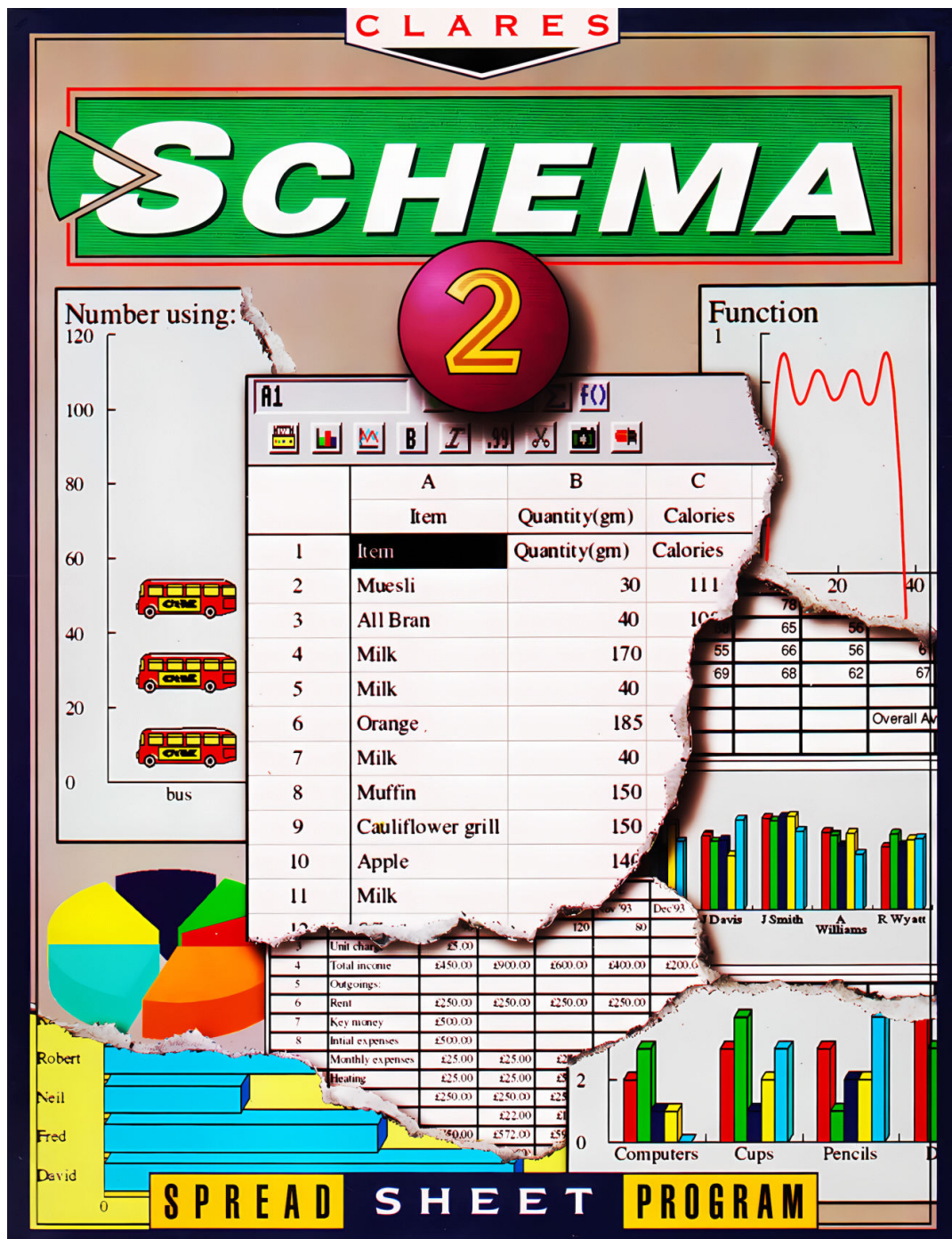




Schema2

Introduction to Macros

ProAction

3QD
DEVELOPMENTS LTD

APDL

Schema2

© CRM Ltd, ProAction & 3QD Developments Ltd 2026
All rights reserved.

No part of this product may be reproduced in whole or part by any means without written permission of the publisher. Unauthorised hiring, renting, lending, public performance or broadcasting of this product or its parts is prohibited.

While every care is taken, the publisher cannot be held responsible for any errors in this product, or for loss of data or consequential effects from the use of this package.

Software and Documentation supplied by
APDL
a part of
3QD Developments Ltd
Single User version downloaded from the free of charge software section of
www.riscos.com

Please note that any free of charge downloads from RISCOS.com are supplied without any technical support package as per the licence agreement.

This document last edited on 29/09/26



If you have found this application and this manual useful please consider making a donation to help cover the costs of cataloguing, documenting, updating and hosting the APDL applications. Scanning the QR code will take you to our donations page.

Contents

Introduction to Macros	5	Graphics	29
What is a macro?	5	4.1 Introduction	29
Uses for Macros	5	4.2 Graph Coordinates	29
The Schema2 Macro language	6	4.3 Drawfiles	29
The Macro Editor	6	4.4 Graphics objects and handles	30
Capturing Macros	6	4.5 Opening, Loading and Saving Drawfiles	30
Graphics macros	6	4.6 Style setting functions	31
Formal description of the Macro Language	6	4.7 Writing components into Drawfiles	33
Evaluation	6	4.8 Utility functions	39
Standard Functions for Macros	6	4.9 Examples	40
Example macros	6		
What is a macro?	6	Interactive Graphics	44
Uses for Macros	6	5.1 Problems of interaction	44
Schema2 Macros - an informal introduction	4	5.2 The Graph Life Cycle	44
1.1 Overview	7	5.3 Editing actions	44
1.2 Program architecture	7	5.4 Hotlinking	45
1.3 Macro header lines	8	5.5 Detailed structure of the Parameter String	45
1.4 Remarks	9	5.6 Setting up graph macros	50
1.5 Local variable declarations	9	5.7 The process of drawing graphs.	51
1.6 Assignments	10	5.8 Parameter vectors	52
1.7 FOR statements	10	5.9 Handling the parameter string	53
1.8 Conditional statements	10	5.10 Parameter string manipulating macros	54
1.9 Return statement	11	5.11 Normalising the data	55
1.10 Ending a macro	11	5.12 Working out the graph layout.	55
1.11 A Practical example	11	5.13 Plotting the graph	56
1.12 WHILE... ENDWHILE	13	5.14 Summary of requirements of an interactive graphics macro	57
1.13 REPEAT... UNTIL	13	5.15 Conclusion	58
1.14 Labels and gotos	13		
1.15 ON_ERROR_EXIT	14	Macro Reference	59
1.16 ON_ERROR_CONTINUE	14	6.1 Lexical Rules	59
1.17 Standard functions	14	6.2 Operands	59
1.18 Macro Capture	14	6.3 Expressions	61
1.19 Compiling and running macros	15	6.4 Macros	61
1.20 Direct links with sheets	17	6.5 Statements	64
1.21 Calling Macros	17	6.6 Error handling in macros	67
1.22 Workspaces and macros	18		
Macro Editor and Compiler Reference	19	Evaluation	69
2.1 Entering the editor:	19	7.1 General considerations	69
2.2 Loading a document for editing	19	7.2 Macros which do not affect spreadsheet cells	69
2.3 Editing	19	7.3 Macros which store values in cells	69
2.4 The Icon Bar Menu Structure	20	7.3 Macros which cause re-evaluation of the spreadsheet	69
2.5 The Main Menu Structure	20		
2.6 The Compiler	23		
2.7 Customising the Menu	23		
2.8 List of hot keys	23		
Capturing Macros	23		
3.1 Macro-based Standard Functions	24		
3.2 Recording Macros	24		
3.3 An example of a captured macro	25		
3.4 Data Input functions	26		
3.5 Converting captured macros	27		

Standard functions available from Macros	72
8.1 Array sizing functions	73
8.2 Graphics functions	74
8.3 Type and error management	74
8.4 Functions which create, destroy and re-size	75
8.5 Functions which enter, move and delete	76
8.6 Functions which affect appearance	77
8.7 Loading and Saving functions	79
8.8 View manipulation functions	80
8.9 Style modifying functions	81
8.10 Print functions	82
8.11 Input functions	83
8.12 Functions to providing and modifying attributes	83
8.13 File functions	86
8.14 Miscellaneous functions	87

Introduction to Macros

This volume is supplementary to the Schema2 Users Guide.

Macros form a powerful extension to the basic facilities provided by the Schema2 spreadsheet. The Schema2 macro language is modelled on BBC BASIC, and macros offer all the power and flexibility in data handling you would expect from a conventional programming language. However, you can only make full use of these features if you know how to write computer programs in BASIC or a similar language.

What is a macro?

When you use Schema2 in the normal way, you handle tasks by giving sequences of commands through the menu or the keyboard. For example:

- Open a new sheet with 26 columns and 5 rows
- Put the string "Jan" into B1
- Select B1..Z1
- Run **fillin** from the main menu
- Put the string "Income" into cell A2
- Print the sheet
- Select B5..Z5
- Draw a graph
- Plant the graph in the sheet
- Print the sheet
- Save the sheet as "Forecast"
- Quit

A macro is a sequence of commands, recorded and stored in a file. Whenever you execute a macro, Schema2 arranges to fetch these commands and obey them, one by one. For example, the sequence of commands we have just listed could be stored as a macro; when executed, this macro would rebuild and print the Forecast spreadsheet whenever you wanted it to do so.

In practice you would not want to make repeated copies of the same spreadsheet; you might like to try out the effects of different interest rates, or use sets of data taken from various sources. Macros can be given the flexibility to do this.

Uses for Macros

Macros can be used in several different contexts, which are described below:

1. Capturing and replaying sequences of commands.

A macro can be used to store and then replay almost any sequence of commands you give Schema2. Such macros are useful if you use a spreadsheet at regular intervals (say weekly) and follow a fixed routine.

2. Augmenting standard functions

You can use macros to add to the set of Schema2 standard functions. For example, Schema2 does not possess a standard function to find the difference, in days, between any two dates. You could easily write a macro to do the job and then use it just like a standard function.

Some of the standard functions provided with Schema2 - notably MATCH, SEARCH and all the Lotus 1-2-3 support functions - are already implemented as macros.

3. Doing routine jobs

Your applications may require parts of spreadsheets to be filled in some standard way, using a fixed set of rules. This can be done with a suitable macro. For example the fillin feature, with its ability to recognise numbers and dates of various sorts, is implemented by a macro.

4. Graph plotting

Macros are invaluable in plotting graphs and pictures. All the graph plotting features of Schema2 are implemented with macros, and you can use macros to develop and install new graph types.

5. Customised applications

A 'customised application' is one in which the eventual user does not need to have any specialised knowledge of macros or even of spreadsheets. All the user needs to do is to follow a very simple set of instructions displayed on the screen, and the answer to the problem (whatever it is) is printed automatically. An example of a customised application would be a chess match supervisor. First, the user keys in the names of the players and is presented with a printed list of matches to be played. Later, he supplies the results of the matches and receives a match analysis with the players ranked in order of merit.

We have organised the remainder of this manual into a series of chapters which cover various aspects of macros. Here are overviews of these chapters:

The Schema2 Macro language

Chapter 1 gives an informal description of the Macro language, with examples. In writing this chapter we have assumed that you are already familiar with BASIC (in any of its variants) or with some other programming language like PASCAL or COMAL. If you have not done any programming at all, you will find the chapter unrewarding, and we advise you to spend some time studying BBC BASIC from one of the standard textbooks before you start on Schema2 macros. The same applies if your programming experience is limited to Machine Code, Assembly Code or a Logic language like PROLOG.

The Macro Editor

Chapter 2 describes the Macro editor. A macro is a sequence of commands, written in a formal language similar to BASIC. Before such a sequence can be used to control Schema2, it must be compiled or translated into a special notation which can be executed directly by Schema2.

Macros are compiled by MacroEdit, a utility program supplied with Schema2. With MacroEdit you can set up new macros, edit, modify and correct existing ones. When you think a macro is ready, MacroEdit will compile it. If it finds any mistakes in your macro it displays them; otherwise it hands the compiled macro over to Schema2 for execution.

In Schema (the predecessor to Schema2) the macro editor was integrated with the spreadsheet program. We have now separated the editor as an application in its own right. This cuts down on memory requirements and reduces loading time for macros.

Capturing Macros

Chapter 3 describes the technique of capturing a sequence of manual operations and turning them into a useful macro for a routine job.

Graphics macros

The techniques of writing macros to display graphics and link them with the graph editor are discussed in Chapters 4 and 5.

Formal description of the Macro Language

Chapter 6 is a full description of the macro language, in formal terms. Use this chapter as reference material.

Evaluation

When macros are executed in the context of a spreadsheet with intelligent evaluation, the exact order of evaluation is subject to certain rules. These rules are discussed in Chapter 7.

Standard Functions for Macros

Schema2 includes many standard functions which are only accessible through macros. Chapter 8 contains descriptions of these functions.

Example macros

The disc supplied with this manual contains the source code of numerous macros, including all the macros supplied as standard with Schema2.

Chapter 1: Schema2 Macros - an informal introduction

1.1 Overview

The Schema2 macro language is a programming language similar to BASIC, designed to run in the context of the Schema2 spreadsheet.

The language is a large subset of BBC BASIC, with a few additions. We assume that you are familiar with the BBC version of BASIC or any other modern, block-structured version of this language. Alternatively, a knowledge of Pascal, C or Ada will be helpful. Our informal introduction starts from this point.

If you can write a BASIC program you will be able to write and use most types of macros quite effectively after reading this chapter. You need then only refer to the rest of the macro manual to look up specific points such as the definitions of standard functions.

Graphics macros are hard to write well. If you plan to follow this road, read Chapters 4 and 5 carefully, and study the examples in detail before you start.

1.2 Program architecture

A macro is a named self-contained segment of code, which does a specific task and delivers a result. Macros are closely similar to named procedures in BASIC. They can be called from within spreadsheet cells, or from the macro menu, from other macros, or (recursively) from themselves.

When you use a macro from inside an expression or formula (these two words mean the same), it is indistinguishable from a standard function call. In fact we have implemented some functions of Schema2 with macros. For example,

`UPPER("deimos and phobos")`
gives

`"DEIMOS AND PHOBOS"`
and is a call to a standard function; but

`L_PROPER("deimos and phobos")`
gives

`"Deimos And Phobos"`
and is a call to a macro.

The difference is that standard functions are a fixed part of the Schema2 program; but macros are written in the macro language, and you can add to them and alter them as you please.

Like standard functions, nearly every macro takes one or more parameters, which are the 'input quantities' to the computation. Each macro type has its own pattern of parameters, some of which are scalars (single quantities) whilst others represent blocks or arrays.

Sometimes parameters are referred to as arguments. We use the two words interchangeably.

Macros do not use line numbers; you exercise control by **if-then-else-endif**, **for**, **while** and **repeat** constructions, and by calling other macros as subroutines. The language lets you label statements and use the notorious **GOTO**, but this practice is not in keeping with current programming techniques. We deprecate the use of **GOTO** except in special conditions such as error handling.

Macros are capable of every type of task, mathematical computations, searching, string handling and graphics. Mathematical macros begin to be useful when they calculate functions which cannot be written down as simple formulae. The sample macro which follows is based on Pythagoras' Theorem; it calculates the hypotenuse of a right-angled triangle given the other two sides:

```
MACRO Pythagoras (a,b)
  = SQR(a^2+b^2)
ENDMACRO
```

Below is a spreadsheet in which this macro has been used. The top row and the left hand column are filled with the numbers 1,2,3,4 and the whole block of cells B2..E5 with the formula

```
= Pythagoras(@a2, b@1)
```

so the cell in C2, for example, displays the square root of (32+22). This ensures that the parameters in each cell are the same as the numbers in column A and row 1 respectively. The results clearly show the famous relationship $3^2+4^2=5^2$ •

	A	B	CD
1	1	2	3 4
2	2	2.83	3.61 4.47
3	3	3.61	4.24 5.00
4	4,	4.47	5.00 5.66

Here is another example, which calculates the range (that is the difference between the largest and the smallest value) over an array of cells. (We have conveniently forgotten that this can also be done with the **max** and **min** standard functions!)

```
MACRO range(a())
REM : Finds range of numbers in an array
LOCAL low, high,j, k, rows, cols
rows = FIRST(a)
cols = SECOND(a)
low = a(0,0)
high = a(0,0)
FOR j = 0 TO rows-1
  FOR k = 0 to cols-1
    IF a(j,k) > high THEN high = a(j,k)
    IF a(j,k) < low THEN low = a(j,k)
  NEXTk
NEXTj
= high - low
ENDMACRO
```

To execute this macro you must first compile it and load it into Schema2. The way to do this is described in Section 1.19 .

When you have loaded the macro you can enter this formula into cell **a1** of a sheet, which contains data:

```
=range(a5..g9)
```

The macro scans the 35-cell block **a5..g9**, extracts the largest and the smallest values, and returns their difference. It illustrates most of the important features of macros, which we discuss in the following sections:

1.3 Macro header lines

The macro is headed by a line

```
MACRO range(a())
```

which starts with the reserved word **MACRO**, followed by the macro name and a list of parameters in brackets. In this case there is only one parameter, an array, but macros can have any number of parameters which may be either arrays or scalars. An array parameter is identified by a pair of empty brackets.

If a macro has no parameters, there should be no empty brackets after the macro name in the header; but when the macro is called from a formula the brackets are still needed. For example:

```
macro text:
  MACRO year_date
    = year(now())
  ENDMACRO
entry in cell:
  =year_date()
```

A macro can have any name you wish, except for reserved words, which include **MACRO**, **IF**, **ELSE**, and any sequence of characters which could be a cell name. Remember cell names can have up to four letters, eg. **LIB3** is a cell name.

You should know that all macros share the same name space, so take care not to duplicate the names of any other macro in the system. This is why many of the standard macros supplied with Schema2 have names which start with CRM. To avoid problems don't start your macro names with these letters.

1.4 Remarks

Any sequence which starts with the characters **REM** (except inside a string) is taken as a comment and is ignored by the compiler. You may wish to use remarks to explain what is happening in your macro.

1.5 Local variable declarations

The next line

```
LOCAL low, high, j, k, rows, cols
```

declares some local variables - **high**, **low**, **rows**, and so on. There is no limit to the number of local variables, and their definitions may be spread out over several lines. Each line must start with the keyword **LOCAL**, and the declarations must be written before any other statements in the macro.

You may use local arrays of one or two dimensions, provided that you state the size when you declare them, thus:

```
MACRO sample (n)
LOCAL forest(30)
LOCAL spinney(n)
LOCAL wood(20,10)
```

This declares :

- a local array of one dimension called **forest**, with 31 elements labelled **forest(0)** to **forest(30)**.
- a local array of one dimension called **spinney**, with **(n+1)** elements called **spinney(0)** to **spinney(n)**. **n** is an input parameter of the function.
- a local array of two dimensions called **wood**, with 21 rows and 11 columns. The corner elements are **wood(0,0)**, **wood(0,10)**, **wood(20,0)** and **wood(20,10)**.

Once a local array has been set up, you can refer to individual elements by using subscripts. The number of subscripts (1 or 2) depends on the number of dimensions in the array.

For example, an element in **wood** is **wood(6,4)**.

Array parameters can also be subscripted; in our example **a(0,0)** is the top left cell in parameter block **a**.

Subscripts are not limited to numbers but can be any expressions. A subscript which is outside the allowable range will produce an error.

You can use any names you wish for local variables and parameters, except reserved words and cell names. The local variables and parameters in any macro share the same name space, but are distinct from variables with the same names in other macros.

The quantities in Schema2 do not have any 'type' permanently attached to them, but may be integers, real numbers, strings, error codes or graphic handles - or even all five at different times. This is why local variable names do not have trailing \$ or % signs.

Local variables and arrays are initialised to zero.

Parameters are like local variables which have been initialised to the calling values; otherwise they behave in the same way as local variables and can be used for assignments, etc. Assigning to a scalar parameter does not alter any value outside the macro; but assigning to an element of an array parameter **does** change the original array. In the technical jargon scalars are called by *value*, but arrays are called by *reference*.

If your macro does a simple calculation on a few values then it is best to use scalar parameters. If your macro does a calculation on many values or modifies the contents of some cells then it is best to use array parameters.

1.6 Assignments

The next four lines,

```
rows = FIRST(a)
cols = SECOND(a)
low = a(0,0)
high = a(0,0)
```

are all assignments. If you prefer you can write them with the reserved word

LET, as in

```
LET rows = FIRST(a)
```

In general you can assign the value of any expression. Expressions are calculated by the same rules as those used in spreadsheets.

FIRST is a standard function which you apply to an array to get the number of rows; **SECOND** tells you the number of columns.

Arrays of one dimension are taken as column vectors, so **FIRST** gives the number of elements, and **SECOND** always returns 1 if applied to a one-dimensional array.

1.7 FOR statements

An example is

```
FOR k = 0 TO cols-1
    .....
NEXTk
```

FOR statements (including nested **FOR** statements) work as in BBC BASIC. The control variable starts at the initial value and then moves up (or down) by the given step size until it 'passes' the final value. The default step size (which need not be stated) is 1.

Notice that it is possible for a **FOR** statement to be executed zero times; this would happen with - say -

```
FOR z = 6 TO 3
    .....
NEXT
```

If a **FOR** statement uses a step size other than 1, the size is indicated by the keyword **STEP**. For example, the **FOR** statement

```
FOR j = 3 TO 6.5 STEP 0.75
    .....
NEXTj
```

will be executed for five values of j : 3, 3.75, 4.5, 5.25 and 6.

If the step size is negative the control variable changes downwards. For example the statement

```
FOR j = 6 TO 3 STEP -1
    .....
NEXTj
```

will run with j = 6, 5, 4 and 3 in that order.

1.8 Conditional statements

Conditional statements, such as

```
IF a(j,k) > high THEN high = a(j,k)
```

work in the same way as in BBC BASIC. If the block of statements to be executed extends over more than one line the special keyword **ENDIF** is used to terminate the block, as in

```
IF p(q) < lowest THEN
    lowest = p(q)
```

```
    low_pointer = q
ENDIF
```

Alternative actions can be prefaced by ELSE, and conditional statements can be nested.

1.9 Return statement

Every macro must return a value; and the command which does it consists of an = sign, followed by the expression to be returned. The return statement in our example is

```
= high - low
```

Some macros (such as the one in the example which follows this section) do not have a sensible return except for an error marker. Nevertheless each macro must return something, even if the value is a dummy.

1.10 Ending a macro

The macro definition ends with the word `ENDMACRO` on a line by itself

1.11 A Practical example

It is often necessary to smooth numerical data by taking moving averages. This process is specially useful if the data is subject to random fluctuations; the smoothed curve is more likely to reveal the general trend.

Smoothing is done by replacing each point by the average of the point itself and its neighbours in either direction. It is normal to use an odd number of points in all so points on both sides of every data point are equally represented.

Special care is needed at the ends of the range, where there are not enough points to do the calculation in the standard way. Suppose that the raw data points are called a0,a1,a2,a3, the smoothed points are b0, b1, b2, b3, and the moving average is to be taken over 5 points.

To cope with the end effects, we set:

```
b0 = average of (a0,a1,a2 )    (3 points)
b1 = average of (a0,a1,a2,a3) (4 points)
b2 = average of (a0,a1,a2,a3,a4) (5 points)
b3 = average of (a1,a2,a3,a4,a5) (5 points)
```

At the end of the array similar arrangements are made.

The macro shown below takes a column of figures as one parameter, and writes a smoothed set into the column given as the second parameter. The third parameter states the number of points to be used in each average.

The macro is concerned to ensure that the data is given correctly; both the array parameters must be columns, they must be the same size, and the number of points must be odd.

