresponding to the key pressed. This can be used to identify special keys in a fully portable manner.
- If the key pressed was *not* recognised and was a character in the ASCII range 0-255 then the ASCII code for the character is returned as a positive number.
- If the key pressed was not recognised and did not return a single ASCII value, **keycode()** returns -1.

keycode(1) returns immediately with a positive integer which indicates the the key that was pressed to exit the last **input** command.

- The value returned will be the entry number in the vdu parameter file which corresponds to the recognised key. Again this mechanism allows full portability between vdu types.
- If there was no previous input command, **keycode()** returns -1.

NOTES

- It is common practice to negate the value returned from **keycode()** when *mode* is one, so that a standard set of manifest constants may be used to define keypress values.
- If **autocr** is on and an **input** is exited with a character keypress, **keycode(1)** will assume CR was pressed.

RETURN VALUE

This function returns an integer in the i2 range and should not be stored in an i1 field.

EXAMPLE

This example demonstrates single keypress input using **keycode(0)**.

```
!temp x, ,i2
!define CR -66
!define ESC -9
!define F1 -81
    x=keycode(0)
    switch(x) {
        case = CR:    message "CR Pressed"
                     break
        case = ESC:   message "ESC pressed"
                     break
        case = F1:    message "F1 pressed"
                     break
```

The following example demonstrates the use of **keycode(1)** to decrement a counter if the user exited the input with an UP-ARROW or to increment a counter if the user exited the input with a DOWN-ARROW or CR.

```
!temp x, ,i2
!temp ctr, ,i4
!temp a,a,a12
!temp b,b,a12
+a,,10,30
+b,,12,30
+x,Key press value,14,30
+ctr,Counter,15,30
!define UPARROW -64
INP          input a,b bs=NXT
NXT          x = -keycode(1)      /* negate so that we can use
                                   consistent !defines for mode 0
                                   and mode 1 of keycode() */

    if x=ESC then exit
    if x=UPARROW then ctr=ctr-1 else ctr=ctr+1
    display a,b,x,ctr
    goto INP
```

Note that in this example the BS trap was present but effectively ignored. Without a BS trap, the user would NOT have been able to exit the **input** statement using the UP-ARROW key.

left()**Return a string left justified****SYNTAX****left(*text_expression*)****DESCRIPTION**

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

RETURN TYPE

This function returns an alpha type.

EXAMPLE

```
!temp lj, ,a30
+a,Name,10;20
    input lj
    lj=left(lj)
    display lj
```

Raise a number to a power

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 from one execution of the program to another, the number generator must be "seeded" with a number that is fairly random at the start of the program. The **sysstime** special temp 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 $(\text{rand}(0) \% 100) + 1$ 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.

```

!temp rnd, ,r8
    interrupts on
    rnd=rand(systime)
TOP  display rnd
    rnd=rand(0)
    goto TOP
/* our only way out !!! */
/* seed the generator */
/* get next random number */

```

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 named 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.

EXAMPLE

Demonstration of Remove

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

SYNTAX

right(*text_expression*)

DESCRIPTION

Right justifies a string within its field width.

RETURN TYPE

This function returns an alpha type.

EXAMPLE

```
!temp rj,,a30  
+rj,Title,10,20
```

```
input rj  
rj=right(rj)  
display rj
```

setstr()

Store a sub-string within another string

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,
    }
prompt : exit
```

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

The following example is complete.

Demonstration of strlen

```
!temp a, ,a50
```

```
+a,String,10,30
```

```
TOP      input a bs=END  
          message "String Length = ": put strlen(a);  
          goto TOP
```

```
END      exit
```

sqrt()

Return the square root of a number

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.

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 value.

EXAMPLE

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

toupper()

Return an upper case version of a string

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 value.

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

Declare the cursor position for a graphics command

!at

SYNTAX

!at *row,column [:] graphics-command*

DESCRIPTION

Sets the cursor position as point of origin for one of the graphics commands. The *row* number starts from the top of the screen, the *column* number starts from the left hand edge of the screen. This declaration also associates the graphics command with a particular screen.

The graphics commands which may be used with **!at** are:

drawbox *depth, width [{ title }]*

hline *length, style*

vline *length, style*

Graphics commands declared in this way form part of the display of the current screen and will be displayed when the screen is turned on and removed when the screen is turned off. The current screen is that defined by the last **!screen** declaration.

When the **drawbox** command is used with **!at**, a title may be optionally specified, enclosed in braces. This title will appear centred on the top line of the box.

NOTES

- If the row or column exceeds the screen limits, the command will be ignored.

EXAMPLE

```
!screen 3
!at 7,20: hline 40,1
```

SEE ALSO

!screen, screen, drawbox, hline, vline.

SYNTAX

!box *delimiters*

DESCRIPTION

This optional statement defines non-standard characters to enclose the screen form fields. If one character is specified then it is used both to open and close each field. If two characters are specified, the first character is used to open the field and the second character is used to close it.

When no **!box** statement is included in the program, **sage** uses the default characters, normally "[]", but this default may be altered using the language configuration program **lcf**.

NOTES

- The box delimiting characters may be effectively disabled (set to spaces) by using a space between two quotation marks.
- If delimiters are set to spaces, the attributes to start and end left and right box delimiters are not issued. These may be forced by setting "**Enter 1 to force left/right box sequences**" (38) in the vdu parameter file to 1.
- Field delimiters are common to all screen overlays throughout the program.

EXAMPLES

!box " "	Disable field delimiters
!box " "	Use vertical bar delimiters
!box "< >"	Use angled brackets
!box :	Use colons as left and right
!box { }	Use braces
!box " "	Use vertical bar delimiters

SYNTAX

!cfile *file_id* [*pathname*]

DESCRIPTION

Declares a Sculptor keyed file which is closed when the program starts. See **!file** for declaring files which are initially open. The *file_id* may be alphanumeric or just numeric and is used as an identifier to refer to the file in subsequent file access commands. The *pathname* may be omitted if the *file_id* is also the file name and the file resides in the current directory. The *file_id* must not be a reserved word.

The index and data files must exist when the program is run (see the program **newkf** for creating new files) and its data dictionary file must exist when the program is compiled. **sage** will look for the file in the current working directory unless a full pathname is supplied as the filename. Both the data file and its associated index file must exist in the same directory.

It is important to note that **sage** 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 **!file** declarations.

The maximum number of files that may be declared in one program using **!cfile** 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.
- A compiler error will result if the *file_id* is not unique, is a reserved word or is used as a temporary field or line label later in the program.

- The **SAGEDATA** environment variable, if declared, is prepended, at run-time, to the name of any file declared within a screen form program.

EXAMPLE

```
!cfile stk ../data/stock  
!cfile control
```

SYNTAX**!depth** *rows***DESCRIPTION**

This optional statement defines the screen depth (number of rows) required. If no **!depth** statement is present, the default **Standard Depth** from the vdu parameter file is used.

The only effect of this statement is to cause **sage** to reconfigure the screen to a larger or smaller number of rows using the sequence **Set standard screen depth** (5) or **Set extended screen depth** (6) respectively from the vdu parameter file.

NOTES

- If the number of rows required exceeds the standard depth (set in the vdu parameter file) the vdu sequence to set extended depth is issued. If there is no extended depth set in the vdu parameter file, this declaration is ignored.
- If the rows required exceeds the extended depth, the extended depth will be set.
- This declaration is evaluated at run-time using the vdu parameter file.

EXAMPLE

!depth 25	If the standard depth is 24 and extended is 25, the vdu will be set to 25 lines.
!depth 25	If the standard depth is 25, the sequence to set standard depth is issued.
!depth 255	if the vdu has a maximum extended depth of 43, this will be set.

SEE ALSO**!width**

SYNTAX

!file *file_id* [*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_id* may be alphanumeric or just numeric and is used to refer to the file in subsequent file access commands. The *pathname* may be omitted if the *file_id* is also the file name and the file resides in the current directory. The *file_id* may not be a reserved word.

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**). **sage** will look for the file in the current working directory unless a full 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**, and **!cfile** is 32. The maximum number of these files that may be open at the same time is also 32, but is dependant on the operating system. Under MSDOS version 3.3 and above up to 32 Sculptor keyed files may be open simultaneously through the use of the **!handles** declaration (see page 7-44).

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.

- The **SAGEDATA** environment variable, if declared, is prepended, at run-time, to any file name specified.

SEE ALSO

!cfile

EXAMPLE

```
!file order
!file cust customer
!file stk ../data/stock
!file inv /usr/ord/data/invoice
```

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 STDP RN. (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. */
```

!highlight

Highlight field areas on input

SYNTAX

!highlight form | field

DESCRIPTION

This declaration causes field areas to be highlighted using the **Start form highlight** (167) and **End form highlight** (168) sequences. Either **form** or **field** may be selected.

If **form** is chosen, the limits of all displayed input field areas are highlighted. This has the effect of showing the width of all fields.

If **field** is chosen only the current input field is highlighted. As soon as input for the particular field is complete, highlighting is switched off and the field is displayed with the **Start normal data** (32) and **End normal data** (33) sequences.

For best visual effect when using highlight mode, the box delimiters may be changed by the programmer to space characters using a **!box** declaration.

NOTES

- The video attributes of the highlight are set by the relevant entry in the vdu parameter file (167 and 168).

EXAMPLE

```
!highlight form
!highlight field
```

SEE ALSO

!box

SYNTAX

!option *modifier*

DESCRIPTION

modifier has one of the following forms:

line *option-row* [,*prompt-row*]

where *option-row* and *prompt-row* are valid vdu row numbers which define the screen rows to be used to display the program options and the option prompt respectively. If the *prompt-row* is not specified, the line after the *option-row* is used to display the option prompt message.

prompt "*text*"

where *text* is the new prompt text. This changes the default prompt text of "Which option do you require".

block

sets the point-&-pick method of selecting an option.

