SCULPTOR

REFERENCE MANUAL

REFERENCE MANUAL

SCULPTOR REFERENCE MANUAL

THE SCULPTOR SOFTWARE DEVELOPMENT SYSTEM

Fourth Edition issued March 1986
Incorporating October 1986 Revision

ISBN 0 947770 02X

Published by
Microprocessor Developments Ltd.
3 Canfield Place
London NW6 3BT
England

TRADEMARK ACKNOWLEDGEMENTS

MS-DOS and MS-NET are trademarks of Microsoft Corp.
Netware is a trademark of Novell Inc.
OS9 is a trademark of Microware Corp.
PC-DOS and PC-NET are trademarks of International Business Machines Corp.
Quix is a trademark of Imtec Computers Ltd.
Ultrix and VMS are trademarks of Digital Equipment Corp.
SCULPTOR is a trademark of Microprocessor Developments Ltd.
UniFLEX is a trademark of Technical Systems Consultants Inc.
Unix is a trademark of AT & T Bell Laboratories
Xenix is a trademark of Microsoft Corp.

PREFACE

The SCULPTOR documentation is supplied in two separate manuals:

- An Introduction to SCULPTOR
- The SCULPTOR Reference Manual

First time users of SCULPTOR are recommended to read the Introduction Manual first until sufficiently confident to attempt to write simple systems. The Reference Manual can then be used to provide details of all the programs and features of SCULPTOR, and will continue to provide an insight into more powerful techniques.

These manuals refer to Version 1.14 of SCULPTOR.

HISTORICAL NOTE

The name of this package was formerly SAGE and has been changed to SCULPTOR in order to avoid confusion with other software products. For the convenience of existing users, program names have not been altered, which is why the screen form interpreter is called **sage** and the report generator is called **sagerep**.

CONTENTS

PAGE

CONTENTS (cont.)

PAGE

CHAPTER 1 An Overview of SCULPTOR

This chapter provides a summary overview of the major SCULPTOR features. The chapter is divided into eight sections.

1.1 GENERAL DESCRIPTION

SCULPTOR comprises a suite of programs designed to enable the rapid development of sophisticated software for a wide variety of applications. At the heart of the SCULPTOR system is a powerful and flexible keyed file technique which provides for very fast retrieval, insertion and deletion of information. Records may be accessed randomly or in ascending key sequence and there are powerful search commands for use when the exact key to a record is not known. The indexing technique (known as a B-tree) keeps the index permanently sorted — it never requires re-structuring. The SCULPTOR Development Menu enables you to select the required program quickly and easily.

Data processing in SCULPTOR is accomplished through two purpose-designed high level languages, one for interactive work through a VDU (CRT) terminal (**sage**) and one for report generation (**sagerep**). Both languages are also suitable for general update programs and contain powerful, high level commands which take care of most normal programming chores, leaving the programmer free to concentrate on the application itself. Since there is a rich set of operators and commands in each language, total flexibility is retained.

A comprehensive set of support programs makes the development and maintenance of a complete software project both quick and easy. There are utilities to describe the structure of files, to create a nested menu system and to check the integrity of the file system and repair it if necessary, although the indexing technique has proved to be extremely robust and file damage is only likely to occur through power or hardware failure.

Further utilities exist to define the parameters for terminal and printer characteristics so that programs run without modification on different terminals and different printers. A special program is provided to configure the system to a different language or different date format. The two programs **sg** and **rg** take the ease of use and speed of SCULPTOR a stage further by creating automatically a Screen Form or Report program from a selected record description file. The **Query** program enables information to be retrieved and reports to be produced from any SCULPTOR file and one associated cross-reference file.

The interactive screen form program (**sage**) can also accept its input from a text file, and the output from the report generator program (**sagerep**) can be redirected to a text file, allowing communication with software written in other languages.

SCULPTOR is ideal for a wide variety of business applications: any system involving retrieval from and updating of large data files, cross-referenced to one another, is particularly suitable. Full data processing including all normal arithmetic calculation is handled efficiently by SCULPTOR, although it is not recommended for complex scientific or mathematical operations.

1.2 DOCUMENTATION CONVENTIONS

Throughout this manual, the following conventions are used in syntax descriptions and examples:

1) Items to be replaced by an appropriate value are enclosed in angle brackets ''< >''.

2) Optional items are enclosed in square brackets ''[]''.

3) An ellipsis ''...'' indicates that the preceding item may be repeated as many times as you wish.

4) A list of items enclosed in braces ''{}'' and separated by slashes ''/'', indicates a choice of items, one of which is to be selected.

5) Since SCULPTOR runs with several different operating systems, standard english words have been used in examples that require operating system commands. The function of the required command will be obvious. For example, on Unix the commands **copy**, **rename** and **delete** are **cp**, **mv** and **rm** respectively.

6) Other words and punctuation characters are to be included as shown.

1.3 THE SCULPTOR PROGRAM SUITE

Twenty main programs are provided in the SCULPTOR Development System. They are:

cf	Compiler for **sage** programs.
cr	Compiler for **sagerep** programs.
decprinter	Decode printer parameters.
decvdu	Decode VDU (CRT) parameters.
describe	Describe record structures.
kfcheck	Keyed file integrity check.
kfcopy	Keyed file copy.
kfdet	Display keyed file details.
kfri	Rebuild keyed file index.
lcf	Language configuration program.
menu	Menu interpreter.
newkf	Initialise a new keyed file.
reformat	Reformat a keyed file.
rg	Automatic report program generator.
sage	Screen form interpreter.
sageform	Print screen form for documentation.
sagerep	Report generator.
setprinter	Setup printer parameters.
setvdu	Setup VDU (CRT) parameters.
sg	Automatic screen program generator.

In addition, a query system is supplied, consisting of a **sage** language program and some supporting programs.

The SCULPTOR Run-time System contains the above programs less **cf**, **cr**, **describe**, **rg**, **sageform** and **sg**.

NOTE: Under the MS-DOS and PC-DOS operating systems, program names are limited to 8 characters. On these systems, the programs **decprinter** and **setprinter** are named **decprint** and **setprint**.

1.4 SCULPTOR KEYED FILES

A SCULPTOR keyed file is kept as two separate disk files, one containing data records and the other an index. All data, including key information, is held in the data file, which means that if the index is lost or damaged it can be rebuilt from an intact data file. The program **describe** is used to specify the record layout of a keyed file and alternative record layouts are also permitted.

Each keyed file has one physical key which may be logically interpreted as several separate data items. There are commands to search the index on part of the key. The index is a multi-level tree which is re-organised every time a record is inserted or deleted. The technique used is very fast and means that the index is permanently up to date. The file grows automatically as new records are inserted and it is not necessary to pre-declare the file size. To avoid unnecessary growth, the space released by deletions is re-used when new records are inserted.

The main data file is created with a filename specified by the programmer. The index file has the same name plus a **.k** extension. It is important to ensure that the correct generation data file and index file are always kept together.

The only restrictions for a SCULPTOR keyed file are:

1) There must be at least one key field.
2) The total record length must be at least 3 bytes.
3) The total key length must not exceed 160 bytes.

1.5 VDU (CRT) PARAMETER FILES

The SCULPTOR system requires a parameter file for each type of VDU terminal being used. Several such parameter files are supplied with the system and new ones may be created with the program **setvdu**. See section 5.8.

The purpose of these parameter files is to make programs independent of the terminals being used. Almost any terminal with cursor positioning and a 'clear screen' command may be used, although SCULPTOR is more effective if protected fields are available. Almost all modern terminals are suitable and different types of terminal may be used on the same system.

Refer to the installation instructions for information on installing VDU parameter files.

1.6 PRINTER PARAMETER FILES

The report generator requires a parameter file to describe the printer being used and the current paper size. Several such parameter files are supplied with the system and new ones may be created with the program **setprinter**. See section 5.9.

The purpose of these parameter file is to make programs independent of the printer being used and to make it easy to use special features such as double-width characters, underlining and compressed print. If the report requests a feature that is not supported by the printer in use, the report generator ignores the request but still prints the report.

Refer to the installation instructions for information on installing printer parameter files.

1.7 CREATING A SCULPTOR SYSTEM

The general procedure for creating a SCULPTOR system is as follows.

1) Use the program **describe** to define the record layouts for each file required. Up to eight alternative record layouts may be described for a file but good design technique should avoid too much use of this facility. **describe** creates descriptor files, identified by a **.d** extension, which are then used by the compilers and some of the utilities.

2) Create new keyed files using the program **newkf**, which reads the descriptor files, calculates the required key and record lengths and initialises the new keyed files.

3) Write the required screen form programs in the **sage** language. These are ordinary text files so you may use any text editor available on your system. The files must have a **.f** extension on the filename and are compiled using the compiler **cf** (compile form). Standard file update programs may be created with **sg** — the automatic screen program generator. The compiled programs are placed in files with the same name stem as the source code but with a **.g** extension. These files are required by **sage** itself and must reside in the working directory or in a local 'bin' directory unless a full pathname is supplied on the **sage** command line.