In all cases the macro returns a string which indicates the outcome of the process. This is either an error report such as "bad dimension" or an indication of success. Returning a string in this way is not a rule in Schema2 (although every macro must return something) but a useful convention. If the macro is invoked from a spreadsheet cell the reply is automatically displayed to the user.

```
REM *****
REM * This takes a column of data and smoothes it by computing
REM * a moving average. The result is written to column b
REM * The number of points to be used in each average is n
REM *****
MACRO smooth_column (a(),b(),n)
LOCAL s      REM half(number of points-1)
LOCAL q      REM main point counter
LOCAL realn  REM number of points actually used
LOCAL z      REM starting point for each average
LOCAL p      REM point counter for each average
LOCAL total  REM to sum point totals
IF SECOND(a) <>1 THEN = "bad dimension"
IF SECOND(b) <>1 THEN = "bad dimension"
IF FIRST(a) <> FIRST(b) then = "bad dimension"
```

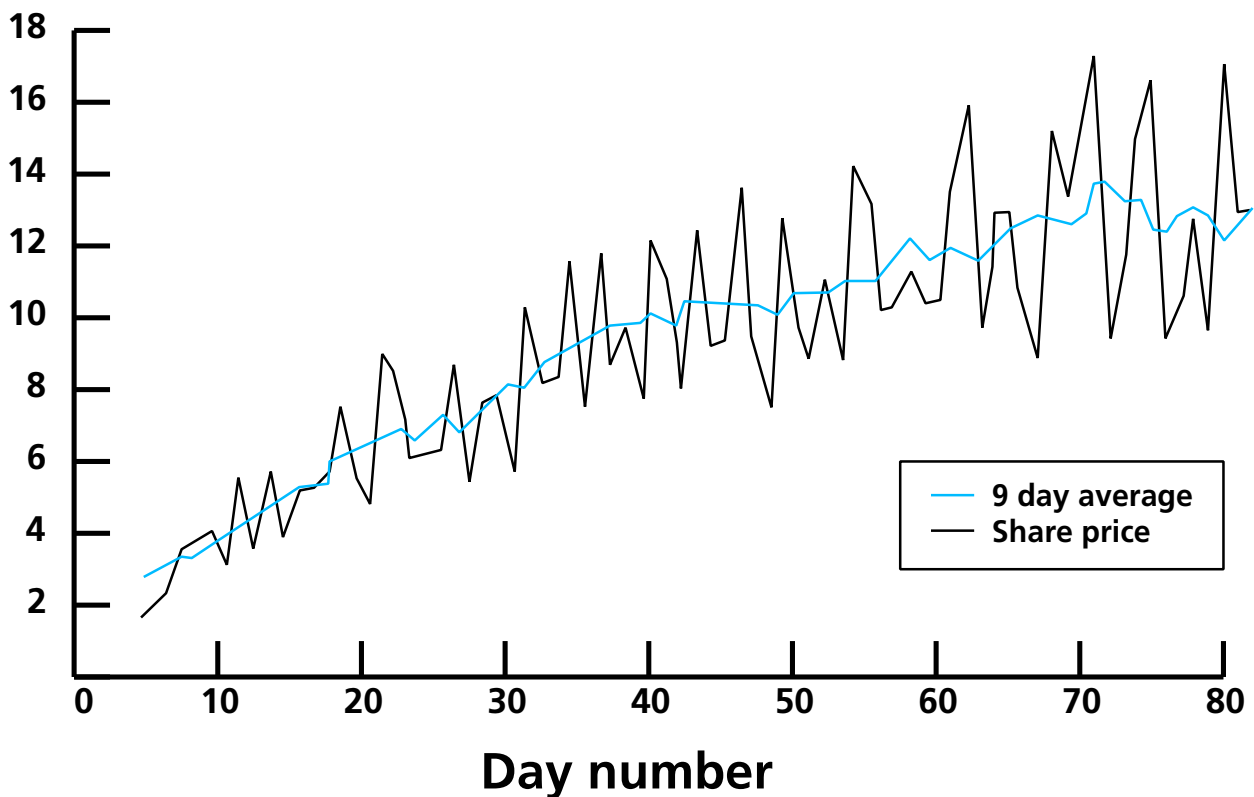
Schema2

```
IF n%2 <> 1 OR n < 1 OR 2*n > FIRST(a) THEN
="bad number of points"
ENDIF
s=(n/2-0.5)
FOR q = 0 to FIRST(a)-1
total=0
realn = n
z = q-s
IF q < s THEN
REM look after end effect at start
reain = n-(s-q)
z=0
ENDIF
IF q >= (FIRST(a) - s) THEN
REM look after end effect at finish
reain = s - q + FIRST(a)
ENDIF
FOR p = 0 TO realn-1
total = total+a(z+p,0)
NEXTp
b(q,0) = total/reain
NEXTq
="completed"
ENDMACRO
```

To test this macro, we set up a sheet with 4 columns and 80 rows. The first column was labelled Day number and filled with the numbers 1-79. The second column was filled with a (fictitious) share price for each day. Then cell D1 was loaded with the formula

```
=smooth_column(b2..b80,c2..c80,9)
```

This filled column C with numbers, and when they were plotted with standard lineplot function we obtained -



1.12 WHILE... ENDWHILE

This is the first of several features not yet described. The **WHILE ... ENDWHILE** construction is an alternative way of controlling repetition. An example is

```
MACRO powers_of_two
LOCAL z,s
z=1
s=0
WHILE z < 1000
s=s+1
z=z*2
ENDWHILE
= s
ENDMACRO
```

The while statement includes an expression. The body of the construction, which consists of all the statements between **WHILE** and its matching **ENDWHILE**, is obeyed repeatedly until the expression turns out to be false. In our example *z* is doubled every time, and takes the values 1 2 4 8 16 32 64 128 256 and 512. When it gets to 1024 the expression in the **WHILE** statement becomes false and the loop terminates.

In practice, make sure that the body of every **WHILE** statement includes an assignment which will eventually make the expression false - otherwise the loop never stops. Remember that the **ESCAPE** key will stop an infinite loop!

1.13 REPEAT... UNTIL

Another way to control iteration is the **REPEAT - UNTIL** construction. This is similar to **WHILE - ENDWHILE** except that the test is made after the body of the loop. This implies that the loop must be executed at least once. For example:

```
MACRO mysqrt(n)
LOCAL w
w=1
REPEAT
w = 0.5 * (w+n/w)
UNTIL ABS(w*w - n) < 0.000001
=w
ENDMACRO
```

1.14 Labels and gotos

Labels and **gotos** let the programmer sequence the commands in a macro in a totally undisciplined manner.

Any statement in a macro can be labelled by preceding it with an identifier, which can be made up using the same rules as for local variable names. The label should be followed by a colon.

The language supports the **GOTO** statement, in which the word **GOTO** is followed by the label of a statement elsewhere in the same macro. When the computer reaches the **GOTO** it immediately jumps to the labelled statement and continues execution from that point. Programs which use **GOTOS** can be difficult to understand. In fact, **GOTOS** are rarely needed as more sophisticated control functions like **IF**, **FOR** and **WHILE**, can do everything necessary. Don't use a **GOTO** unless you really cannot reach your aim in some other way.

As an example here is the equivalent of a **WHILE - ENDWHILE** loop using **gotos**:

```
MACRO horrible
LOCAL z,s
s=0
z=1
start: IF z<1000 GOTO continue
GOTO end
continue: s=s+
z=z*2
GOTO start
end: =s
ENDMACRO
```

1.15 ON_ERROR_EXIT

This command is used to help with error control.

You will remember that expressions and standard functions can sometimes produce errors as their results; for example, the expression produces the result

```
2/(4-4)
produces the result

Division by zero
```

If a macro executes the command **ON_ERROR_EXIT**, then any attempt to assign an error to a local variable or a spreadsheet cell is trapped, and makes the macro terminate. It returns the error it would have assigned.

The command can prevent a macro trying to calculate with invalid numbers.

Consider this macro:

```
MACRO function (fa)
LOCAL z,sum,t,term
ON ERROR EXIT
REM Find the sum of the first 10 terms of the series
REM 1, z, z*z\2, z*z*z\2^,... where z = sqrt(1-fa*fa)
z = sqrt(1-fa*fa)
sum = 0
term = 1
FOR t=1 TO 9
sum=sum+term
term = term*z/t
NEXTt
= sum
ENDMACRO
```

If the parameter supplied as **fa** is negative, the function returns with the error message "Square root of negative number" without executing any of the code which follows the line with **sqrt**. The macro detects an error condition when it tries to assign a value to **root**, and immediately returns the error. **ON_ERROR_EXIT** only affects the macro in which the command was called.

1.16 ON_ERROR_CONTINUE

This command cancels the effect of **ON_ERROR_EXIT**.

1.17 Standard functions

Macros have access to a large library of standard functions which are not available for standard spreadsheet expressions. These functions can be divided into several groups:

1. Every manual operation on a spreadsheet, such as planting a formula into a block of cells or adding some rows, has an equivalent standard function which can be called from a macro.
2. There is a set of enquiry functions which lets a macro find out every detail about a given spreadsheet - its size, and the dimensions, contents and style attached to each cell.
3. A set of input functions allows the user to communicate directly with a running macro.
4. A set of file functions allows macros to manipulate data directly from files.
5. A large set of graphics functions is used to produce graphs and charts.
6. There is a set of miscellaneous functions including **SYS**, a direct call to a named **SWI** in the operating system.

The input functions are discussed in Chapter 3. Graphics functions are described in Chapters 4 and 5, and the other standard functions are specified in Chapter 8

1.18 Macro Capture

Any sequence of manual operations can be captured and made into a macro. Details of macro capture are discussed in Chapter 3.

1.19 Compiling and running macros : The !MacroEdit program

MacroEdit is the tool for editing and compiling macros. MacroEdit is described in Chapter 2. Here we'll tell you enough about it to let you get started..

MacroEdit fulfils two roles:

- It is a simple text editor, which lets you type and edit macros.
- It is a compiler, which can translate macros into a form directly comprehensible to Schema2. Macros must be compiled before Schema2 can execute them.

MacroEdit modes

You can run MacroEdit in either of two modes:

- If memory is tight, you can run MacroEdit by itself. It will produce compiled macros which you will eventually have to load by dragging their icons into Schema2.
- If you have plenty of memory, you can run MacroEdit at the same time as Schema2. In this mode macros are automatically copied into Schema2 as soon as they are compiled, so that they can be tested immediately.

The text editor section of MacroEdit is based on the standard Edit utility. You should note the following differences:

- MacroEdit uses a cut-copy-paste system with a clipboard to manipulate segments of text
- MacroEdit has no undo facility; to avoid mishaps you should save rather more often than with Edit.

Using MacroEdit

To use MacroEdit take the following steps:

1. Create a directory to take your macros. Open this directory and park it in a corner of the screen.
2. Double-click on the MACroeDit application icon.
3. Open a MacroEdit window:
 - a) To insert a new macro, click on the MacroEdit bar icon to get a blank window, and type the new macro.
 - b) To load an existing macro, drag its file icon to the MacroEdit icon on the icon bar.

Alternatively, if the text of the macro already exists, you can simply double-click on the text icon, and the text appears in a MacroEdit window.

4. Type or edit your macro, and then read over it to find as many errors as you can.

If you have captured the macro you could skip this stage since captured macros can often be compiled and run without alteration. Otherwise you may find the following check list useful:

- The macro must start with the word **MACRO**.
- The macro must end with the line **ENDMACRO**.
- The macro must contain at least one return statement which starts with =. The statement just before endmacro is usually (but not necessarily) a return statement.
- All local variables must be declared.
- The results of all standard functions and macro calls must be assigned to variables even if you don't do anything with the values.

5. When you believe your macro is correct, save it by choosing **File > Save** on the MacroEdit main menu and using the normal save dialogue. The text of macros is saved in files of type **SheetMc**.

At this stage you may find it convenient to choose a file name which is the same as the name of the macro. You are limited to names of 11 letters.

6. Attempt to compile the macro by pressing **F8** or choosing **Compile** from the main MacroEdit menu.

If the macro is syntactically correct (that is, it contains no errors of grammar) the compilation succeeds. MacroEdit writes a file which contains a version of your macro in a form that Schema2 can load. This file

is called a **compiled macro file**. It has file type **SMacro**.

This file is saved in the same directory as the **source** (original text) version of the macro. The name of the compiled version is derived from the name of the source; if the source is called **xyz** the compiled version is called **=xyz**

7. If the macro compiles correctly you can test it with Schema2. If Schema2 is loaded the compiled version of the macro will have been transferred automatically. Otherwise load Schema2 and drag the compiled macro file to the icon bar icon. Now enter a macro call into a cell. For example:

If you have just compiled the pythagoras macro, fill a cell with the expression

```
= Pythagoras(12,5)
```

The result should be 13.

If you have just compiled the smoothing macro, fill a column (say column E1 to E20) with a sequence of numbers, and then fill cell with the formula

```
=smooth_column(E1..E20,D1..D20,20)
```

Now make a line plot of cells D1..E20.

Checking and Testing

Check your macro extremely carefully by giving it simple sets of data to which you know the answer in advance. Give it a number of tests, designed so that - at the very least - each line of code in the macro is exercised at least once.

When you start your macro it may produce a result. It might also be caught in making an error (such as going outside the bounds of an array). In this case, the error is displayed in a box, with information to help you track it down.

A macro can also get stuck in an infinite loop, where nothing seems to be happening. You can break in to such a loop with the **ESCAPE** key.

If your macro runs but does not give the right results, return to MacroEdit, make any changes you need, and try it again.

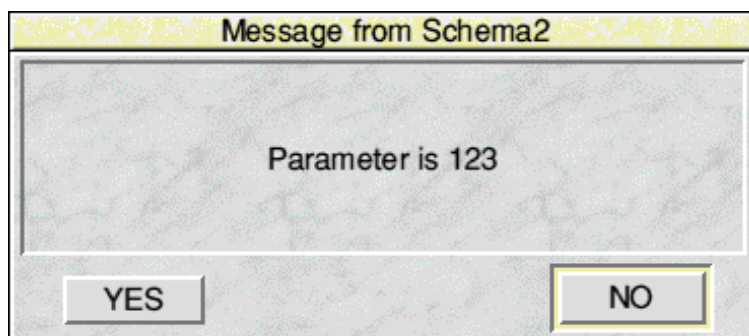
Sometimes it is not obvious why a macro has failed. Schema2 does not have a debugger, but you can seed a macro with calls to the QUERY function, which can be made to display the value of a variable and then wait until you type something and press **RETURN**. For example, if you set up the following macro:

```
MACRO show_query(p)
LOCAL res
res = QUERY(" Parameter is " cat p,"YES","NO")
=res
ENDMACRO
```

and call it by

```
= show_query(123)
```

the macro displays



The macro then waits until you click one of the buttons, and returns 1 (for "YES") or 0 (for "NO").

If a macro is not grammatically correct (macros are hardly ever right the first time you compile them) MacroEdit stops at the first error, marks the place with the cursor and puts a description of the error in the box at the bottom of the window. Correct the error, save the new version of the program and try again.

1.20 Direct links with sheets

Macros can read or write directly to the cells in a spreadsheet. The cell can be named in full, including the sheet name, as in

```
!Heathrow.B5
```

If the sheet name is omitted, the cell is assumed to be in the same sheet as the one from which the macro was called.

As a simple example, consider this macro, whose job is to watch over cell B2 and give a warning if the user tries to enter a value outside the range 18-65

```
MACRO watchb2(x)
LOCAL res
IF b2 < 18 or b2 > 65 THEN
res = QUERY("Cell b2 is out of range. Force default?", "Yes", "No")
IF res THEN b2 = 40
ENDIF
=0
ENDMACRO
```

A call to this macro is entered into a cell in a remote corner of the sheet. The parameter is set as B2:

```
= watchb2(b2)
```

The parameter is actually a dummy, and its only purpose is to ensure that the macro is evaluated every time B2 changes value.

As long as the new value of b2 is within range, the macro is silent; but if the user types an incorrect number the macro issues a warning and offers the chance to substitute a valid number.

References to cells can be indexed as if they were the names of two-dimensional variables. For example B3(2,8) is actually a reference to cell J5. The indices can be arbitrary expressions.

Most things that you can do with direct references to spreadsheets you can do with array parameters. One of the advantages of using array parameters, rather than direct references, is that they can make your macro more general, working on any spreadsheet.

1.21 Calling Macros

This section describes several ways in which macros can be called to execute.

Inclusion in cell formulae

Entering a formula or expression in a spreadsheet cell which includes a macro call will execute that macro. For example, consider a macro called initials, which takes a string parameter like "Percy Bysshe Shelley" and returns a string like "P.B.Shelley". Load cell A2 with the string "Tyrannosaurus Rex". The call

```
=initials(A2)
```

entered into a cell, will evaluate the macro and give the result "T. Rex".

There are no restrictions on the number or type of parameters to macros which are called from cells. The line

```
FUNCMACRO
```

immediately after the header line will make the macro appear in the standard function menu.

Calls from macros

An expression in a macro can call another macro.. There is no restriction on the number or type of parameters. Recursion is allowed, up to a limit of 50 calls.

Calls from the main menu

There are two ways to make a macro callable from the menu,

1. Include the keyword **EXECMACRO** immediately after the macro header line. When the compiled macro is loaded into Schema2, its name is automatically included in the submenu reached when you choose **Utilities -> Macro** from the main menu.
2. To install a macro elsewhere in the menu structure you will need to set up your own workspace and edit the menu definition file (See Techniques 27 in the main manual). Insert a line which reads

```
DOMACRO, macroname
```

where macroname is the name of the macro.

Calling a macro from the icon ribbon

To make a macro callable from the icon ribbon you must:

1. Design or find a suitable icon and install it in one or more of the ribbon_s, ribbon_s22 or ribbon_s23 files (as appropriate)
2. Link the macro to the icon by entering a new line in the ribbon_m file : name of icon, **DOMACRO** name of macro

An example is

```
clearstyles, DOMACRO Clear_Styles
```

A macro included in the main menu or icon ribbon must have one of three possible parameter arrangements:

- No parameters
- One block parameter. When the macro is called, this parameter is the selected area of the spreadsheet.
- One block parameter and one normal parameter. When the macro is executed the block is the selected area of the spreadsheet, and the scalar is the leaf name of the spreadsheet as a string.

Graphics macros

A macro can be included in the graphics dialogue menu by including a line beginning with the word **GRAPHMACRO**. Further details are given in Chapter 5.

Boot macros

When the system loads a spreadsheet with a given name - say **xxx** - it looks for a macro called **BOOT_xxx** in the same directory. If found, it executes this macro immediately after the spreadsheet is loaded.

A Boot macro can have none, one or two parameters, just like a menu macro. When the boot macro is called, the first parameter (if any) is a block which represents the whole of the spreadsheet which has just been loaded.

1.22 Workspaces and macros

When you load a compiled macro file Schema2 remembers the name of that file. If you save a workspace then Schema2 records a reference to the file in the workspace file, in the same way as spreadsheet names are recorded.

If you load a workspace file which contains references to compiled macro files then Schema2 loads those compiled macro files.

Chapter 2 : Macro Editor and Compiler Reference

This chapter gives a more complete description of MacroEdit than we had space for in Chapter 1. Treat it as a reference document.

MacroEdit is a special application program designed to support the writing and compilation of Schema2 macros. It consists of a closely coupled editor and compiler, with direct links to Schema2.

The main reason for using the Macro editor, rather than **Edit** is that it is closely connected to the compiler; syntax errors are reported on the editor window and the cursor is automatically moved to the right position for a correction.

The Schema2 Macro Editor is a plain text editor designed for the development of macros. It is broadly similar to **Edit**, except in three respects:

- It does not provide the undo feature
- It uses the cut-copy-paste model instead of **Move** and **Copy**.
- It does not offer multiple views of the same document

The editor conforms to Acorn RISC OS standards for window handling.

2.1 Entering the editor:

You can start up the editor in any of the following ways:

- Double click on the 'MacroEdit application icon.
- Double click on any document of type SheetMc. This document will automatically be loaded for editing, provided that RISC OS has 'seen' a directory with the [MacroEdit icon.

2.2 Loading a document for editing

- You can create a blank document by clicking on the MacroEdit icon on the icon bar.
- You can load a document of type **SheetMc** or **Text** by dragging its file icon to the MacroEdit icon on the icon bar.
- You may load a document of type **SheetMc** or **Text** by dragging its file icon to an empty document window.

Up to 50 distinct documents may be open at the same time, subject to memory limitations. No document may be opened more than once at the same time.

2.3 Editing

A document may have lines up to 256 characters long, and the number of lines is limited only by the available memory.

Every document window contains a cursor, which indicates the position at which any change is carried out. You can move the cursor with the cursor control keys, or by clicking with the mouse. The cursor is invisible if scrolled out of view. All keyboard keys including the printing characters, SPACE, NEWLINE and BACKSPACE may be used to construct or edit text.

You can use the mouse to mark any contiguous block of text. The marked block is displayed in reverse video, and can be manipulated with the Edit group commands, described in section 2.5.2 below.

You can merge any file of type SheetMc or Text with the current context of a window, simply by placing the cursor and dragging the file icon to the window.

2.4 The Icon Bar Menu Structure

The icon bar menu has only two entries:

Info

The entry gives information about the program.

Quit

This closes the program. Quit gives you a warning if any document currently open has not been saved.

2.5 The Main Menu Structure

Many entries have keyboard short cuts, which are shown in the usual way, following the menu item names. Thus F5 means Function key 5 and C means Ctrl C.

The main menu is shown.

The File menu structure

This includes three entries:

Info (I)

This entry gives information about the current file.

Save (F3)

This saves the current document as a SheetMc file

Save as text

This saves the current document as a Text file.

The Edit menu

This comprises 6 entries:

Cut (^X)

Removes the highlighted block from the document and places it in the clipboard. This destroys the previous contents of the clipboard.

Copy (^C)

Copies the highlighted block from the document and places it in the clipboard. This destroys the previous contents of the clipboard but does not change the document itself.

Paste (^V)

Inserts the current contents of the clipboard into the document, at the cursor position. This does not affect the contents of the clipboard.

Delete (^K)

Removes the highlighted block from the text. The clipboard is not affected.

Select all (^A)

Highlights all the text.

Clear (^Z)

Removes the highlight

The Utilities Menu

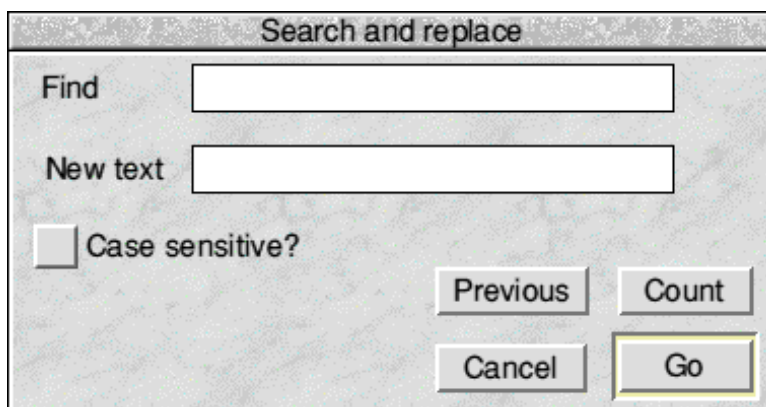
The Utilities menu contains 7 entries:

Search (F4)

This lets you search a file for any sequence of characters, provided they are all on the same line. The search starts at the current position of the cursor. You may optionally replace any sequence of characters with any other sequence, either once or through to the end of the file.

The Search Dialogue

The search menu item leads to the dialogue shown.



The fields of the search dialogue are:

Find (a writable area). Use the Find area to enter the sequence to search for. This sequence is called the target string. As well as all printing characters, you may use the following special symbols:

- \. Stands for any character.
- \a Stands for any letter or digit
- \d Stands for any digit
- \n Stands for the beginning or end of a line and must be placed first or last in your sequence
- * Stands for any sequence of characters terminated by (and not including) the character following the *. If * is the last item the sequence extend until the end of the line.
- \q (where q is any character other than . a d n or *) stands for q. This is mainly used to specify the \ character.

Examples:

- c\..t matches with cat, cot, cut, c7t,c#t
- c\at matches with cat, cot, cut, c7t
- c\d..t matches c7t
- \ncat matches with cat at the beginning of a line
- cat\n matches with cat at the end of a line
- c*s matches with cats, catamarans, etc.

New text (another writable area). This area is used if the target string needs to be replaced. The new sequence is entered into the area. Any characters may be used, and there is one special sequence:

- \& stands for a copy of the target string.

The New text area may also be left empty.

Case sensitive If this button is on, the comparison will be case sensitive.

Previous. If this is selected the Find and New text areas will automatically be loaded with the most recent values used in this document. go. When this button is clicked the search actually starts, at the current position of the cursor.

If the target string cannot be found the message not found is displayed and the cursor returns to its original position.

If an occurrence of the target string is found, the occurrence is highlighted and the Text found dialogue will open. This is described below.

Count. If this button is clicked the system works through the entire document, searching for instances of the target string. If found, the last occurrence is highlighted; otherwise the cursor remains in its initial position.

After a **count** command the Text found dialogue opens.

Cancel. This cancels the dialogue without any action. Clicking the mouse anywhere outside the dialogue has

the same effect.

The Text found dialogue

The Text found dialogue appears after a successful search. If the count option was used, the lower line will show the number of occurrences.

The dialogue has five buttons:

Stop. This terminates the find-replace sequence.

Continue. This searches on for the next occurrence of the target string. If there are no more occurrences the lower field says, not found.

Replace. The target string is replaced with the specified new text (which may be null), and the system searches for the next instance of the target string.

Last replace. The current instance of the target string is replaced with the specified new text (which may be null), and the find-replace sequence ends.

End of file replace. All instances of the target string are replaced.

Goto (F5)

This dialogue lets you position the cursor at the line with any given number within the document - such as line 100.

Indent (F4)

Use this dialogue to edit the highlighted block. It offers three choices:

Add Spaces: Inserts a given number of spaces at the start of each line in the block

Add Characters: Inserts a given character string at the start of each line in the block

Remove characters: Removes a given number of characters from each line in the block.

The dialogue does not appear if no block is highlighted; instead a warning is displayed.

Format

This is a toggled entry. If it is set, and you enter a newline after a line which contains initial blanks, the same number of blanks is inserted at the beginning of the new line. You may remove some or all of these blanks with the **Backspace** key. Use **Format** to help with laying out block-structured code.

Save as default

This preserves the current setting of the format marker and the document colours. These settings are used as defaults when the document is next loaded.

Foreground

This leads to a menu which lets you choose the foreground colour for the window. Only the 16 wimp colours may be used.

Background

This leads to a menu which lets you select the background colour for the window. Only the 16 wimp colours may be used.

Note that any colour combination, once selected, can be made the default choice by using the **Save as default** option.

2.6. The Compiler

Call the compiler with the **Compile** option on the main menu or with the F8 key. If the source text is grammatically correct the compiler generates a **SMacro** file. If a copy of Schema2 is present in the machine this code is automatically loaded into Schema2 and can be used immediately.

If the source text contains any syntax errors the compiler stops as soon as it detects the first error. An error message is displayed at the bottom of the window, and the cursor is placed near the point at which the error was detected. (Note: this need not be the place where the error actually occurred!)