NOTES

- If a **!option** declaration with the same modifier occurs more than once, the last declaration encountered is used.
- To suppress the prompt text completely, use **!option prompt ""**
- The *prompt-row* is also the default row for the **prompt** command.

EXAMPLE

```
!option line 20,22
!option text "Please choose one"
```

SEE ALSO

optline, opthelp

SYNTAX

!record *file_id* *pathname*

DESCRIPTION

This command is used to declare an alternative record layout for the file referred to by the *file_id* (previously declared in a **!file** or **!cfile** statement). The *pathname* identifies an alternative descriptor file created with the program **describe**. Both sets of fieldnames may be referred to in subsequent program statements. Up to eight **!record** statements may be associated with each 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 display and input the appropriate set of fields depending on the 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 **sage** 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.

Note that the **clear** command (with no arguments) initialises each file's record buffer according to the main record layout (i.e. the **!file** declaration) with alphanumeric fields being initialised to spaces and numeric fields to nulls.

NOTES

- Using an alternative record layout does not require any additional files at run-time. The descriptor (.d) file for the alternate record layout 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.

EXAMPLE

File **std** has been described (and created with **newkf**) with a record length of 6 bytes as follows:

```
Key : std_key,Key,i1
Data: std_data,Data,a5
```

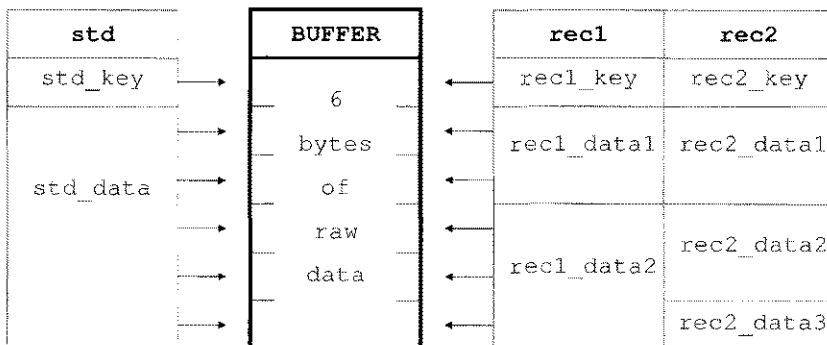
File **rec1** defines an alternative record layout for **std** and has been described as follows:

```
Key : rec1_key, ,i1
Data: rec1_data1, ,i2
      : rec1_data2, ,a3
```

File **rec2** defines an alternative record layout for **std** and has been described as follows:

```
Key : rec2_key, ,i1
Data: rec2_data1, ,i2
      : rec2_data2, ,i2
      : rec2_data3, ,i1
```

Within a program, a single record buffer is maintained for each keyed file. The main file layout and any alternate record layouts are mapped onto that buffer based on the field name used.



The only storage space that is allocated is that required for the buffer. The use of the field name will determine the method by which the bytes are accessed in that buffer. From the diagram above, using the field name `rec1_data1` will access the second and third bytes of the buffer as an `i2` value, similarly, using `std_data` will access the second through sixth byte of the buffer as an `a5` value.

Clearing `std_data` and displaying the field `rec2_data3` will show that it contains the value 32. This is because the **clear** has filled the buffer with spaces (5 of them) and the byte (`i1`) value for the space character is 32.

Alternate record layouts provide immense flexibility but require considerable care in their implementation.

SYNTAX

!screen *number* [, *number* ...]

DESCRIPTION

number may be an integer constant in the range 1-8.

Up to 8 screen overlays may be declared for use within the one **sage** program. Each overlay has access to the whole display area and may contain screen fields, graphic lines and graphic boxes.

All graphics declarations and box definitions which follow a **!screen** declaration belong to that screen and are automatically drawn when the screen is turned on or redrawn, and removed when the screen is turned off.

A numbered screen is switched on by use of the **screen** command. When a screen associated with a **!at ... : drawbox** declaration is switched on the underlying display is temporarily cleared to the extent of the graphics rectangle. The data that was within the rectangle is stored and will be re-displayed when the overlaid screen is turned off.

An error occurs if an attempt is made to execute an **input** statement for a field associated with an inactive screen. In this case the message "**Attempt to access an inactive screen**" is displayed and control returns to the prompt line.

Use of multiple screen numbers causes the graphics and screen field declarations which follow to belong to all the numbered screen overlays.

EXAMPLE

```
!screen 3
!at 5,3 : drawbox 12,10
+code,,13,12

!screen 5,7,8
!at 7,8 : drawbox 10,10
+std_name,,12,20

!screen 1,5
!at 7,8 : drawbox 10,12
```

SYNTAX

!scroll *heading_line, depth*

DESCRIPTION

Defines an area on the screen in which data is displayed in columns. The scroll area may be used to simultaneously display the values in subscripted fields or to display data from more than one record at a time; for example from a transactions file. Only one scroll area may be defined in a program.

The heading line indicates the screen line on which the field headings are to be displayed as column headings. The depth parameter specifies the number of fields required in each column. A field is included in the scroll area by giving it a line co-ordinate equal to the heading line in the **!scroll** statement.

Note that a field whose line co-ordinate exceeds the heading line but is within heading line + depth, displays as a single field, so the scroll area need not reserve the whole width of the screen.

The line within the scroll area on which data is displayed is controlled by the special temporary field **scrline**, whose value is maintained with the **scroll** command. If the value in **scrline** exceeds the depth of the scroll area, then a wrap around takes effect automatically.

NOTES

- If an array field is used without an explicit subscript, the current value of **scrline** is used to determine the array element to use.
- The scroll area will be present on all screen overlays. If a screen field in any screen has a row co-ordinate equal to *heading_line*, then it will be replicated *depth* times at it's column position.

EXAMPLE

This example is complete except for the posting section and serves to demonstrate three major points.

- The array `stk_code` is considerably larger than the depth of the scroll area so automatic array paging in the scroll area is demonstrated.
- The field `stk_ctr` does not form part of the scroll area, and is displayed to identify the array line number being edited.
- The scroll area is global to all screen overlays used, and the LOOKUP subroutine demonstrates use of the scroll area on a second screen overlay.

This example has been simplified to conserve space.

```
Stock System : Goods Received Input
.   stk_gri.f
.
.   Receives a list of stock codes and quantities in
.   stk_code and stk_qty arrays and posts them to stock file.
.

!file STK    stock                      /* lookup and posting */
!scroll 6,15
!temp ctr,,i2                          /* general counter */
!temp key,,i2                          /* key press */
!temp stk_ctr,Entry number,i2         /* current entry line*/
!temp stk_max,Last entry,i2          /* last entered line */
!temp stk_code,Stock codes,a9(100)    /* allow 100 max */
!temp stk_qty,Quantity recd.,i4(100)
!screen 1
+stk_codes,,6,6
+stk_qty,,6,20
+stk_ctr,,9,50
+stk_last,,11,50
!screen 2
!at 4,40 : drawbox 18,38
+f_stk_code,,6,48                      /* stock code from file */
+f_stk_desc,,6,60                      /* short description */
!define ESC      -9                    /* ESC entry in VDU file */

..... Program Options : I=Input, P=Post, E=Exit
*I=Input

      clear
      gosub STK_ARRAY_INPUT
      end
```

*P=Post

```
if stk_last=0 then \  
    error "No data input" : sleep 2 : end  
gosub POST          /* not included here */  
end
```

*E=Exit

```
if stk_last<>0 then \  
    prompt "Data entered! - ABANDON " no=SAFE  
exit  
SAFE      end
```

..... SUBROUTINES

STK_ARRAY_INPUT

```
stk_ctr=0 : stk_last=0 : ctr=0  
on local F1 gosub LOOKUP  
for(stk_ctr=1; \  
    stk_ctr<=dim(stk_code);  
    stk_ctr=stk_ctr+1) {  
a_LOOP      scroll1 stk_ctr  
            display stk_ctr-stk_qty  
            input stk_code, stk_qty bs=a_BS eoi=a_EOI  
            display stk_code, stk_qty  
            if strlen(stk_code)=0 or stk_qty=0 then \  
                stk_ctr=stk_ctr-1 : continue  
            if stk_ctr>stk_last then stk_last=stk_ctr  
            continue  
a_BS        stk_ctr=stk_ctr-1  
            if stk_ctr<1 then goto a_EOI  
            goto a_LOOP  
a_EOI      if stk_last<>0 then \  
                prompt "Entries complete" no=a_LOOP  
            return  
        }  
return
```

LOOKUP

```
screen 2 on  
rewind STK : ctr=0
```

L_LOOP

```
next STK nsr=L_RET : display f_stk_code,f_stk_desc  
ctr=ctr+1 : scroll1 ctr  
if ctr%16=0 then \  
    gosub GETKEY : \  
    if key=ESC then screen 2 off : return  
goto L_LOOP
```

L_RET

```
return
```

GETKEY

```
key=keycode(0) : return
```

SYNTAX

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

DESCRIPTION

Declares a temporary field for use within the program. *name* may be any valid field name. *heading* is optional and will be the heading that is used by default if the field is placed on the screen. 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 7-4).

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.

A temporary field may be declared with a *type&size* of **a0**, i.e. an alphanumeric field of zero length. If a screen field is declared for the field, the field itself is suppressed, but the heading appears to the left of the field's imaginary position and is a convenient way of displaying static textual information on the screen. Headings are not removed by a **clear** command.

NOTES

- No temporary field may exceed 32,767 bytes in size.
- When a global clear is performed, any temporary fields in screen fields are cleared but temporary fields which are not shown on the screen are not cleared.
- Field headings and formats that contain punctuation characters should be enclosed in quotes.
- The scope of a temporary field is the entire program.

Special temporary fields

The following special temporary fields are available. These are automatically declared but may be redeclared if required.