4) Write the required report programs in the **sagerep** language. These are ordinary text files and may be created with any available text editor. The files must have a **.r** extension on the filename and are compiled using the compiler **cr** (compile report). Simple, single file report programs may be created with **rg** — the automatic report program generator. The compiled programs are placed in files with the same name stem as the source code but with a **.q** extension. These files are required by **sagerep** itself and must reside either in the working directory or in a local 'bin' directory unless a full pathname is supplied on the **sagerep** command line.

5) If you subsequently alter a file's record layout by deleting fields or inserting new ones, or by changing the size or type of any field, then the file must either be re-initialised or reformatted and all programs which reference that file must be recompiled. Failing to do so can corrupt existing data in the file. See section 5.2.

1.8 FILE INTEGRITY CHECKS

The SCULPTOR keyed file system is very robust and has been thoroughly tested over several years. However, the multi-level tree index can be corrupted if an update routine is interrupted by power or hardware failure or by uncontrolled task termination when the system is incorrectly shut down. The update routines ignore all normal software interrupts.

A utility program called **kfcheck** is provided which checks the integrity of keyed files. It should be run as part of the start up procedure every time the system is switched on, or if the system is permanently on, it should be run once each day. If damage is discovered, the utility **kfri** is normally able to repair it, although prolonged use of a damaged file can lead to serious loss of data.

See section 5.1 for details on running **kfcheck**.

CHAPTER 2 DESCRIPTION OF RECORD LAYOUTS (describe)

This chapter explains the program **describe** which is used to define the record layout for a SCULPTOR keyed file. The chapter is divided into eight sections.

2.1 INTRODUCTION TO DESCRIBE

The **describe** program is used to define the record layout for a SCULPTOR keyed file. The program creates a text file of the record descriptions which has the same file name as its corresponding data file, but with a **.d** extension. The associated index file will also have the same name, but with a **.k** extension. The descriptor file is used by the compiler programs and by some of the utility programs.

A record layout is described in terms of data items called fields. Each field may have a name, heading, type&size, print format and validation list. A field's name must uniquely identify it. There are two sets of fields in each record — key fields and data fields. Index entries are composed by concatenating the data values from the key fields in the order in which they are defined. Record data is composed by concatenating the data values of both the key fields and the data fields.

Alternative record layouts may be described for the same keyed file. The descriptor files for each of these alternatives must be given unique names, each alternative record layout being defined in a separate call to **describe**.

When describing alternative record layouts, it is permissible to give a field the same name as a field on the main record or other alternative record, but it only makes sense to do this if the fields are identical and occupy the same byte positions in both records. The compilers will issue a warning and select only one of the field descriptions. You may prefer to avoid the warning messages by using unique field names throughout.

If **describe** is used to alter an existing descriptor file then all programs that reference that file must be recompiled to incorporate the changes. If fields are deleted, new fields inserted or if the size of an existing field is altered, then recompilation is essential to avoid file corruption, and the existing keyed file must either be re-initialised or reformatted. See section 5.2.

2.2 SYNTAX AND OPTIONS (describe)

describe <filename>

A descriptor file always has a **.d** extension but this does not have to be typed on the command line. If the file does not already exist then the following appears on the screen:

```
For each field enter:
name,heading,type&size,format;validation
Type h for help.

KEY FIELDS
  1:
```

The order in which key fields are described is crucial, since index sequential access depends upon the content of the key. Keys are sorted by direct byte comparison, which means that if the sequential access order is important, you should not include numeric fields which may contain negative values, or any floating point fields (**r8** and **m8**)

An alternative record layout must have an identical key length to the main record and it is recommended that the key structure is also identical. If you choose to have a different key structure, ensure that the byte string value of a main record key cannot be the same as that of an alternative record key.

When all key fields have been described, pressing RETURN displays the prompt for data fields. Record length will be the sum of the sizes of the key fields and the data fields. It is advisable to describe alternative record layouts such that they have the same length as the main record, by including filler fields as necessary, but the system will not complain if they are shorter.

When all data fields have been entered or if **describe** is used on an existing descriptor file, the following options are displayed on the screen:

List,Change,Delete,Insert,Abandon,Save,Help:

Select the required option by typing its initial letter and RETURN. The options are explained on page 2-4.

List	Lists the descriptions of all fields.
Change	Prompts for the number of the field to be changed and displays its current description. A new description may then be typed. Pressing RETURN alone retains the existing description.
Delete	Prompts for the number of the field to be deleted. You may go back to the option list by pressing RETURN only. When a field is deleted, all following fields are immediately renumbered. Therefore it is best to work in descending field number order when deleting several fields in one go.
Insert	Prompts for the field number after which insertion is to begin. If you specify the last numbered key field then the prompt **Key or data?** is issued; type **k** or **d** as appropriate. To insert at the beginning, enter field number 0. You may now insert as many fields as you wish. Press the RETURN key to redisplay the option list. Note that all fields following the ones that you insert will be renumbered.
Abandon	Exits from **describe** without saving the current entries or amendments. The prompt **Abandon (y/n)?** is issued to confirm this intention. Type **y** or **n** and RETURN.
Save	Exits from **describe** and saves the descriptors.
Help	Displays a short description of the syntax with an explanation of field types. This option may also be invoked by entering **h** RETURN instead of a field name.

describe has an option to list the information stored in descriptor files. Its output may then be redirected to a file or piped to a printer and used in documentation. With this option, multiple filenames are permitted on the command line. Example:

```
describe -l *.d ¦ spooler
```

2.3 FIELD NAMES

Field names may contain alphabetic characters (upper and lower case), the numerals 0 through 9 and the underscore character, but the first character may not be a numeral. SCULPTOR keywords may not be used as field names. (Refer to Appendix B for a list of reserved words.)

Each name must be unique and, although it is possible to use the same field name in different descriptor files, confusion will occur in any program which references both those files. The compilers issue a warning message when identical field names are encountered and select only one field description. To avoid possible duplication it is advisable to use a file identifying prefix on all field names.

Example field names from a sales ledger file.

```
sl_name
sl_accno
sl_amount
sl_date
```

Illegal fieldnames:

1st_name	(Must not start with a numeral)
p-item	(Hyphen not permitted)
vat@15%	(Illegal characters @ and %)

The following names are reserved for special variables:

```
arg
date
day
lines_left
month
pageno
scrline
systime
task
time
tstat
ttyno
year
```

2.4 FIELD HEADINGS

Field headings are used to identify boxes on the **sage** screen form and for column headings on reports. There is no restriction on the characters that may be used in headings, subject to the following special cases:

1) A null heading, indicated by two consecutive commas, causes the field name to be used as the heading.

2) A single space implies a blank heading.

3) If the heading is to contain commas or semi-colons then it must be enclosed in quotation marks (`''` or `"`). If the heading is to contain quotation marks then it must be enclosed in the other type of quotation mark.

The field heading specified with **describe** may be overridden in a **sage** or **sagerep** program.

Examples:

```
c_name,Name,
c_addr1,Address,
c_addr2, ,
c_status,,
c_code,''0 = Exempt,1 = Inclusive,2 = Exclusive'',
```

2.5 FIELD TYPE AND SIZE

Field type&size is specified by a single alphanumeric character indicating the field type together with a number which indicates the size in bytes, e.g **a20** is an alphanumeric field of 20 bytes and **m4** is a money field of 4 bytes. A data field may also be dimensioned by enclosing the number of occurrences required in parentheses after the type and size, e.g. **d4(10)**. See the **scroll** command in **sage** and **sagerep** for details on indexing a dimensioned field. Key fields may not be dimensioned.

Type	Size		Description
a	0 to 255		Alphanumeric field which may contain any character.
d	4		Date field stored internally as a day number (starting 1/1/0001).
i	1,2 or 4		Integer stored in binary. Range according to size:
	1		0 to 255
	2		-32,767 to +32,767
	4	*	-2,147,483,647 to +2,147,483,647
m	4 or 8		Money with a main currency unit equal to 100 of its subsidiary unit (e.g. Sterling and Dollars). Stored internally as an integer in the lower unit. Range according to size:
	4	*	-21,474,836.47 to +21,474,836.47
	8		Floating point giving up to 15 digits in the lower currency unit.
r	8		Floating point (real) number.

* For historical reasons and to retain upward compatability, the range of values that may be stored in 4-byte integers on 6809 Uniflex systems is -1,073,741,823 to +1,073,741,823.

DATE FIELDS

Since date fields are stored as day numbers, adding or subtracting an integer value from a date adjusts its value by that number of days. Dividing a date by 7 and taking the remainder (use the % operator) gives the day of the week (0 = Sunday). Dates are input and displayed in standard formats, e.g. 31/12/84 or 31/12/1984, the input routines adding 1900 to any two-digit year. The language configuration program **lcf** may be used to define the required standard date format (refer to Section 5.3).

MONEY FIELDS

Money fields are stored in the lower currency unit which helps to avoid rounding errors, **m4** fields being stored in long integers and **m8** fields in floating point. Whenever a value is stored into an **m8** field, Sculptor adds 0.5 (halfpenny or half cent) and then stores the 'floor' value, thus removing any fractional portion. This operation is not performed on **r8** fields.

2.6 FIELD FORMATS

A format may be attached to a field and is used to determine the precise way in which the field is printed and displayed. If no format is specified, a standard default is applied to suit the field's type and size. Any format specified with **describe** may be overridden in a **sage** or **sagerep** program. A format specification which includes commas or semicolons must be enclosed in quotation marks.