2.7 Customising the Menu

The menu structure for !MacEdit can be customised in the same way as a menu structure for Schema!. The file to edit is called EDWIN_M, and may be found in the !MacroEdit directory.

Don't modify this file unless you have made a backup.

2.8 List of hot keys

F3	Save
F4	Search
F5	Goto
F8	Compile
FI	Info
F4	Indent
X	Cut
C	Copy
V	Move
K	Delete
A	Select all
Z	Clear selection

Chapter 3: Capturing Macros

This chapter discusses a technique for capturing sequences of user actions, turning them into macros, and using these macros to build simple applications.

3.1 Macro-based Standard Functions

Schema2 supports a special library of standard functions which can only be called from within macros. These functions fall into various groups, of which only two are of interest in this Chapter;

- User action functions
- Data input functions

Each user action function mimics a particular user action or group of actions; for example, the call to the PUTBLOCK function

```
res = PUTBLOCK( "A1+1" , !Trout.A2..B7)
```

imitates precisely the following actions

Choose sheet **Trout**

Select cell block **A2..B7**

Enter the formula **=A1+1**

The result in res can be used to check that all these actions were carried out without any mishap, such as a syntax error in the expression. Every significant user action can be imitated by an action function, executed by a macro.

3.2 Recording Macros

The Schema2 icon menu has an entry labelled Record Macro, with three sub- entries marked Start, Stop and Abort.

During a normal Schema2 session you can switch on macro recording at any time with the Start entry. All significant operations which follow are recorded in a macro file. Each item in the file is a call to the appropriate user action function, such as

```
res = INPUTBLOCK( !Mysheet.B5..J17,1)  
res = SETSHEETINFO( !Another.A1..A1, "labels", 0)
```

You can end the macro recording process by either of the other two entries: **Stop** displays a dialogue where you can enter a suitable macro name. You can then save the macro file into a directory, or straight to the MacroEditor icon on the main icon bar. Here you can edit the macro and compile it so that it can be run with a spreadsheet. The set of operations you recorded is automatically fitted out with header and trailer statements to make them into a valid macro.

Abort simply discards the collected set of operations.

Any macro you construct this way should be regarded as raw material. At best, an unchanged captured macro will simply replay the exact sequence of operations which produced it, and sometimes (specially when you change the name of a sheet during macro capture) the macro may even fail to do as much as that. However, a few simple alterations to a macro (such as replacing constant sheet names by parameters) can make it into a useful general-purpose procedure.

3.3 An example of a captured macro

	A	B	C	D	E	F
1	Place	Zurich	Distance from	(Miles)	Latitudes	Longitudes
2	Latitude	47	Athens	1008.4	37.91	-23.6
3	Longitude	-8	Berlin	444.1	52.5	-13.2
4			Brussels	324.8	51	-4.2
5			Copenhagen	624.8	55.6	-12.6
6			Dublin	763.3	53.3	6.27
7			Lisbon	979.0	38.8	7.9
8			London	481.6	51.5	0.2
9			Luxembourg	199.3	49.6	-6.1
10			Madrid	733.7	40.5	3.7
11			Paris	289.8	48.9	-2.4
12			Rome	412.8	41.9	-12.4
13			The Hague	384.8	52.1	-4.5

Sheet GCD, shown above, is a spreadsheet which calculates the distance from any point on the earth's surface to each of the 12 (at time of writing) European Union capitals. You will find this sheet on the companion disc. Skip the following section (in small print) if you are not interested in how the distance is calculated.

For the mathematically minded, the distance calculated in the sheet is actually the Great Circle distance, which is the shortest distance over the earth's surface, assuming the earth is a perfect sphere.

Consider two places X and Y. X has latitude AX and longitude BX; and Y has latitude AY and longitude BY. Latitudes range from +90 (at the North Pole) to -90 at the South Pole. Longitudes are based on Greenwich and vary to +180 going West, or to -180 going East.

If the mean radius of the earth is 3950 miles, the great circle distance between X and Y is given by

$$3950 * \arccos(\sin(AX)*\sin(AY)+\cos(AX)*\cos(AY)*\cos(\text{abs}(BX-BY)))$$

This formula is correct for radians, but needs some correction factors if degrees are used as basic units.

Load the sheet and carry out the following steps. If you make a mistake, choose Record macro > Abort from the icon menu and start again.

1. Choose Record macro > Start from the icon menu.
2. Enter the name of a place, its latitude and its longitude into cells B1, B2 and B3, respectively. We suggest Northwich, which is roughly at latitude +54.2 and longitude +2.5.

(Here positive numbers mean North of the Equator and West of Greenwich, respectively).

As soon as you enter the values. Column D displays the distance from Northwich to each of the 12 capitals.

3. Select block A1..D13 and print the selection.
4. Discard sheet GCD.

5. Bring up the icon menu, choose Record macro > Stop, call the macro file BOOT_GCD and drag the icon to the same directory as GCD.
6. Double-click on the BOOT_GCD file icon. The captured macro appears as:

```
MACRO boot_gcd(sel())
EXECMACRO
LOCAL res, target
ON ERROR EXIT
res = PUTBLOCK("\Northwich\","",!Distances.B1..B1)
res = PUTBLOCK("52.4",!Distances.B2..B2)
res = PUTBLOCK("2.5",!Distances.B3..B3)
res = SETUPIMAGEPRINT(!Distances.AL..D13,1,142,1,0,0,0,0,0,1)
res = PRINT(!Distances.AL.D13)
res = DELETESHEET(!Distances.AL..J28)
=res
ENDMACRO
```

In this macro, the lines

```
MACRO boot_gcd(sel())           (header statements)
EXECMACRO
LOCAL res, target
ON ERROR EXIT

and

=res                           (trailer statements)
ENDMACRO
```

are inserted automatically. You can see how the other lines correspond to your actions. Printing takes two commands: **SETUPIMAGEPRINT** and **PRINT**.

If you ever need to write or edit a call to **PUTBLOCK**, you will need to understand the way that the first parameter is laid out. The rules are not self-evident; they have been chosen to be compatible with macros originally written for Schema.

- Numbers are written inside double quotes, as in "55.2".
- Expressions are written inside double quotes, without the = sign, as in "B2+C3"
- Strings (segments of text) are written inside two layers of double quotes. A quote mark inside a quotation must be written as \" so every string must be preceded by the sequence \" and ended with the sequence \" \" as in \"Northwich\"

3.4 Data Input functions

When you modify a captured macro, it is usually necessary to let the user replace some fixed quantity (such as a sheet name or the name of a person) with a value of their own choice. Schema2 has two data input functions, which can be called from macros, to make this possible.

QUERY

This function lets the macro put a 'yes-or-no' question to the user and receive a reply. The query function takes three parameters:

- the name of the question
- the wording for the affirmative reply
- the wording for the negative reply.

When the function is called from a macro, Schema2 displays a dialogue box and eventually returns a reply : 1 for Yes, and 0 for No. For example, the call

```
res = QUERY("Question?", "Yes", "No")
```

displays the dialogue and eventually sets **res** to 1 for yes, or 0 for no.

INPUT

The input standard function lets the user supply the value of any variable. Numbers and strings can be used equally well.

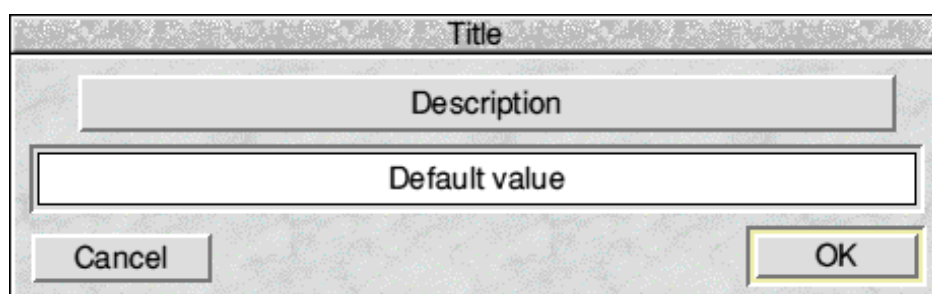
The function takes five parameters:

- The title of the dialogue
- A description of the question
- A default answer (which may be the empty string "")
- The wording for the box which completes the dialogue
- The wording for the box which cancels the dialogue

To give an example, the call

```
res = INPUT("Title","Description","Default value","OK","Cancel")
```

displays the dialogue shown below.



You can replace the words 'Default value' by any string or number. When you click on OK or press Return, the typed value is assigned to **res**. If you click on Cancel, the function returns the value Undefined.

3.5 Converting captured macros into customised applications

Our aim is to make the Great Circle sheet macro into an application. We would like to simplify matters so that the user can click a specified icon, enter the data into labelled dialogue boxes (not blank cells), and receive a printed answer. The user should not need to know anything about spreadsheets, or be given any chance to spoil the sheet by entering data into the wrong places.

To do this, we must modify the macro so it uses the **input** standard function to obtain the name of a place, its latitude and longitude.

To do this for the name of the place, change the line

```
res = PUTBLOCK("\Northwich\","",!Distances.B1..B1)
into
name =INPUT ("Distances","Name of place","", "OK", "Cancel")
res= PUTBLOCK("\" cat name cat "\", !Distances.B1..B1)
```

Note that the PUTBLOCK command uses the **cat** operator to enclose the first parameter in an extra layer of double quotes.

Similarly, the two PUTBLOCK calls which set up the latitude and longitude must be replaced by

```
reply = INPUT("Distances","Latitude of place","", "OK", "Cancel")
res = PUTBLOCK(reply, !Distances.B2..B2)
```

and

```
reply = INPUT("Distances","Longitude of place","", "OK", "Cancel")
res = PUTBLOCK( reply, !Distances.C2..C2)
```

The last instruction to be added should be

```
res = QUITPROGRAM
```

This ensures that Schema2 closes itself down at the end of the run.

The new version of the macro reads:

```
MACRO boot_gcd(sel())
EXECMACRO
LOCAL res, name,reply
ON ERROR_EXIT
name =INPUT ("Gcd","Name of place","", "OK", "Cancel")
res= PUTBLOCK("\\" cat name cat "\", !Gcd.B1..B1)
reply =INPUT("Gcd"," Latitude of" cat !Gcd.b1,"53", "OK", "Cancel")
res = PUTBLOCK(reply,!Gcd.B2..B2)
reply = INPUT("Gcd","Longitude of " cat !Gcd.b1,"0", "OK", "Cancel")
res = PUTBLOCK(reply,!Gcd.B3..B3)
res = SETUPIMAGEPRINT(!Gcd.A1..D13,1,142,1,0,0,0,0,0,0,1)
res = PRINT(!Gcd.A1..D13)
res = DELETESHEET(!Gcd.A1..J28)
res = QUITPROGRAM
=res
ENDMACRO
```

Save the modified macro, compile it and make sure that it is loaded into Schema2 in its new form. Don't run it, or it will delete the GCD sheet. Now save a workspace in the same directory as GCD and BOOT_GCD. Prepare instructions for the user, on the following lines:

To find the distance of a place from each of the 12 EU capitals:

1. Find out the name, latitude and longitude of the place.
2. Double-click the GCD workspace icon in the GCD directory.
3. Enter the items of data as you are asked for them.
4. Wait for the printer.

When you click on the workspace, Schema2 loads the sheet GCD. It then recognises BOOT_GCD as a boot-macro and runs it immediately, before letting you do anything to the sheet. The macro ends by shutting down the sheet and deleting Schema2. At no time do you have any chance to make unintended alterations.

Apart from your ingenuity, there is no limit to the complexity and versatility of macro-based customised applications.

Chapter 4: Graphics

4.1 Introduction

In this Chapter we assume that you are writing macros, which will eventually be used by another person, whom we call the user.

Graphics functions in Schema2 macros can be divided into two main groups.

The Drawing functions

The drawing functions let you create a graph or picture of almost any type. To draw a picture, you must write a macro which calls a series of primitive functions.

The macro sets up a blank graph of a given size, and uses various commands to plot lines, rectangles, segments of text, polygons and curves. The macro can also import ready-made Drawfiles and plant them in the graph, at any position and to any scale.

The positions and sizes of the various objects are often taken from the cells in a spreadsheet. This allows the picture to be a graphical representation of the data in the sheet.

A graph drawn in this way is purely passive; it cannot be edited in any way by the user unless the right edit support functions are included.

The Edit support functions

The edit support functions help you write macros which provide the following features:

1. When a graph has been drawn, it can be edited by the user, in the way described in Sections 20-24 of the main manual. The user can change colours, fonts and axis parameters, and can add decorations in the form of lines, boxes, ellipses, text and Drawfiles imported from other sources.
2. The decorated graph can be hotlinked into a spreadsheet.
3. When the data pictured in the graph is changed, the graph is replotted automatically. The new version incorporates all the changes and additions made by the user.

We describe the graph drawing functions in Chapter 4, and the edit support functions in Chapter 5.

4.2 Graph Coordinates

In graphs distances are measured in OS-units. Each OS-unit is 1/180 of an inch. The coordinate origin is the lower left hand corner of the window. Horizontal distances are measured to the right, and vertical distances upwards.

4.3 Drawfiles

In RISC OS computers, the basic graphical object is a Drawfile. This is a file of commands for drawing a picture. Individual plot commands draw graphic components such as text, Drawfiles, and 'paths' which can be lines, polygons, circles or curves of any shape. The paths may be (but need not be) filled with colour. There is no command to plot a sprite directly, but the system will render sprites included in Drawfiles. There is no reason not to plot a Drawfile which contains only a sprite.

Drawfiles are fully defined in the RISC OS Programmer's Reference Manual, but in most cases you will not need to know the details of Drawfile formats. The graphics primitives work by setting up a Drawfile and filling it with various components. When the macro which created the Drawfile terminates, the Drawfile is rendered (that is, converted into a visible picture) automatically.

As an alternative, you can save a Drawfile and then print it or render it with another application such as !Draw.

If a graph drawing macro fails for any reason (perhaps because it is given unsuitable data) it should return a fault or a string which describes the problem. Error handling is simplified by the command

```
ON_ERROR_EXIT
```

which (as we explained in Chapter 1) ensures that a macro terminates whenever it tries to assign an error to a variable. The macro returns the error it tried to assign.

We now describe the graph drawing primitives, and provide some simple examples. You can find the code for the examples in directory Graphs 1. Some of the primitives are only included to ensure that macros originally written under Schema will also run under Schema2. These primitives are described in a smaller typeface at the end of each section, and their use is not recommended in new macros.

4.4 Graphics objects and handles

In Schema2, every Drawfile has a unique identifier known as a 'handle', which is used as a parameter for all operations on that Drawfile. A handle is a type of object in its own right, in the same class as integers, strings and faults. A handle can be assigned to a variable, or passed as a parameter, and macros can return handles as their results. When a handle is assigned to a spreadsheet cell, the Drawfile it refers to is drawn in that cell. Arithmetic on handles is forbidden.

The simplest way to draw a graphic object in a cell is to set up a macro which makes a Drawfile and returns its handle, and then to enter a call to that macro into the cell. This will be illustrated in due course.

4.5 Opening, Loading and Saving Drawfiles

This group of functions lets you open an empty Drawfile, load an existing Drawfile, and save a Drawfile to the disc. You will know that every standard function returns a value, which is often but not always an error marker. In the descriptions which follow, values which are error markers are conventionally assigned to a variable called *res*. Values needed for some other purpose are assigned to *w* or *q*.

w = GSTART(x,y) **Open a graph**

This function starts a graph by creating an empty Drawfile. The graph size in OS units is set by *x* (horizontal dimension) and *y* (vertical dimension). The function returns a handle, or graph identifier. If an error is returned it means that the command failed.

The handle is used to draw objects into the graph. Usually, when the graph is complete, the macro will either return the graph's handle or save the graph with the SAVEGRAPHQ function.

Example:

```
MACRO MyGraph
ON ERROR EXIT
w = GSTART(500,300)
REM : Create a graph 500 OS-units wide and 300 units high
REM : Exit is automatic if GSTART fails
REM : Now we draw the graph using w
REM : Now the graph is complete we return its handle
= w
ENDMACRO
```

Any cell that has a call to MyGraph in the form of the formula **=MyGraph()** will have the graph rendered in it.

w=GRAPHLOAD("name")

This function loads the Drawfile identified by the name, and returns a handle. Once loaded, this graph can be used like any other other graph.

The "name" must be the full path name of a Drawfile

If the command fails (perhaps because the file cannot be found) it returns an error.

Example:

```
ON ERROR EXIT
q = "ADFS::Disc4.$.Drawings.Horse"
w = GRAPHLOAD(q)
REM : Try to open ADFS::Disc4.$.Drawings.Horse
```

w = FETCHGRAPH("identifier")

This function loads a Drawfile which is already known to the system, and returns a handle. The parameter is a unique system-generated identifier. Once loaded, this graph can be used like any other graph.

This command will be discussed further in Chapter 5

res = SAVEGRAPH(w, "Name")

The Drawfile identified by the handle w is saved as a file of the given name. The type of the file is Drawfile. The function returns 0 if the process succeeded, or an error if it failed (for example, because the disc was full).

Example:

```
ON ERROR EXIT
q = "ADFS::Disc4.$.Drawings.Pegasus"
res = SAVEGRAPH(w,q)
REM : Save graph w as ADFS::Disc4.$.Drawings.Pegasus
```

w = GOPEN(marker,x,y,w,h,sx,sy) *Deprecated*

If marker = 0, this creates and opens a new window of dimensions (w,h) units. When the object is eventually rendered the window will have its top left hand corner at (x,y) and the scroll bars will be set to sx,sy.

If the marker is the handle of an existing graphical object, that object is cleared but not destroyed.

In both cases GOPEN returns the handle of an empty graphics object, or a fault if the operation failed.

The graphics object created by GOPEN is not displayed until macro execution is complete.

res = GCLOSE(w) *Deprecated*

This function does nothing.

4.6 Style setting functions

The functions in this group do not create any new objects in a Drawfile; they simply determine the properties or styles of objects yet to be created. A style command is linked to a specific handle. This makes it possible to assign two different styles to two Drawfiles at the same time

res = FONT(w," font-name", height, width)

This function sets the typestyle for commands which create text objects.

Parameter details are:

w The handle of a Draw file

font-name The name of a font mounted on your machine

height The height of the font in points (1/72 of an inch)

width The width of the font in points

The function returns 0 if it succeeds, or an error if it fails (for example, if the font is not loaded on the machine).

Note: You can vary the aspect ratio of the text by specifying the height and the width independently.

Example:

```
f="Bodini.Bold.Slant"
res = FONT(w, f, 24,14)
```

res = LINEATTRIBUTES (w, style, width, colour, path style)

This function sets the line style for commands which create objects with lines. Parameter details are:

w The handle of a Draw file

style This parameter is an integer in the range 1 to 5. The values have the following meanings;

1. Continuous line
2. Dotted line
3. Line with short dashes

4. Line with medium length dashes and gaps
5. Line with long dashes and gaps

width The width of the line, in 1/256 OS-units. so that a width of 1024 gives a line 4 OS-units wide. A line width of 0 indicates that the line is to be drawn as thin as possible for that plotting device. To preserve compatibility with Schema, certain other widths are interpreted in special ways:

- 1 Invisible line
- 2 Plot with width 2. This is a special value understood by the RISC OS rendering code.
- 3 Line 5 OS-units wide ("Medium")
- 4 Line 10 OS-units wide ("Wide")

colour The colour parameter is an eight-digit hex value, preceded by **Ox** which signifies the following digits as a hex number, which specifies the saturation of each of the three primary colours:

- top two digits : BLUE (maximum ff)
- Next two digits: GREEN (maximum ff)
- next two digits: RED (maximum ff)
- bottom two digits : Always 00 except for 'transparent'

A value of -1 (Oxffffffff) signifies transparency.

path style The path style description word controls the way that line segments are joined and terminated. The path style is only significant for lines which have visible thickness. The path style is an integer number which is made up from independent bit-fields, as follows (bit 0 is the least significant):

Bit(s)	Description
1-0	join style 0 - mitred joins 1 - round joins 2 - bevelled joins
3-2	end cap style 0 - butt caps 1 - round caps 2 - projecting square caps 3 - triangular caps
5-4	start cap style 0 - butt caps 1 - round caps 2 - projecting square caps 3 - triangular caps
6	winding rule 0 - non-zero 1 - even-odd
7	must be 1
15-8	reserved and must be zero
23-16	triangle cap width in units of 1/16 of the line width
31-24	triangle cap length in units of 1/16 of the line width

Example:

```
res = LINEATTRIBUTES (w,l,1024,Ox00FFFF00, Ox8080007C)
```

This sets up Drawfile w to draw lines with the following attributes:

Line style : solid
Line width : 4 OS-units
Colour : yellow (green and red)
Path style: both ends of the line have triangular cap (arrowheads) which
are 32 OS-units high and 32 OS-units wide (Ox80=(4/16)*32)

4.7 Writing components into Drawfiles

Every function in this group (except for some auxiliary functions) adds an object to a Drawfile. When the Drawfile is rendered this appears as a visible object.

res = LINE(w,x,y,dx,dy)

This function draws a straight line in the style set by the most recent **LINEATTRIBUTE()** function. The parameters are:

w	The handle of the graph
x,y	The start location of the line
dx,dy	The horizontal and vertical displacement of the end point from the start point

Example: The command `res = LINE(w, 300,450,200, -25)` plots a line starting at (300,450) and ending at (500,425).

The macro

```
MACRO cross
LOCAL w,res
ON ERROR EXIT
w = GSTART(200,200)
res = LINEATTRIBUTES(w,1,8192,0,0x94)
res = LINE(w,25,25,150,150)
res = LINE(w,25,175,150,-150)
REM : Macro returns a handle to the graph
=w
ENDMACRO
```

plots a thick X with rounded line ends. To see it in action:

1. Enter and compile the macro
2. Enlarge cell A1 to about 2 inches square
3. Fill A1 with the formula `=cross((its)`

res = BOX(w,x,y,width,height,colour)

This function draws a rectangular box with its edges parallel to the borders of the window. The parameters are:

w	The handle of the graph
x,y	The position of the bottom left corner of the box
width	The width of the box
height	The height of the box
colour	The colour of the inside of the box. This is an RGB value similar to the one used for lines. -1 signifies transparency.

The rectangle is drawn in the given colour with a border using the current line style and colour. The line centre coincides with the edge of the box so the box is one line width higher and wider than the given dimensions. For example:

```
res = BOX(w,200,600,300,400,0xff000000)
```

draws a blue box with its corners at (200,600) and (500,1000)

res = POLY(w,points,number,colour)

This call adds a closed polygon to the graph. The parameters are:

w	The handle of the graph
points	A two-dimensional array or a block of spreadsheet data. The block must be two rows high and may include any number of columns. Each column gives the x and y coordinate of one points in the polygon.
number	The number of points to be plotted from the array. Points beyond this number are ignored.
colour	The colour of the inside of the box. This is an RGB value similar to the one used for lines. -1 signifies transparency.

The polygon is drawn with the line type specified by the most recent **LINEATTRIBUTES()** function. The first and last points are joined and the polygon is filled with the specified colour (which may be transparent). If the polygon is self-intersecting, the way it is filled is determined by the winding rule.

Note that if points is a local array, it must be declared as

```
LOCAL points(1,n)
where n depends on the number of points.
```

Example:

```
MACRO star
LOCAL j,k,w, res,points(1,5)
ON_ERROR_EXIT
REM : set points of the star
FORj=0T04
k = 144*j
points(OJ) = 250+150*sin(k)
points(1J) = 250+150 * cos(k)
NEXTj
w = GSTART(500,500)
res = BOX (w,5,5,490,490,OxC0000000)
res = LINEATTRIBUTES(w,1,4,-1,Ox80)
res=POLY(w,points,5,OxffffT00)
=w
ENDMACRO
```

This macro draws a 5-pointed yellow star on a blue background.

res = OPENPOLY(w,points,number)

This call adds an open polygon to the graph. The parameters are:

w	The handle of the graph
points	A two-dimensional array or a block of spreadsheet data. The block must be two rows high, and may include any number of columns. Each column gives the x and y coordinate of one of the points in the polygon. See the note in section 4.7.3.
number	The number of points in the array

The polygon is drawn with the line type specified by the most recent

LINEATTRIBUTES() command. The first and last points are not joined; the polygon is left open and is not filled.

res = POLYMARKER(w, points, number, type, size, colour)

This call decorates the vertices of a polygon with markers. The parameters are:

w	The handle of the graph
points	A two-dimensional array or a block of spreadsheet data. The block must be two rows high, and may include any number of columns. Each column gives the x and y coordinate of one of the points in the polygon. See the note in section 4.7.3.
number	The number of points in the polygon
type	The shape of the marker. The codes are 1. X 2. + 3. triangle 4. diamond 5. square
size	The size of the marker in points (1/72 of an inch)
colour	The colour of the interior of the marker. All markers are drawn with black outlines, and X and + have no interiors.

res = ELLIPSE(w, ex, cy, start angle, end angle, rx, ry, maa, colour)

This function draws a segment of an ellipse, bounded by the two radii and part of the circumference. The

parameters are:

w	The handle of the graph
cx,cy	The coordinates of the centre of the ellipse
start_angle	The starting angle of the segment. Angles are measured anticlockwise, starting from due East.
end_angle	The ending angle of the segment
rx,ry	The major and minor axes of the segment
maa	The angle between the major axis and the horizontal
colour	The fill colour of the segment.

The segment is bounded with a line drawn in accordance with the most recent **LINEATTRIBUTE()** command.