!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 : sage cust ABC 1000

```
var1 = arg[1]           /* "sage"           */
var2 = arg[2]           /* "cust"          */
scroll 3 : var3 = arg    /* "ABC"           */
var4 = arg[4]           /* "1000"          */
```

!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. It 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 inputbuf, ,a80

During an **input** command, a function key may be pressed which causes execution of a subroutine. Changes to the current input field, however, have not yet been stored in that field when control passes to the subroutine. This temporary field contains the data in the current input field at the moment the function key was pressed. It may not be altered.

!temp keyboard, ,a12

The keyboard type in use. Read from the second line of the vdu parameter file. Most commonly used to provide keyboard independence for help files by using the keyboard name as part of the help file name.

!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(s) for use with **get** and **put** commands. This field may be directly assigned. If multiple character separators are required, the separator field may be redefined.

EXAMPLE

The following example demonstrates reading a quote delimited file. The first and last quote must be discarded and the separator characters are "," (quote, comma, quote).

```
!temp separator,,a3
    separator="",""          /* note single quotes      */
    open #1,"myfile.qdf" read /* no error checking !! */
    xa=inchar(1)             /* throw away quote char */
LP   get #1,ta,tb,tc,td,te err=QUIT /* get the data          */
    xe=getstr(te,1,strlen(te)-1) /* throw away quote      */
    ...      process the data accordingly
    goto LP
QUIT  ...
```


EXAMPLE NOTES

- If the field `te` shown above was a numeric field, the last quote character would be ignored anyway.
- 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 separators 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, rather than the device name. 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 vduname, ,a12

The name of the current vdu. Read from the first line of the vdu parameter file.

The following special temporary fields are used by the **decdate** and **encdate** commands.

!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 *integer*

DESCRIPTION

This optional statement defines the screen width (number of columns) required. If no **!width** statement is present, a default of 80 columns is assumed.

The only effect of this statement is to cause **sage** to reconfigure the screen to a larger or smaller number of columns in cases where the terminal supports such a feature. If the required width exceeds the standard screen width, the extended screen width is set (if available).

The available widths for a vdu are defined in the vdu parameter file by the **Standard width** and **Extended width** entries. The code sequences to place the vdu in these widths are defined by the entries **Set standard screen width**(4) and **Set extended screen width**(3).

EXAMPLE

```
!width 132
```

SEE ALSO

!depth

SYNTAX**!zeros on | off****DESCRIPTION**

When a zero value numeric field is displayed using **display** or **highlight** or when it is redrawn, the zero value is normally shown using the format defined for the field. The **!zeros** declaration allows the display of zeros to be suppressed within a screen form program.

Setting zeros **on** is the default and will display all zero values regardless of the format applied. Setting zeros **off** will suppress display of zeros only if the format string does not force use of zeros, ie, "#####0.00".

The compiler, **cf**, or the pre-processor, **spp**, may also be used to force suppression of zeros using the **-z** switch on the command line. The **!zeros** declaration will, however, over-ride this command line switch.

EXAMPLE

This short example is complete.

```

Program Title
!temp f1, ,m4
!temp f2, ,m4
!temp f3, ,m4,##,##0.00
+f1,F1,10,30
+f2,F2,12,30
+f3,F3,14,30
!zeros off
    display f1,f2      /* will display nothing */
    f1=23
    display f1,f2,f3   /* will display 23.00 in f1
                        and 0.00 in f3 */
    prompt
    exit

```

SYNTAX

at row, column

DESCRIPTION

This command positions the cursor to the specified row and column. It may be used prior to any command which produces output that is position definable, as shown below:

- When used with the graphics commands - **clearbox** **drawbox**, **hline**, and **vline**, **at** sets the cursor start position as required.
- When used with **message** or **prompt** in a multi-statement line, it changes the default position of the text.
- When used with the **put** command to channel zero, **at** will define the cursor position to commence the output of characters. Note that any command which changes the cursor position will affect **put**.
- When used with the **exec** command, **at** causes the output from the called program to appear at a particular place on the screen.
- When used with the **vdu** command, **at** enables special use of position specific features of the terminal.

NOTES

- The default position of the option line and option prompt may be defined by the **!option** declaration.

EXAMPLES

```
at 5,20: drawbox 8,25
at 15,30: message "Updating stock file"
at 23,2: put "Press F1 for help text"
at top-1,left+2: put account_name
at 1,1
exec "sagerep stocklist qume| lpr"
at 20,1:vdu 52
```

SYNTAX

autocr on | off

DESCRIPTION

If **autocr** is **off**, all input into screen fields must be completed with the RETURN or DOWN-ARROW key.

If **autocr** is **on**, typing the last character in the screen field completes the input and the RETURN key is not required, although it can still be used to complete input to a screen field before the last character position is reached.

By default, **autocr** is **off**.

NOTES

- If **autocr** is **on**, use of the **keycode(1)** function will return the code for RETURN if a field was exited by typing the last character.

EXAMPLE

```
autocr on
autocr off
```

SYNTAX

autogoi on| off

DESCRIPTION

If **autogoi** is **off**, a **gosub** command which follows a fieldname in an **input** statement is executed only if the field is changed by the **input** statement.

If **autogoi** is **on**, a **gosub** command which follows a fieldname in an **input** statement is executed on every input to the field, whether or not its value is changed.

In either case, the **gosub** command is only executed if the cursor is moved out of the field in a forward direction. Using Backspace or UP-ARROW to exit a field does not cause execution of the subroutine.

By default, **autogoi** is **off**.

EXAMPLE

```
autogoi on
```

SEE ALSO

input

SYNTAX

autohelp on | off

DESCRIPTION

If **autohelp** is **off**, the help message attached to an input field is displayed only if the user presses the **F1** key while the cursor is in the field. If **autohelp** is **on**, the help message attached to an input field is displayed whenever the cursor is moved into the field.

By default, **autohelp** is **off**.

NOTES

- If the **F1** key has been redefined in the program, it will not operate as described above until a **clearkey F1** is encountered.

EXAMPLE

```
autohelp on
```

Send the terminal bell character

bell

SYNTAX

bell

DESCRIPTION

Outputs the vdu file entry **Bell character**(117). This is normally set to ^G (hex 07) which is the ASCII code for the bell character.

EXAMPLE

```
if s__type<>5 then bell : message "Invalid type"
```

SYNTAX

cancel on | off

DESCRIPTION

If cancel is on, the operator can abort any input operation and return to the option prompt by pressing the CANCEL key (as defined in the vdu parameter file). If cancel is off, the CANCEL key is ignored.

The default state of **cancel** is on. It should be turned off if a related series of inputs and file updates is taking place which must be completed without interruption.

NOTES

- The CANCEL key is valid only during an **input** statement and should not be confused with the keyboard INTERRUPT key which, if enabled, will be recognised at any time.
- If the **Sent by CANCEL key(7)** entry in the vdu parameter file is blank, the CANCEL key will be ignored.
- If **cancel** is **off** and interrupts are **on**, the user may still exit using the interrupt key as defined in the vdu parameter file.

EXAMPLE

This example shows a series of inputs and writes to a file during which cancel is turned off.

```
cancel off
interrupts off
scroll 1
IL1  input item - qty eoi=IL2
      insert ordlines key=ordno,scrline
      scroll: goto IL1
IL2  prompt "All items entered" no=IL1
      cancel on
      interrupts on
```

SYNTAX

chain *text_expression*

DESCRIPTION

Terminates **sage** 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. In this case the new shell or command processor must be explicitly chained, with the command to execute passed to it as a parameter.

NOTES

- Multiple commands cannot form part of a **chain** statement.

EXAMPLE

```
chain "sage ordlines " + ordno
. following called with a shell as redirection required
chain "sh -c sagerep calc pvdu >/dev/tty"
chain "command /c dir >dir.out"
```

SEE ALSO

exec, execu

check

Check that a record is currently selected for update

SYNTAX

```
check file_id [ nrs = label ]
```

DESCRIPTION

Checks that a record has been read from the specified file and not written back, cleared or unlocked. If a record is available, control passes to the next statement, otherwise the error **No record selected** is displayed and control passes to the option prompt. This error may be trapped by using the **nrs** clause, in which case control passes immediately to the label indicated.

NOTES

- **check** functions on files open for reading only, except that the record cannot be written back.

EXAMPLE

```
*a=Amend
      check cust nrs=A2
A1    input c_name - c_status
      prompt "Amendments correct" no=A1
      write cust
      clear: end
A2    message "Please supply a customer name"
      input c_name
      find cust : display c_name - c_status
      goto A1

*d=Delete
      check cust /* use the default response */
      prompt "Are you sure " no=D1
      delete cust
      clear
D1    end
```

Clear all fields, specified screen fields or the option line

clear

SYNTAX

clear [*field_list* | **optline**]

DESCRIPTION

If a *field_list* is specified, clears the screen of data displayed in those fields and re-initialises the corresponding record and temporary fields. Alphanumeric fields are set to spaces and numeric fields to zero. The *field_list* may consist of individual screen fields separated by commas, ranges of screen fields separated by hyphens or a combination of the two. Screen fields are those declared by program lines which commence with +.

Fields in the scroll area are cleared only on the row indexed by the current value of the special temp **scrline**. See **!scroll** and **scroll** for further details.

If **optline** is specified, clears the option line. The option line will be redisplayed automatically the next time the user is prompted for an option.

The **clear** command with no arguments clears all screen fields and all unprotected messages, re-initialises all record buffers (except those which have been preserved using the **preserve** command) and unlocks all locked records. In this case, although their screen fields are cleared, the data held in corresponding temporary fields is not lost. The record buffers are initialised according to the **!file** record layouts, alphanumeric fields being set to spaces and numeric fields to zero.