ALPHANUMERIC FIELDS

Data is normally output precisely as input, the number of characters equalling the field size. The following special format characters modify the input or output:

e	Suppress echo of input characters (**sage**).
l	Force typed input into lower case.
u	Force typed input into upper case.
s	Remove leading spaces on printing (**sagerep**).
t	Remove trailing spaces on printing (**sagerep**).

DATE FIELDS

Date formats may use the characters d, m and y to designate day, month and year respectively. The year portion may be two or four digits or may be omitted altogether. However, it is generally better to omit date formats and allow the default value set in the **sage** and **sagerep** programs to take effect. By doing this, the program becomes independent of date format and may be run without recompilation in other countries. If this option is taken, it is wise to design screen forms on the assumption that dates require space for a four-digit year. The default format set in **sage** and **sagerep** may be altered with the language configuration program **lcf**. Example date formats:

```
dd-mm-yy
mm/dd/yyyy
dd.mm.yy
''yy,mm,dd''
''dd;mm;yy''
dd mm yy
dd/mm      (Valid for printing but not for input)
```

NUMERIC FIELDS

The output width exactly equals the number of characters in the format specification. The following format characters have special meaning, other characters in the format are output unchanged.

`#` Designates a digit position where leading zeros are to be space filled.

`*` Designates a digit position where leading zeros are to be '*' filled.

0 Designates a digit position where leading zeros are to be printed.

`,` Designates a comma to group significant digits.

`.` Designates position of decimal point.

A format is processed from its right hand end. Once digit selection has started (first `#`, `*` or 0), the first non-special character terminates the numeric part of the format. If the last digit designator was `#`, all further characters in the format are now floated right and leading spaces are used to make up the required width.

Examples:

`#`	Single digit.
`#####.##`	Two decimal places.
`''##,###''`	Comma printed if number exceeds 999.
000000	Number printed with leading zeros.
`***,***,***.**`	Number printed with leading asterisks.
`''$###, ###.##''`	Floating dollar sign.
`''# ###,###''`	Floating pound sign (note the space).
`''Lit ########''`	Floating 'Lit' for Italian Lire.

DEFAULT FORMATS

If no format is specified for a field, then a default is applied as follows:

Field type&size	Default format
d4	That set with **lcf**.
i1	###
i2	#####
i4	#######
m4	#######.##
m8	#######.##
r8	#######.##

There is also a default date format set in the compilers **cf** and **cr** and in **describe** itself. The only purpose of the date format in these programs is to indicate how date constants are to be interpreted in program code and in validation lists.

EXTENDED ALPHANUMERIC FORMATS

The format definition on an alphanumeric field can be extended to apply to assignments to that field as well as to inputs. This is done by using a + character in the format definition. The + should follow any standard format characters used. For example, the format character **l** causes all input to the field to be folded to lower case, but allows upper case assignments. Using the format **l+** causes both inputs and assignments to be folded to lower case.

Optionally, the + can be followed by a numeric format which overrides the default used when numeric data is assigned to the field. If the data being assigned is to be formatted as either money or date, then either **d** for date or **m** for money must precede the + sign. The **d** means that the value being assigned is a day number, and the **m** means that the value has two implied decimal places.

Examples:

```
!temp ukey,,a5,us+
!temp anumber,,a12,"+ ###,###,###"(numeric)
!temp amoney,,a9,"m+ #####.##"     (money)
!temp adate,,a8,"d+dd/mm/yy"       (use this date format)
!temp ddate,,a8,d+                 (use default date format)
```

2.7 VALIDATION

Input data is always automatically validated according to field type and size. Further validation may be defined by attaching a validation list to a field description. Cross-field validation is possible by programmed testing of data inputs using the command language.

AUTOMATIC VALIDATION

Integer Fields

Only integers within the range appropriate to the field size are accepted.

Money Fields

For **m4** fields, amounts must be either whole numbers or fractions with exactly two digits following the decimal point. Note that although **m8** fields are not validated to exactly two decimal places, they will be rounded to that precision. If greater precision is required with money values then **r8** fields must be used.

Date Fields

Day, month and year must be input in the order specified in the date format, but most punctuation characters are accepted as separators regardless of the one used in the actual format. If a two digit year is input then 1900 is added. Input dates are fully checked with proper consideration for leap years, although a correction for the loss of several days back in the eighteenth century is not made.

VALIDATION LISTS

In addition to automatic validation, it is possible to attach a validation list to a field descriptor. The list may consist of individual values and ranges of values, items in the list being separated by commas with ranges indicated by hyphens. **sage** then accepts data for that field only if the value is included in the list. Validation lists are not checked by **sagerep**.

It is frequently convenient to include **no input** as a valid value in a validation list. This may be done with two consecutive commas.

Alphanumeric data is validated by comparing the input text for equality with single items in the validation list and on a greater than and less than basis against a range. Although input data is padded with spaces to fill the length of the field, these are ignored for comparison purposes, so care must be exercised with alphanumeric validation lists. For example, if a two byte field is to begin with a letter in the range A to M, it is insufficient to validate the field **A-M** since **MB** will be considered invalid. Specifying **AA-MZ** overcomes this problem. Examples:

Alphanumeric fields:

```
,,Y,N
MR,MRS,MISS,MS,DR
A-M,X,Z
```

Numeric fields:

```
0-100
,,0,100-999
11-19,21-29,31-39
-500--100,100-500
```

Money fields:

```
,,0-9999.99
1-9999999
```

Date fields:

```
,,1/1/80-31/12/99
1/5/1980-27/8/1982
```

2.8 DESCRIBE PROGRAM EXAMPLES

A SIMPLE STOCK FILE:

```
KEY FIELDS
        1:st_Stock Code,i2,####

DATA FIELDS
        2:st_desc,Description,a20
        3:st_supp,Supplier Code,a9
        4:st_cost,Cost Price,m4,"####.##";0-1999.99
        5:st_sale,Sale Price,m4,"####.##";0-2999.99
        6:st_stklev,Stock Level,i4,"###,###,###"
        7:st_deldat,Delivery Date,d4(3)
        8:st_ordqty,Order Quantity,i2(3)
        9:st_cat,Category,a1;A,B,C,D,E
        10:st_rol,Reorder Level,i2
        11:st_eoq,Economic Order Qty.,i2
```

A SIMPLE NAME AND ADDRESS FILE:

```
KEY FIELDS
        1:na_sur,Surname,a20,u
        2:na_fname,First Name,a12,u
```

CHAPTER 3 SCREEN FORM PROGRAMS (SAGE)

This chapter explains the use of the **sage** program to generate Screen Form programs. The chapter is divided into eleven sections. Sections 3.1 to 3.8 explain the overall structure and main features of **sage**; Sections 3.9 and 3.10 explain all the Declarations and Commands; Section 3.11 explains compilation and running of **sage** programs.

3.1 INTRODUCTION TO SAGE

sage itself is an interpreter which means that its actions are controlled by a code which it interprets as it runs. This code is produced by passing source code programs through a compiler called **cf** (compile form).

The **sage** language is designed primarily for processing data interactively on a screen form, although a **sage** program can be written to function without operator intervention. The data being processed may be stored in several different keyed files.

The general principle of operation is that a screen form is defined containing boxes in which data may be input and displayed. Beneath this form is a menu line which presents the operator with a choice of actions. When an option is selected, the appropriate routine in the **sage** program is entered. When the routine terminates, the operator is again able to select an option.

The **sage** language elements are specifically designed for the screen based processing of keyed data files. There are many very powerful commands, making it possible to write a sophisticated program quickly and easily. Most of the hard work associated with placing a form on the screen, inputting valid data, displaying formatted data and file manipulation is handled automatically by **sage**. Although the **sage** language has many sensible default actions, the programmer is free to override them.

The following sequence shows how **sage** is used in a simple SCULPTOR application:

1) Use the **describe** program to define record layout (i.e. the logical structure of the file).

2) Use the **newkf** program to create the new (empty) data file and the index file (i.e. the physical structure of the file).

3) In any available text editor, write your source code program using the **sage** language declarations and commands.

4) Run the **cf** program to compile the source code file into an object program.

5) Run the new **sage** program.

Note: Steps 3 and 4 can be performed automatically using the **sg** program to create a standard screen form program. Refer to Chapter 5 for details of the **sg** program.

3.2 SAGE PROGRAM STRUCTURE

A **sage** program is written as a text file using a standard editor. The language is line-oriented and the compiler recognises six different line types:

1) The first physical line in the program, regardless of its leading character, is taken as the title line. When the program is run, the text on the title line is displayed centralised on the top line of the screen. The title line must be present but it may blank.

2) Lines commencing with a full stop ''.'' are comment lines and are ignored by the compiler. Completely blank lines are also ignored by the compiler (except for the title line).

3) Lines commencing with an exclamation mark ''!'' are declarations. It is normal to place these after the title line.

4) Lines commencing with a plus sign ''+'' are box definitions and should follow the declarations.

5) Lines commencing with an asterisk ''*'' introduce an 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 SCULPTOR keyword then it is taken to be a line label. Multiple statements, separated by colons, may be placed on the same line.