Notes: If an ellipse is complete (the is, the start and end angles are the same) the radii are omitted.

A circle (or part of a circle) can be drawn as a special case of an ellipse.

Angles are measured in degrees or radians, depending on the current angle mode. By default the angle mode is determined by the user, with the values dialogue. The angle mode used in running a macro is the mode of the spreadsheet from which it was called. This is insecure, and to avoid problems you can use the special function

```
res = MACROANGLE(n)
```

This sets up a temporary angle mode: degrees for n=0, and radians for n=1. The mode lasts until the current macro terminates; then reverts to its original value. **Example:** The following macro draws a perspective view of a 6-bladed fan:

```
MACRO fan
LOCAL vv,resJ,k
ON ERROR EXIT
res = MACROANGLE(0)
w = GSTART(400,400)
res = LINEATTRIBUTES(w,1,1,0,0x80)
FOR j = 0 TO 5
REM We can set k to be angle in degrees
REM since we have called macroangle
k = 60*j
res = ELLIPSE(w,200,200,k+10,k+50,180,120,37,0xff00)
NEXT j
=w
ENDMACRO
```

res = STRING(w,x,y,text,fore,back)

This function plots a string of characters using the font and dimensions set up by the most recent **FONT** command. The parameters are

w	The handle of the graph
x,y	Position of the lower left hand corner of the first character to be drawn
text	A string of text
fore	The text colour to be used
back	The background against which the text is to be displayed.

The background colour is important if the characters are small enough to use anti-aliasing.

The best position for a text item often depends on its size. For example, you may wish to centre the item in a given area by arranging for longer items to start further towards the left. Suitable enquiry functions are **TEXTWIDTH** and **TEXTHEIGHT**, which are described in section 4.8.

res = PICTURE(w,q,x,y)

The next three functions, **PICTURE**, **SCALEDPICTURE** and **SQUEEZEDPICTURE** all let you place a copy of another Drawfile into your picture. In each case, one of the parameters is the handle of the Drawfile you want to copy. This Drawfile might have been opened and plotted independently, or loaded from a file with the **GRAPHLOAD** or **FETCHGRAPH** functions.

The parameters are:

w	The handle of your picture
q	The handle of the Drawfile to be copied
x,y	The position of the bottom left-hand corner of the Drawfile.

The Drawfile is copied full-size.

4.7.9 **res = SCALEDPICTURE(w,q,x,y,width,height)**

This function copies a Drawfile into your picture, scaling it so that it fits into a given rectangle without distortion. Unless the target rectangle is an exact fit, a band, either at the top or at the right is left empty.

The parameters are:

w	The handle of your picture
q	The handle of the Drawfile to be copied.
x,y	The position of the bottom left-hand corner of the Drawfile.
width	The width of the rectangle
height	The height of the rectangle

res = SQUEEZEDPICTURE(w, q, x, y, width, height)

This function copies a Drawfile into your picture, scaling the axes independently so that it fits exactly into a given rectangle. In general the Drawfile is distorted, and its aspect ratio is not preserved.

The parameters are:

w	The handle of your picture
q	The handle of the Drawfile to be copied.
x,y	The position of the bottom left-hand corner of the Drawfile.
width	The width of the rectangle
height	The height of the rectangle

Path drawing functions

The group of commands defined below let you draw paths of any complexity. The term 'path' includes straight lines, sequences of linked lines, polygons which may be self-intersecting, curves which approximate circles or ellipses, or any combination of these.

All curves other than straight lines are drawn using Bezier polynomials, which are described in most books on Computer Graphics. Schema2 provides functions for plotting circular and elliptical arcs. If you need curves of a more complex sort, and cannot approximate them by chains of short straight lines, you will need to calculate your own Bezier polynomials.

In a Drawfile a path is made of two major components: a head and a body. The head contains fixed-format information such as the details of the path edge (which is a line) and the fill colour. The path body consists of an arbitrary number of sub- paths, each composed of one or more elements. An element is made of an element type number, called a tag, followed by a fixed number (possibly none) of coordinate pairs. Some element types are:

- 0: End the path. No coordinate pairs
- 2: Move to an absolute position without drawing. The tag is followed by one coordinate pair. This completes the previous sub-path.
- 4: End the current subpath, leaving a gap. No coordinate pairs
- 5: Close the current subpath with a line. No coordinate pairs
- 6: Draw a Bezier curve, starting at the current position. Three coordinate pairs are given, Control point 1, control point 2, and the final position.
- 8: Draw a line to the given position. One coordinate pair is supplied.

Here is an example of a path which draws a diamond with its coordinates at (50,100), (100,150), (150, 100) and (100,50)

```

2 50 100 (Move to point at the left)
8 100 150 (Draw a line to next point)
8 150 100 (Draw a line to next point)
8 100 50 (Draw a line to next point)
5 (Close and end the current sub-path)
0 (End the path)
```

The commands in this group help you construct path bodies, and display the corresponding path objects. A path body is built in a local array, and is similar to a Drawfile path body except that all coordinates are given in OS-units. Drawing straight lines needs no particular help. Curves are a different matter and we provide the following functions:

n = PARTCIRCLE(cx, cy, start, end, radius, z())

This function generates the path elements needed to draw part of a circular arc. The parameters are:

ex, cy	The centre of the circle
start	The start angle of the arc (anticlockwise from due East)
end	The end angle of the arc
radius	The radius of the arc, in OS-units
z	An array into which the path is to be written. The path may be up to 30 elements long.

The function returns the length of the generated path.

n = PARTELLIPSE(cx, cy, start, end, x radius, y radius, rotation, z())

This function generates the path needed to draw an arc or an ellipse. The parameters are:

ex, cy	The centre of the ellipse
start	The start angle of the arc (anticlockwise from due East)
end	The end angle of the arc
x radius	The major radius of the ellipse, in OS-units
y radius	The minor axis of the ellipse in OS-units
rotation	The rotation of the major axis from the horizontal
z	An array into which the path is to be written. The path may be up to 30 elements long.

The function returns the length of the generated path.

The arrays returned by **PARTCIRCLE** and **PARTELLIPSE** have the same structure. Suppose the array is called P, in the simplest case (a curve which extends for 90 degrees or less), the elements are :

P(0), P(1):	The coordinates of the start of the curve
P(2):	6, meaning that a Bezier curve follows
P(3),P(4):	The first control point for the Bezier curve
P(5),P(6):	The second control point for the Bezier curve
P(7),P(8):	The end-point of the curve.

If the curve extends for more than 90 degrees, the last seven elements are repeated as many times as necessary. For example, a curve which subtended 200 degrees would need three Bezier groups, and would use up a total of $3 \times 7 + 2$ or 23 elements. When an array generated by **PARTCIRCLE** or **PARTELLIPSE** is being copied into a path, the initial position should be omitted if it coincides with the final position of the previous move. Otherwise, a tag (2 or 8) is needed to move to the first position.

res = DRAWOBJECT(w, path, elements, fill colour)

This function tries to create an object from the given array path, which contains the stated number of elements. The object is filled with the given fill colour.

The parameters are:

w	The handle of the graph
path	An array holding a path
elements	The number of elements in the path
fill colour	The colour used to fill the path

The function fails and returns an error if the path is found to be invalid. If the function succeeds it outputs the object to the graph defined by handle w. Many aspects of the object (such as the winding rule) can be controlled by setting the line attributes before drawing the object.

The functions in this group are used to output objects of irregular shapes such as perspective view of the slices in pie charts.

Example: The following macro draws a picture of a Russian Easter egg balanced on its point. The upper half is a semicircle, and the lower half is an ellipse.

Note the following points:

1. The path array is made long enough to hold up to 30 words for each section of the curve, plus a few more.
2. The initial position of the semicircle is retained, and the first command in the path is an invisible move to this position (element type 2).
3. The ellipse is planned as if it were facing the left (start-angle 90 degrees, end- angle 270 degrees). Then the major axis is rotated by 90 degrees, giving a downward pointing ellipse.
4. The initial position of the elliptical arc is discarded because the pen is already in the right place.
5. The path is terminated by elements of type 5 and 0.
6. The macro uses `p` as a path index, and is pedantic about incrementing it every time a new value is loaded. Clearly some compression could be achieved, but only at the cost of making the macro obscure.

```
MACRO easter_egg
LOCAL w,res,p,j
LOCAL path (70)
LOCAL dump(30)
ON ERROR EXIT
res = MACROANGLE(0)
w = GSTART(300,300)
p=0
path(p) = 2 : REM : Move to start
p=p+1
res = PARTCIRCLE(150,150,0,180,75,dump)
FORj=0TOres-1
path(p) = dump(j) : p=p+1
NEXTj
res = PARTELLIPSE(150,150,90,270,100,75,90,dump)
REM : discard initial position (already there)
FORj=2TOres-1
path(p) = dump(j): p=p+1
NEXTj
path(p) = 5: p=p+1
path(p) = 0: p=p+1
res = DRAWOBJECT(w,path,p,Oxff880000)
res = FONT(w,"Trinity.Bold",60,30)
res = STRING(w,98,90,"XB",Oxffff00,Oxff880000)
=w
ENDMACRO
```

This macro produces the image shown.

res =TEXTCOLUMNOBJECT(w, x, y, width, height, text, fore, back)

This function renders the text as a text column object. The parameters are

w	The handle of the graph
x,y	The lower left-hand corner of the containing rectangle
width	The width of the rectangle
height	The height of the rectangle
text	The text to be plotted
fore	The foreground colour
back	The background colour

Text column objects are fully described on page 4-464 of the RISC OS 3 Programmer's Reference manual. Writers of ordinary macros are unlikely to need this function.

res =ARC(w,cx,cy,start_angle,end_angle,r,colour)

Deprecated; use ELLIPSE

This function draws a segment of a circle, bounded by the two radii and part of the circumference. The parameters are:

w	The handle of the graph
----------	-------------------------

cx,cy	The coordinates of the centre of the ellipse
start_angle	The starting angle of the segment, anticlockwise from due East
end_angle	The ending angle of the segment
r	The radius of the circle
colour	The fill colour of the segment.

The segment is bounded with a line drawn in accordance with the most recent `lineattributes` command.

res = TEXT(w,x,y,text,colour) (deprecated; use STRING)

This function plots a string of characters using the font and dimensions set up by the most recent **FONT** command. The background colour is indeterminate, and the text sizes set by **FONT** command are multiplied by Murphy constants to make the function compatible with the Schema version : 0.375 for the width and 0.4267 for the height. The parameters are

w	The handle of the graph
x,y	The position of the lower left hand corner of the first character to be drawn
text	A string of text
colour	The text colour to be used

res = TCO(w,x,y,width,height,text,colour)

Deprecated: use TEXTCOLUMNOBJECT

This function renders the text as a text column object. The parameters are:

w	The handle of the graph
x,y	The lower left-hand corner of the containing rectangle
width	The width of the rectangle
height	The height of the rectangle
text	The text to be plotted
colour	The colour used for the text

Text column objects are fully described on Page 4-464 of the RISC OS 3 Programmer's Reference manual.

4.8 Utility functions

The functions in this group are useful in drawing graphs.

q = TEXTWIDTH(w,text)

This function delivers the width needed to plot the given string according to the details given in the most recent **FONT** function. The parameters are:

w	The handle of the current graph
text	The text being measured.

The width is delivered in OS-units.

q = TEXTHEIGHT(w,text)

This function delivers the height needed to plot the given string according to the details given in the most recent **FONT** function. The parameters are:

w	The handle of the current graph
text	The text being measured.

The height is delivered in OS-units.

q = FORMAT(value, format)

This function takes a numerical value and translates it into a character string, using the supplied string as a custom format. The meaning of custom format strings is described in *Techniques: Section 9 Custom formats*. Parameters are:

value	A number or expression
--------------	------------------------

format A string which is used as a custom format

Example :

```
q = FORMAT (3.14159265, "£%0.2f")
gives the string £3.14
```

res = GROUP(w), res = ENDGROUP(w)

These two functions are always used as a pairs. All objects sent to a Drawfile between **GROUP** and its matching **ENDGROUP** are grouped together. When the Drawfile is subsequently edited the group is selectable as a compound object.

GROUP and ENDMETHOD can be nested.

4.9 Examples

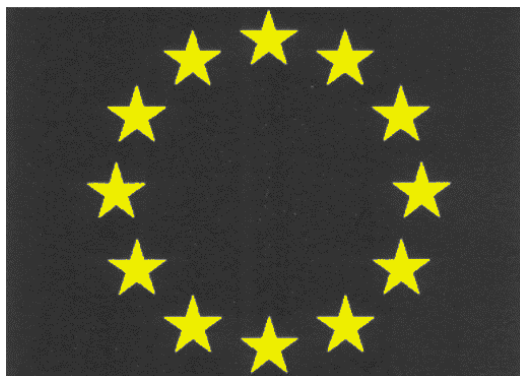
The first example draws a ring of yellow stars on a blue background. The number of stars is supplied as a parameter.

If this parameter is 12, the macro draws the flag of the European Union as it is in December 1993. As countries join or leave the flag can easily be adjusted. The macro uses a sub-macro to draw an individual star. The program is

```
MACRO small_star (w,x,y,z)
REM : this macro plots a small yellow star in graph w at position x,y
REM : and with dimension
LOCAL j,k, res,points(1,5)
ON ERROR EXIT
FOR j = 0 to 4
k = 144*j
points(OJ') = x+z*SIN(k)
points(1^') = y+z* COS(k)
NEXTj
res = LINEATTRIBUTES(w,1,4,-1,0x80)
res=POLY(w,points,5,0xffff00)
=w
ENDMACRO

MACRO flag(number_of_stars)
REM : Main macro.
LOCAL res,wJ,k
ON ERROR EXIT
w = GSTART(700,500)
res = BOX (w,5,5,690,490,0xE0000000)
k = 360/number_of_stars
FORj = 0 TO number_of_stars-1
res = small_star(w,350+200*COS(J'*k),250+200*SIN(j*k),40)
NEXTj
=w
ENDMACRO
```

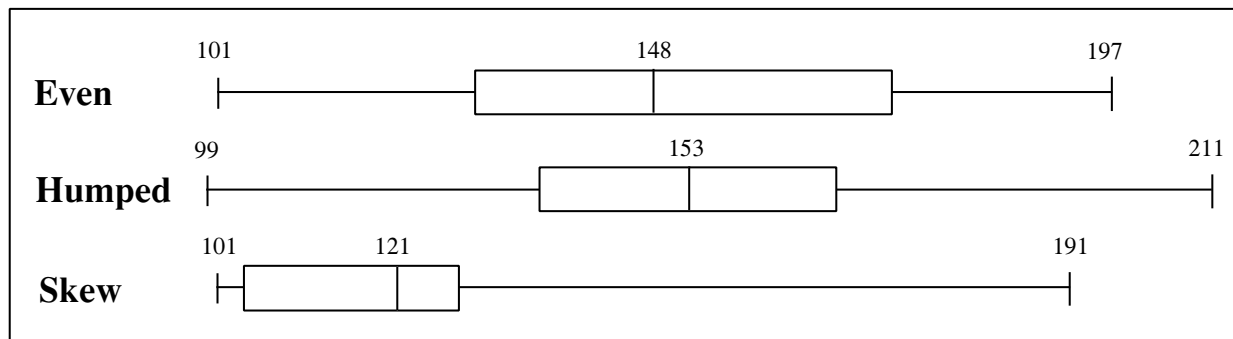
Entering the formula = *flag*(12) in a cell will display the image shown in the cell



The second example draws a box and whiskers chart for an array of data. A box and whiskers chart shows the distribution of a set of numbers, and lets you see at a glance:

- * The average
- * The highest and lowest value
- * The range which contains half of all the numbers

A typical box and whiskers diagram for three sets of data is -



The data in the first set is more or less evenly distributed between the extreme values of 101 and 197; but the second set has a 'humped' distribution, in which most of the numbers cluster round the mean value of 153. The third set is a skew distribution. The actual expressions used to produce these sets were;

Even : =100 + rand(100)
 Humped: =gauss(150,25)
 Skew: = 100 + 1/exp(rand(5))

A simple way to draw a box and whiskers diagram is to sort the data into ascending order. This makes it easy to pick off the extreme values and the upper and lower quartiles, which enclose the middle half of the values.

The complete code for the box and whiskers diagram used to produce the figure is shown below. It uses several sub-macros, which we describe as we go:

The first sub-macro sorts an array of data. It uses the shell-sort algorithm, which is one of the fastest available. We note that the standard function sort is one of the few which is not available on local arrays.

```
MACRO shell_sort (a(),n)
REM : sorts array a(n elements) into ascending order
REM; Code adapted from The C Programming Language, Kernighan
REM: & Richie
LOCAL gap,iJ,temp
gap = n/2
WHILE (gap > 0)
for i = gap to n-1
J = i-gap
WHILE (j>= 0 AND a(j) > a(j+gap))
temp = a(j)
a(j) = a(j+gap)
a(j+gap) = temp
J^J'-gap
ENDWHILE
NEXTi
gap = int(gap/2)
ENDWHILE
=0
ENDMACRO
```

The next sub-macro draws and labels a single box and whisker. It takes 11 parameters (by no means a large number in graphics code), which are used as follows:

window is the graph handle
q is the original array of data, taken from a spreadsheet. It probably has several rows, of which only one is to be used on this call. The

	first column may (or may not) be occupied by a label.
r	is the relevant row number in the data
c	is the starting column number in the data block (it must be over 1)
n	is the number of values

x,y,w and h define the window area where the diagram is to be drawn. The enclosing rectangle has two opposite comers at (x,y) and (x+w, y+h).

m and cc define a transformation. Each value in the data must be multiplied by m and then added to cc to get the corresponding x coordinate.

The transformation is needed because we always want the graph to be a particular width (such as 1100 units) but there is no guarantee that any actual data set will span this width.

```
MACRO draw bw(window,q(),r,c,n,x,y,w,ri,m,cc)
REM window is the current window, and the macro draws a
REM horizontal box-and-whisker diagram in the box
REM delineated by x,y,w and h
REM the raw data is taken from q(r,c) to q(r,c+n-1) - n elements
REM in all. m and cc are used to transform values to x=coordinates
REM using the formula
REM
REM      coordinate = m * value + cc
REM
LOCAL res, a(n),j,top,bottom,box_top
LOCAL box_bottom,mean,smean,tt,ww
REM:
REM : first copy values to local array j
ON ERROR EXIT
FOR j = 0 TO n-1
a(j) = q(rj+c)
NEXTj
REM : next, sort them
res = shell sort(a,n)
top = a(n-1)*m+cc
bottom = a(0)*m+cc
box_top = a(0.75*(n-1))*m+cc
box_bottom = a(0.25*(n-1))*m+cc
mean = AVERAGE(a)
smean = mean *m + cc
res = LINEATTRIBUTES(window,1,1,0,0x80)
REM : Draw components of whisker (8 solid lines)
res = LINE (window,x+bottom,y+0.1*h,0,0.4*h)
res = LINE (window,x+top,y+0.1*h,0,0.4*h)
res = LINE (window,x+bottom,y+0.3*h,box_bottom-bottom,0)
res = LINE (window,x+box_top,y+0.3*h,top-box_top,0)
res = LINE (window,box_bottom,y+0.1*h,0,0.4*h)
res = LINE (window,x+box_top,y+0.1*h,0,0.4*h)
res = LINE (window,x+box_bottom,y+0.5*Pr^box_top-box_bottom,0)
res = LINE (window,x+box_bottom,y+0.5*h,box_top-box_bottom,0)
REM : Draw a dotted line to show mean
res = LINEATTRIBUTES(window,3,1,0,0x80)
res = LINE (window,x+smean,y+0.1*h,0,0.4*h)
REM : Attach labels centred over values
res = FONT (window," Trinity.Medium", 12,12)
tt = FORMAT(a(0),"%0.0f")
ww = TEXTWIDTH(window,tt)
res = STRING(window,x+bottom-ww/2,y+0.60*h,tt,0,0xffffffff00)
tt = FORMAT(mean,"%0.0f")
ww =TEXTWIDTH(window,tt)
res = STRING(window,x+smean-ww/2,y+0.60*h,tt,0,0xfiffff00)
tt = FORMAT(a(n-1)," %0.0f")
ww =TEXTWIDTH(window,tt)
res = STRING(window,x+top-ww/2,y+0.60*h,tt,0,0xffffffff00)
=0
ENDMACRO
```

Finally, we show the main part of the macro. The parameter is the area of the sheet on which the graph is based.

The function allows for any number of data sets up to 7, and lets the user put labels in the first column (but does not insist on this). The variable labels is set to 1 if labels are present; else 0.

Note that the listing of the macro shows some statements split over two lines. To compile correctly these statements must occupy one line only.

```
MACRO bar_whisker(z())
LOCAL w,rows,columns, grand_max,grand_min,j,k,m,cc,res,labels
ON_ERROR_EXIT
rows = first(z)
IF rows > 7 THEN = "Too many rows of data"
columns = SECOND(z)
IF TYPE(z(0,0)) = 3 THEN labels = 1 ELSE labels = 0
REM : Find largest and smallest values overall
grand_max = z(0,labels)
grand_min = z(0,labels)
FOR j = 0 TO rows-1
FOR k = labels TO columns -1
IF z(j,k) > grand_max THEN grand_max = z(j,k)
IF z(j,k) < grand_min THEN grand_min = z(j,k)
NEXTk
NEXTj
IF (grand_max = grand_min) THEN = "All numbers the same"
REM : set up scale transformation so that smallest value appears
REM : at x=100, and largest at x=900
m = 800 / (grand_max - grand_min)
cc = 100 - grand_min * m
REM : Open a window of the right size
w = GSTART(1100,100 * rows)
REM: Draw a box-whisker diagram for each row
FORj=0TORows-1
res = FONT(w,"Trinity.Bold",18,18)
IF labels THEN res=STRING(w,16,100*(rows-j-1)+20,
z(j,0),0,0xffffffff00)
res = draw_bw(w,zj,labels,columns-labels,100,
100 * (rows-j-1),800,100,m,cc)
NEXTj
= w
ENDMACRO
```

Chapter 5 Interactive Graphics

5.1 Problems of interaction

The previous chapter showed you how to write macros for purely passive graphs. In this chapter, we explain how to build macros for graphs which can be edited by the user, and replotted in a variety of ways. This lets you add new types of graph to the standard set provided with Schema2.

To conform to Schema2 standards, new graphs must have certain key features:

1. The colour scheme of every graph, when it is first plotted, should correspond to the default graph colours as they were set in the preferences dialogue.
2. Other features of the graph, such as the form of the axes, the selection of data, the overall size and - in some cases - the 3D effect, must all be capable of being set by the graph dialogues.
3. Once displayed, it should be possible for the user to add decorations, and to change the underlying aspects of the graph such as its colour, the style of lettering, and so on.
4. When a graph is hotlinked to a sheet, subsequent versions should reflect all the changes and additions made during editing.

These requirements are not easy to implement. Graph macros must integrate data taken from a spreadsheet with instructions given in the graph dialogues, and take account of additions and alterations made during graph editing.

The macros used to produce the standard graphs are recorded on the companion disc. They are worth careful study.

5.2 The Graph Life Cycle and the Parameter String

To draw a picture, a graph macro needs two types of data:

- Live data, which is the subject matter of the graph, and which is taken from a block in a spreadsheet.
- Stylistic information, which defines the way the graph is drawn. This information is supplied to the macro as a sequence of coded characters called a parameter string.

The parameter string carries all the information needed to customise the graph. For example it includes colour data, instructions about fonts and character sizes, and information about the overall dimensions of the graph.

The parameter string, in various forms, is used throughout the life of a graph.

1. The string in its initial form is assembled just before the graph is drawn for the first time, using the default colour data and the replies given to questions in graph dialogues.
2. The graph drawing macro interprets the string, and draws the graph accordingly. For example, information about the size of the graph, its colours, and the choice of data from the selected block are all taken from the parameter string. The string itself is divided up into its component parts, which are attached to various parts of the Drawfile as 'tags'. Thus the legend box and each of the axes has its own descriptive tag.
3. As the user edits the graph, the various tags are modified to reflect the changes. New objects are given their own new tags.
4. At some point, the graph may need to be recalculated and drawn again. Before this happens, the existing Drawfile is scanned, and the tags are collected together and formed into a new parameter string. This string is the one used for the recalculation.

In a redraw, it is the graph macro which draws the decorations originally added by the user.

5.3 Editing actions

When the user edits the graph, there are several types of action:

1. The user can add a new component, such as a geometrical object, an imported Drawfile or a segment of text. Such an object is appended to the end of the Drawfile, and a suitable tag is attached. The tag contains information to allow the component to be redrawn if necessary.
2. The user can drag an existing component to a new position. When this is done, the tag attached to the component is modified to reflect the new position of the object.
3. The user can delete an object. When an object is removed from the Drawfile its tag is also taken away.
4. The user can 'top' an object. When an object is topped, it is placed last in the Drawfile, and its tag moves

with it.

5. The user can make an alteration which forces the graph to be redrawn. This might involve a change of axes, a change of scale, a colour change, or a change in the style of lettering. In each case the appropriate tag is altered, and the tags in the Drawfile are collected into a parameter string. Then the graph is recalculated using the new string.

5.4 Hotlinking

Eventually the graph may be hotlinked to a spreadsheet cell. What is actually planted in the cell is a call to the original graph macro, with two parameters:

1. The original selected block.
2. The new parameter string, captured from the Drawfile after the graph has been edited.

The intelligent evaluation mechanism ensures that whenever any cell in the data block changes the graph is recalculated. The new parameter string ensures that the new version of the graph displays all the changes and decorations.