NOTES

- Other information, such as the current position of each file and match keys set up by the **find** command, is not destroyed by **clear**.
- If a field is not declared on a screen, it may not be cleared using **clear field_list**.

- Fields on an inactive screen may be cleared but there will be no visible effect until the screen is switched on.

EXAMPLE

```
clear  
clear optline  
clear o_custno, st_code-value, o_total
```

SEE ALSO

preserve

SYNTAX

`clearbuf file_id`

DESCRIPTION

The specified file buffer is cleared and the currently selected record, if any, is unlocked. The buffer is initialised according to the **!file** record layout, alphanumeric fields being set to spaces and other fields to zero.

NOTES

- This command does NOT affect the screen display until the next **display**, or **highlight**, when the new (cleared) values of the fields will be shown. Specifically, turning a screen **on** or replacing screen contents by turning an overlaying screen **off** does not redisplay data, and consequently may show data in the field when the buffer is actually empty.
- Commonly used when inserting to ensure that a file buffer is empty at the start of processing each record.

EXAMPLE

```
clearbuf stk
```

SEE ALSO

clear

clearbox**Clear a rectangle on the screen****SYNTAX****clearbox** *depth, width***DESCRIPTION**

Clears a rectangle on the screen which is *depth* rows deep and *width* columns across starting at the current cursor position. The **at** command may first be used to position the cursor.

This command is typically used to clear the area inside a box which has been drawn using the **drawbox** command. It may also be used to clear the entire box including its graphic border, or any other area of the screen.

EXAMPLE

```
at 5,10 : drawbox 10,30
at 8,11 : put "This text is inside the box"
at 10,12 : prompt "Okay to clear text"
at 6,11 : clearbox 8,28
at 10,12 : prompt "Okay to clear box"
at 5,10 : clearbox 10,30
```

SEE ALSO

drawbox

SYNTAX

clearkey *key-id* [, *key-id*] ...

DESCRIPTION

Disables function keys or special keys which have been programmed using the **on** command. *key-id* may be any of **F1** through **F32**, **TAB**, **BACKTAB**, **DEL_LINE**, **INS_LINE**, **SCRL_UP**, **SCRL_DN**, **PG_UP** or **PG_DN**.

If **F1** is disabled it reverts to its default use as the HELP key.

NOTES

- It is proper practice to disable a function or special key when processing statements called by that key. If the key is left active and there is an **input** statement in the routine, there is a danger of recursion. If this happens more than a few times, the program will abort with the error "Input statements nested too deeply".
- When an **on local fkey** is enabled and an **end** statement is encountered, the defined *fkey* is automatically cleared.

EXAMPLE

```
clearkey F2, F3, PG_UP
```

SYNTAX

close *file_id*

DESCRIPTION

Closes the specified keyed 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 **clear** command still operates on a closed file's record buffer but this may be prevented by using the **preserve** command.
- An attempt to close a file which is already closed is ignored.

EXAMPLE

```
close control
```

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.
- The number of files which may be concurrently opened is governed by the operating system, but is usually not less than 16.

EXAMPLE

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

SEE ALSO

!handles,close, open #

decdate**Decode a date field to day, month and year parts****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

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

Test of Month/Day of Week

```
!temp monthtext, ,a36
!temp daytext, ,a21
!temp tdate,Test Date,d4
!temp final, ,a15
!temp dow,Day of week,a3
!temp mon,Month,a3
!temp dt,Day text,a2,+00
!temp yt,Year text,a4
+tdate,,10,35

    monthtext="JanFebMarAprMayJunJulAugSepOctNovDec"
    daytext="SunMonTueWedThuFriSat"

INP  input tdate bs=QUIT : decdate tdate      /* get parts */
    dow=getstr(daytext,(((tdate%7)+1)*3)-2,3) /* day      */
    mon=getstr(monthtext,(month*3)-2,3)      /* month    */
    dt=day : yt=year
    final=dow+" "+mon+" " "+dt+" " "+yt      /* build string */
    message "Result: " : put final;
    goto INP

QUIT exit
```

SYNTAX

```
delete file_id [ nrs = label ]
```

DESCRIPTION

Deletes the currently selected record from the specified file. If no record is currently selected, then the error **No record selected** is displayed and control passes to the option prompt. This error may be trapped by using the **nrs** trap, in which case control passes to the label indicated.

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

```
*d=Delete
                                prompt "DELETE THIS RECORD: Are you sure" no=d__END
                                /* DELETE RELEVANT TRANSACTIONS FIRST */
                                find trans key=st_code nsr=d_NOTRAN
d__LOOP      delete trans : match trans nsr=d_NOTRAN
                                goto d__LOOP
d__NOTRAN    delete stk
                                clear
d__ENDend
```

SYNTAX

display *field_list* | **optline**

DESCRIPTION

Displays the current values of the specified screen fields. The field list may consist of individual field names separated by commas, ranges of field names separated by hyphens, or a combination of the two. Screen fields are those declared by program lines which commence with a plus sign "+". The display takes place in the order specified in the field list.

Using the keyword **optline** in place of a field list will display the option line. The option line is always redisplayed when an option ends.

NOTES

- If a field's contents have been changed *by assignment*, the field should be displayed prior to being input to ensure that the user is being shown the current data.
- Fields from an inactive screen may be displayed, but will not be shown until the screen is turned on (see the **screen** command).
- The vdu sequences **Start Normal Data** (32) and **End Normal Data** (33) are issued prior to and after the data is displayed.

EXAMPLE

```
*f=Find
      clear s_code
      input s_code bs=F1
      read stock
      display s_code - s_rol
      end
```

SYNTAX

drawbox *depth,width*

DESCRIPTION

Draws a rectangle on the screen which is *depth* rows deep and *width* columns across starting at the current cursor position. The cursor position is left unchanged.

depth and *width* may be constants, field names or expressions.

The rectangle is drawn using the line graphics characters which are defined in the vdu parameter file. The area inside the box is not cleared. To clear it, use the **clearbox** command.

The **clearbox** command may also be used to erase the entire box including its graphics border. A global **clear** command will not erase the border on vdu's which support protected graphics characters.

NOTES

- A drawbox is not associated with any screen overlay so will not be removed when a screen is turned off or redrawn when a screen is turned on. The **redraw** command will not redisplay boxes drawn with the **drawbox** command.
- If the drawbox is associated with a screen (using the **!at row, col : drawbox depth, width** mechanism), the box *will* be cleared when drawn and *will* be redrawn with the **redraw** command, but *depth* and *width* may only be integer constants.
- The **at** command may be used to position the cursor prior to issuing a **drawbox** command.
- A title may be defined for a box which is associated with a screen using the **!at** declaration. See **!at** for syntax details.

EXAMPLE

```
at 5,10 : drawbox 10,30
at boxrow-1,boxcol-1 : drawbox boxdepth+2,boxwidth+2
at boxrow,boxcol : clearbox boxdepth,boxwidth
```


SYNTAX

editmode *modeval*

DESCRIPTION

Sets the edit mode for a subsequent **input** command. The default edit mode is zero. *modeval* is a binary code with its bits interpreted as follows:

Bit 0

Suppress the clearing of fields on first keypress. If this bit is not set, the contents of a field are erased on the first keypress unless that key is an edit key or a cursor movement key.

Bit 1

Select insert or overwrite mode. If this bit is not set, field editing starts in insert mode. If this bit is set, field editing starts in overwrite mode.

Bit 2

Suppress the **<INS>** (insert) and **<OVT>** (overwrite) messages. If this bit is not set, either **<INS>** or **<OVT>** is displayed in the bottom right hand corner of the screen whenever editing is in progress.

Bits	Value	Mode
0 0 0	0	alphas cleared, insert, messages on
0 0 1	1	alphas not cleared, insert, messages on
0 1 0	2	alphas cleared, overwrite, messages on
0 1 1	3	alphas not cleared, overwrite, messages on
1 0 0	4	alphas cleared, insert, messages off
1 0 1	5	alphas not cleared, insert, messages off
1 1 0	6	alphas cleared, overwrite, messages off
1 1 1	7	alphas not cleared, overwrite, messages off

EXAMPLE

```
editmode 3
editmode SYS_EDITMODE
```

encdate

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 *date_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 statement execution and passes control back to the option prompt. It is permissible for statement execution to fall through an option title line into the logic for the following option. Therefore, care should be taken to include an **end** statement at the conclusion of each option, unless such continuation is intended.

NOTES

- If the program contains no options, an **end** will cause termination of the program.
- The internal subroutine stack is cleared by an **end** statement, so that any pending subroutine **returns** are properly cleared.
- An **end** statement is not required before the first option in a program if that program contains no initialisation statements.

EXAMPLE

Example of end

```
!file stk
+s_code,,10,20
+s_rol,,11,20

*n=Next
      next stk
      display s_code - s_rol
      end

*a=Amend
      etc...
```

SYNTAX

error *text_expression*

DESCRIPTION

Displays an error message in the bottom, left-hand corner of the screen. The text expression may be a string constant, an alphanumeric field or a concatenation of several such items using the + and / operators. If the value of a numeric field is required in an error message, it must first be assigned to an alphanumeric field.

The text is displayed bracketed by the **Start error message**(27) and **End error message**(28) sequences in the vdu parameter file, and is erased as soon as fresh input is received, another message is displayed or a **clear** command with no field list is given.

Certain error conditions, unless trapped, automatically display an error message and return control to the option prompt.

EXAMPLE

```
error "Sale price must exceed cost price!"
atemp = max_disc
error "Maximum discount is " + atemp + "%"
```

SYNTAX

exec[u] *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.

Before executing the command, **exec** issues the **Ignore protection and Enable Scroll**(20) and the **Reconfigure VDU**(12) sequences defined in the vdu parameter file. On regaining control, it issues the **Configure VDU**(11) and the **Honour Protection and Disable Scroll**(19) sequences. If there is a possibility that the screen form will be damaged by the program being executed, the **newform** or **redraw** command should be used to redisplay it. No vdu sequences are issued by the **execu** (execute unseen) command.