Program statements may extend over more than one physical line by terminating each line that is to be continued with a backslash ''\''. This is particularly useful when several statements are to be controlled by an **if ... then** command.

A typical **sage** program has the following structure:

<program title line>
<declarations>
<box definitions>
<initialisation statements>
<option title line>
<option statements>
<option title line>
<option statements>
.
.
<subroutines>

Note that a set of statements may be placed before the first option title. These statements are given control after the screen form has been displayed but before the option prompt is first issued. In effect they behave as an automatically selected first option and must terminate in the same way as a normal option (**end**, **exit**, etc.). This set of initialisation statements is optional and may be omitted altogether.

Subroutines do not have to be grouped at the end of the program. Subject to the flow of logic being correctly controlled by **end** statements and by programmed branches, the code for a subroutine may be anywhere in the program. For example, it can be more convenient to place a subroutine immediately after the logic for a particular option.

3.3 SAGE BOX DEFINITIONS

All fields in the program for which values are to be input or displayed must be given a box definition:

+ <field name>,[<heading>],<row>,<col>[,<format>]

The field name may be from a declared keyed file (see **!file** and **!record**) or be a temporary variable (see **!temp**) and defines the field whose value is input and displayed in that box.

If the row number is not equal to the row number in a **!scroll** declaration, a box is created on the screen, the first character position **within the box** being at the specified row and column number (counting from 1). The box is bracketed by the default box delimiters (normally ''[]'') unless a **!box** declaration is included in the program to specify alternative box delimiters. The default box delimiters may be altered by using the language configuration program **lcf**.

If a heading is included in the box definition then it is displayed to the left of the box. If no heading is included (two consecutive commas) then the heading defined when the field was described is used. A blank heading may be obtained by leaving a single space between the commas.

If the row number equals the row number in a **!scroll** declaration then a column of boxes is created on the screen. In this case the first box in the column is on **row + 1** and the heading is displayed centrally over the column on the specified row. The number of boxes below the heading equals the depth defined in the **!scroll** declaration. Note that all scroll area box declarations must be consecutive.

In the case of alphanumeric fields, the box width equals the field width. In the case of any other field type, the box width equals the number of characters in the format specification. If no format is included then the one defined when the field was described is used. If none was defined there, then a default applies (see section 2.6).

Because input and display of a range of fields proceeds in box definition order, it is normally sensible for the boxes to be defined in logical order of screen position. However, screen boxes can be defined in any order.

If a box declaration is given to a field of zero length, i.e. an **a0** field, then no box is displayed, but its heading is still displayed. This exception is useful for the purpose of creating sub-titles on the screen form. Because there is still an imaginary box on the screen, the column co-ordinate must be chosen carefully — it should be 3 greater than the end column of the text to be displayed.

When designing a screen form the top row may be used but care must be taken not to interfere with the title. Remember to keep at least the bottom three rows free of boxes — the minumum requirement for the menu line, option prompt line and messages.

EXAMPLES:

```
+ date,Today's date,2,70
+ st_code,,5,20
+ st_desc, ,5,40
+ cat,"M = Material,L = Leather",7,50,u
+ value,,9,40,###### . ##
```

```
f = find n = next i = insert a = amend d = delete e = exit
                Which option do you require?
```

3.4 SAGE OPTIONS

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

*<code> = <description>

The code may be any one or two printable characters. The number of options in a **sage** program should be fairly small (normally not more than 6) and the description of each should be brief since the option title lines are merged into a single menu line which is displayed on the screen.

Following an option title line are the program statements to be obeyed when that option is selected. When **sage** requires the operator to select an option, it displays an option prompt beneath the menu line and awaits a response. The reply is validated and then the appropriate section of the program is given control. A typical menu line and prompt might look like this:

f = find n = next i = insert a = amend d = delete e = exit
Which option do you require?

The processing of a selected option continues until terminated in one of the following ways:

1) An **end** statement is encountered. This returns control to the option prompt.

2) An **exit** statement is encountered which terminates the program completely.

3) An untrapped error condition occurs. This causes a suitable error message to be displayed and then returns control to the option prompt.

4) The operator cancels the option during an input operation. This also returns control to the option prompt. The cancel feature may be turned off — see the command **cancel**.

If the first character of an option code is * then that option is not included in the menu line displayed on the screen but the option may still be selected by the user. For example:

```
**d = delete
```

To call such an option, the leading "*" must be typed. In the above example the **= delete** is clearly superfluous as it will never be displayed, but it effectively serves as a program comment.

NOTES

1) DON'T forget to include an option to exit from the program. This is the only safe way out of **sage**.

2) The program statements for each option must terminate with an **end** command. If there is no **end** command for an option, statement execution can continue through an option title line into the logic for the following option.

3) When checking option codes, **sage** folds all characters to the same case. Consequently, the operator may respond to the option prompt in either upper or lower case.

4) It is possible to create a **sage** program which has no options, all logic being in initialisation statements. Ensure that such programs terminate with an **exit** command at the proper place.

3.5 SAGE EXPRESSIONS AND OPERATORS

sage supports a comprehensive set of arithmetic and relational operators. These may be used to form expressions involving keyed file fields, temporary variables and constants.

Precedence

Operators have precedence as set out below, but sub- expressions may be enclosed in parentheses to force a particular order of evaluation. The available operators are listed below. The operators grouped together have equal precedence and the groups are listed in descending order of precedence.

Group 1:	
-	Negate (unary minus)
Group 2:	
*	Multiply
/	Divide
%	Remainder (after integer division)
/	String concatenation (with trailing spaces stripped from left operand)
Group 3:	
+	Add
-	Subtract
+	String concatenation (with trailing spaces retained from left operand)
Group 4:	
=	Equal to
<	Less than
>	Greater than
< =	Less than or equal to
> =	Greater than or equal to
< >	Not equal to
ct	Contains
bw	Begins with

Group 5:

and	Logical 'and'

Group 6:

or	Logical 'or'

The operators **ct** and **bw** apply to alphanumeric data only. **ct** yields true if the right operand is a sub-string of the left operand. **bw** yields true if the left operand begins with the right operand. Examples:

if ''Smithson'' bw ''Smith''	(true)
if ''Smithson'' bw ''son''	(false)
if ''ABCDEF'' ct ''CD''	(true)
if ''ABCDEF'' ct ''BD''	(false)

Relational expressions are most commonly used with the **if** command, but it is also permissible to have a relational expression as part of an assignment statement; its value is 1 if true or 0 if false.

CONSTANTS

Numeric and alphanumeric constants may be freely included in expressions. Numeric constants may be integer or floating point and positive or negative; a numeric constant is floating point if it includes a decimal point. Alphanumeric constants, sometimes called strings, must be enclosed in either single or double quotation marks (" or '''). Examples:

123	(integer)
-50	(integer)
34.451	(floating point)
100.0	(floating point)
'Thursday'	(alphanumeric)
''today's''	(alphanumeric)
''''	(null string)

FIELD TYPE CONVERSION

Expressions may include fields of different types. Each operation examines the types of its two operands and if they differ, an automatic type conversion is performed according to the following 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.

In the following expression, the two division calculations will be evaluated first, without type conversion, the result of the integer division will then be converted to floating point for the multiplication and finally the product converted to alphanumeric for the assignment:

```
alpha = (int1 / int2) * (real1 / real2)
```

All internal integer arithmetic is performed with long integers to minimise the possibilty of overflow in an intermediate result.

Arithmetic may be freely performed on dates which are treated as day numbers, counting 1/1/0001 as day 1. Useful operations on dates are addition and subtraction of a number of days and taking the remainder after division by 7 (% 7) to get the day of the week (0 = Sunday).

When operating on money fields remember that their data value is in the LOWER currency unit, e.g. pence or cents.

When calculating ratios or percentages using integer variables the result will be integer.

i.e.

```
real = int1 / int2 * 100
```

will not give the required answer. Instead, use:

```
real = (100.00 * int1) / int2
```

3.6 THE KEY= CLAUSE

Commands which read records according to a supplied key value have an optional **key=** clause. If the clause is omitted, then data from the file's natural key fields is used to construct the key. The natural key fields are those defined in the main record layout.

The purpose of the **key=** clause is to simplify the program when the required key data is being supplied from other than the natural key fields, e.g. from fields on another file or from temporary variables.

If the **key=** clause is present then a key is constructed from the named fields. These should normally have the same type and size as their counterparts in the natural key. If they do not, the following rules apply:

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

2) If the number of fields named in the clause 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, 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 the main record. Bearing in mind the fact that the alternative record layout simply overlaps the main record layout, the recommended method is to assign values to the key fields named in the alternative record layout and to omit the **key=** clause.

3.7 TRAP CLAUSES

Whenever **sage** encounters an exception condition, it applies an appropriate default action. For example, if no record is located in a **read** command, the error **No such record** is displayed and control is returned to the option prompt.

Traps enable the programmer to specify an alternative action if the default is not suitable. All trap clauses have the general syntax:

<trap code> = <label>

The trap code identifies the condition being trapped and the label specifies the program line to be given control if the exception condition occurs. A trap clause forms part of the command line to which it relates and the allowed traps for a particular command are specified in the syntax of the command.