If you hotlink a graph to a cell, and then click on the cell to display its formula in the input area, you will see the parameter string as part of the formula. This is a useful window into the working of the graph macro.

Much of the system we have described is automatic; but there is still plenty of work for the macro writer to do.

5.5 Detailed structure of the Parameter String

The lengthy details which follow are included here because you will eventually need them to write an effective graph macro. On first reading, a quick scan will give you enough background to move on to the next section.

You will notice that the form of the parameter string is economical with characters. This is because Schema2 has a general limit on string length of 512 characters, and we are concerned to squeeze as much information as possible into this small capacity.

Groups

At the highest level, the parameter string can be divided into a sequence of groups. Each group corresponds to a tagged object in the graph, and consists of the following sequence:

Code letter open bracket labelled values close bracket

for example:

C(#Homerton.Medium|v600swO)

The code letter gives the type of object, according to the following code:

A	The whole graph
B	The X axis (horizontal)
C	The Y axis (vertical)
D	A data block (such as a line in a line diagram)
E	A graph title
F	A legend block
a	A line
b	A rectangle
c	A circle
d	An included Drawfile
e	A segment of text
f	An ellipse

The upper case code letters correspond to items in the graph created by the macro. The lower case code letters correspond to items originally added to the graph by the user.

Each labelled value is a single identifying character followed by a value, which may be any of

- A string terminated by a 1. If a string contains the character I, it is represented as II.
- A decimal number.
- A hexadecimal number (using lower case letters).

- Nothing (the character is a Boolean flag).

In our example,

```
#Homerton.Medium| v600 s w0
```

are all labelled values.

The order of labelled values in a group is not significant.

Colours are generally represented as 8-digit hexadecimal numbers. Taken in pairs from the left, the digits represent the saturation values of each of the colours Blue, Green and Red on a scale of 0 to 255 (ff). The lower two digits are always zeros unless the colour is transparent, which is coded as -1 (hex ffffffff). For example, the code 4080COOO represents wine, a colour which is 25% blue, 50% green and 75% red.

We now list the various types of labelled values with details of their use.

General items (can occur in a group of any type)

Character	ASCII code (hex)	Type of value	Meaning
X	(0x58)	Decimal	Horizontal start position
Y	(0x59)	Decimal	Vertical start position
x	(0x78)	Decimal	Horizontal extent
y	(0x79)	Decimal	Vertical extent
m	(0x6D)	Hex	Background colour
K	(0x4B)	Hex	Fill colour
G	(0x47)	None	Do not draw this group if G is present

Text items (can occur in groups of types A,B,C,E,F,and e)

#	(0x23)	String	Font name
H	(0x48)	Decimal	Point size
I	(0x49)	Hex	Aspect ratio (times 10010)
J	(0x4A)	Hex	Text colour
p	(0x70)	Decimal	Alignment (0=left,1=centre,2=right)
&	(0x26)	String	String to be displayed
k	(0x6B)	Hex	Text background colour

Items used only in groups of type A (whole graph)

@ (0x40)	String	7 colours, each coded as 6 hex digits. The groups represent:
		background colour
		data colour 1
		data colour 2
		data colour 3
		data colour 4
		data colour 5
		margin colour

Example:

```
@dddddd0000ff00ff00ff000000fffffffff00fffffl
```

gives:

background : grey
data colour 1 red
data colour 2 green
data colour 3 blue
data colour 4 yellow
data colour 5 cyan
margins white

Note: this is the only time where colours are represented by six digits not eight.

i (0x69) Hex	Overall horizontal graph size
j (0x6a) Hex	Overall vertical graph size

g (0x67)	Hex	Outside borders. The value is coded as four 2- digit hex numbers, arranged as follows: LLTTRRBB
	where	LL is the outside left border TT is the outside top border RR is the outside right border BB is the outside bottom border.

Example: The item g10181020 specifies outside side borders of 16 OS-units, an outside top border of 24 OS-units, and an outside bottom border of 32 OS-units.

h (0x68)	Hex	Inside borders. The coding is the same as for item g
f (0xA3)	String	String from the 'other' box in graph options dialogue
P (0x50)	Decimal	Index of vector in data block to be used as the X axis
Q (0x51)	Decimal	Index in data block of 1 st dependent data set
R (0x52)	Decimal	Index in data block of 2nd dependent data set
S (0x53)	Decimal	Index in data block of 3rd dependent data set
T (0x54)	Decimal	Index in data block of 4th dependent data set
U (0x55)	Decimal	Index in data block of 5th dependent data set
u (0x75)	None	Take rows as vectors. If u is absent take them as columns

Example:

uP3Q4R2S0

means: Take row 3 as the X-axis, and rows 4, 2 and 0 as dependent data sets 1,2 and 3 respectively.

Note codes P to U are only present if the data selection dialogue has been used.

Axis items (used only with groups B and C)

s (0x73)	None	Use logarithmic scale
t (0x74)	Decimal	Add minor ticks
v (0x76)	Decimal	Upper limit on values plotted
w (0x77)	Decimal	Lower limit on values plotted

Line description items

L (0x4C)	Hex	Line colour
M (0x4D)	Decimal	Line thickness
N (0x4E)	Decimal	Style (solid/dotted/dashed)
O (0x4F)	Hex	Path style word

Imported Drawfile items

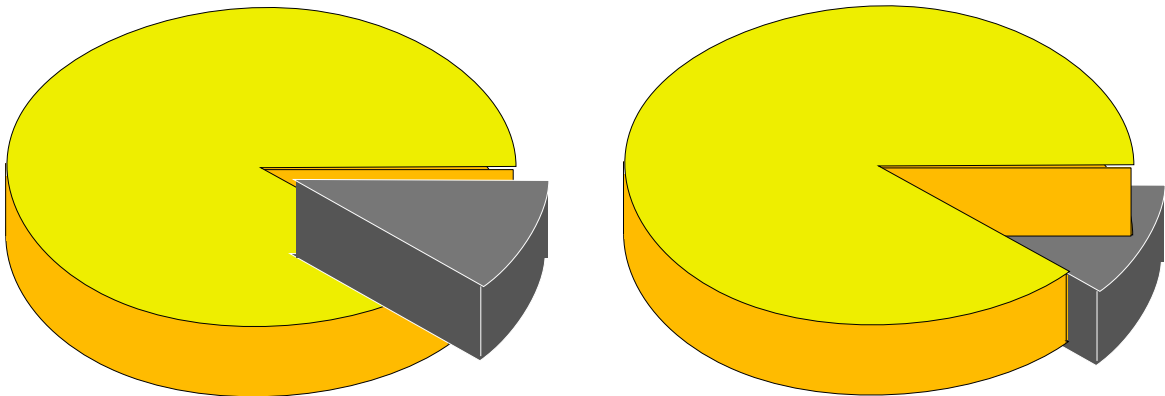
\$ (0x24)	String	Handle of imported Drawfile. The string obtained from this labelled value should be used as the parameter to the FETCHGRAPH() function. The graph handle returned should be used with the relevant function from the PICTURE() family of functions.
A (0x41)	Decimal	Squeeze rule: 0 - full size 1 - scale but do not distort 2 - distort to fit box

Miscellaneous items

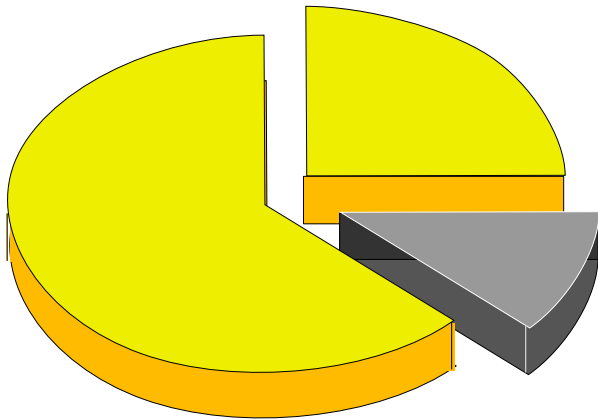
W (0x57)	None	If present, use 3D effect
V (0x56)	None	If present, draw background grid - except in pie charts in which case Explode.
o (0x6F)	Dec	Column width (%) for histograms
q (0x71)	Dec	Marker type (1-5)
r (0x72)	Dec	Marker size
n (0x6E)	Hex	Marker colour

Pie-Chart items

Pie charts with 3D effects can be awkward to draw. Consider a pie-chart of the values 23 and 3, and suppose that you tried to render the slices in sequence. Possible results are shown below.



You can see that in each case, one slice of the pie partially occludes the other one. Whichever order you render the slices on the screen, the diagram looks wrong. To solve this problem it is necessary to divide the larger slice into two sections, and then render the most distant sections first. The right order in this case is shown below.



Furthermore, each slice may or may not be exploded. This affects the way it is drawn.

For a pie chart, the parameter string includes instructions on how to draw the figure. There is a group of type D for each section of the chart, with the following components:

H	(0x48)	Decimal	This section belongs to slice n of the chart
K	(0x4B)	Hex	The top colour of this section
V		None	This section is exploded

The diagram in the first figure consists of only two slices, but needs three sections. The sections are rendered in the order they occur in the parameter string, which is `D(HOKdddddd00)D(HLK77777700V)D(HOKdddddd00)`. This shows that sections 1 and 3 belong to slice 0, and section 2 belongs to slice 1.

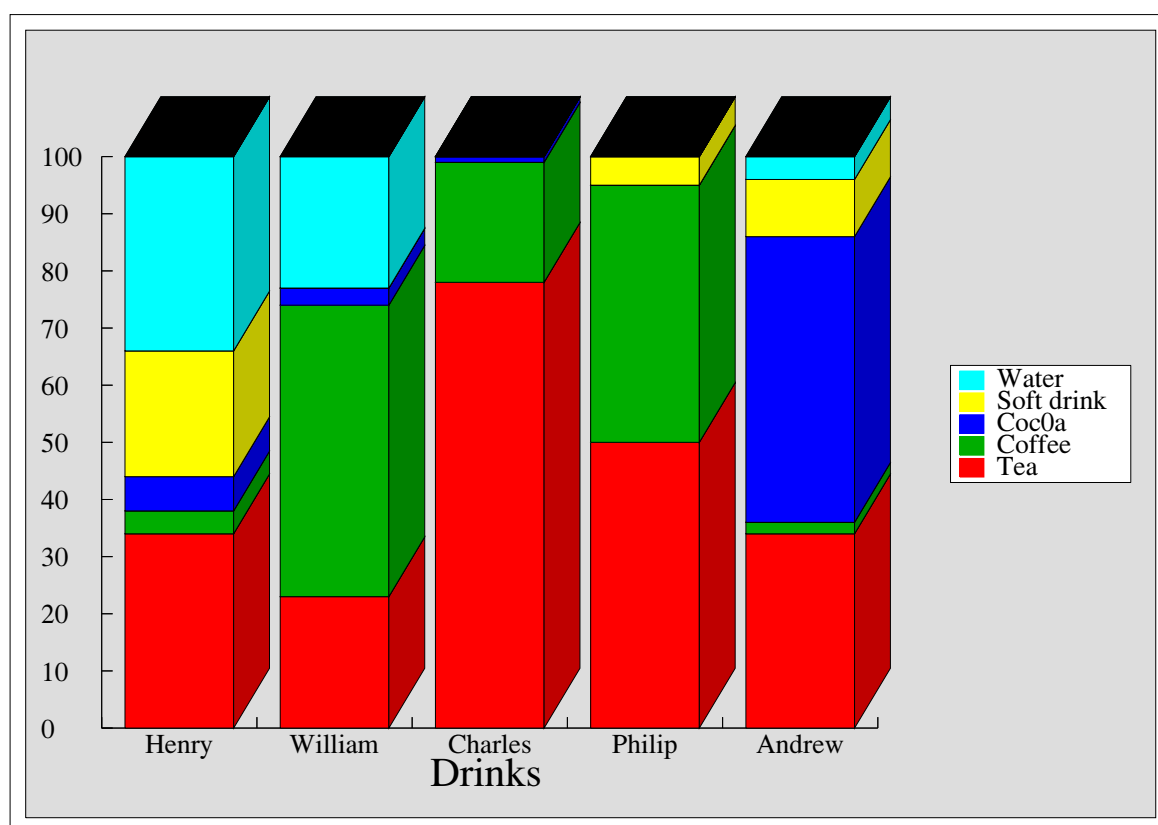
An illustration

To illustrate the concept of the parameter strings, we display the strings used for a typical graph, both before and after editing.

The spreadsheet used as a base is shown below.

	A	B	C	D	E	F
1	Drinks	Henry	William	Charles	Philip	Andrew
2	Tea	34	23	78	50	34
3	Coffee	4	51	21	45	2
4	Cocoa	6	3	1	0	50
5	Soft drink	22	0	0	5	10
6	Water	34	23	0	0	4

The initial graph is a stacked histogram drawn over A1..F6, using 3D. The graph is shown below.



The groups in the initial parameter string are:

```
A (@ddddddO000ffO0ffO0ffO00000fffffrO0ffffli550j320H12WuPO)
B ()
C ()
D (q0)
```

Where values are not supplied, the graph macro takes standard default values. The user now makes the following changes:

1. Adds 2 ticks to the vertical axis. The C group now reads C(t2)
2. Changes the labelling of the horizontal axis to 18-point Homerton.Bold, in red, with an aspect ratio of 85%. The B group now reads

B(I55H18#Homerton.Bold|Jdd)

3. Changes the graph colours to :

Background: off-white
 Colour 1: Yellow
 Colour 2: Dark grey
 Colour 3: Light grey
 Colour 4: Blue
 Colour 5: Red
 Border: Green

The A group is now

A (@f7f7f700eeee999999dddddffbbOOOOOoddOOccOOli550i320H12WuPO)

4. Changes the character size in the legend box and moves it to a new place. The parameter string now contains the group

F(X613Y318H24)

5. Adds a text, a line, a circle, an ellipse, a box and a Drawfile. The parameters string is extended to hold the following groups:
 text:

e(X332Y720H18JccOOkeaeaeal5a#Corpus.Boldl&TheJabberwocky)
 line: a(X808Y700x212y-8LddNIM2000dObOOOb4)
 circle: c(X1138Y160xl50yl32KffbbOOLddM200)
 ellipse: f(X1094Y496xl98y76KddLddM200)
 box: b(X1096Y628xl62y96KeeeeLddM200)
 Drawfile: d(X198Y464x240yl44A2\$x05980900l)

To get a feeling for parameter strings we suggest that you create a graph, make some simple changes and additions, and hotlink it into a spreadsheet cell. If you then examine the expression this cell, you will find the modified version of the parameter string.

5.6 Setting up graph macros

To set up a new graph macro so that the graph can be edited and hotlinked you must include it in the Graph menu. To do this, follow these rules:

Graphics macro parameters

Ensure that the macro has two parameters:

- The first parameter is a block which represents the selected area.
- The second parameter is the parameter string described in section 5.5

The GRAPHMACRO line

The second line of the macro should be

GRAPHMACRO "Name", Flag word, ""

Name is the title of the macro you would like to appear in the graph menu.

The **Flag word** is an integer composed of independent binary digits. The flag word is used whenever a user first asks for a graph of a specific type. One group of digits in the word controls the graph dialogues and determines which areas are to be enabled. Another group of digits helps to specify the initial parameter string passed to the graph.

Individual digits in this word are given in the table below. It might be helpful to display one or two standard graphs, note carefully how the various fields in the graph dialogues are enabled, and then compare your notes with the **GRAPHMACRO** lines in the standard graph macros.

Name	Value	Meaning
Dialogue control:		
GDDATA	0x1	Allow data selection
GDMULTIPLE	0x2	Allow more than one dependent data set
GDXVALUES	0x8	Enable setting of X axis limits and log/lin toggle
GDYVALUES	0x20	Enable setting of Y axis limits and log/lin toggle
GDOPTIONS	0x40	Enable options dialogue
GDPAPERSIZE	0x80	Enable paper size and include i and j items in the parameter string
GD3D	0x100	Enable 3D button
GDPICTURE	0x200	Enable picture field
GDOTHER	0x400	Enable other field
Parameter string control:		
GDLEGEND	0x800	Include a F() group in the parameter string
GDCOLOURS	0x1000	Include the default graph colours in the & item in the parameter string
GDXAXIS	0x4	Include a B() group in the parameter string
GDYAXIS	0x10	Include a C() group in the parameter string

The third parameter of **GRAPHMACRO** is an empty string, reserved for future expansion.

Returning a graphic object

Make sure that your macro returns the handle of the graph it has drawn.

Example : A typical graph macro might start:

```
MACRO polar_plot(a(),s)
GRAPHMACRO "Polar plot", 0x1000,""
LOCAL w
w = GSTART(500,600)
and end with
=w
ENDMACRO
```

5.7 The process of drawing graphs.

In this section we describe the way that the standard Schema2 graphics macros set about drawing graphs.

We suggest you use the same methods for your own graph. You will be able to use many segments of code developed for the standard macros.

In this chapter we refer to a macro which is called only by other macros as a submacro.

The main steps in drawing a graph are these:

1. The main groups in the parameter string are scanned and their contents placed in arrays. Missing values are represented by suitable defaults. There is an array for the whole graph, each of the axes, and the legend box. We refer to these arrays as the graph description vector, the axis description vectors, and so on. There are submacros to do this job, which you will find in the utilities file.
2. The data used in the graph is collected and arranged into canonical form, which is independent of gaps in the original data and its arrangement into rows or columns. The collection process uses the information supplied in the data dialogue about data sets and their orientation.

All further work is done on this normalised data set.

3. The layout of the graph is worked out. This is usually a long and complex process as it involves the size of the graph, the dimensions of the axes, an allowance for legends and consideration of the data itself. In some cases it is necessary to start with plausible assumptions and to iterate to a good layout by repeating calculations more than once. There are submacros to plan axes from ranges of data values. The size of text segments can be found by using the **TEXTHEIGHT** and **TEXTWIDTH** functions.
4. The graph is drawn according to the layout. This is relatively straightforward, since there exist submacros

to handle the drawing of axes and legend boxes. A key point is that every object you draw such as an axis must have a tag attached to it using the **SETTAG** function (see below). If no tag is attached the macro will be unable to redraw the graph when called to do so.

5. Finally, the parameter string is scanned for added items such as labels, lines, imported graphs, and so on. In each case the details are extracted from the parameter string and used to draw the object. There is a submacro to handle this task in the utilities file, called `CRM_tail`.

We now explain the steps in further detail.

5.8 Parameter vectors

Contemporary programming techniques would use structures to hold details of the sets of values given in the parameter string. The Schema2 macro language does not support structures, but can offer a viable alternative.

First, no variable has a fixed type associated with it, so that any element of an array can hold a number, a string or a graphic object with equal ease.

Second, the macro language can use the define statement to attach names to values. For example, the statement

```
define POINTS_ONLY,1
```

causes the token **POINTS_ONLY** to be handled just like the value 1.

These two features permit the use of pseudo-structures, defined as follows:

```
define PERSON_SIZE,3
define NAME,0
define ADDRESS,!
define AGE,2
define WEIGHT,3
```

and later,

```
local boss(PERSON SIZE)
boss(NAME) = "James Watson"
boss(ADDRESS) = "34 Gunter Gardens, Glasgow G40 8TT"
boss(AGE) = 77
boss(WEIGHT) = 67.6
```

Defined values remain in scope until the end of the current file of macros.

The graph description vector

All the description vectors are very similar. The axis description vectors are derived from the graph description vector. The graph description vector includes 43 elements. They are

define PSIZE,43	Size of the array
define FOGR,0	Character font to be used
define FSGR,1	Character size to be used
define FAGR,2	Aspect ratio for characters
define TCGR,3	Text colour
define LCGR,4	Line colour
define LWGR,5	Line width
define PDGR,6	Line path description
define PSGR,7	Line style
define BGGR,8	Background colour
define PWGR,9	Total graph width (OS-units)
define PHGR,10	Total graph height (OS-units)
define BWGR,11	Reserved for future use
define DCOLGR,12	First of 5 Data set colours
define GRGR,17	Set if graph uses a grid
define ROWS,18	Number of rows in normalised data set
define COLUMNS,19	Number of columns in normalised data set
define TDGR,20	Set if graph is in 3D format
define GRAPHTYPE,21	Graph type (a value as defined above)

define XLABEL,22	The label for the X axis (if any)
define YLABELS,23	Labels for the data sets (if any)
define THICKNESSGR,29	3d thickness
define FRGR,30	Column width for histograms
define OLGR,31	Outside left margin
define OTGR,32	Outside top margin
define ORGR,33	Outside right margin
define OBGR,34	Outside bottom margin
define ILGR,35	Inside left margin
define ITGR,36	Inside top margin
define IRGR,37	Inside right margin
define IBGR,38	Inside bottom margin
define MTGR,39	Marker type
define MSGR,40	Marker size
define PIGR,41	Handle of file to be inserted
define MAGR,42	Margin colour

5.9 Handling the parameter string

One of the key parts of any graph macro is the extraction of values from the parameter string. This could be done with the standard functions such as FINDQ and MID(), but to speed up matters Schema2 provides a set of special functions for this task, which are described below. The function names all start with the letters PS for "Parameter String".

u = PSREGISTER(v)

This function registers a parameter string for analysis and returns an integer handle. A parameter string must be registered before any of the other functions in this group can be executed.

There is a limit of about 10 on the number of strings which can be registered at the same time. In practice it is rare to be handling more than one string at a time. The only way to de-register a string is to use **PSSTEP()** (see 5.9.2). Note that strings must be de-registered or they will prevent other strings from being registered; with the result that Schema2 will fail after plotting ten graphs.

q = PSSTEP(u)

This function locates and selects the next group in the string registered as u, and returns the ASCII equivalent of the group identifier (A to F or a to f). If the string is newly registered the function selects the first group. If there are no substrings left, the function de-registers the string and returns 0.

The other functions in this group all work on the selected group of a registered string.

z = PSEXSUB(u)

This function delivers a copy of the selected group in the string registered as u. If no group is selected the function returns a null string.

The letters EX in the name of this function stand for EXtract.

z = PSCONTAINS(u, s)

This function searches the selected group for an occurrence of a value of type s. s is a one-character string. The function returns 1 (True) if the value is found, and 0 (False) if not.

This function is mainly used for fields with no values, such as G.

z = PSEXDEC(u, s, default)

This function searches the selected group for an occurrence of a value of type s, which should be followed by a decimal number, s is a one-character string. If found, the function returns the value of the following number; if not, it returns the default value given as the third parameter.

z = PSEXHEX(u, s, default)

This function searches the selected group for an occurrence of a value of type s, which should be followed by a hexadecimal number, s is a one-character string. If found, the function returns the value of the following number; if not, it returns the default value given as the third parameter.

z = PSEXSTR(w, s, "default")

This function searches the selected group for an occurrence of a value of type s, which should be followed by a string, s is a one-character string. If found, the function returns the value of the string; if not, it returns the default value given as the third parameter.

5.10 Parameter string manipulating macros

These functions are not quite versatile enough to jump around the parameter string in the way needed by some of the graph macros. Some useful submacros are provided in the Utilities macro file, and they are quoted in full, as an illustration of the PS standard functions:

```
REM given a parameter string s and a group code, this function
REM opens the string and tries to select the substring with
REM the given code. If found it exits with a handle.
REM If not, it returns -1
REM Note that after the handle has been used the string must
REM closed by calling close_string
MACRO CRM find_substring(s,key)
LOCAL v,q
v = PSREGISTER(s)
IF ISERROR(v) THEN = v
REPEAT
q = PSSTEP (v)
IF ISERROR(q) THEN = -1
IF q = CRM_code(key) THEN = v
UNTIL q = 0 OR ISERROR(q)
=(-1)
ENDMACRO

REM : This function closes the string with current handle v
MACRO CRM_close_string(v)
LOCAL q
REPEAT
q = PSSTEP (v)
IF ISERROR(q) THEN = 0
UNTIL q=0
=0
ENDMACRO
```

The work of extracting parameters for the graph description vector is done by the function **CRM_get_gr_params**, which is also in the Utilities file. This is called with two parameters: an empty graph description vector and a handle to a registered parameter string, where the selected group is of type A.

Part of this function is

```
macro CRM_get_gr_params(p(),v)
localq
p(FOGR) = psexstr(v,"#", "Trinity.Medium")
p(FSGR) = psexdec(v,"H",12)
p(FAGR) = psexhex(v,"I",100)
p(TCGR) = CRM_psexcol(v,"J",0)
p(LCGR) = CRM_psexcol(v,"L",0)
p(LWGR) = psexhex(v,"M",256)
p(PDGR) = psexhex(v,"N",1)
p(PSGR) = psexhex(v,"O",0)
p(PWGR) = psexhex(v,"i",800)
p(PHGR) = psexhex(v,"j",600)
q = psexhex(v,"g",0x05050505)
p(OLGR) = (q & 0xff000000) » 22
p(OTGR) = (q & 0xff0000) » 14
p(ORGR) = (q&0xff00)»6
p(OBGR) = (q & 0xff) « 2
p(GRGR) = pscontains(v,"V")
.....
```

The graph description vector is set up by code like:

```
MACRO plotgraph(a(),s)
ON ERROR EXIT
LOCAL P(PSIZE)
LOCAL v,res
```

```
v = CRM_find_substring(s,"A")
res = CRM_get_gr_params(P,v)
res = CRM_close_string(v)
```

The description vectors for the axes and for the legend box are loaded by similar code, except that default values are often taken from the main graph description vector.