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. Failing 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/yy"
...
txt=date
exec "-sagerep putdate pvdu "+"'+txt+''"
if tstat<>0 then message "PUTDATE FAILED" : \
    prompt : exit
```

Dates are, however, best passed as day numbers as follows:

```
!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

- The termination code of a child process is contained in the **tstat** special temp.
- If a shell (or command processor) has been used to execute a program, that shell may not pass back its child process termination status.
- 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.

EXAMPLE

```
*e=Exit
prompt "Were the shown values correct " no=EX_ERR
exit
EX_ERR exit 9
```

SYNTAX

```
find file_id [ 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. If no matching key is found, the error **No such record** is displayed and control passes to the option prompt. This error may be trapped by using the **nsr** trap, in which case control passes to the label indicated.

If the located record is currently locked by another user, the message **Waiting...** is displayed and the read is retried every three seconds until successful. This status may be trapped by using the **riu** trap, in which case control passes to the label 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 file's 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:

- 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.
- If the natural key field is numeric (including dates) and the supplied data is non-zero, then that key field must match exactly.
- If the natural key field is numeric (including dates) and the supplied data is zero, then any value in that key field matches.

NOTES

- A **key=** clause cannot be used to specify values for secondary key fields only.

EXAMPLE

```
*f=Find
  clear
  input surname,firstname,dob      /* this is key order */
  find addr nsr=f__NOADDR riu=f__INUSE
```

In the above example, surname and firstname are alphanumeric and dob is a date field (date of birth).

If the user inputs part of the surname, part of the firstname and no dob, then the first record which matches the supplied parts of both surname and firstname is read, regardless of dob.

If the user inputs firstname and dob but no surname, then the first record which matches the supplied part of firstname and has the required dob is read, regardless of surname, but in this case the search may be slower, since surname is the most significant part of the key and many surnames may have to be checked until a match is found.

```
. Data file structure
. df_key1, ,a10
. df_key2, ,i2
. df_key3, ,i4
. df_data, ,a30
. To find based on df_key3 ...
  df_key1=""
  df_key2=0
  df_key3=123                      /* value to find */
  find DF nsr=fd_NOT ON FILE riu=fd_IN_USE
  display df_key1-df_data
```

If a secondary key find is performed, as above, Sculptor has to search the index file from its start until a matching key is found. The retrieval time will depend on the size of the key and the number of keys checked but will be considerably faster than a record by record, sequential search.

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.

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 valid separator is defined in the special temp **separator**. By default, **separator** is an **a1** field and has the value "," but it may be redefined to hold multiple characters, all of which then define the field separator.

The data items read are assigned to the corresponding fields in order. If there are more fields than data items read, the extra fields are given a null value. If there are more data items than fields, the extra data items are ignored.

NOTES

- The fields may be of any type. The input data is converted as required (see page 7-7 for further details).
- The separator and the end of line characters are not stored in the fields.
- 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.

- Input from the keyboard is a special case. Characters are read until a RETURN character is typed and the entire line is stored in the first field. Separator characters are treated as ordinary characters and 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 displayed and the program aborts.
- See **!temp errno** for a complete list of error codes returned.

EXAMPLE

This example demonstrates reading a text file into a **Sculptor** keyed file. Each record in the text file has this structure:

```
Surname, Forename, Group
entry1, entry2, ... (minimum 1, maximum 5)
```

The **Sculptor** keyed file has this structure:

```
Key:      vf_surname, Surname, a30
          vf_dup, Duplicate key check, i2
Data:     vf_group, Group, a12
          vf_forename, Forename, a30
          vf_ent, Entries, i4(5)
```

The **vf_dup** field is required to handle potential duplicate key values read in from the text file. The program to read this file follows:

Read In VF File

```
!file VF      vf_store
!include <errors.h>
!temp ctr, i2

interrupts on : on INTERRUPT gosub QUIT
open #3, "VF_INPUT.DAT" read err=OPENERR
LOOP
  get #3, vf_surname, vf_forename, vf_group err=READERR
  get #3, vf_ent[1], vf_ent[2], vf_ent[3], \
        vf_ent[4], vf_ent[5] err=READERR
  vf_dup=-1
  TRYINSERT
    vf_dup=vf_dup+1 : insert VF re=TRYINSERT
  clearbuf VF
  goto LOOP
OPENERR
  at 23,1 : put "Error ";errno;" opening file"
  sleep 3 : exit
READERR
  if errno=FEOF then exit
  at 23,1 : put "Error ";errno;" reading file"
  sleep 3 : exit
QUIT
  prompt "INTERRUPT : do you wish to quit " no=RET
  exit
RET
  return
```

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

```
*f=Find
    clear
    input item bs=F1
    find item
    gosub DISP
F1
    end

*n=Next
    clear
    next stk
    gosub DISP : end

DISP
    display item - rol
    if stklev < rol then error "Below re-order level"
    return
```

SYNTAX

goto *label*

DESCRIPTION

Transfers control to the statement at the line indicated. It is permissible to jump from the code in one option to the code in another 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 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

```
*f=Find
      clear
      input item
      find stk
      goto DISP

*n=Next
      clear
      next stk
DISP display item - rol
      end
```

SYNTAX

hangup on | off

DESCRIPTION

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

If **hangup** is **on** and **sage** 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 **cllocal** option of the Unix command **stty**.

EXAMPLE

```
hangup off
```

SYNTAX

highlight *field_list*

DESCRIPTION

Positions the cursor at the first character position in each screen field specified in the field list and sends the **Start Highlight data** (36) sequence defined in the vdu parameter file. The data is then displayed and the sequence defined as **End Highlight Data** (37) is sent. The command may be used to highlight specific data values on the screen.

This command will only work if the vdu terminal is capable of supporting it. Some terminals do not have the ability to highlight areas of the screen. On terminals which use embedded attributes, the use of **highlight** may cause the data to be shifted by one character.

NOTES

- The **highlight** command is similar to the **display** command, but uses the entries **Start highlight data** (36) and **End highlight data** (37) from the vdu parameter file to display the data.

EXAMPLE

```
highlight st__stklev,st__rol
```

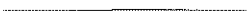









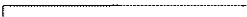

SYNTAX

hline *len, style*

DESCRIPTION

Draws a horizontal line at the current cursor position of length *len* characters using the graphics characters defined in the vdu parameter file. *style* is a parameter in the range 0-16 which defines the style of the line as follows.

- 0** Draw a blank line using spaces. Often used to remove an existing line.
- 1-12** Draw a thin line with start and end characters according to the following table. If the start and end characters do not match this table, check that the graphics characters are correctly defined in the vdu parameter file.

1		7	
2		8	
3		9	
4		10	
5		11	
6		12	

- 13** Draw a line using block graphics character 1 in the vdu file.
- 14** Draw a line using block graphics character 2 in the vdu file.
- 15** Draw a line using block graphics character 3 in the vdu file.
- 16** Draw a line using block graphics character 4 in the vdu file.

EXAMPLE

```
at 12,20 : hline 30,4
```


Conditionally execute a statement

if

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 following 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
bw	begins with (alpha only)
ct	contains (alpha only)
and	logical and
or	logical or

EXAMPLE

```
if costpr<=salepr then error "Sale price must exceed cost"

if d_date > date and (cat = "A" or cat = "B") then \
    status = 1 : goto D50

if options ct "A" then \
    gosub OPTA : display flda \
else if options ct "B" then \
    gosub OPTB : display fldb \
else \
    clear flda, fldb
```

SYNTAX

```
input field_list [ bs=label ] [ eoi=label ] [ ni=label ]
```

DESCRIPTION

Positions the cursor to each field in the *field_list* in turn and accepts input from the user. If the input is valid, the data is assigned to the corresponding record or temporary field. If the input is not valid, the bell is sounded and re-input is awaited. The *field_list* may consist of individual screen fields separated by commas, ranges of screen fields separated by hyphens or a combination of the two. Screen fields are those declared by program lines which commence with a plus sign "+".

Input proceeds in the order specified in the *field_list*. A range of fields *fieldm-fieldn* indicates all screen fields declared in the program between *fieldm* and *fieldn* inclusive. The **input** command may only reference fields on active screens. Any attempt to input to a field on an inactive screen causes the error **Attempt to access an inactive screen** to be displayed and control passes to the option prompt.

Input into fields in the scroll area takes place on the row indexed by the current value of the special temp **scrl**ine. See **!scroll** and **scroll** for further details.

Input data is validated firstly according to the data type of the field and secondly against any validation list associated with the field. Date fields are fully checked for date validity.

There is an option to attach a subroutine to an individual field in the *field_list*. This is done by appending a **gosub** clause to the field name, as in the following example:

```
input    field1 gosub CHECK_FIELD1, \
         field2-field5, \
         field6 gosub CHECK_FIELD6, \
         field7
```

Note that the subroutine cannot be attached to a range of fields, only to individual fields.

By default, a field subroutine is called only if the input changes the content of the field. The **autogoi** command may be used to force a field subroutine to be called whether or not the field content changes. (See **autogoi** for details.) Automatic validation is performed before the field subroutine is called.

On return from a field subroutine, input proceeds with the next field in the list unless either an **inputerr** command or a **skip** command was called within the subroutine. An **inputerr** command informs Sculptor that the input data is not valid and must be re-input. A **skip** command instructs Sculptor to move to another field in the input list. It is possible to skip forwards or backwards. See **inputerr** and **skip** for details.

A field subroutine may be used to further validate the input data or for some other purpose such as reading and displaying the description of a code. It is also possible to execute another **input** command within a field subroutine. A maximum of seven **input** commands may be nested at any one time. In such nests, **inputerr** and **skip** commands only affect the active **input** command.

The **ni=label** clause specifies a label to which control is transferred if no fields are changed by the **input** command and if, prior to the **input** command, a **display** command was used to display data in each field. The **ni** trap will not occur if no **display** command has followed the last **clear** command.

Various editing keys are available during input. In this manual, the full name of these keys is used. The actual key assigned to each task is defined in the vdu parameter file in use.

BACKSPACE

Deletes the character to the left of the cursor and moves all characters to the right of the cursor one position left. If the backspace key is used at the beginning of a field, any previous content of the field is redisplayed and input reverts to the preceeding field in the input list. If the backspace key is used at the beginning of the first input field and a **bs=label** clause is present, control passes to the line indicated, otherwise the bell is sounded and the cursor does not move.

DELETE CHARACTER

Deletes the character under the cursor and moves all characters right of the cursor one position left. On the PC keyboard this is normally the DEL key.

DELETE TO END OF FIELD

Erases from the current cursor position to the end of the field. This key is normally ^Z.

RETURN

The RETURN key is used to terminate the input in each field. The data in the field will be validated unless a **display** command was previously used to display data in the field and no attempt has been made to change the data. If **autocr** is on, the effect of typing the last character in a field is the same as pressing the RETURN key. (See **autocr** for details.)

END OF INPUT (EOI)

The EOI key has the same effect as the RETURN key unless the **eoil=label** clause is present, in which case control passes to the line indicated. When this clause is present, the EOI key will not function if the data in the field has been changed. This ensures that the data is properly validated. On the PC keyboard this is normally the ESC key.

CANCEL

If cancel is on (see **cancel**) and the user presses the CANCEL key, an automatic **clear** command is executed and control passes to the option prompt. If cancel is off, the CANCEL key has no effect. This key is normally ^X.

TOGGLE INSERT/OVERTYPE MODE

Pressing this key switches the editing mode between insert mode and overtype mode. When a character is typed in insert mode, existing characters which are under and right of the cursor are pushed right. If a character is pushed past the end of the field it is lost. When a character is typed in overwrite mode, it overwrites the existing character under the cursor. The current mode is shown by the INS/OVT indicator in the bottom right corner of the screen. The **editmode** command may be used to turn this off. On the PC keyboard, this is normally the INS key.

UP ARROW

The UP ARROW key moves the cursor to the previous field in the input list and, if the field is not numeric, maintains the position of the cursor relative to the start of the field.

DOWN ARROW

The DOWN ARROW key moves the cursor to the next field in the input list and, if the field is not numeric, maintains the position of the cursor relative to the start of the field. Validation and field subroutine calls are performed as with the RETURN key.

LEFT ARROW

Moves the cursor one character position left without erasing a character. If used at the beginning of a field, moves the cursor to the previous field in the input list.

RIGHT ARROW

Moves the cursor one character position right without erasing a character. The cursor cannot be moved out of the field.

NOTES

- If a **skip** command is used prior to using **input**, the skip will take effect immediately the next **input** command is executed.
- If, during input, a function key is used to call a subroutine, the content of the current input field will not have been stored but the current input data may be found in the special temp **inputbuf**. This data may be examined but not changed.
- When a subroutine is executed from a function key during input, the current scroll line number is automatically saved and is restored when the subroutine returns.
- Use of the Left-Arrow or Up-Arrow key to exit an input field will cause the BS trap to be executed if the field is the first in the field-list. Use of the Backspace key to exit the first field will always result in the previous field contents being shown in the field. Use of the Left-Arrow or Up-Arrow key however, may result in a BS trap with changed field contents. Please bear this in mind when processing the BS trap.

- Turning a screen off within a subroutine when fields which appear on that screen are being input will give the error "Attempt to access inactive screen" when the subroutine returns. Control will then return to the option prompt.

EXAMPLE

```
input st_code, st_desc-st_costpr, \
    st_eoq-st_r01 bs=IN10 eoi=IN90
display st_stklev
input st_stklev ni=NO_CHANGE
input name-addr, cy_code gosub DISPLAY_COUNTRY
```

The following example shows an input routine which makes use of scroll areas and subroutine validation. This example assumes the files referenced exist, but is otherwise complete.

Example INPUT program

```
.    input.f

!file CUST customer          /* customer details */
!file CAT  category         /* category descriptions */
!scroll 13,5                 /* for customer address */
!temp ctr, ,i2              /* general counter */
!option block               /* pick & point option line */

+cust_key,,8,20
+cust_name,,10,20
+cust_address,,13,20        /* cust_address is an a30(5) */
+cust_cat,,20,20
+cat_name,,20,35           /* name lookup from cat file */
    end

*i=Insert    { Insert a record onto the customer file }
            clear : autogoi on
i_GETKEY    input cust key bs=KWIT eoi=KWIT
            testkey CUST nsr=i_NAME : goto i_GETKEY
i_NAME      input cust_name bs=i_GETKEY eoi=KWIT
            for( ctr=1; ctr<=dim(cust_address); ctr=ctr+1 ) {
i_LOOP      scroll ctr
            input cust_address bs=i_LOOPUP eoi=KWIT
            continue
i_LOOPUP    ctr=ctr-1 : if ctr=0 then goto i_NAME
            goto i_LOOP
            }
            input cust_cat \
            gosub CHECK_CAT \
            eoi=KWIT bs=i_LOOPUP

            prompt "Is this record correct  " no=i_GETKEY
            write CUST
KWIT        autogoi off : clear : end
```

```
CHECK_CAT  read CAT key=cust_cat nsr=cc__NOCAT
cc_RET      display cat name : return
cc__NOCAT   cat_name="NOT FOUND" : inputerr : goto cc_RET
```

```
*e=Exit      { Exit this program }
              exit
```


SYNTAX

```
insert file_id [ 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. The key must be unique. If a record having the supplied key already exists, then the error **Record exists** is displayed and control passes to the option prompt. 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 data in the file's natural key fields 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 in a program will cause the keyed file to be opened in update mode.
- If the file is not currently open, an operating system error will occur.
- 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 in chapter 10).

EXAMPLE

```
*i=Insert
      input c_name - c status
      insert cust re=I1
      message "New customer recorded"
      end
I1    error "Customer already on file"
      end
```

SYNTAX

interrupts on | off

DESCRIPTION

If interrupts are on and **sage** 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, **sage** 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

inputerr

DESCRIPTION

The **inputerr** command is meaningful only when executed in a subroutine called by a **gosub** command in an **input** statement. At all other times an **inputerr** command is ignored.

Executing **inputerr** indicates that the data in the current input field is not valid. When the validation subroutine returns to the **input** statement, the cursor will not move forward out of the current input field until a new value has been entered which does not cause the validation subroutine to execute an **inputerr** command.

EXAMPLE

```
...  
input stkcode gosub CHECK_STKCODE, stkdsc.  
...
```

```
CHECK_STKCODE  
    testkey stock nsr=BAD_STKCODE  
    return  
  
BAD_STKCODE  
    error "No such stock code"  
    inputerr  
    return
```

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 7-7). 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 or logical expression, or part 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
```

SYNTAX

lock *file_id* [*riu=label*]

DESCRIPTION

Any existing record on the file which is locked by this program is first unlocked. The command then attempts to place a read lock on the entire file. If the lock succeeds, execution continues with the next statement.

If any record on the file is currently locked by another process, the error **Record in use** is displayed and control passes to the option prompt. This error may be trapped by 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 error **No record selected** will occur.
- On VMS, the **lock** command is ignored.

EXAMPLE

```
lock transfile riu=BUSY
```

SYNTAX

```
match file_id [ nsr = label ] [ riu = label ]
```

DESCRIPTION

Unlocks any existing locked record on the file then reads 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, the error **No such record** is displayed and control passes to the option prompt. This error may be trapped by using the **nsr** trap, in which case control passes to the line indicated. An unsuccessful match leaves the record buffer unaltered.

If the located record is currently locked by another user, the message **Waiting...** is displayed and the read is retried every three seconds until successful. This status may be trapped by using the **riu** trap, 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.

NOTES

- A **match** operates on the key supplied to the last **find** command on the relevant file.
- If a **find** command on the file has not previously been performed, **match** will not find a record.

EXAMPLE

```
match cust nsr=DONE  
match stock
```

SYNTAX

message *text_expression*

DESCRIPTION

Displays a message in the bottom, left-hand corner of the screen. The text expression may be a string constant, an alphanumeric field or a concatenation of several such items using the **+** and **/** operators. If the value of a numeric field is required in a message, it must first be assigned to an alphanumeric field.

If **message** is preceded by an **at** command in a multiple statement, the message is displayed at the current cursor position instead of the bottom left hand corner of the screen. A message on the message line remains displayed until another message or error message is issued or a **clear** command with no field list is executed. If **autohelp** is **on**, messages are cleared when an **input** statement is executed.

NOTES

- Messages placed on the screen do not form part of the declared data for that screen and are not redrawn when the screen is redrawn or removed when the screen is turned off.

EXAMPLE

```
message "New record inserted for " + c_name
message "Updating sales ledger..."
message "" (Clear the current message)
at 8,40: message "This is line 8 column 40"
at 12,10: vdu 50: \
           message "This is at line 12 column 10":vdu 51
```

SEE ALSO

redraw, at, put, error

SYNTAX

newform

DESCRIPTION

Clears the screen and re-displays the background screen form. This command is useful if an **exec** statement has been used to call a program which may have destroyed the screen form.

NOTES

- This command does not redisplay graphics or data field contents. The **redraw** command may be used to redisplay the screen form, data and associated graphics.

EXAMPLE

```
exec "sage invoice"  
newform
```

SEE ALSO

redraw

SYNTAX

```
next file_id [ nsr = label ] [ riu = label ]
```

DESCRIPTION

Unlocks any existing locked record on the file and 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, the error **No such record** is displayed and control passes to the option prompt. 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 message **Waiting...** is displayed and 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 **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

```
*n=Next
  clear
  next cust riu=N1
  display c_name - c_status
end

N1  error "Record in use" :end
```

SYNTAX

nextkey *file_id* [**nsr = label**]

DESCRIPTION

Unlocks any existing locked record and 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. 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, the error **No such record** is displayed and control passes to the option prompt. This error may be trapped by using the **nsr = label** clause, in which case control passes to the line indicated.

EXAMPLE

```
*s=Search
      message "Enter known part of name"
      input target bs=S9 :rewind cust
S1    message "Searching..."
S2    nextkey cust nsr=S8

      if surname ct target then goto S3
      goto S2
S3    message "": display surname
      prompt "This one" no=S1:read cust
      end
S8    message "End of file reached"
S9    end
```

Configure function keys to select program options

on

SYNTAX

on *key_id* **option* [, *key_id* **option*] ...

DESCRIPTION

The **on** command configures one or more function keys to select program options.

key_id is a function key code in the range **F1-F32**.

**option* is a program option code.

Once configured, the option can be selected by pressing the function key whenever the program is at the option prompt. Pressing the key at other times has no effect.

A function key remains configured until the program exits, it is reconfigured for some other purpose or a **clearkey** command is used to disable it.

EXAMPLE

In this example, pressing F10 at the option prompt will cause the program to exit.