Available traps are listed below. They are explained in more detail in the descriptions of the commands which accept them.

Trap Code	Meaning
bs	Backspace past first box
eoi	End of input
ni	No input
no	'no' reply to 'prompt'
nrs	No record selected
nsr	No such record
re	Record exists
riu	Record in use
yes	'yes' reply to 'prompt'

3.8 SAGE BOX LISTS AND FIELD LISTS

BOX LISTS

Several commands require a box list. This is a list of box names, separated by commas and may also include ranges of boxes, indicated by hyphens. A range implies all the boxes defined in the program between (and including) the named boxes, in order of definition. A box is defined by a line commencing with ''+'' (see section 3.3).

Examples:

1) **input name,addr1-addr4,cat**

2) **clear t_date-t_amount**

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 declared in the program or may be a temporary variable.

Example:

read ord key = ordno,lineno

3.9 SAGE DECLARATIONS

This section describes the declaration statements available in the **sage** language. They are listed on the pages indicated below.

!box

Define non-standard box delimiters.

SYNTAX:

!box <delimiting characters>

DESCRIPTION:

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

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

The box delimiting characters may be set to spaces by enclosing a space in quotation marks.

EXAMPLES:

1) !box < >

2) !box :

3) !box " "

Declare a file which is initially closed.

SYNTAX:

!cfile <file identifier> [<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 identifier 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 identifier is also the filename. The file identifier must 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.

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 16.

EXAMPLES:

1) !cfile control

2) !cfile supp suppliers

3) !cfile cont /usr/common/control

!depth

Define screen depth.

SYNTAX:

!depth <integer>

DESCRIPTION:

This optional statement defines the screen depth (number of lines) required. If no **!depth** statement is present, a default of 24 lines is assumed.

The only effect of this statement is to cause sage to reconfigure the screen to a larger or smaller number of lines in cases where the terminal supports such a feature.

EXAMPLE:

```
!depth 16
```

Declare a file which is initially open.

SYNTAX:

!file <file identifier> [<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 identifier 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 identifier is also the filename. The file identifier must 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 16. The maximum number of files that may be open at the same time, and hence the maximum number of **!file** declarations, varies according to the operating system but is normally not less than six. See Appendix A (Implementation Differences) for details of your particular system.

EXAMPLES:

1) !file stock

2) !file cust customers

3) !file tax /usr/john/income_tax

!record

Declare an alternative record layout.

SYNTAX:

!record <file identifier> <pathname>

DESCRIPTION:

This command is used to declare an alternative record layout for the file referred to by the file identifier (previously declared in a **!file** 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.

Alternative record layouts must have the same key length as the main record and it is 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). Since alphanumeric fields are initialised to spaces and numeric fields to nulls, this can be important.

Alternative record layouts are useful not only for completely different record types contained in the same file, e.g. a control file, but also for redefining the structure of individual fields to enable access to their component parts.

EXAMPLE:

```
!file control
!record control control1
!record control control2
!record control control3
```

Define a scroll area.

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 boxes required in each column. A box 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 box whose line co-ordinate exceeds the heading line but is within heading line + depth, displays as a single box, 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 variable **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.

EXAMPLE:

```
!scroll 10,5

(5 rows under headings at line 10)
```

!temp

Declare a temporary field.

SYNTAX:

!temp <name>,[<heading>],<type&size>[(<dimension>)]
[,<format>]

DESCRIPTION:

Declares a temporary field for use within the program. The syntax is similar to that used with the program **describe**, to which reference should be made for full details, except that no validation list is permitted.

A temporary field may be subscripted, in which case the element accessed is determined by the current value of the special variable **scrline**. If **scrline** 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 keyfile fields, but note the special treatment of temporary fields by the **clear** command.

A temporary field may be declared with a **type&size** of **a0**, i.e. an alphanumeric field of zero length. If a box is defined for the field, the box itself is suppressed, but the heading appears to the left of the box's imaginary position and is a convenient way of displaying static textual information on the screen.

EXAMPLES:

1) !temp total,Total,m4

2) !temp bf,Brought forward,a0

3) !temp cat,Category,a2(10)

4) !temp status,,i1

SPECIAL TEMPORARY FIELDS

Certain special variables, some of which are operating system dependent, may be referenced by defining them as temporary fields. These are:

!temp arg,,a0
Command line arguments.

!temp date,[<heading>],d4
The system date.

!temp scrline,[<heading>],i2
See the command **scroll**.

!temp systime,[<heading>],i4
System time in seconds.

!temp task,[<heading>],a5
The current task number.

!temp time,[<heading>],m4
The current time: hours.mins.

!temp tstat,[<heading>],i1
Child task termination status.

!temp ttyno,[<heading>],i2
tty port number.

!temp day,[<heading>],i1
Day for encdate/decdate commands.

!temp month,[<heading>],i1
Month for encdate/decdate commands.

!temp year,[<heading>], i2
Year for encdate/decdate commands.

!temp vduname,[<heading>],a12
Name of vdu in use (from parameter file).

arg returns a command line argument. Whenever it is referenced, the particular argument returned depends on the current scroll line number. For example, if the command line is:

```
sage cust abc
```

then **sage** is returned if the scroll line number is 1, **cust** is returned if it is 2, and **abc** is returned if it is 3. Reference to a non-existent argument returns a null string. The values in **arg** cannot be altered. See **scroll** for information on setting the scroll line number.

date returns the system date and is updated each time it is referenced, so that programs running over midnight can still access the correct date. However, if a value is stored into the field **date**, then automatic updating stops.

scrline returns the current scroll line number. Its value cannot be altered by direct assignment (see **scroll**).

systime returns the system time in seconds. Its value cannot be altered by direct assignment. Since its base value is somewhat arbitrary, **systime** should only be used to calculate time differences.

time returns the time of day and is updated each time it is referenced. Note the use of a money field as a convenient way of presenting hours and minutes.

task returns the current task number on multi-tasking operating systems.

tstat returns the termination code of the last child task (see the **exec** command).

ttyno returns the port number (or other unique number) on multi-user systems. On single-user systems it returns zero.

Note: On systems such as Unix System V which support cluster controllers, there is no port number as such. On these systems, **ttyno** returns a unique number based on the major and minor device numbers.

day, month and **year** are used by the commands **decdate** and **encdate**.

Define screen width.

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.

EXAMPLE:

!width 132

3.10 SAGE COMMANDS

This section describes the command language from which **sage** program statements are constructed. The commands are listed on the pages indicated below.

(cont.)

SAGE COMMANDS (CONT.)

at

Position the cursor.

SYNTAX:

at <row>,<col>

DESCRIPTION:

Positions the cursor to the specified row and column. One use for this command is just prior to an **exec** command, to cause the output from the called program to appear at a particular place on the screen. Another use is in conjunction with the **vdu** command to make special use of the features of the terminal.

If the **at** command precedes a **message** command in a multiple statement line, the message is displayed at the current cursor location instead of the bottom left hand corner of the screen.

EXAMPLES:

1)
```
at 1,1
exec "sagerep stocklist qume ¦ lpr"
```

2)
```
at 5,20: message "This is at row 5 column 20"
```

3)
```
at 20,1: vdu 52
```

Enable or disable automatic RETURN on input.

SYNTAX:

autocr {on/off}

DESCRIPTION:

If **autocr** is **off**, all input into the screen boxes must be completed by using the RETURN key. If **autocr** is **on**, typing the last character in the screen box completes the input and the RETURN key is not required, although it can still be used to complete input to a screen box before the last character position is reached.

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

EXAMPLE:

autocr on

cancel

Enable or disable the cancel option.

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 (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 is to be completed without interruption.

EXAMPLE:

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

```
        cancel 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
```

Chain new program.

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.

The text expression cannot include I/O redirection, pipes or other special shell features. Multiple commands cannot form part of a 'chain' statement.

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.

See also the **exec** command.

EXAMPLE:

1) chain "sage ordlines " + ordno

2) chain "menu main"

3) chain "sage calc"

check

Check that a record is selected.

SYNTAX:

check <file identifier> [nrs = <label>]

DESCRIPTION:

Checks that a record has been read from the specified file and not written back or cleared. 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 = <label>** clause, in which case control passes to the line indicated.

EXAMPLES:

1)
```
*a = Amend
        check cust nrs = A2
A1      input c_name - c_status
        prompt "Amendments correct" no = A1
        write cust
        clear: end
A2      input c_name
        find cust
        goto A1
```

2)
```
*d = Delete
        check cust
        prompt "Are you sure" no = D1
        delete cust
        clear
D1      end
```

Clear all or specified boxes.

SYNTAX:

clear [<box list>]

DESCRIPTION:

If a box list is specified, clears the screen of data displayed in those boxes and re-initialises the fields associated with them, alphanumeric fields being set to spaces and numeric fields to zero. The box list may consist of individual box names separated by commas, ranges of box names separated by hyphens, or a combination of the two. Subscripted fields and boxes in the scroll area are cleared only on the row identified by the current value of the special variable **scrline**. (See the **scroll** command.)

If no box list is specified then all boxes are cleared, the message line is erased, all file buffers are re-initialised and any records currently held are unlocked. However, in this case, although their screen boxes are cleared, the data held in any associated temporary fields is not lost. The file buffers are initialised according to the **!file** record layouts, alphanumeric fields being set to spaces and numeric fields to zero.