5.11 Normalising the data

Data normalisation is designed to produce the following arrangement:

1. The data actually used for the graph is copied into an array, where each data set is represented by a column. The number of rows is the same as the number of values in each data set. The number of rows and columns in this array are loaded into elements **ROWS** and **COLUMNS** of the graph description vector.
2. The label for the X-axis (if any) is copied into element **X_LABEL** of the graph description vector.
3. The labels for each of the data sets (if any) are copied into elements **Y_LABELS**, **Y_LABELS+1**,... of the graph description vector.

The work is done by two functions

in the Utilities file:

```
point_plot_row_data if the original data sets are rows
point_plot_column_data if the original data sets are columns
```

These functions take account of certain special cases. For example, if no X data set is specified the functions set up a dummy X data set with the values 1,2,3 ...

If no Y data sets are specified the functions take sensible default decisions.

One small problem is caused by the following aspect of the Macro language:

It is convenient to declare the normalised data array in function **point_plot_row_data**. However, a Schema2 macro cannot return a local array to its calling code. It is therefore necessary for **point_plot_row_data** to call the actual plotting routines as submacros so that they can be passed the normalised data block. A section of code at the end of **point_plot_row_data** is:

```
.....
IF p(GRAPHTYPE) = PIE CHART THEN
=CRM_pie_chart_plot(data,s,p,v)
ENDIF
IF p(GRAPHTYPE) = PICTOGRAM THEN
=CRM_picto_plot(data,s,p,v)
ENDIF
.....
```

When you add any new graphics functions, you must use augmented and renamed copies of **point_plot_row_data** and **point_plot_col_data**. Don't try to modify the original versions.

5.12 Working out the graph layout.

When you calculate the layout of a graph, it is vital to know the dimensions of the objects you plan to show. The existing macro system provides some useful support:

1. The position and size of the 'live' graph area - that is, the area inside the borders where you can safely plant objects - can be found from the graph description vector, using the overall graph and border dimensions.
2. The size of any string you plan to display can be discovered by using the standard functions **TEXTHEIGHT** and **TEXTWIDTH**. For these to work correctly you must call an appropriate **FONT** command first.

Under the right conditions, the macro **CRM_finddims** (Utilities file) will do this for you. The function returns a string in which two values are embedded. Its format is

```
x=hs y=vs
```

where *hs* and *vs* are the horizontal and vertical sizes, respectively. A sample reply would be

```
x=156y=18
```

The function **CRM_extract** (Utilities file) can be used to extract individual values from such a reply.

The device of returning a string with labelled, embedded values is used wherever a function must return more than one answer.

Plotting the axes is another problem. The Utilities file contains two functions which are useful for designing axes:

CRM_linear_scale is used to plan a linear scale. The parameters are:

smallest, largest size	The extremes of the range to be plotted
	The number of OS-units available for the scale

The function chooses two round numbers, *ll* and *hh* which enclose the range, and which will be used as the limits of the axis. The function then suggests a number of intervals for the scale, and calculates coefficients which will serve to convert any value to a displacement from the position of the smallest value.

To give an example, suppose that **CRM_linear_scale** is called with the range extremes 2.8, 9.4 and a size of 1000 OS-units. The function might return axis limits of 2 and 10, to be divided into 8 intervals. The conversion coefficients will be 125, and -250. This would ensure that the value 2 will transform to

$$2 \times 125 - 250 = 0$$

and the value 10 to

$$10 \times 125 - 250 = 1000$$

CRM_linear_scale returns its results as a string of the form

```
a=125b=-250i=8ii-1011=125
```

CRM_log_scale works the same way but plans a logarithmic axis with an integral number of decades.

The axes file includes several functions which help to plan axes axes of various types.

5.13 Plotting the graph

If your macro has completed the design process so that all the dimensions of the graph are readily available, the actual rendering of the graph is laborious but straightforward. The axes and legend files hold useful submacros for this purpose.

Before you start building the graph you must arrange two matters:

1. The graph as a whole must carry a **tag string** which corresponds to the 'whole graph' group (type A).
2. The graph carries a **flag word** which controls the menu items to be enabled when the picture is subsequently edited. The flag word includes the following bits:

Name	Value	Purpose
DRAG_ALLOWED	0x1	If set, the object can be dragged
SELECT_ALLOWED	0x2	If set, the object can be selected
TEXT_EFFECTS	0x4	Text effects enabled
FILL_EFFECTS	0x8	Fill effects enabled
LINE_EFFECTS	0x10	Line effects enabled
GRAPH_COLOUR_ALLOWED	0x20	Colour changes allowed
THREE_DEE_ALLOWED	0x40	Three dimensional display permitted
GRID_ALLOWED	0x80	Background grid allowed
EXPLODE_ALLOWED	0x100	Explode allowed (for a pie chart)
COLOURS_ALLOWED	0x200	Colour changes allowed
BAR_WIDTH_ALLOWED	0x400	Bar width allowed (for histogram)
POINT_DETAILS_ALLOWED	0x800	Point details (for scatter diagram and marked lines)

TICKS_ALLOWED	0x1000	Ticks on axes allowed
TOP_ALLOWED	0x4000	Item may be topped
DELETE_ALLOWED	0x8000	Item may be deleted
REDRAW_NEEDED	0x40000	Graph must be redrawn when item is changed

res = SETTAG (tag, flag word)

This command sets up a tag string and a flag word. Both items are picked up by the next Drawfile command (usually **GSTART** or **GROUP**) and attached to the current object.

The tag string is an array of characters which obeys the rules for a group in the parameter string. A suitable tag can nearly always be found in the current parameter string, but it may sometimes be necessary to synthesise a tag string. The flag word is a collection of bits with the meanings given above. For example you would only expect to set **EXPLODE_ALLOWED** for pie-charts or other diagrams which support exploded views.

The tag and flag word for a Drawfile could be set by the following sequence. Note that v is the current string handle.

```
local w,v,res,tag
v = CRM_find_substring(s,"A")
tag = PSEXSUB(v)
res = CRM_close_string(v)
res = SETTAG(tag, THREE_DEE_ALLOWED | ...
... | REDRAW_NEEDED)
w = GSTART(WIDTH,HEIGHT)
```

Certain groups of objects, such as axes or legend boxes, must be identified so that the user can select and edit them. This is done with the **GROUP** and **ENDGROUP** commands.

If the group is given its own tag string and flag word, it will be selectable during editing. When the group is selected, the flag word overrides the flags attached to the graph as a whole. The tag string enables the macro to redraw the object in the same way if the graph needs to be redrawn.

The following code could be used to draw a horizontal axis (group code B). u is the parameter string, and w is the handle of the current Drawfile. The group is copied from the parameter string:

```
LOCAL v,w,res
v = CRM_find_substring(s,"B")
k = PSEXSUB(v)
res = CRM_close_string(v)
res = SETTAG(k,TICKS_ALLOWED | SELECT_ALLOWED |
TEXT_EFFECTS | DELETE_EFFECTS | REDRAW_NEEDED )
res = GROUP(w)
..... (Commands to draw the components of the axis)
res = ENDMETHOD(W)
```

Note that the flag word does not include **DRAG_ALLOWED**; the user is not permitted to move the axes about.

To end this section, we again emphasise that a graph will only be redrawn correctly if the graph macro is meticulous in attaching the correct tag string both to the graph as a whole, and to its individual components. If a tag is badly constructed or omitted, the error will not show until the graph is replotted, so replotting is an essential part of the test process for new graphs.

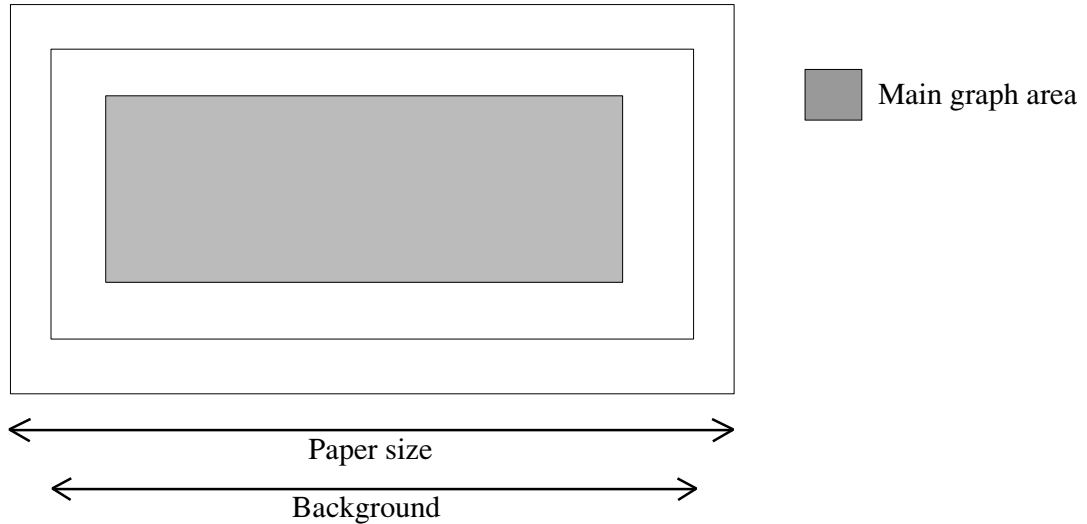
5.14 Summary of requirements of an interactive graphics macro

Here we list the things your macro must do in order to appear consistent to the user. Most of the points listed here are dealt with earlier in this chapter; they are included here to help with error finding. We do not deal with the issues of good design of graphs and aesthetics.

It is possible not to support some features, by greying them out on the dialogues and macros. However the more general you make your graph macro the more useful it is to the user.

1. The macro must take two parameters, a data area and parameter string
2. The macro must contain the **GRAPHMACRO** line with the flag word properly set. Features which your macro does not support should be greyed out on the graph dialogue.

3. The macro must return the handle of the graph produced or, if unable to produce a graph, a suitable error or string.
4. The macro must use the **SETTAG()** function to set a tag string and a correct flag word for the entire graph. The tag string should usually be the same as the A() group in the parameter string (obtainable by calling the **PSEXSUB()** function).



5. The macro should draw a box the size of the paper (as specified by the i and j items in the A() group) in the border colour. Then it should draw a box which covers the paper except for the outer borders in the background colour. 6. All further items should be restricted to an area bounded by the inner borders (the main graph area in figure 5.5), unless they have explicit X and Y items in their group (in which case the user dragged them to that point).
7. The parameter string should be scanned for information on how to draw the relevant parts of the graph.
8. Each selectable object in the graph must have an associated **tag string** and **flag word** set by the **SETTAG()** function, which must convey sufficient information to enable the object to be re-plotted. If there was a parameter string group associated with the group that should usually be used as the tag string.
9. All items added to the graph by the user must be drawn in the manner of the **CRM_tail macro**.

5.15 Conclusion

Graph plotting is not easy; there are many geometrical computations to be done and you are obliged to use primitive commands of extreme simplicity.

Before you do much serious work on graph design, we suggest you study the sources for the standard Schema2 graph macros. The code is far from perfect, but these programs have two solid merits: they work, and they define a lower level of acceptability which you can improve on.

Chapter 6 Macro Reference

This chapter forms a reference manual for Schema! macros. Each concept is described concisely, and may be illustrated by examples if appropriate.

6.1 Lexical Rules

Macros are written using elementary tokens or words such as numbers, strings, mathematical operators, and so on. Most of the token types are the same as those used in Schema! expressions. Upper and lower case letters are not distinguished except inside strings.

Spaces

Spaces are significant in two contexts:

- Inside strings(as in "This is a reference manual")
- As token terminators

(e.g., **NOT NOT** is two occurrences of the token **NOT**, but **NOTNOT** is a single token)

Reserved words

The following tokens are used only in macros:

BOOTMACRO	DEFINE	ELSE	ENDIF
ENDMACRO	ENDWHILE	EXECMACRO	FOR
FUNCMACRO	GOTO	GRAPHMACRO	IF
LET	LOCAL	MACRO	GRAPHMACRO
NEXT	ON_ERROR_CONTINUE	ON_ERROR_EXIT	REPEAT
REM	STEP	THEN	TO
WHILE			

Names

Names are used for macros, parameters and local variables. A name always starts with a letter and may continue with any sequence of letters, digits and the the under-score symbol `_`. In distinguishing macro names no regard is paid to the case of any letter; **CRM**, **crm**, **Crm** and **cRm** all represent the same name.

There are certain exclusions to names:

- No name may be the same as a reserved word.
- No name may be the same as a possible cell name, either in the A1 or the R1C1 notation.
- No name may be the same as a standard function.
- Names should be distinct from the names of all other macros. It is legal but not advisable to use names which begin with the characters **CRM_**. Such names may interfere with the standard graphics system.

Examples:

```
A xy Totals Best_of_3
```

Counter-examples:

```
ELSE AND G5 DFY1045 R5C77 crm_bug
```

Comments

Any sequence which starts with the reserved word **REM** and ends with a newline is taken as a comment or remark, and ignored. This does not apply if the sequence **REM** occurs inside a string.

6.2 Operands

Operands are of the same basic types as in the spreadsheet: numbers, strings, block designations and error markers. Numbers can either be integers (exact quantities in the range -2147483648 to +2147483647) or real numbers with a precision of 15 digits and a range of $1e-306$ to $1e306$ approximately. Strings may contain up to 512 characters.

Operands occur in expressions, which have exactly the same meaning as they do in spreadsheets. There are

several ways in which operands can be specified:

- Constants
- Local variables
- Parameters
- Cell names
- Block references
- Graphics

Constants

Constants are written the same as in spreadsheet expressions. Examples are

```
23 45.76 2.4e-5
" A string \"inside\" another"
```

Local variables

A local variable is a named quantity used inside a macro. There are two types of local variable - scalar and array.

- A scalar local variable is specified in an expression just by giving its name.
- A complete local array is specified by giving its name.
- An element from a local array is specified by the array name followed by one or two subscripts in brackets. Each subscript can be any expression, provided the value of the subscripts is within the size of the local array.

Examples of elements in local arrays are:

```
values(5)
matrix(4,5)
table(xx,yy+1)
```

If a local variable contains a block as its value then it may be subscripted like a local array. For example

```
var_1 = WHOLESHEET("Sheet1")
var_2 = var_1(1,2)
```

assigns the value of cell C2 in spreadsheet Sheet 1 to the local variable var_2. See section 6.4.3 for a discussion on the declaration of local variables.

Note that a variable name cannot be the same as the name of the macro even if it is in a different case.

Parameters

A parameter is a named quantity used to pass data into a macro. There are two types of parameter - scalar and array. In general scalar parameters represent single quantities such as numbers or strings, and array parameters represent groups of quantities, such as local arrays or blocks of spreadsheet cells.

- A scalar parameter can be used in an expression just by giving its name.
- An array parameter is also used by giving its name. It must occur in the right context, such as a call to a standard function or macro which expects an array parameter.
- An element from a parameter array is specified by the array name followed by one or two subscripts in brackets. Each subscript can be any expression, provided that its value is always within the size of the parameter array.

Examples of elements in parameter arrays are:

```
list(5)
contour(4,5)
map(xx,yy+1)
```

See section 6.4.1 for a discussion on the declaration of parameters.

Cell names

A macro can specify a spreadsheet cell name as an operand. The standard way to write a cell name is as a foreign reference, namely:

- A ! sign
- The name of a spreadsheet
- A point
- The name of the cell.

an example is:

```
!January.G7
```

If two cell names are combined to make a block, the spreadsheet name can be omitted in the second cell, as in

```
!Yearly_tot.H8..J20
```

The macro language permits indexing of cell names. The cell names can be followed by two indices in brackets. The first index is a row offset and the second one, a column offset. For example,

```
!January_sales.J8(6,4)
```

is actually a reference to **!January_sales.N14**

The indices may be arbitrary expressions, with positive or negative values, provided they do not produce references to cells off the edge of the spreadsheet. If a macro is called from a cell in a given sheet, the name of that spreadsheet serves as a default and can be omitted from cell references. Examples of this usage are

```
G55     A1(j,k)
```

If a macro is called without an implicit link to any spreadsheet then cell references without spreadsheet names are illegal and generate run-time faults.

Graphic handles

Certain functions return values which represent images. These variables are of type graphic, and they can only support a limited set of operations:

- A graphic can be passed as a scalar parameter to other macros
- A graphic can be assigned to a local variable, to an element of a local array, or to a spreadsheet cell.
- A graphic can be returned as the result of a macro.
- The function **TYPE()**, if applied to a graphic object, yields 6.

6.3 Expressions

In general, expressions are governed by the same rules as expressions in spreadsheets. However there is a wider range of operand types (local variables and arrays, parameters) and there are over 100 standard functions which are available only from within macros. They are described in chapters 3,4,5 and 8.

Examples of expressions within macros include:

```
a+1
p3+q7 + IFj=5 THEN name ELSE 56.6 ENDIF
whole(1,2)..whole(3,4)
ncr(pd(j),92) - ncr(92, pd(j))
NOW()
```

6.4 Macros

A Macro is a named, self-contained segment of code analogous to a function in BBC BASIC, C or Pascal. Macros communicate with their environment through parameters, through named spreadsheet cells which serve as global variables, and by delivering results.

Every macro must include the following three components, in the given order:

- A header line.

- One or more statements.
- A terminator line.

A macro may include a descriptor line to indicate which menu (if any) it is to appear in. If present this is placed immediately after the header line.

A macro may also include one or more local declaration lines. If present, they must be placed between the header line and any statements.

Two simple macros are:

```
(1)      macro within(a,b,x)      REM header line
        = (x>=a)and(x<=b)        REM statement
        endmacro                 REM terminator

(2)      macro ncr(a,b)           REM header
        local s,t                REM local declaration
        s=1                     REM statements follow
        for t=1 to b
        s = s * (a-t+1)/t
        next t
        = s
        endmacro                 REM terminator line
```

The Header line

The header line of a macro consists of the reserved word **MACRO** followed by the name of the macro. Macro names are subject to all the normal rules about names, but as all macros in an application share the same name space no macro name should clash with any other macro name. The standard macros use 'private' sub-macros which all start with the characters **CRM_**, so do not use standard macro names or any name starting with **CRM_**. If the macro has any parameters then the macro name is followed by a list of the parameter names in brackets. The parameters order must be the same as those in the macro call.

A parameter can be either a scalar variable (a single quantity) or an array or block name. Arrays and blocks are indicated by empty pairs of brackets after the name. If a macro has no parameters, the macro name is the last item on the line. As explained in section 6.2.3, parameters can be used as operands in expressions. If a parameter is a scalar, the call is by value; the value supplied to the macro is a copy of the original. No changes made to this copy will be reflected in the original. On the other hand, if the parameter is an array the call is by reference, and the value supplied to the macro is essentially a pointer to the original array. Changes to individual elements of the parameter will be result in changes in the original.

This is best illustrated by example:

```
MACRO funny(a, b())      REM a complete macro
a =123                  REM a is a scalar parameter
b(1) = 45.88             REM b is an array parameter
=0
ENDMACRO

.....                 REM Part of a macro
z = 65.7
y(1) = "Hello"
res = funny(z,y) REM Call to funny
.....
```

After funny has been executed, z = 65.7 and y(1) = 45.88

Examples of macro headers are

```
macro timeout
macro search(area(), value,type)
```

The Descriptor line

The descriptor line is only needed if the macro is to be callable from a menu.

It can take one of three forms:

a) EXECMACRO

This implies that the macro name is to be made into an item in the main menu tree. The name appears in the submenu reached by **Utilities > Macro**. When the name is selected and clicked the macro is executed. Any quantity returned by the macro is ignored unless it is an error, which is displayed in an error box.

An example might be:

```
MACRO start (a())
EXECMACRO
LOCAL x
REPEAT
x =INPUT("STARTING","First name?"," ","OK","CANCEL")
UNTIL TYPE(x) =3
a1=x
REPEAT
x =INPUT("STARTING","Second name?"," ","OK","CANCEL")
UNTIL TYPE(x) =3
a2=x
=0
ENDMACRO
```

b) FUNCMACRO

This implies that the macro name is to be made into an item in the function menu tree. The name appears in the submenu reached by the Macros entry. When the item is selected and clicked the name is copied into the formula being entered, just like a standard function name.

An example of a function macro is

```
macro larger (a,b)
FUNCMACRO
if a>b then =a else =b
endmacro
```

c) GRAPHMACRO

The word GRAPHMACRO is followed by a string constant, a number constant and a string constant, in that order.

This specifies that the macro name is to be included in the graph menu. The string constant gives the menu entry, and the number constant is a flag word which controls the fields of the graph dialogue. The flag word is described in Chapter 5. The second string constant is reserved for future use.

An example of a graphmacro descriptor line is

```
GRAPHMACRO "Whiskers", 0x28D, " "
```

Local Declarations

A local declaration is written as the word local followed by a list of names for local variables and arrays. The various items are separated by commas.

To specify an array, the name should be followed by dimensions in brackets. One- and two-dimensional arrays are allowed.

In each case the dimension gives the highest usable subscript. The lowest subscript is always 0.

The dimension can be a parameter to the function, or any expression involving a parameter.

Several local declarations can be written one after the other, but they must all be placed before the first statement.

Each macro has its own name space for local variables and parameters. Names for local variables must obey the general naming rules and should not clash with other names in the same macro or use the same name as the macro itself, but need not be different from local names in other macros.

Local variables and arrays are all initialised to zero.

Examples of local declarations are

```
MACRO twist (b(),x,y)
LOCAL z,q(5), risk(5,7),image(x,y)
LOCAL qq(x'*y)
```

6.5 Statements

There are only a few basic types of statement:

- Assignment
- Selection (IF)
- Iteration control (FOR, REPEAT,WHILE)
- Jumps (GOTO)
- Return statement
- Error control statements

As in BBC BASIC, statements generally begin with keywords. Statements are normally placed one to a line, but several short statements can be placed on one line and separated with colons.

Assignment

There are two forms of assignment statement with equivalent meanings:

```
LET <left_hand_side> = <expression>
```

and

```
<Left_hand_side> = <expression>
```

Here **<left_hand_side>** denotes a local variable or array element, a parameter or parameter array element, a spreadsheet cell (possibly indexed) or a user name from a spreadsheet.

<expression> is any expression.

The effect of the statement is to calculate the expression and assign its value to the **left_hand_side**.

Examples are

```
let a = 7
b5 = sum(q)
area(2,k) = area(0,k) * area(1,k)
```

Selection

Selection is managed by IF statements, of which there are two types:

The short type is always written on one line and takes one of the forms

```
IF <expression> THEN <simple sequence>
IF <expression> THEN <simple sequence> ELSE <simple sequence>
```

where **<expression>** is any expression which yields a scalar value and **<simple sequence>** is any sequence of assignment statements and possibly a **GOTO** or a **RETURN** statement, separated by colons.

The **<expression>** usually but not necessarily includes a relational operator such as = or <.

To evaluate a short IF statement the system first calculates the value of the expression. If the expression is a string it is converted into a number. If the result is anything other than zero, the system goes on to evaluate the statements in the first **<simple sequence>**, and skips the second **<simple sequence>** (if any). If the expression does evaluate to zero, the system skips the first **<simple sequence>** and evaluates the second one (if present).

Examples of short IF statements are:

```
IF a > 5 then a=a-5
IF D4 = "ACORN" THEN D4 ="OAK":D5="MAPLE"
if ab < ac then xy = ac else = ab
```

The long version of the IF statement always occupies several lines. The first line is always

```
IF <expression> THEN
```

and the last one is always

```
ENDIF
```

In between these two lines you can place any sequence of statements, without restriction. In particular, IF statements can be nested as required. To evaluate this form of IF statement the system computes the expression and then executes the sequence only if the result of the expression is not zero.

An example of this form is

```
IF !January.A5 = !February.A5 THEN
FOR k = 1 TO 10
x(k) = 0
NEXTk
ENDIF
```

In another form, the IF and ENDIF lines can enclose two sequences separated by the word ELSE. If the expression evaluates to a non-zero value only the first sequence is executed; if zero, then only the second.

An example occurs in:

```
MACRO FACT (n)
IF n = 0 THEN
=1
ELSE
= n * FACT(n-1)
ENDIF
ENDMACRO
```

Iteration

The macro language provides three ways of controlling iteration. In the following description:

- **<expression>** stands for any expression which produces a scalar value.
- **<sequence>** stands for any sequence of commands which is complete with regard to matching constructions; that is, all opening brackets are matched by closing brackets, each compound IF by ENDIF, and so on.
- **<local>** stands for any local variable or parameter but not a cell or array element.

The REPEAT construction

```
REPEAT
<sequence>
UNTIL <expression>
```

The given sequence is repeated until the expression evaluates to a non-zero value. One execution of the **<sequence>** is guaranteed.

An example is

```
a=0
b=0
REPEAT
b=b+1
a=a+1/b
UNTIL b=100
```

The WHILE construction

```
WHILE <expression>
<sequence>
ENDWHILE
```

At the beginning of each cycle the system evaluates the **<expression>**. If the result is non-zero, it executes the **<sequence>** and restarts the cycle. Otherwise it skips the **<sequence>** and continues at the line after **ENDWHILE**.

If the <expression> is zero at the start the <sequence> is not executed at all. An example is

```
a=1
WHILE ABS(a*a-x) > 0.000001
a=0.5*(a+x/a)
ENDWHILE
```

The FOR construction

```
FOR <local> =<expression1> TO <expression2> STEP <expression3>
<sequence>
NEXT <local>
```

Here the <sequence> is executed for each of several values of the <local>, starting at <expression1>, changing in steps of <expression3> until <expression2> is reached but not passed. <expression3> may be of either sign. It is also possible to omit <expression3> and the preceding **STEP**; in this case a step size of +1 is assumed.