```
on F10 *e
*e=exit
    exit
```

SYNTAX

on global *key_id* **gosub** | **goto** *label* [, ...]

DESCRIPTION

The **on global** command configures one or more function keys to call a subroutine or to transfer control to a specified line. Once configured, a key remains active until the program exits, it is reconfigured for some other purpose or a **clearkey** command is used to disable it. *key-id* may be any of F1-F32, **BACKTAB**, **DEL_LINE**, **INS_LINE**, **PGDN**, **PGUP**, **SCRL_DN**, **SCRL_UP** or **TAB**.

gosub *label* is a subroutine to be called when the key is pressed.

goto *label* is a line to which control is to be transferred when the key is pressed.

The key is effective whenever the program is waiting for input in an **input** command. Pressing the key at other times has no effect. If the key calls a subroutine, the **input** command continues at the point it was interrupted when the subroutine returns.

NOTES

- Any changes to the current input field will not have been stored when a function key is pressed. The special temp **inputbuf** contains the contents of the current input field at the moment the function key was pressed.
- It is proper practice to disable a function key when processing statements called by that key. If the key is left active and there is an **input** statement in the routine, there is a danger of recursion. If this happens more than a few times, the program will abort with the error "Input statements nested too deeply".

EXAMPLE

```
on global F2 gosub DO_CUST, F3 gosub DO_ITEM, F10 goto EXIT
```

SYNTAX

on INTERRUPT *gosub label*

DESCRIPTION

The **on INTERRUPT** command specifies a subroutine to be called when a keyboard interrupt is received. Interrupts can only be received if they have been enabled - see the **interrupts** command.

When a keyboard interrupt occurs, **sage** first completes the processing of the current line, which may be a multiple statement line, then calls the specified subroutine. When the subroutine returns, processing continues with the next statement.

An **input** statement can be interrupted whilst it is waiting for input. In this case, input continues from the point where it was interrupted when the subroutine returns.

NOTES

- On Unix type systems, the keyboard interrupt can be redefined by changing the entry **Interrupt Code**(174) in the vdu parameter file.

EXAMPLE

```
on INTERRUPT gosub PROCESS_INTERRUPT
```

SYNTAX

on local *key_id* **gosub** | **goto** *label* [, ...]

DESCRIPTION

The **on local** command configures one or more function keys to call a subroutine or to transfer control to a specified line. Once configured, a key remains active until the current option ends, the program exits, it is reconfigured for some other purpose or a **clearkey** command is used to disable it. When an **end** is encountered, the on local keys are cleared.

key-id may be any of F1-F32, **BACKTAB**, **DEL_LINE**, **INS_LINE**, **PGDN**, **PGUP**, **SCRL_DN**, **SCRL_UP** or **TAB**. With **gosub**, *label* is a subroutine to be called when the key is pressed. With **goto**, *label* is a line to which control is to be transferred when the key is pressed.

The key is effective whenever the program is waiting for input in an **input** command. Pressing the key at other times has no effect. If the key calls a subroutine, the **input** command continues at the point it was interrupted when the subroutine returns.

NOTES

- Any changes to the current input field will not have been stored when a function key is pressed. The special temp **inputbuf** contains the contents of the current input field at the moment the function key was pressed.
- It is proper practice to disable a function key when processing statements called by that key. If the key is left active and there is an input statement in the routine, there is a danger of recursion. If this happens more than a few times, the program will abort with the error "Input statements nested too deeply".

EXAMPLE

```
on local F2 gosub DISPLAY_NAME, F3 gosub DISPLAY_PHONE
```

SYNTAX

open *file_id*

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 program's 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 but note that the **clear** command still operates on a closed file's buffer unless the **preserve** command has been used.

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. An attempt to open a file which is already open is ignored.

EXAMPLE

open stock

SEE ALSO

!handles, **close**, **open #**

SYNTAX

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

DESCRIPTION

Opens a sequential file on *channel*, which must be a numeric expression in the range 1-32. *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 but if it does exist, the file pointer will be positioned at the end of the file.

If a file needs to be open to read and write, 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.

NOTES

- The number of sequential files which may be opened concurrently is governed by the operating system, but is usually not less than 16.

EXAMPLE

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


SYNTAX

opthelp on | off

DESCRIPTION

Help text can be attached to an option definition by including the required wording within { } on the program line that defines the option. This help text is available if a block style option line is being used (**!option block**).

If **opthelp** is **on**, any help text which is attached to the currently highlighted option is displayed on the message line.

If **opthelp** is **off**, the help text is not displayed. This is the default state.

NOTES

- If **opthelp** is **on**, the message line will be cleared when an option ends, whether or not there is any help text to be displayed.

EXAMPLE

```
!option block
    opthelp on
*i=insert {Enter a new stock record} /* will be shown */
```

pause

Suspend the program and wait for an alarm interrupt

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

SYNTAX

preserve *file_id*

DESCRIPTION

Stops the global **clear** command (with no field list) from clearing the specified field's record buffer. Takes effect for the rest of the program.

The preserve command may be applied to both open and closed files.

EXAMPLE

preserve control

SYNTAX

```
prev file_id [ nsr = label ] [ riu = label ]
```

DESCRIPTION

Unlocks any existing locked record and 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, the error **No such record** is displayed and control passes to the option prompt. 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 message **Waiting...** is displayed and 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 operating systems, the **record in use** status can only occur if the file is open in update mode.

NOTE

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

EXAMPLE

```
*p=Previous
      clear
      prev cust riu=N1
      display c_name - c_status
N1      end
```

SYNTAX

prevkey *file_id* [**nsr** = *label*]

DESCRIPTION

Unlocks any existing locked record and 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, the error **No such record** is displayed and control passes to the option prompt. 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 stock nsr=BOF
```

prompt

Prompt for yes / no reply

SYNTAX

prompt [*text expression* [**no** = *label*] [**yes** = *label*]]

DESCRIPTION

By default the display of the text expression is centralised beneath the menu line. The line on which it appears may be changed by previously using the **at** command on the same multi-statement line as the **prompt** command. The text will have **(y/n)?** appended and a valid reply is awaited. Only an upper or lower case **y** or **n** is accepted.

The **no** = *label* and **yes** = *label* clauses enable the programmer to designate the next line to be executed according to the operator's reply. If the particular reply is not trapped, execution continues with the next program statement.

If the **prompt** command is used alone (without any parameters), it displays nothing but awaits a key press before continuing with the next statement.

The language configuration program **lcf** may be used to alter the **(y/n)?** text in **sage** itself. If this is done, note that the characters replacing **y** and **n** must be in exactly the same place in the text.

EXAMPLE

```
*d=Delete
  check cust
  at 5,15:prompt "Are you sure" no=D1
  delete cust
  message "Customer deleted. Press a key"
  prompt
  clear
D1  end
```

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 (normally the screen).

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. By default, **separator** is an **a1** field and has the value ",". It may be assigned a new value or redefined as a wider field. 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.

NOTES

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

EXAMPLE

This short program creates a standard merge file for the Microsoft Word™ v4.00 word processor. This file format requires a header record defining the field names to be used, and data records separated by CR/LF. A comma is used as the field delimiter.

As the stored data itself may contain commas, the text fields are enclosed in quotes in this example. The fields to be output should be stripped of trailing spaces before being enclosed in quotes. This is done here by applying the `r` format modifier.

Create MS-WORD 4.00 merge file

```
.
.   wordmrge.f
.
.   Creates MSWORD v4.00 merge file
```

```
!file ADD address
```

```
!temp ctr,,i4
```

```
!temp q,,a1
```

```
+ctr,Records Output,10,35
```

```
.   The r modifier (strip trailing spaces) is forced here
.   by placing fields on a dummy screen. This could also
.   have been achieved using the describe program
```

```
!screen 8
```

```
+add_name,,5,30,r
```

```
+add_add1,,7,30,r
```

```
+add_add2,,8,30,r
```

```
+add_add3,,9,30,r
```

```
+add_pc,,10,30,r
```

```
q=''' : message "Processing"
```

```
open #1,"MERGE.DAT" write err=OPENERR
```

```
gosub HEADER
```

```
LOOP ctr=ctr+1 : display ctr
```

```
next ADD nsr=DONE
```

```
put #1,q;add_name;q,q;add_add1;q, err=WRITERR
```

```
put #1,q;add_add2;q,q;add_add3;q, err=WRITERR
```

```
put #1,q;add_pc;q,add_spent err=WRITERR
```

```
goto LOOP
```

```
HEADER      put#1,"name,street,town,county,postcode,spent"
             return
```

```
DONE        close #1 : exit
```

```
OPENERR     at 23,1 : put "Error ";errno;" opening file ";
             close #1 : prompt : exit 9
```

```
WRITERR     at 23,1 : put "Error ";errno;" writing file ";
             close #1 : prompt : exit 9
```


SYNTAX

```
read file_id [ key = field list ] [ nsr = label ] [ riu = label ]
```

DESCRIPTION

Unlocks any existing locked record on the file and then reads the record whose key exactly matches the supplied key. If no matching key is found, the error **No such record** is displayed and control passes to the option prompt. This error may be trapped by using the **nsr = label** clause, in which case control passes to the line indicated. An unsuccessful **read** leaves the record buffer unaltered.