Note that other information, such as the current position of each file and any match keys set up by the **find** command is not destroyed by **clear**.

See also the **preserve** command.

EXAMPLE:

```
clear name,addr-status,t_code
```

clearbuf

Clear a file's record buffer.

SYNTAX:

clearbuf <file identifier>

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.

EXAMPLE:

clearbuf stk

Close a file.

SYNTAX:

close <file identifier>

DESCRIPTION:

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

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

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 that is already closed is ignored.

EXAMPLE:

close control

decdate

Decode a date.

SYNTAX:

```
decdate <expression>
```

DESCRIPTION:

Decodes a date into day, month and year components. The expression must yield a valid day number and will normally be a simple date field. The decoded values are placed in the special temporary variables **day**, **month** and **year** which must be declared (see **!temp**).

EXAMPLE:

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

Delete a record.

SYNTAX:

delete <file identifier> [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 = <label>** clause, in which case control passes to the line indicated.

EXAMPLE:

```
*d = Delete
        prompt "Are you sure" no = D1
        delete stk
        clear
D1      end
```

Display data.

SYNTAX:

```
display <box list>
```

DESCRIPTION:

Displays the current values of the specified fields in their associated boxes on the screen. The box list may consist of individual box names separated by commas, ranges of box names separated by hyphens, or a combination of the two. The display takes place in the order specified in the box list.

The data is displayed with the attributes defined by the **Start Normal Data** and **End Normal Data** sequences in the vdu parameter file. This reverses the effect of the **highlight** command.

EXAMPLE:

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

Encode a date.

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** must be declared (see **!temp**).

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

EXAMPLE:

```
decdate date
day = 31: month = 12
encdate eoy                (last day of current year)
```

end

End the current option.

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.

EXAMPLE:

```
*n = Next
        next stk
        display s_code - s_rol
        end

*a = Amend
        etc.
```

Display an error message.

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** and **End error message** sequences in the VDU parameter file (see section 5.8), and is erased as soon as fresh input is received, another message is displayed or a **clear** command with no box list is given. Certain error conditions, unless trapped, automatically display an error message and return control to the option prompt. These are:

Message	Cause
No such record	Attempt to read a non-existent record.
Record exists	Attempt to insert a record with a duplicate key.
No record selected	Attempt to write back or delete with no record selected.

The language configuration program **lcf** may be used to alter the text of these error messages in **sage** itself.

EXAMPLES:

1)
```
error "Sale price must exceed cost price!"
```

2)
```
atemp = max_disc
error "Maximum discount is " + atemp + "%"
```

exec
execu

Execute a child task.

SYNTAX:

exec <text expression>
execu <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 variable **tstat** contains the child tasks' termination code.

Before executing the command, **exec** issues the **Ignore Protection and Enable Scroll** and the **Reconfigure VDU** sequences defined in the VDU parameter file. On regaining control, it issues the **Configure VDU** and the **Honour Protection and Disable Scroll** sequences. If there is a possibility that the screen form will be damaged, the **newform** 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. For further details see appendix A (Implementation Differences).

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 the system by preceding the command with a - as in example 3 below. A shell is required if the command involves I/O redirection, pipes, shell expansion or multiple commands.

EXAMPLES:

1) exec "kfcheck *.k"

2) exec "sagerep printinv " + ptr + "¦" + spooler

3) exec "-sage stock"

Terminate the program.

SYNTAX:

exit [<numeric expression>]

DESCRIPTION:

Terminates the program and returns control to the calling task. The optional numeric expression may be used to pass back a termination code (default zero if omitted).

EXAMPLE:

```
* e = Exit
        exit
```

find

Find and read a record.

SYNTAX:

find <file identifier> [key = <field list>] [nsr = <label>]
[riu = <label>]

DESCRIPTION:

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 = <label>** clause, in which case control passes to the line indicated. An unsuccessful find 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 = <label>** clause, in which case control passes to the line indicated. 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. See section 3.6 for full details. The **find** command differs from **read** by not requiring an exact key. The rules are:

1) If the natural key field is alphanumeric, then trailing spaces in the supplied data are ignored and only the leading characters must match the corresponding characters in that key field, e.g. if "Smith" is supplied then "Smithson" will match but "Smythe" will not.

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

3) If the natural key field is numeric (including dates) and the supplied data is zero, then any value in that key field matches.

See also the **match** command.

EXAMPLE:

```
*f = Find
        input surname,firstname,dob
        find addr
        .
        .
```

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.

getstr

Extract a sub-string.

SYNTAX:

getstr(<source>,<pos>,<len>)

DESCRIPTION:

getstr returns the sub-string which starts at character position **pos** in the string **source** and is up to **len** characters long. It can be included as part of an expression.

source can be either an alpha field or a string constant. **pos** and **len** can be either numeric constants or numeric expressions.

The first character in **source** is position 1. If **pos** is less than 1 or greater than the length of **source**, then a null string is returned. If **len** is greater than the number of characters remaining, then **getstr** returns only those available.

EXAMPLES:

```
orderno = getstr(custno,1,n) + aseq

aday = getstr("SunMonTueWedThuFriSat",3*day-2,3)

message "Sub-code is: " + getstr(code,5,4)
```

Call a subroutine.

SYNTAX:

gosub <label>

DESCRIPTION:

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

Up to 200 subroutines may be nested at any one time. Each subroutine must always be exited eventually using a **return** statement. Repeated use of **goto** commands to exit a subroutine will ultimately cause the error message **Too many nested gosubs** to be displayed.

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
```

goto

Transfer control to another line.

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.

EXAMPLE:

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

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

Highlights specific data items.

SYNTAX:

highlight <box list>

DESCRIPTION:

Displays the current values of the specified fields in their associated boxes on the screen, bracketed by the display attribute defined by the **Start Highlight** and **End Highlight** sequences in the vdu parameter file. The command can 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, and some of those which do are unsuitable or impose limitations. The most suitable type of terminal is one which has non-embedded display attributes.

For this reason, the **highlight** command should not be used if truly portable software is required.

See the **display** command which reverses the effect of **highlight**.

EXAMPLE:

highlight st_stklev,st_rol

if ... then ... else

Conditionally execute a statement.

SYNTAX:

if <expression> then <statement> [else <statement>]

DESCRIPTION:

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

The statement which follows **else** can be another conditional statement, but, if the statement which follows **then** is a conditional statement, the first expression must not have an **else** clause. An **else** clause always relates to the immediately preceding **if**.

Conditional expressions can include all supported arithmetic, relational and logical operators and parentheses can be used to force a particular order of evaluation. Relational and logical operators available are:

<	Less than.	bw	Begins with (alphanumeric data only).
>	Greater than.	ct	Contains (alphanumeric data only).
=	Equal to.	and	Logical and.
< =	Less than or equal to.	or	Logical or.
> =	Greater than or equal to.		
< >	Not equal to.		

EXAMPLES:

1)
```
if salepr < = costpr then\
        error "Sale price must exceed cost": goto 11
```

2)
```
if d_date > date and (cat = "A" or cat = "B")\
        then status = 1
```

3)
```
if flaga \
        then flda = val: display flda \
        else if flagb \
                then gosub DISPLAY B
                else gosub DISPLAY C
```

Input data from screen boxes.

SYNTAX:

input <box list> [bs = <label>] [eoi = <label>] [ni = <label>]

DESCRIPTION:

Positions the cursor to each box in turn and awaits input. If the input is valid then its value is assigned to the field associated with the box. If the input is not valid, then the bell is sounded and re-input is awaited. The box list may consist of individual box names separated by commas, ranges of box names separated by hyphens, or a combination of the two. Input takes place in the order specified in the box list.

Data is validated according to the field associated with the box, firstly for correct type and secondly against any validation list associated with the field.

During input, certain keys have special functions:

RETURN (ENTER on some keyboards)
The RETURN key is used to terminate the input in each box. If used at the beginning of a box, no other characters having been typed, then input to that box is skipped subject to valid data being present in the box. If there is no validation list then a blank box is considered valid.

BACKSPACE
The BACKSPACE key erases the last character typed. If it is used at the beginning of a box, then the cursor is moved to request input at the previous box in the list. Attempting to backspace beyond the first box in the list is rejected and causes the bell to sound. However, if the **bs = <label>** clause is present, control is transferred to the line indicated.

(cont.)

EOI (End of Input)
The EOI key (as defined in the VDU parameter file) has the same effect as the RETURN key unless the **eoi** = <**label**> clause is present, in which case control is transferred to the line indicated.

CANCEL
If **cancel** is on (see the command **cancel**), pressing the CANCEL key (defined in the VDU parameter file) causes a **clear** command to be executed and control to pass to the option prompt. If **cancel** is off then the CANCEL key is ignored.

If the **ni** = <**label**> clause is present and if no fields are updated by the **input** command, then control is transferred to the line indicated. The **no input** condition is only satisfied if valid data is displayed in the boxes and no new values are entered. Skipping the boxes after a **clear** command implies that the associated fields are being updated with null values and does not count as **no input**. Further, this trap should be used with considerable care — it is easy to miss the implications of returning to the input statement for amendments after valid data has been entered.

EXAMPLES:

1) `input c_name,c_addr-c_status bs = I2 eoi = I5`

2) `input t_no ni = D5`

Insert a new record.

SYNTAX:

insert <file identifier> [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 = <label>** clause, 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. See section 3.6 for full details.

EXAMPLE:

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

interrupts

Enable or disable interrupts.

SYNTAX:

interrupts {on/off}

DESCRIPTION:

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

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

Certain types of terminal require **sage** to work in **raw** mode. In such a case the operating system does not check for special characters and keyboard interrupts are not recognised.

The default state for **interrupts** is off.

EXAMPLE:

interrupts on

Assign a value to a field.

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. The expression may include all supported arithmetic, relational and logical operators and parentheses may be used to force a particular order of evaluation. A relational or logical expression, or part expression, yields 0 if false and 1 if true.

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

EXAMPLES:

1) `let fullname = firstname / " " + surname`

2) `total = qty * price * (1 + vatrate)`

3) `roflag = stklev < rol`

match

Find and read the next matching record.

SYNTAX:

match <file identifier> [nsr = <label>] [riu = <label>]

DESCRIPTION:

Returns the next record whose key matches the key supplied to the previous **find** command applied to the designated 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 = <label>** clause, in which case control passes to the line indicated. An unsuccessful read 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 = <label>** clause, in which case control passes to the line indicated. The **record in use** status can only occur if the file is open in update mode.

EXAMPLE:

```
*fc = Find customer

        input c_name bs = FC3
        find cust nsr = FC2
FC1     display c_name - c_status
        prompt "Correct customer" yes = FC3
        clear
        match cust nsr = FC2
        goto FC1
FC2     error "Sorry - can't find customer"
FC3     end
```

Display a message.

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, as in examples 4 and 5 below, the message is displayed at the current cursor position instead of the bottom left hand corner of the screen.

A message remains displayed until another message or error message is issued or a **clear** command with no box list is executed.

See also the **error** command.

EXAMPLES:

1) **message "New record inserted for " + c_name**

2) **message "Updating sales ledger..."**

3) **message ""** (Clear the current message)

4) **at 8,40: message "This is at line 8 column 40"**

5)
```
at 12,10: vdu 50: \
   message "This is at line 12 column 10": vdu 51
```

newform

Re-display the screen form.

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.

EXAMPLE:

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

Read the next record.

SYNTAX:

next <file identifier> [nsr = <label>] [riu = <label>]

DESCRIPTION:

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. The **record in use** status can only occur if the file is open in update mode.

See also **rewind**.

EXAMPLE:

```
*n = Next
        clear
        next cust riu = N1
        display c_name - c_status
        end
N1      error "Record in use"
        end
```

nextkey

Read next key only.

SYNTAX:

nextkey <file identifier> [nsr = <label>]

DESCRIPTION:

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 it 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 record is the one whose key 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
        promptyn "This one" no = S1
        read cust
        .
        .
        end
S8      message "End of file reached"
S9      end
```

Open a file.

SYNTAX:

open <file identifier>

DESCRIPTION:

Opens a file that was initially declared closed with **!cfile** or has been closed with the **close** command.

Closing and reopening 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 is used.

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 open at one time varies according to the operating system but is normally at least six. If an attempt is made to open too many files at the same time then the program will abort. An attempt to open a file that is already open is ignored.

EXAMPLE:

```
open stock
```

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**.

This command is available only with Unix and certain similar operating systems.

See also the **wakeup** command.

Preserve file buffer from global clear.

SYNTAX:

preserve <file identifier>

DESCRIPTION:

Stops the **clear** command (with no fieldlist) from clearing the specified file'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
```

prev

Read the previous record.

SYNTAX:

prev <file identifier> [nsr = <label>] [riu = <label>]

DESCRIPTION:

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 **record in use** status can only occur if the file is open in update mode.

EXAMPLE:

```
*p = Previous
        clear
        prev cust riu = N1
        display c_name - c_status
        end
N1      error "Record in use"
        end
```

Prompt for yes/no reply.

SYNTAX:

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

DESCRIPTION:

The text expression is displayed centralised beneath the menu line, with **(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**> traps 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.

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
        prompt "Are you sure" no = D1
        delete cust
        clear
D1      end
```

read

Read a record.

SYNTAX:

read <file identifier> [key = <field list>] [nsr = <label>]
[riu = <label>]

DESCRIPTION:

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. 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. See section 3.6 for full details.

EXAMPLES:

1)
```
*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
```

2)
```
        ...
        .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
        ...
```

Read key data only.

SYNTAX:

readkey <file identifier> [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. See section 3.6 for full details.

EXAMPLE:

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

return

Return from a subroutine.

SYNTAX:

```
return
```

DESCRIPTION:

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

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
```

Reposition a file at its start.

SYNTAX:

rewind <file identifier>

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.

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

EXAMPLE:

```
rewind trans
```

scroll

Reset the scroll line number.

SYNTAX:

scroll [<**expression**>]

DESCRIPTION:

Resets the special variable **scrline** according to the value of the expression, as follows:

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

The scroll line number defines the line within the scroll area on which scroll data will be displayed and is the index value for all subscripted fields. See the **!scroll** declaration for details of setting up the scroll area.

By declaring **scrline** with **!temp**, its value may be directly referenced but it may not be altered by direct assignment.

The value in **scrline** may exceed the depth of the scroll area. If it does, data is displayed and indexed on a wrap around basis.

(cont.)

EXAMPLE:

The following example displays records from a transactions file in a scroll area that is five lines deep. At the end of each batch, the user is prompted to reply **y** for the next batch or **n** to terminate the display.

```
*dt = Display transactions
                rewind trans
DT1             clear: scroll 1
DT2             next trans nsr = DT3
                display t_date - t_amount
                if scrline%5 then scroll: goto DT2
                prompt "Next batch" yes = DT1
DT3             message "End of transactions file"
                end
```

setstr

Set a sub-string.

SYNTAX:

setstr(<dest>,<pos>,<len>,<source>)

DESCRIPTION:

setstr overwrites characters in the alpha field **dest** with characters from the string **source**. Overwriting starts at character position **pos** in **dest** and continues for **len** characters, or until the end of **dest** is reached, or until the end of **source** is reached, whichever occurs first.

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

The first character in **dest** is position 1. If **pos** is less than 1 or greater than the length of **dest**, then the command is ignored.

EXAMPLES:

```
setstr(code,1,4,prefix)

setstr(sent,p + 1,n,word)
```

Suspend the program for a number of seconds.

SYNTAX:

sleep <**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
```

testkey

Check for record with specified key.

SYNTAX:

testkey <file identifier> [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. See section 3.6 for full details.

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,fstname 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
```

unlock

Unlock a record.

SYNTAX:

unlock <file identifier>

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**).

EXAMPLE:

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

Send an alarm interrupt to a process.

SYNTAX:

wakeup <task id>

DESCRIPTION:

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

A simple way of determining the task id of another program is to cause each participating program to write its id into a shared file.

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

This command is available only with Unix and certain similar operating systems.

EXAMPLE:

wakeup re_id

Write a record back.

SYNTAX:

write <file identifier> [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 eoi = A2
A2              prompt "Amendments correct" no = A1
                write cust
                clear: end
```

Send a vdu control sequence to the screen.

SYNTAX:

vdu <n>

DESCRIPTION:

Sends control sequence number **n** from the vdu parameter file to the screen at the current cursor position. The cursor can be positioned using the **at** command before the vdu control sequence is called.

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 one of these sequences, but it is primarily intended for use with sequences 50 through 59, which are available for any use the programmer requires.

EXAMPLE:

at 7,28: vdu 59

3.11 COMPILING AND RUNNING A SAGE PROGRAM

After creating the text source file of a program it must be compiled using the program **cf**. The source code file must have a **.f** extension and the compiler requires access to the descriptor files (**.d** extension) corresponding to each **!file** and **!record** declaration. Unless these have pathnames, the compiler expects to find them in the current local directory. The syntax for calling **cf** is:

cf <program name>

The **.f** extension on the program name does not have to be typed since the compiler assumes it. If the compilation is successful, a file is created with the same name stem as the program but with a **.g** extension. **sage** requires this output file in order to run the program. If the compilation is unsuccessful then any existing **.g** file is unaltered and one or more error messages are output. These indicate the line number at which the error was detected and point to the offending part of the line.

The command line for running a **sage** program is

sage <program name> [<arguments>]

where <**arguments**> is an optional list of parameters, separated by spaces, that may be referenced through the special temp **arg**.

Refer also to the installation instructions on setting up vdu parameter files.

CHAPTER 4 REPORT GENERATION (SAGEREP)

This chapter explains the use of the **sagerep** program to produce printed reports. The chapter is divided into seven sections. Sections 4.1 to 4.4 explain the overall structure and main features of **sagerep**; Sections 4.5 and 4.6 explain all the Declarations and Commands; Section 4.7 explains the compilation and running of **sagerep** programs.

4.1 INTRODUCTION TO SAGEREP

sagerep is a powerful program which takes care of much of the complex logic associated with producing reports, leaving the programmer free to concentrate on the information to be printed. Although designed primarily as a report generator, **sagerep** is sufficiently flexible to undertake other tasks, such as global file updating and, by redirecting standard output, the creation of text files for input to programs written in other languages.