If the sign of (<expression2> - <expression1>) is different from the sign of <expression3> or its default value, the sequence will not be executed at all. Some examples are shown below. They all assume that suitable local variables have been declared:

```
REM sets a(1) to A(10) inclusive to zero
FORj=1 TO 10
a(j)=0
NEXTj
REM sets a(1), a(4) and a(7) to zero
FOR j=1 TO 9 STEP 3
a(j)=0
NEXTj
REM adds up the elements in array w
s=0
FOR r= 0 TO 4
FOR c = 0 TO 9
s = s+w(r,c)
NEXTc
NEXTTr
```

Labels and GOTOS

Any statement in a macro, provided that it is the first statement in a line, can be labelled by preceding it with a <label name>.

A <label name> is an arbitrary but distinct local name, not used for any other purpose.

The **GOTO** command is written

```
GOTO <label name>
```

When executed a **GOTO** breaks the normal sequence of commands and transfers control to the named statement. The following example shows how a typical **FOR** statement could be decomposed into simpler constituents:

```
FOR j=a TO b STEP c
<sequence>
NEXTj
```

This is defined as equivalent to

```
j=a-c
xxx: j=j+c
IF SIGN(c)=1 AND c>b THEN GOTO yyy
IF SIGN(c)=(-1)AND c<b THEN GOTO yyy
<sequence>
GOTO xxx
YYY: .....
```

It is not possible to jump from one macro to another using **GOTO**.

The RETURN statement

RETURN is used to return a value to the calling sequence. It is written

```
=<expression>
```

When obeyed, the system calculates the value of the expression and returns it to the calling expression. Execution of the current macro ceases, and all local variables and scalar parameters are destroyed,

Every macro must contain at least one **RETURN** statement, but may include more than one.

An example is

```
= sum/n
```

6.6 Error handling in macros

There are two error control statements that may occur in a macro, **ON_ERROR_EXIT** and **ON_ERROR_CONTINUE**.

Possible Sources of error

Errors in macro execution can arise from 4 different categories:

Errors in assignment.

This may be an attempt to assign a value to a non-existent array element or spreadsheet cell. Execution of all macros and all other evaluation is terminated immediately and a suitable error marker returned to the calling macro or cell.

Stack overflow.

This implies that there is not enough memory available for local variables or arrays. By far the most common cause of stack overflow is unintentional recursion or recursion running out of control. When stack overflow occurs execution of all macros and all other evaluation ceases and a suitable error marker is returned.

Depression of the Escape key.

This event is handled exactly like a stack overflow error; the macro is stopped and an error marker is returned.

Typically the Escape key is used during macro development, to stop macros which are stuck in infinite loops.

Errors in expressions.

This category includes arithmetic errors such as overflow or division by zero; incorrect parameters for standard functions, and calls to undefined functions. Errors in expressions generate 'error' values, which can be assigned to variables or cells. The normal action is to treat the errors as special values and continue with execution.

The error control statements allow this rule to be varied.

ON_ERROR_EXIT

When **ON_ERROR_EXIT** is executed by a particular macro, it ensures that if the macro attempts to assign an expression error value, the macro is terminated immediately, returning the error value in question.

Consider the following macro, which was written to illustrate this point:

```
macro error_demo (a)
  local z
  if a then ON ERROR EXIT
  z=1!0
  =3
endmacro
```

The call =error_demo(0) produces 3

but =error_demo(1) generates "Division by zero"

ON ERROR CONTINUE

When **ON_ERROR_CONTINUE** is executed it switches expression error handling back to normal if **ON_ERROR_EXIT** had been executed.

The effects of these two commands are local to the specific macro in which they occur. It is usual to make **ON_ERROR_EXIT** the first executable statement.

Chapter 7 Evaluation

7.1 General considerations

In the Intelligent Automatic mode a spreadsheet is equivalent to a functional language - that is, whenever a value or group of values is changed anywhere, all other values which depend on those values are automatically updated. The order of evaluation is partially defined so as to ensure that each relevant expression need be evaluated only once.

On the other hand the Schema2 macro language, like BASIC, is a procedural language - the order of evaluation is tightly controlled by the code itself.

When the two systems are used together in a spreadsheet, their evaluation rules could conflict, and we need safeguards to avoid endless cycling and other unwanted effects. The rules are arbitrary, and represent a compromise towards controlling a problem which is basically insoluble.

Fortunately the special rules only apply in a small minority of cases, so that in practice they can be ignored. For purposes of reference they must be stated.

7.2 Macros which do not affect spreadsheet cells

The large majority of macros (including all the graphics macros) do not affect any cell of a spreadsheet except for the one where the result appears. In this case there is no interaction, and no need for special rules.

7.3 Macros which store values in cells

Macros are permitted to store values to spreadsheet cells, using assignment statements such as

```
B3 = J+1
```

For this purpose, spreadsheet cells are treated in the same way as local variables, except that the contents persist after the macro has completed its execution. The assignment is done at the time the statement is executed, and the change to the cell does not trigger a chain of automatic evaluations of other cells. Assignments to cells do not cause the sheet's "modified" flag to be set,

To give an example, consider the macro:

```
MACRO place
A1=17
=0
ENDMACRO
```

Suppose a sheet already contains the following

```
A1:      3
A2 :    =A1+1 (currently 4)
```

Now arrange for macro place to be executed (perhaps by calling it from a menu). The result will be:

```
A1:      17
A2:       4
```

Note that A2 has not been re-evaluated.

7.3 Macros which cause re-evaluation of the spreadsheet

Macros which contain certain standard functions require special treatment, since they have the same effect as user actions which would require re-evaluation. The standard functions are all those which directly affect the contents or structure of the spreadsheet; we call them evaluating functions.

ADDCOLUMNS	ADDROWS	COPY	CUTCLIP
DELETE	DELETECOLUMNS	DELETEROWS	MOVE
PARTLOAD	PASTECLIP	PUTBLOCK	REPLICATECLIP
SETSHEETINFO	SORT	WIPE	

To illustrate the problem, consider the PUTBLOCK() function, which puts an expression into each cell of a given block. It is equivalent to the user action of selecting a block, entering a valid formula and clicking on the tick

icon.

Whenever this action takes place all the cells in the block need to be re-evaluated; but so may other cells which are outside the block but whose values depend, directly or indirectly, on cells within the block.

Now suppose one of the cells being re-evaluated calls a macro which uses **PUTBLOCK()**. It is obvious that left unchecked, this arrangement could lead to an infinite regression, which could crash the system. The mechanism described manages to avoid that without restricting the system in any practical sense.

The system recognises three levels of evaluation : zero, one and two.

Evaluation level zero

Level zero is used to evaluate a macro which is called directly from a menu, such as a graphics or procedure macro.

If a macro being executed at level zero calls an evaluating function, the re-evaluations arising from that function are immediately executed at level one. All cells therefore take on their correct values by the time the next statement is executed in the macro.

The formulae of cells are never evaluated at level zero.

Evaluation level one

Level one is used for evaluations resulting from direct user commands, such as changing the dimensions of the sheet, cutting or pasting its contents, and planting formulae into cells or blocks. Level one is also used for evaluations which arise from evaluating functions called in macros at level zero.

Level one is the normal level for the evaluation of cell formulas and macros called from them.

If an evaluating function is called from a macro which is being executed at level one the consequent re-evaluations will be executed at level two after all evaluations at level one are complete. This will be rare, as it is rarely sensible to place a macro which alters the spreadsheet's structure into a spreadsheet cell.

Evaluation level two

Level two is only used for evaluations resulting from evaluating functions called in macros at level one. Level two evaluation takes place immediately after all level one evaluation is complete. If an evaluating function is called from a macro which is being executed at level two then no consequent re-evaluations will take place.

Example

The EvalExamp directory contains a macro called Add_totals (source is in the AddTots file). The purpose of this macro is to enter a new row of totals under some columns of data. To use it:

1. Load the EvalExamp workspace (loads the files : =AddTots and Data) into Schema2.
2. Select B3..C7 in the Data spreadsheet.
3. Menu and choose Utilities » Macro > Add_totals.

You will see that the new cells B8..C8 will have SUM() formulae in them now.

When the line

```
res =ADDDROWS(bottom_row, 1)
```

in the macro is executed the whole spreadsheet is re-evaluated at level one before the next line is executed. There are no cells with calls to macros containing evaluating functions in the spreadsheet, so no level two evaluation takes place.

When the PUTBLOCKQ is executed then cells B3..C8 are evaluated at level one. If however instead of calling the macro from the menu we had opened a second spreadsheet, Unfilled, and entered into cell !L)ntitled.AI the formula:

```
=Add_totals(!Data.B8..C8)
```

then a different series of evaluations would have occurred. The cell AI and thus Add totals would have been

evaluated at level one. When the line

```
res =ADDDROWS(bottom_row, 1)
```

in the macro is executed the whole of the spreadsheet would be marked as needing to be re-evaluated. However the next statement would be executed before this happened. The same would happen with the call to PUTBLOCKQ. When the macro finished execution the whole of Data would be re-evaluated at level two, giving the correct values. However as !Untitled.A1 depends on cells in Data it would also be re-evaluated at level two. Thus an extra unwanted row would be inserted. Further re-evaluations would not occur.

Chapter 8 Standard functions available from macros

Schema2 has 111 functions which can be called from macros but not directly from cells in spreadsheets. These functions do not appear on the function menu. In general, macro functions work like the functions on the menu. Each one expects a certain pattern of parameters and delivers a result.

Many of the macro functions correspond exactly to user actions. Some of these functions can modify a spreadsheet and cause re-evaluation to occur. The rules about re-evaluation are set out in Chapter 7.

Parameters

The standard functions use a variety of parameter types, described below.

Array: A local array or a block of cells from a spreadsheet.

Example : MACRO Mode(AreaO)
 LOCAL d(20,30)

d, !Porpoise.D4..G6 and Area are all arrays.

Attribute name: A string which defines which effect, cell information or spreadsheet information you are interested in finding out about or modifying. The capitalisation of the word is not significant.

Examples: "customformat", "CustomFormat", "XSize"

Block: A block of cells from a spreadsheet

Example : MACRO Embolden(Area())
 LOCAL d(20,30)

Area and 'Porpoise.D4..G6 are blocks (provided the macro is called with a block as the parameter), but d is not.

Cell attribute: The attribute name for an attribute of a cell, given as a string

Examples : "x", "CellWidth", "CustomFormat"

Cell attribute strings include **Effect** strings. The other strings and their descriptors are described in section 8.12

Colour: An integer number usually expressed in hexadecimal form. The first two digits give the blue strength (between 00 and FF), the second two the green, the third two the red and the last two are 00. For further explanation see the Custom formats section of the main manual.

Example: OxFFOOF00 is Magenta.

Effect: The attribute name for an effect, given as a string

Examples : "font", "customformat"

Effect name strings and the descriptors that go with them are:

String	Value	Meaning
Alignment	String	L = left, R = right, C = centre, J = justify
BgColour	Colour	Colour for background
Bold	Flag	1 = bold; 0 = not bold
CustomFormat	String	A custom format string
FgColour	Colour	Colour for foreground
FontHeight	Number	Font height in points
FontName	String	Font name
FontWidth	Number	Font width in points
Italic	Flag	1 = italic; 0 = not italic
VAligment	String	T = top, B = bottom, M = middle, F = fit graphs

Descriptor: The value for an attribute name, which can be a colour, distance, flag, style or string
Examples: "Trinity.Bold", OxFFOOS00

Distance: A number giving a measure in os units (1/180* of an inch)
Example: 20 (=0.111 inches)

Flag: A Boolean parameter : 1 or 0
Example: 1

File handle: The handle returned when a data file is opened with the OPE() function.

File type:	An integer number defining the type of a file, usually expressed as a hexadecimal number. Standard values are: OxFF TSV OxD FE CSV OxD BO WK1 OxC 26 ASheet (Normal Schema2 spreadsheet file) OxA FF Drawfile
Int:	An integer Example: 350
Path:	The full path name of a file, quoted as a string. It is often useful to use the variable <Workspace\$Dir> when all your data is in the same directory. Examples: "ADFS:ST506Disc4.\$\$.sheets.jansales" "<Workspace\$Dir>.Summary"
Spreadsheet:	The identity of a spreadsheet. This is quoted as a block, but the block details are ignored Examples : !Manatee.Al..S50 , !Manatee.H6..H6 These examples are equivalent
Sheet attribute:	The attribute name for an attribute of a spreadsheet, given as a string Examples : "AngleMode", "printdraft" Sheet attribute strings and the descriptors that go with them are defined in section 8.12.
String:	A string Example: "Cyclones bring bad weather"
Style:	The name of a style, quoted as a string Example: "Accountant"
Value:	A number or a string Examples: 45 78.19 "Best"
View handle:	A unique identifier for a view

Results

Many of the functions described generate a 0 if the function succeeds or an error if it fails. We do not describe the possible errors in detail. In the examples, the result of every such function is assigned to variable res.

Where a function generates an interesting value it is described, and in the example is assigned to a variable with a name other than res.

8.1 Array sizing functions

FIRST (Array)

Purpose: Delivers the first (or only) dimension of an array or block

Parameter: Array is the name of the local array or block

Example:

```
LOCAL x(5,7), y(18),
Zdim=FIRST(x) gives 6
dim = FIRST(y) gives 19
dim = FIRST(z) gives 0
dim = FIRST(!Dolphin.Al..C7) gives 7
```

SECOND (Array)

Purpose: Delivers the second dimension of an array or block

Parameter: The name of the local array or block

Example:

```
LOCAL x(5,7), y(18),
Zdim = SECOND(x) gives 8
dim = SECOND(y) gives 1
dim = SECOND(z) gives 0
dim = SECOND(!Dolphin.Al..C7) gives 3
```

8.2 Graphics functions

These functions are all defined in Chapters 4 and 5. Section numbers are:

Function name	Reference	Function name	Reference
ARC	4.7	POLYMARKER	4.7
BOX	4.7	PSCONTAINS	5.7
DRAWOBJECT	4.7	PSEXDEC	5.7
ELLIPSE	4.7	PSEXHEX	5.7
ENDGROUP	4.5	PSEXSTR	5.8
FONT	4.6	PSEXSUB	5.7
FORMAT	4.8	PSREGISTER	5.7
GCLOSE	4.5	PSSTEP	5.7
GOPEN	4.5	SAVEGRAPH	4.5
GRAPHLOAD	4.5	SCALEDPICTURE	4.7
GROUP	4.8	SETTAG	5.10
GSTART	4.5	SQUEEZEDPICTURE	4.7
LINE	4.7	STRING	4.7
LINEATTRIBUTES	4.6	TCO	4.7
OPENPOLY	4.7	TEXT	4.7
PARTCIRCLE	4.7	TEXTCOLUMNOBJECT	4.7
PARTELLIPSE	4.7	TEXTHEIGHT	4.8
PICTURE	4.7	TEXTWIDTH	4.8
POLY	4.7		

8.3 Type and error management

The functions in this group are used to manage and report on types and errors.

TYPE(parameter)

Purpose: To return the type of the given parameter
Parameter: A value of any type
Result: The type of the value, coded as an integer:
1 - Integer
2 - Number with a fractional part
3 - String
4 - Fault
5 - Block name
6 - Graphic object
7 - Local array

Examples:
TYPE(3.45) gives 2
TYPE(" Schema!") gives 3
TYPE("G1..J17") gives 5
TYPE(1/0) gives 4

ISERROR(parameter)

Purpose: To determine whether a value is an error
Parameter: A value of any type
Result: 1 if the value is a fault; otherwise 0

ERRORCODE(value)

Purpose: To produce a string describing an error
Parameter: A value of any type
Result: If the parameter is a fault, a suitable message. Otherwise, the message "This is not a fault"

Examples:
ERRORCODE(1/0) gives "Division by zero"
ERRORCODE(ASIN(3.4)) gives
"Argument for trig function out of range"
ERRORCODE(1) gives "This is not a fault"

NA()

Purpose: To produce the fault value "N/A"

Parameters: None
Example: `ERRORCODE(NAQ)` gives "N/A"
Comment: "N/A" is used to mean that a value is absent.

REPORTERROR(string1, string2, int)

Purpose: To display a RISC OS error message
Parameters: String1 defines a box title, String2 defines the box contents Int is a flag; the error box is only displayed if this parameter is zero or greater.
Example: `REPORTERROR("Title", "Message", 0)` displays a RISC OS error message box with the given title, and bearing the given message. Execution of the macro continues when the user clicks on the OK button.

8.4 Functions which create, destroy and re-size a spreadsheet

ADDCOLUMNS(Block, Int)

Purpose: Add columns to the spreadsheet
Parameters: Block identifies a column, Int is the number of columns to insert after this column
Result: The columns are added and the entire spreadsheet re-evaluated.

Example: `res = ADDCOLUMNS(!Untitled3.H1..H29, 7)`

ADDROWS(Block, Int)

Purpose: Add rows to the spreadsheet
Parameters: Block identifies a row, Int is the number of rows to insert after this row
Result: The rows are added and the entire spreadsheet re-evaluated.

Example: `res = ADDROWS(!Untitled3.A3..S3, 5)`

CREATESHEET (Name, Int1, Int2, Flag)

Purpose: Create a new spreadsheet
Parameters: Name gives the name of the spreadsheet. If it is the empty string (" ") then the function gives the new spreadsheet a name of the form UntitledN.
Int1 gives the number of rows. Int2 gives the number of columns. If Int1 and Int2 are both 0 then the size of the spreadsheet is determined by the default document.
Flag: 0 : Do not open a view on the spreadsheet
1: Open a view of the spreadsheet after creation

Examples: `new_sheet = CREATESHEET(" ", 0, 0, 1)` is equivalent to clicking on the Schema2 icon.
`new_sheet = CREATESHEET("WorkArea", 20, 30, 0)` creates a hidden spreadsheet, called WorkArea, with 20 rows and 30 columns.

DELETECOLUMNS(Block)

Purpose: Delete columns from the spreadsheet
Parameters: Block identifies one or more columns to be deleted
Result: The columns are removed and the spreadsheet re-evaluated.

Example: `res = DELETECOLUMNS (!Untitled3.I1..J34)`

DELETEROWS(Block)

Purpose: Delete rows from the spreadsheet
Parameters: Block identifies one or more rows to be deleted
Result: The rows are removed and the entire spreadsheet re-evaluated.

Example: `res = DELETEROWS(!Untitled3.A3..Q4)`

DELETESHEET(Spreadsheet)

Purpose: Discard a spreadsheet
Parameters: Spreadsheet identifies the spreadsheet to be deleted
Result: The spreadsheet is discarded from memory without checking if it was modified. This

function can only be used in macros called from the menu or icon ribbon. If used as a result of a cell being evaluated it returns an error and does not discard the spreadsheet

Example: `res = DELETESHEET(!Sheet1.A1..A1)`

8.5 Functions which enter, move and delete formulas and data

CUTCLIP(Block)

Purpose: Cut a block from a spreadsheet and send it to the clipboard
Parameters: Block identifies the block to be cut
Result; The area cut is deleted, as in the DELETEDQ function, and re-evaluated.

Example: `res = CUTCLIP(!Untitled3.B6..J12)`

COPY(Block1, Block2, Int1, Int2)

Purpose: Copy data and formulas from one location to another
Parameters: Block1 identifies the area to be copied, Block2 identifies the top left cell of the target area
Int1 gives the number of horizontal replications, Int2 gives the number of vertical replications
Result: The area is copied the required number of times and re-evaluated.
This is equivalent to **COPYCLIP()** followed by **REPLICATECLIP()** except that the contents of the clipboard are not modified at all.

Example: `res = COPY(!Sheet1.B3..C4, !Sheet2.A1..A1,2,2)`

COPYVALS(Block1, Block2, Int1 Int2)

Purpose: Copy data from one location to another
Parameters: Block1 identifies the area to be copied, Block2 identifies the top left cell of the target area
Int1 gives the number of horizontal replications, Int2 gives the number of vertical replications
Result: The data in the area is copied the required number of times. The formulas in the source block are not copied, so that when a value in the source block subsequently changes, the alteration is not reflected in the target block.

Example: `res = COPYVALS(!Sheet1.B3..C4, !Sheet2.A1..A1,2,2)`

COPYCLIP(Block)

Purpose: Copy a block from a spreadsheet and send it to the clipboard
Parameters: Block identifies the block to be copied

Example: `res = COPYCLIP(!Untitled3.B6..J12)`

DELETE(Block)

Purpose: Clear a block of cells from a spreadsheet
Parameters: Block identifies the block to be deleted
Result: The area is deleted and re-evaluated.

Example: `res = DELETE(!Untitled3.D7..D10)`

Note: See WIPE (section 8.5.11)

MOVE(Block1, Block2)

Purpose; Move data from one location to another
Parameters: Block1 identifies the area to copied, Block2 identifies the top left cell of the target area
Result: The area is moved and re-evaluated. This is equivalent to **CUTCLIP(Block1)** followed by **PASTECLIP(Block2,1, 0,0)** except that the contents of the clipboard are not modified at all.

Example: `res = COPY(!Sheet1.B3..C4, !Sheet2.A1..A1,2,2)`

PASTECLIP (Block, Flag1, Flag2, Flag3)

Purpose: Paste or move data from the clipboard
Parameters: Block identifies the top left cell of the target area
 Flag1 is 1 if this is a move operation; else 0
 Flag2 is 1 if this is a values-only operation; else 0
 Flag3 is 1 if the data is to be transposed; else 0.
 If Flag1 is 1 then Flag2 and Flag3 must be 0.
Result: The clipboard is pasted in and the affected area(s) re-evaluated.

Examples:

```
MACRO DoPaste(Target0)
LOCAL res
res = PASTECLIP(target ,0,0,0)
is equivalent to Edit > Paste, while
res = PASTECLIP(target,0,1,1)
would be equivalent of Edit > Paste values I Transpose.
```

PUTBLOCK(String,Block)

Purpose: Enter a formula into a spreadsheet block (This is the macro equivalent of selecting a block, keying an expression into the input area, and clicking the tick icon) Parameters: String is the formula or data to be entered (note that if the expression is itself a string another pair of quotes is needed).
 Block is the target address

Examples:

```
res = PUTBLOCK("1",!Untitled3.A1..A1)
res = PUTBLOCK("A1+1",!Untitled3.B1..Q1)
res = PUTBLOCK("\"Hello world'V", target)
```

REPLICATECLIP (Block, Int1 Int2)

Purpose: Replicate data from the clipboard
Parameters: Block identifies the top left cell of the target area
 Int1 gives the number of horizontal replications
 Int2 gives the number of vertical replications

Example:

```
res = REPLICATECLIP(!Untitled3.I5..I5,1,2)
```

WIPE(Block)

Purpose: Remove data from an area leaving styles and effects
Parameters: Block identifies the area to wiped
Result: All formulae and values are removed from the area. However unlike **DELETE()** the styles and effects are left. The area is re- evaluated.

Example:

```
res = WIPE(!Sheet3.B3..C4)
```

8.6 Functions which affect the appearance of a spreadsheet

BIGCELL(Block, Flag)

Purpose: To create or destroy a big cell
Parameters: Block gives the area to be converted
 Flag is 0 if a big cell is to be created; 1 if it is to be destroyed

Examples:

```
res=BIGCELL(!Dolphin3.B5..D6,0) rem make a big cell
res=BIGCELL(!Dolphin3.B5..B5,1) rem unmake it
```

BORDERBITS(Block, Int1, Flag, Int2)

Purpose: To set the cell borders of a block of cells
Parameters: Block identifies the block of cells, Int1 gives the type of action:
 0: clear all border bits
 1: clear and set borders

2: add borders to those already in place
3: clear specified borders
Flag indicates whether Int (below) is valid:
0: use parameter bits given in 1
1: ignore 1 and use a single border for the entire area
Int2 if Flag = 0, Int2 indicates which borders to set or clear:
1: left
2: right
4: top
8: bottom
The value for Int2 is obtained by adding the values for the relevant borders.

Examples:
`res = BORDERBITS(!Untitled3.B2..C3,1,1,0)` set the border around B2..C3
`res = BORDERBITS(area, 2, 0,12)` add top and bottom borders to all cells in area.

CHANGEHEIGHT(Block, Distance)

Purpose: To change the height of a block of rows
Parameters: Block specifies a block of rows
Distance specifies a height in OS-units

Example: `res = CHANGEHEIGHT(!Untitled3.A5..Q5, 88)`

CHANGESTYLE(Block, Style)

Purpose: To change the style of a block of cells
Parameters: Block specifies a block of one or more cells
Style is the style to be applied
Result: The style is applied to the block. To edit a style use the **MODIFYSTYLE()** function.

Example: `res = CHANGESTYLE(!Untitled3.C3..C3, "Heading")`

CHANGEWIDTH(Block, Distance)

Purpose: To change the width of a block of columns
Parameters: Block specifies a block of columns
Distance specifies a width in OS units

Example: `res = CHANGEWIDTH(!Untitled3.C1..E28,192)` will set columns C, D and E to be 1.07 inches wide.