If the requested record exists but is currently locked by another user, the message **Waiting...** is displayed and 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 data in the file's natural key fields is used as the key. If the **key=** clause is present, a key is constructed using data from the named fields.

EXAMPLE

```
*f=Find Item
  input st_code.
  read stk_nsr=F1
  riu=F2
  ...
F1  error "Item not recorded": end
F2  error "In use - please try later":end

... Another example
.Input order number and read details
  input o_ordno
  read orders
.Now read customer name and address
  type = "C"
  read cust key=type,o_custno
...
```

SYNTAX

readkey *file_id* [**key** = *field list*] [**nsr** = *label*]

DESCRIPTION

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, the error **No such record** is displayed and control passes to the option prompt. This error may be trapped by using the **nsr** = *label* clause, in which case control passes to the line indicated.

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 data in the file's natural key fields is used as the key. If the **key**= clause is present, a key is constructed using data from the named fields.

EXAMPLE

```
.Position file to read first "TT" item
    st_code = "TT"
    readkey stk nsr=N1
N1    next stk nsr=N5
    .
    .
    .
```

Redraw the screen display.

redraw

SYNTAX

redraw

DESCRIPTION

Clears the screen and re-displays all currently active screens and all graphics displayed by graphics declarations (**!at ... drawbox**, etc).

Graphics displayed using graphics commands and text displayed with the **put** command is not redisplayed. Text output with the **message** command is redisplayed only if it is on the standard message line.

EXAMPLE

```
...
    exec "sysadm users -g " + t_uname
    redraw
...
```

return

Return from a subroutine

SYNTAX

return

DESCRIPTION

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

NOTES

- All pending **returns** are maintained on an internal stack and a "Stack overflow" error will occur if subroutines are nested more than 90 levels deep.
- An **end** command clears the internal stack and thus clears any pending **return** statements.

EXAMPLE

```
*f=Find
  clear
  input item bs=F1
  find item
  gosub DISP
F1  end

*n=Next
  clear
  next stk
  gosub DISP
  end

DISP display item - rol
  if stklev < rol then\
    error "Below re-order level"
  return
```

SYNTAX

rewind *file_id*

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 trans
```

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 and only the integer part is used. This is the default state with rounding **off**. With rounding **on** the fractional part is rounded to the nearest integer by adding 0.5 prior to truncating.

For example, if the value 5.6 is assigned to an integer field, the result is 5 if rounding is off 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.

By default, **rounding** is **off**.

EXAMPLE

```
rounding on
real = 5.6
int = real
display int          /* will display 6 */

rounding off
int=real
display int          /* will display 5 */
```

SYNTAX

screen *number* **on** | **off**

DESCRIPTION

Switches the specified screen overlay on or off. *number* is a screen identifier in the range 1-8 as defined in a **!screen** declaration. (Fields and graphics declarations which do not follow an explicit **!screen** declaration are deemed to be part of screen 1.)

When a screen is switched on, all fields, values and graphics which are part of that screen are displayed. This will erase other data in the same area of the screen.

When a screen is switched off, all underlying fields, values and graphics which were erased when the screen was switched on are redrawn.

When the program starts, screen 1 is on and all other screens are off. See the **!screen** declaration for further information.

EXAMPLE

```
screen 2 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 1).

The value in **scrline** is the default index value for all subscripted fields which are not explicitly indexed when referenced. It is also the row within the scroll area on which **clear**, **display** and **input** commands operate on screen fields which are displayed in the scroll area.

If the value in **scrline** exceeds the depth of the scroll area or exceeds the dimension of a subscripted field, a wrap around takes effect.

NOTES

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

EXAMPLE

```
*i=input
IN10 input ordno bs=IN90
    scroll 1
IN20 input st_code, st_qty bs=IN40 eoi=IN30
    if (scrline < MAXLINE) then scroll : goto IN20
IN30 prompt "All correct" no=IN20
    gosub STORE_ORDER
    clear : end
IN40 if (scrline > 1) then scroll -1 : goto IN20
    goto IN10
IN90 prompt "Abandon this order" no=IN10
    clear : end
```

SYNTAX

skip *number*

DESCRIPTION

Skips the specified number of fields during an **input** command. *number* may be a constant or an expression.

A **skip** command may be executed within a subroutine called from within an **input** statement or may be executed prior to an **input** statement. If used within an **input** statement, **skip** operates immediately the subroutine returns to the **input**. If used prior to an **input**, **skip** takes effect as soon as the next **input** statement is encountered.

If *number* is positive, the current position in the field list of the **input** command is moved forward by the specified number. A **skip 1** takes input to the next field. A **skip 2** misses out the next field. If **skip** would move the input beyond the last field, the **input** command terminates normally.

If *number* is negative, the current position in the field list is moved backward the specified number of fields. A **skip -1** takes the input to the previous field. If **skip** would move the input beyond the first field, the **input** command executes a **bs** (backspace) trap if there is one, otherwise it stops on the first input field.

EXAMPLE

```
input type gosub CHECK,value,description
...
CHECK    if type=2 or type=3 then skip 2
         return
```

SEE ALSO

input

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

A typical use of the **sleep** command is to give the operator time to read a message before clearing the screen:

```
message "Order deleted"
sleep 4
clear
end
```

SYNTAX

```
testkey file_id [ 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, the error **No such record** is displayed and control passes to the option prompt. This error may be trapped by using the **nsr = label** clause, in which case control passes to the line indicated.

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 data in the file's natural key fields is used as the key. If the **key=** clause is present, a key is constructed using data from the named fields.

EXAMPLE

Typical use is in an insert option to test if the record already exists prior to inputting all the data fields:

```
*i=Insert
I1   input surname,firstname bs=I9
      testkey cust nsr=I2
      error "Customer already recorded"
      goto I1
I2   input addr1 - status bs=I1
      insert cust message "New customer inserted"
I9   end
```

SYNTAX

unlock *file_id*

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**).

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.

EXAMPLE

```
*f=Find
    input flight_no
    read resv
    gosub DISP
    prompt "Hold" yes=F1
    unlock resv
F1    end
```

validhelp

Enable or disable validation list as help message

SYNTAX

validhelp on | off

DESCRIPTION

If **validhelp** is **on** and input data fails to pass the validation list attached to the field, the validation list is automatically displayed as a help message.

If **validhelp** is **off**, the validation list is not available as a help message.

By default, **validhelp** is **off**.

EXAMPLE

```
validhelp on
```

SEE ALSO

autohelp

SYNTAX

vdu *numeric_expression*

DESCRIPTION

This command sends control sequence number **n** from the vdu parameter file. The cursor can be positioned using the **at** command before the vdu control sequence is sent.

The control sequences in the vdu parameter file are numbered from 1 upwards, starting with the sequence **Position Cursor**. The **vdu** command can be used to send any of these sequences, but some sequences, such as **Position Cursor**, are meaningless unless accompanied by additional output data.

The sequences from **44** to **59** are available for any purpose, the sequences from **126** to **134** are available for attribute control specifically, and the sequences from **154** to **164** are available for user graphics characters.

EXAMPLE

```
at 7,28: vdu 59
```








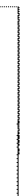




SYNTAX

vline *len, style*

DESCRIPTION

Draws a vertical line at the current cursor position of length *len* characters using the graphics characters defined in the VDU parameter file. *style* is a parameter in the range 0-16 which defines the style of the line as follows.

- 0** Draw a blank line with spaces. May be used to remove an existing line.
- 1-12** Draw a thin line with start and end characters according to the following table. If the start and end characters do not match this table, check that the graphics characters are correctly defined in the VDU parameter file.

											
1	2	3	4	5	6	7	8	9	10	11	12

- 13** Draw a line using block graphics character 1 in the VDU file.
- 14** Draw a line using block graphics character 2 in the VDU file.
- 15** Draw a line using block graphics character 3 in the VDU file.
- 16** Draw a line using block graphics character 4 in the VDU file.

EXAMPLE

```
at 5,60 : vline 8,14
```


SYNTAX

wakeupt *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 **wakeupt** 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

```
read process key=task          /* get my entry      */
wakeupt proc_generate          /* wake generate */
tmp_gen=proc_update
read process key=proc_generate
proc_end=tmp_gen              /* tell gen to wake upd */
write process
read process key=tmp_gen
proc_end=task
write process                  /* tell update who i am */
pause                          /* wait for wakeup from update */
```

write

Write a record back to a keyed file

SYNTAX

```
write file_id [ re = label ] [ nrs = 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 error **No record selected** is displayed and control passes to the option prompt. This error may be trapped by using the **nrs = label** clause, in which case control passes to the line indicated.

If any key data has been altered since the record was read, then a new record is inserted and 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.

If key data has been altered but a record with the new key value already exists on the file, then the error **Record exists** is displayed, the old record is not deleted and control passes to the option prompt. This error may be trapped by using the **re = label** clause in which case control passes to the line indicated.

EXAMPLE

```
*a=Amend
  check cust
A1  input c_name - c_status eci=A2
A2  prompt "Amendments correct" no=A1
    write cust
    clear
    end
```