To make the best use of **sagerep** it is important to understand how it operates. The explanation on the following pages is supplemented by a flowchart (see page 4-5). Note also that **sagerep** is a batch processor and that, with the exception of a simple input and display facility, it has no interactive features.

The sequence below shows how **sagerep** is used in a simple SCULPTOR application:

1) In any available text editor, write your source code program using the **sagerep** language declarations and commands.

2) Run the **cr** program to compile the source code file into an object program.

3) Run the new **sagerep** program.

Note: Steps 1 and 2 can be performed automatically using the **rg** program to create a standard report program. Refer to Chapter 5 for details of the **rg** program.

SAGEREP STATEMENTS

Statements are grouped into sets, each set being executed at the appropriate time in the overall processing logic. Within any one set, statements are executed in the order in which they are declared within the program. The sets are:

DRIVING FILE
!file

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

TITLE STATEMENTS
!title

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

PAGE HEADING STATEMENTS
!heading

FOOTNOTE STATEMENTS
!footnote

ON STARTING STATEMENTS
!on starting

ON ENDING STATEMENTS
!on ending

FINAL STATEMENTS
!final

MAIN STATEMENTS
The main body of statements.

STATEMENT EXECUTION SEQUENCE

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

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

The **on ending** statements are checked. If any require executing, **sagerep** temporarily backtracks to the state that existed before the current set of records was read, but any temporary data modified by the **on ending** statements is carried forward.

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

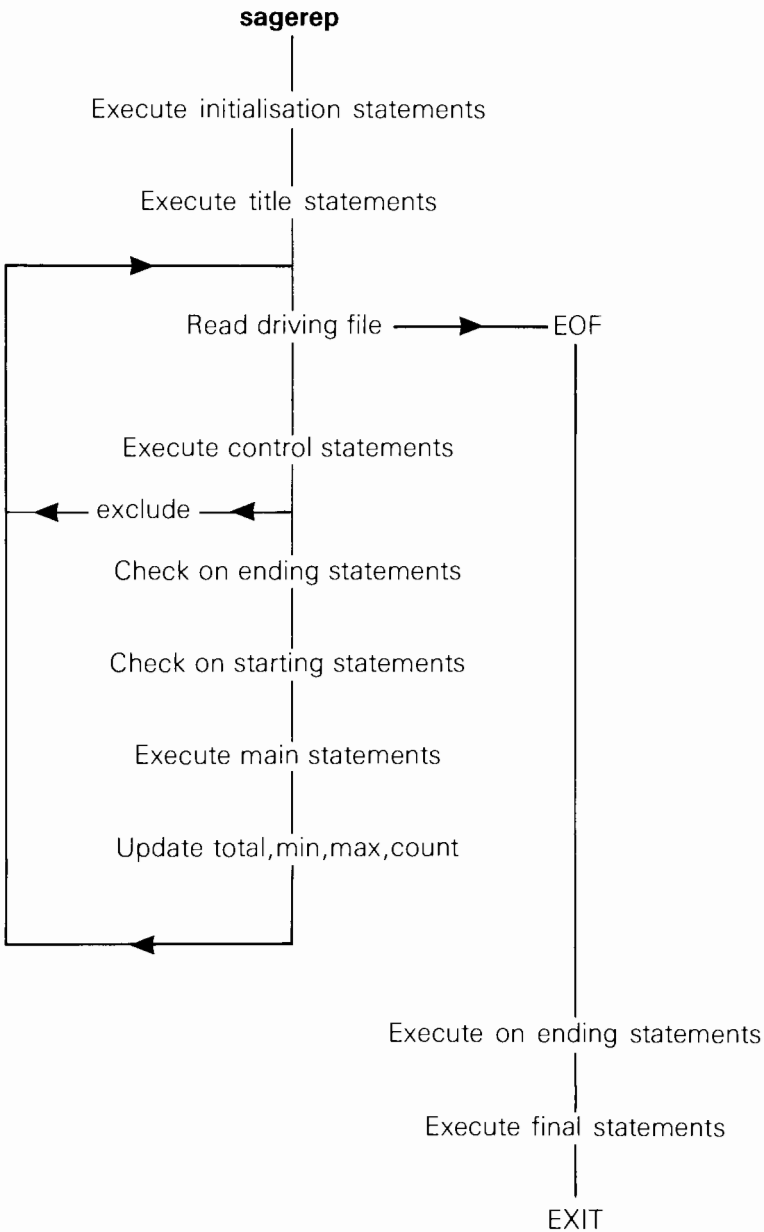
Finally the main block of statements is executed, following which all total, min, max and count function accumulators are updated.

When an end of file condition occurs on the driving file, the **on ending** statements are executed, followed by the final statements.

Heading statements are executed at the top of each page and footnote statements at the bottom of each page. Since these two statement sets can be invoked at any time, care should be taken not to cause unintentional changes. For example, it is normally wise to ensure that **scrline** is not altered, so if the **scroll** command is used in headings, the following code is recommended:

```
!heading scrsave = scrline
<other heading statements>
!heading scroll scrsave
```

SAGEREP FLOWCHART



4.2 SAGEREP PROGRAM STRUCTURE

A **sagerep** program is written as a text file using a standard editor. The language is line-oriented and the compiler recognises four different line types:

1) Lines commencing with a full stop ''.'' are comment lines and are ignored by the compiler. Completely blank lines are also ignored by the compiler.

2) Lines commencing with a plus sign ''+'' are format definitions and should appear first in a program.

3) Lines commencing with an exclamation mark ''!'' are declarations.

4) Other lines are program statements. If the first word on the line is not a field name or SCULPTOR keyword then it is taken to be a line label. Multiple statements, separated by colons, may be placed on the same line.

Both declarations and program statements may extend over more than one physical line by terminating each line that is to be continued with a backslash ''\''. This is particularly useful when several statements are to be controlled by an **if ... then** command.

A typical **sagerep** program has the following structure:

<format definitions>

<declarations>

<main statements>

<subroutines>

Since **sagerep** reads the driving file automatically, the main block of statements does not require a command to obtain the next record. There is also no need to program a loop into the main statements, their execution for each selected record being implied. However, if one or more subroutines are included in the program, then the main statements must terminate with an **end** command. The following example shows how straightforward a **sagerep** program which produces neat output can be:

```
!heading printh *st_code,*st_desc,*st_stklev,*st_costpr
!heading print
        keep 6
        printh st_code,st_desc,st_stklev,st_costpr
```

It is possible to write a **sagerep** program which reads all required records and does all processing explicitly by using a **!title** declaration to call a subroutine containing the required logic and having an **exit** command as a single main statement. A program like this cannot make use of any control statements such as **!on starting**, although **!heading** statements are still effective. Example:

```
!title gosub MYLOGIC

        exit

MYLOGIC\
.
.
.
return
```

4.3 SAGEREP FORMAT DEFINITIONS

By default, the heading and format used for a field are the ones defined when the field was described (see the **describe** program in Chapter 2). A particular report program may wish to override these and can do so by including a format definition for the field:

+ <field name>,[<heading>][,<format>]

Either a heading or a format or both may be specified.

4.4 SAGEREP EXPRESSIONS AND OPERATORS

sagerep supports a comprehensive set of arithmetic and relational operators. These may be used to form expressions involving keyed file fields, temporary variables and constants.

PRECEDENCE

Operators have precedence as set out below, but sub-expressions may be enclosed in parentheses to force a particular order of evaluation. The available operators are listed below. The operators grouped together have equal precedence and the groups are listed in descending order of precedence.

Group 1:

-	Negate (unary minus)

Group 2:

*	Multiply
/	Divide
%	Remainder (after integer division)
/	String concatenation (with trailing spaces stripped from left operand)

Group 3:

+	Add
-	Subtract
+	String concatenation (with trailing spaces retained from left operand)

Group 4:

=	Equal to
<	Less than
>	Greater than
< =	Less than or equal to
> =	Greater than or equal to
< >	Not equal to
ct	Contains
bw	Begins with

(cont.)

Group 5:

and	Logical **and**

Group 6:

or	Logical **or**

The operators **ct** and **bw** apply to alphanumeric data only. **ct** yields true if the right operand is a sub-string of the left operand. **bw** yields true if the left operand begins with the right operand. Examples:

```
if ''Smithson'' bw ''Smith''     (true)
if ''Smithson'' bw ''son''       (false)
if ''ABCDEF'' ct ''CD''          (true)
if ''ABCDEF'' ct ''BD''          (false)
```

Relational expressions are most commonly used with the **if** command, but it is also permissible to have a relational expression as part of an assignment statement; its value is 1 if true or 0 if false.

CONSTANTS

Numeric and alphanumeric constants may be freely included in expressions. Numeric constants may be integer or floating point and positive or negative; a numeric constant is floating point if it includes a decimal point. Alphanumeric constants, sometimes called strings, must be enclosed in either single single or double quotes ('' or ''''). Examples:

```
123          (integer)
-50          (integer)
34.451       (floating point)
100.0        (floating point)
'Thursday'   (alphanumeric)
''today's''  (alphanumeric)
''''         (null string)
```