HIDECOLS(Block)

Purpose: To hide some columns of a spreadsheet
Parameter: Block identifies the columns to be hidden

Example: `res = HIDECOLS(!Untitled3.G1..H30)`

HIDEROWS(Block)

Purpose: To hide some rows of a spreadsheet
Parameter: Block identifies the rows to be concealed

Example: `res = HIDEROWS(!Untitled3.A4..Q4)`

REVEALCOLS (Spreadsheet)

Purpose: To reveal all hidden columns of a spreadsheet
Parameter: Spreadsheet identifies the sheet

Example: `res = REVEALCOLS(!Untitled3.A1..Q30)`

REVEALROWS (Spreadsheet)

Purpose: To reveal all hidden rows of a spreadsheet
Parameter: Spreadsheet identifies the spreadsheet

Example: `res = REVEALROWS(!Untitled3.A1..Q30)`

SETEFFECT(Block, Effect, Descriptor)

Purpose: To set a new effect into a block of cells

Parameters: Block is the target block, Effect names the effect and must be one of the strings below, Descriptor is the value of the effect

Examples:

```
res=SETEFFECT(!Untitled3.B2..C2,"fontwidth",18)
res=SETEFFECT(!Untitled3.B2..C2,"alignment",1)
res=SETEFFECT(!Untitled3.B2..C2,"fgcolour", OxFF00FF00)
res=SETEFFECT(!Untitled3.C3..C3,"customformat", " %alf%0.2f")
```

INPUTBLOCK(Block,Flag)

Purpose: To set or clear the input block frame round a block of cells

Parameters: Block is the block to be marked or cleared
Flag is 1 to set the frame, 0 to clear it

8.7 Loading and Saving functions

LOAD(Path)

Purpose: Load a file of a given name

Parameter: Path is the path name of the file, the file can be of any type that Schema2 is capable of loading.

Result: If an error occurs then the function will return the error. If the file loaded was a workspace or a macro file then the function returns 0, otherwise it returns a block which covers the whole spreadsheet created. This block can then be used for all future references to the spreadsheet. If you want a view to be opened on the spreadsheet then you should use the **NEWVIEW()** function.

Example:

```
new_sheet = LOAD("<WorkSpace$Dir>.Table")
REM If spreadsheet successfully loaded open a view
IF TYPE(new_sheet) = 5 THEN
res = NEWVIEW(new_sheet, 0,0,0,0,0,0)
REM Now reveal any hidden data
res = REVEALROWS(new_sheet)
res = REVEALCOLS(new_sheet)
ENDIF
```

PARTLOAD (Block, Path)

Purpose: Load a spreadsheet of a given name into another spreadsheet, already loaded

Parameters: Block marks the top left corner of the area into which the new spreadsheet is to be loaded
Path is the path name of the spreadsheet to be loaded

Example: `res=PARTLOAD(!Geo.C3..C3,"ADFS::ST506Disc4.$.sheets.China")`

PARTSAVEFILE (Datablock, Path, File type)

Purpose: Save a part of a spreadsheet as a file of given name and type

Parameters: Datablock identifies the spreadsheet to be saved
Path is the full path name of the file
File type is the file type number (normally 0xC26 for ASheet)

Examples:

```
res=PARTSAVEFILE(!March.b3..k99,"ADFS::Floppy1.$.Data", 0xDFE)
will save the datablock to disc Floppy 1 as a CSV file.
res=PARTSAVEFILE(!Summary.B2..D5,"<WorkSpace$Dir>.KeyValues", 0xC26)
will save part of Summary as a spreadsheet in the Workspace directory.
```

SAVEFILE (Spreadsheet, Path, File type)

Purpose: Save a spreadsheet as a file of given name and type

Parameters: Spreadsheet identifies the spreadsheet to be saved

Path is the path name of the file

File type is the file type number (normally OxC26 for ASheet). If the spreadsheet is saved as an ASheet file then a new leaf name for the spreadsheet is extracted from the path name and the spreadsheet is marked as unmodified.

Example: `res=SAVEFILE(!deco.Al..J28,"ADFS:MyFloppy$.Sheet2",OxC26)`

SAVEWS (Path)

Purpose: To save the current workspace

Parameter: Path is the pathname of the workspace

Result: The workspace is saved and <WorkSpace\$Dir> is reset.

Example: `res = SAVEWS("ADFS:ST506Disc4$.sheets.ws")`

8.8 View manipulation functions

Each spreadsheet can have up to four views open. As views are created and destroyed the ordering of the views can change. To manipulate a view the macro functions use a view handle, which is returned by the **NEWVIEW()** and **VIEWHANDLE()** functions. The view handles are not affected by other views being created and destroyed. The number of open views can be obtained from the **GETSHEETINFO()** function.

DELETEVIEW (Spreadsheet, View handle)

Purpose: To delete a view

Parameters: Spreadsheet identifies the spreadsheet to which the view is attached
View handle is obtained from the **NEWVIEW()** or **VIEWHANDLE()** functions

Example: `res = DELETEVIEW(!Bt.al..Q30, handle)`

GETVIEWINFO (Spreadsheet, View handle, Int)

Purpose: To get information about a view

Parameters: Spreadsheet identifies the spreadsheet to which the view belongs
View handle is obtained from **NEWVIEW()** or **VIEWHANDLE()**
Int is a code to specify which attribute is needed:

Code	Attribute
0	The x scroll-bar position of the view.
1	The y scroll-bar position of the view.
2	The xO position of the view.
3	The yO position of the view.
4	The x1 position of the view.
5	The y1 position of the view.
6	The depth of the view.

Values for codes 0 to 5 are returned in OS-units. xO and yO refer to the lower left of the window, x1 and y1 to the upper right. The depth attribute returns a number. Lower values indicate that the view is higher up the stack of views on the screen.

Result: The value of the attribute, or a fault if the view does not exist or the attribute number is out of range.

Example: `res = GETVIEWINFO(!Bt.al..Q30, handle, 3)`

NEWVIEW (Spreadsheet, Pos1, Pos2, Pos3, Pos4, Pos5, Pos6)

Purpose: To set up a new view of a given spreadsheet

Parameters: Spreadsheet identifies the spreadsheet
Pos1 and Pos2 give the initial x and y slider bar positions in the new window
Pos3 and Pos4 give the x and y screen position of the lower left corner of the new window
Pos5 and Pos6 give the x and y screen position of the upper right corner of the new window
All positions are in OS-units.

If all the positions are zero a default new view position will be opened.

Result: The function returns the handle of the new view, or a fault if the view could not be opened.

Example: `res = NEWVIEW(!Bt.Al..Q30,0,0,50,50,600,500)`

VIEWHANDLE(Spreadsheet, Int)

Purpose: To find the handle of a view with a given number
Parameters: Spreadsheet identifies the spreadsheet to which the view belongs Int is a view number. If a spreadsheet has n views then they are numbered 0 to n-1. The ordering is arbitrary.
Result: A view handle of a view, or a fault if there is no view with the given view number

Example:

```
REM Find the highest view for a sheet
n_views = GETSHEETINFO(area, "views")
FOR i = 0 TO n_views-1
  REM Get the handle
  handle = VIEWHANDLE(area, i)
  this_depth = GETVIEWINFO(area, handle, 6)
  IF i = 0 OR this_depth < highest THEN
    highest = this_depth
    highest_view = handle
  ENDIF
NEXT
```

8.9 Style modifying functions

To apply a style to an area use the CHANGESTYLEQ function.

CLEAREFFECTS(Block)

Purpose: Clears all effects from the specified block
Parameter: Block gives the block to be cleared

Example: `res=CLEAREFFECTS(!Dolphin.G4..K19)`

DELETESTYLE(Spreadsheet, Style)

Purpose: To delete a style from a spreadsheet
Parameter: Spreadsheet is the spreadsheet from which the style is to be deleted. Style is the style to be deleted

Example: `res = DELETESTYLE(!Porpoise.Al..Al,"Heading")`

MODIFYSTYLE(Spreadsheet, Style, Effect, Descriptor)

Purpose: To modify a component of a style
Parameters: Spreadsheet identifies the spreadsheet where the style is held
 Style selects the style to be changed
 Effect is the name of the effect to be altered
 Descriptor is the new value of the effect

Example: `res=MODIFYSTYLE(!Dolphin.Al..Al,"Footnote","bgcolour",Ox88fiff00)`

NEWSTYLE (Spreadsheet, Style1 Style2)

Purpose: Sets up a new style, which is initially a copy of an existing style
Parameter: Spreadsheet is the spreadsheet to which the styles are attached
 Style1 is the style to be copied, Style2 the name of the new style

Example: `res = NEWSTYLE(!Porpoise.Al..Al,"Plain","Footnote")`
 creates a new style call Footnote in spreadsheet Porpoise which is a copy of Plain.

8.10 Print functions

COLPAGEBREAK(Block)

Purpose: To mark a **COLUMN** as the last one on a page
Parameter: Block identifies a column

Example: `res = COLPAGEBREAK(!Porpoise.J1..J30)`

FILEPRINT(Block, Path)

Purpose: To file a printable image of a given block, in accordance with the parameters set by **SETUPIMAGEPRINT** or **SETUPDRAFTPRINT**
Parameter: Block is the area to be printed, Path is the full path name of the file

Example: `res=FILEPRINT(!DoIphin.A1..S30,"ADFS::HDisc4.$.dump")`

PRINT(Block)

Purpose: To print a block in accordance with parameters set by **SETUPIMAGEPRINT** or **SETUPDRAFTPRINT**
Parameter: Block gives the area to be printed

Example: `res = PRINT (!Dolphin.G5..K20)`

ROWPAGEBREAK(Block)

Purpose: To mark a row as the last one on a page
Parameter: Block identifies a row

Example: `res = ROWPAGEBREAK(!Porpoise.A20..S20)`

SETUPDRAFTPRINT(Spreadsheet, Int1, Flag, Int2)

Purpose: To set up parameters for a print command in draft mode
Parameters: Spreadsheet is the spreadsheet to be printed
Int1 gives the number of copies
Flag is set to 1 for condensed printing
Int2 gives the maximum number of characters per line

Example: `res = SETUPDRAFTPRINT(!Dolphin.A1..A1,4,1,132)`

SETUPIMAGEPRINT (Spreadsheet, Int,, Int2, Flag1, Pos1, Pos2, Pos3, Pos4, Flag2, Flag3, Flag4)

Purpose: To set up parameters for a print command in image mode
Parameters: Spreadsheet is the spreadsheet to be printed
Int1 gives the number of copies
Int2 gives the percentage magnification (100 = full-size)
Flag1 is true if landscape printing is needed
Pos1 gives the top margin to be left blank on each page
Pos2 gives the bottom margin, Pos3 gives the left margin
Pos4 gives the right margin
All distances are in OS-units
Flag2 is set for horizontal centering
Flag3 is set for vertical centering
Flag4 is set for collated output

Example: `res=SETUPIMAGEPRINT(!PORPOISE.A1..A1,2,110,0,25,25,25,25,1,1,0)`

8.11 Input functions

These functions are defined in Chapter 3.

8.12 Functions to provide information and modify attributes

These functions use cell attributes and sheet attributes to enquire about and modify cells contents and spreadsheets. Cell attribute strings include the Effect strings listed near the beginning of this Chapter. The other strings and the descriptors that go with them are:

Border	Int	Border of cell as supplied to BORDERBITS() function.
Bottom	Distance	Coordinate of the cell bottom in OS-units
CellHeight	Distance	Height of the row in OS-units
CellWidth	Distance	Width of the column in OS-units
CustomFormat	String	
Expression	String	Formula for the cell.
Formatted	String	Cell value formatted according to the number format for the cell.
Left	Distance	Co-ordinate of the left hand of the cell
Protected	Int	-1 protected, 1 = not protected, 0 = protection disabled.
StyleName	StyleStyle	for the cell
X	Int	Column number of the cell (A=1)
XDim	Int	Number of cells horizontally taken up by this cell or the big cell which overlays it.
Xroot	Int	If cell is overlaid by a big cell then column number of big cell, otherwise column number of the cell itself
Y	Int	Row number of the cell
YDim	Int	Number of cells vertically taken up by this cell or the big cell which overlays it.
YRoot	Int	If cell is overlaid by a big cell then row number of the cell, otherwise the row number of the cell itself

Sheet attribute strings and the descriptors that go with them are:

Addresses	Flag	Cell address mode.
AngleMode	Flag	The spreadsheet's angle rule (radians=1)
Bottom	Int	Bottom row of the current block/cell in the sheet.
ECRMode	Flag	The spreadsheet's empty cell rule
EvalMode	Int	The spreadsheet's evaluation mode.
Formulas	Flag	Display cell formulas/values.
Grid	Flag	The spreadsheet's grid (l=on)
Labels	Flag	The spreadsheet's labels (l=on)
Left	Int	Left-most column of current block/cell in the sheet.
NameLock	Flag	If a sheet has this attribute, the leaf name under which it is saved is the same as its sheet name
Mag	Int	Scale sheet is shown at (percent)
PathName	String	Full path name under which the sheet is stored. The name of a new, unstored sheet is the empty string.
PrintBottomM	Distance	Bottom paper margin
PrintCollate	Flag	Print collated.
PrintCPL	Int	Characters per line in draft printing.
PrintDraft	Flag	Print in draft or normally
PrintHCentre	Flag	Print horizontally centred.
PrintLeftM	Distance	Left paper margin
PrintMag	Int	Print scale (percent)
PrintOrient	Flag	Print Orientation
PrintRightM	Distance	Right paper margin
PrintTopM	Distance	Top paper margin
PrintVCentre	Flag	Print vertically centred.
Right	Int	Right-most column of the current block/cell in sheet.
Security	Flag	The spreadsheet's security flag (l=on)
SelectPages	Flag	Show page breaks for selection
SheetName	String	The name of the sheet
Top	Int	Top row number of current block/cell in the sheet.
TimeMode	Flag	0= Schema, 1= Lotus
Views	Int	The number of open views on this sheet.
XSize	Int	The number of columns in the sheet (x- dimension)

YSize Int The number of rows (y-dimension)

CELL(Block, Cell attribute)

Purpose: To find out about different aspects of a particular cell.
Parameters: Block specifies the cell, if the block is bigger than one cell then the top left hand cell is taken.
 Cell attribute is one of the strings listed above.
Results: The function returns a value as requested.
Examples: font_used = CELL(!Sheet2.B2..B2, "FontName")
 block_starts = CELL(area, "X")

GETSHEETINFO(Spreadsheet, Sheet attribute)

Purpose: To find out about different aspects of a spreadsheet.
Parameters: Spreadsheet is the spreadsheet being enquired about.
 Sheet attribute is one of the strings listed above.
Results: The function returns a value requested.
Example: total_cells = GETSHEETINFO(sheet_block, "xsize") *
 GETSHEETINFO(sheet_block, "ysize")

MACROANGLE(Flag)

Purpose: To control interpretation of angles within macros.
Parameter: Flag if 0 then interpret angles as degrees, otherwise as radians.
Result: All angles are interpreted as defined by the Flag until the current macro finishes executing.
 In general the interpretation of angles in macros depends on the spreadsheet from which the macro was called. This enables macros such as:

```
MACRO Sec(x)
=1/COS(x)
ENDMACRO
```

to work both in spreadsheets with angles in degrees and with angles in radians. However you want to put in a constant angle (as is common in graphing macros) then the macro should behave the same from whatever sheet it is called. Thus:

```
MACRO draw_segment(graph, x, y)
res = MACROANGLE(0)
res = ARC(graph, x, y, 45, 90, 160, -1)
```

will always draw an arc from 45 degrees to 90 degrees. If data is read in by a macro from a sheet which may be in radians or degrees the following procedure may be adopted:

```
MACRO Process_data(block)
LOCAL res, flag, value
res = MACROANGLE(1)
REM Read our data
value = block(0,0)
flag = GETSHEETINFO(block, 0)
IF flag =0 THEN
value = RAD(value)
ENDIF
REM Now process it
```

SETSHEETINFO(Spreadsheet, Sheet attribute, Descriptor)

Purpose: To modify many different aspects of a spreadsheet.

Parameters: Spreadsheet is the spreadsheet being modified.

Sheet attribute is one of the following Sheet attribute strings:

Addresses	AngleMode	ECRMode	EvalMode	Formulas
Grid	Labels	Mag	Namelock	PrintBottom
PrintCollate	PrintCpl	PrintDraft	PrintHCentre	PrintLeftM
PrintMag	PrintOrient	PrintRightM	PrintTopM	PrintVCentre
Security	SelectPages	TimeMode		

Descriptor is a value appropriate to the Sheet attribute which becomes the new value for that attribute of the spreadsheet.

Results: Changing some attributes will result in the whole spreadsheet being re-evaluated.

Example: The return value is 0 or an error.

```
new_sheet = CREATESHEET(" Results", 30,40,1)
REM Now make spreadsheet have no labels or grid
REM and use radians for angles
res ^ SETSHEETINFO(new_sheet, "Labels", 0)
res = SETSHEETINFO(new_sheet, "Grid", 0)
res = SETSHEETINFO(new_sheet_block, "anglemode",1)
```

SHEETNAME(Spreadsheet)

Purpose: To give the name of a spreadsheet.

Parameter: Spreadsheet is a block in a spreadsheet.

Result: Returns a string giving the name of the spreadsheet.

Example: name = SHEETNAME(area)

Comment: A call to this function has the same effect as

```
name = GETSHEETINFO(area, "Sheetname")
```

WHOLESHEET(String or Spreadsheet)

Purpose: To give a block referring to the entire spreadsheet.

Parameter: The parameter is either a sheet name as a string or a block in the spreadsheet. The parameter may also be an empty string.

Result: Returns a block referring to the whole of the spreadsheet. This block can be used to reference all the cells. If the string is empty the result refers to the currently selected sheet.

Examples:

```
MACRO GetValue(name, x, y)
REM To find the value of 'name.xy
local whole
whole = WHOLESHEET(name)
cell_xy = whole(y,x)
REM To set all rows to be half an inch high
whole = WHOLESHEET(area)
res = CHANGEWIDTH(whole, 90)
```

CHANGENAME(Sheet,String)

Purpose: To change the name of a sheet
Parameters: Sheet identifies the sheet to be changed
String is the new name
Result: 0 if the change took place successfully

Example:

```
MACRO CHANGENAME(z)
REM Changes name of selected sheet to
ZLOCAL qq
qq = WHOLE SHEET(" ")
= CHANGENAME(qq,z)
ENDMACRO
```

CHANGESELECTION(Block, Flag)

Purpose: To change the current selection
Parameters: Block is the new selection
If flag is set to 2 the sheet is automatically scrolled so that the selection comes into view.

8.13 File functions

The following functions permit macros to open files and to read and write data in various formats. The chief purpose of the file functions is to provide macros with data which cannot be easily computed - e.g. certain mathematical tables or numerical descriptions of coastlines and other geographical regions.

When a file is opened successfully, the system returns an integer handle which is then used in all further references to that file. The file support system allows access to five files simultaneously.

Files can be read and written at either of two 'levels': character by character or item by item, where each item is either a number or a string. It is not generally useful to mix the two levels.

Files are treated as serial; there is no 'random access' mechanism.

Note it is often easier with small files to load them as TSV or CSV files and manipulate them in a spreadsheet, saving them likewise.

Items in files in the **READ()** and **WRITE()** correspond to values in CSV files. For a description of CSV files see Appendix D of the main manual.

CLOSE(File handle)

Purpose: Closes a file, flushing any buffered data
Parameters: File handle is a handle obtained from the **OPEN()** function.
Result: This function closes the file with the given handle and frees the handle. When this function has been executed the handle is meaningless and should not be used any more. If another file is opened it may be allocated the same handle.

Example: `res = CLOSE(handle)`

OPEN(Path, Flag)

Purpose: Start using a file and obtain a handle for it
Parameters: Path is the name of the file
Flag is either 0 in which case the file is opened for reading, or 1 in which case it is opened for writing.
Result: If successful the file is opened and a handle to that file is returned. This handle should be used by subsequent **READ()**, **READCH()**, **WRITE()** and **WRITECH()** functions. The file should be closed with the **CLOSE()** function. If unsuccessful an error is returned.

Example: `handle = OPEN(" ADFS: :Floppy2.$.Mydata" ,1)`

READ(File handle)

Purpose: Read an item from a file
Parameters: File handle is a handle obtained from the **OPEN()** function.
Result: This function reads an item up to the next comma or new line from the file. If the end of file is reached it returns an error.

8.14 Miscellaneous functions

MODIFIEDSHEETSO

Purpose: To determine how many spreadsheets are modified and unsaved.
Result: Returns the number of modified spreadsheets.

Example:

```
REM Now quit after checking
n = MODIFIEDSHEETSO
IF n>0 THEN
reply=QUERY("Some spreadsheets not saved","Cancel","Quit")
IF reply = 0 THEN n = 0
ENDIF
IF n=0 THEN res = QUITPROGRAM(0)
```

QUITPROGRAM(Int)

Purpose: To quit Schema2.
Parameter: Int is a value returned to RISC OS.
Result: Schema2 terminates execution. No check is done on whether there is unsaved data (use **MODIFIEDSHEETSO** to check for that). Use this function with caution.

Example: res = QUITPROGRAM(0)

SELECTSHEET(String)

Purpose: Change the default spreadsheet.
Parameter: String is the leaf name of a spreadsheet.
Result: Subsequent references within the macro to cells without an explicit reference will be taken to be from this spreadsheet. This function is to provide compatibility with original Schema macros.

SORT(Block, Flag, Int1, Int2, Int3, Int4, Int5, Int6, Int7, Int8, Int9, Int10)

Purpose: To sort a block in a spreadsheet.
Parameters: Block is a block to be sorted. Local arrays cannot be sorted by this function, use the sorting macro given in chapter 4 instead.
Flag is 0 for sorting by row, 1 for by column.
Int1 is the row or column number of the first sort key field in the block. It refers to a column if sorting is by row and a row if sorting is by column.
Int2 is a number which determines how the sorting on the first key field is to be done:

Value	Meaning
2	Key fields are strings, sort case-insensitive, descending
3	Key fields are strings, sort case-insensitive, ascending
4	Key fields are strings, sort case-sensitive, descending
5	Key fields are strings, sort case-sensitive, ascending
6	Key fields are numbers, sort descending
7	Key fields are numbers, sort ascending

Int3 is the index of the second sort key. If there is only one sort key it is 0, otherwise it is like Int .
Int4 determines how sorting on the second key is to be done. If there is only one sort key it is 0, otherwise it is like Int1
Int5 and Int6 refer to the third sort key.
Int7 and Int8 refer to the fourth sort key.
Int9 and Int10 refer to the fifth sort key.

Result: The block is sorted according to the parameters and re-evaluated.

Example: SORT(!Sheet2.B2..E10,1,5,7,2,3,3,8,4,8,0,0)

Sort the block B2..E10: first sort key is column E, descending numbers; second sort key is column B, descending case-insensitive strings; third sort key is column C, ascending numbers; the fourth and final sort key is column D, ascending numbers.

SYS(String, Array1, Array 2)

Purpose: To access SWIs.

Parameter: String is the name of the SWI.

Array1 is a column array of up to 15 elements giving the parameters of the SWI. Element 0 corresponds to register R0, element 1 to R1 etc. If the value is a number then an integer is put in the register. If the value is a string then a pointer to it is put in the register. If the value is a fault or anything else the function gives a fault result.

Array2 is a column array of up to 16 elements into which the results of the SWI are copied. The function interprets the value according to the type of the item that was in the corresponding element of Array2. If it was a number then the register is assumed to be an integer. If it was a string then the register is assumed to be a pointer to a string which is copied into Schema2's memory. If it was anything else an error is reported.

Result: Calls the SWI and fills Array2 with the results unless there was an error. Note there is no way to provide a buffer for SWIs to return a value in; the SWI should not modify any strings that it is passed as parameters. Users should be aware that this function can have many side effects (including crashing the machine).

Example:

```
MACRO FreeMemory
LOCAL in(2), out(3)
REM Set up arrays
in(0) = (-1)
in(1) = (-1)
out(0) = 0
out(1) = 0
out(2) = 0
res = SYS("Wimp_SlotSize", in, out)
= out(2)
```

Index

A

ADDCOLUMNS(Block, Int)	75
ADDROWS(Block, Int)	75
Assignment	64

B

Background	22
BIGCELL(Block, Flag)	77
Boot macros	18
BORDERBITS(Block, Int1, Flag, Int2)	77

C

Calling a macro from the icon ribbon	18
Calls from macros	18
Calls from the main menu	18
Cell names	61
CELL(Block, Cell attribute)	84
CHANGEHEIGHT(Block, Distance)	78
CHANGENAME(Sheet,String)	86
CHANGESELECTION(Block, Flag)	86
CHANGESTYLE(Block, Style)	78
CHANGEWIDTH(Block, Distance)	78
Checking and Testing	16
CLEARAFFECTS(Block)	81
CLOSE(File handle)	86
COLPAGEBREAK(Block)	82
Comments	59
Constants	60
COPY(Block1, Block2, Int1, Int2)	76
COPYCLIP(Block)	76
COPYVALS(Block1, Block2, Int1 Int2)	76
CREATESHEET (Name, Inti, Inta, Flag)	75
CUTCLIP(Block)	76

D

DELETE(Block)	76
DELETECOLUMNS(Block)	75
DELETEROWS(Block)	75
DELETESHEET(Spreadsheet)	75
DELETESTYLE(Spreadsheet, Style)	81
DELETEVIEW (Spreadsheet, View handle)	80

E

ERRORCODE(value)	74
Evaluation level one	70
Evaluation level two	70
Evaluation level zero	70

F

FILEPRINT(Block, Path)	82
FIRST (Array)	73
Foreground	22
Format	22

G

GETSHEETINFO	
(Spreadsheet, Sheet attribute)	84
GETVIEWINFO	
(Spreadsheet, View handle, Int)	80
Goto (F5)	22
Graphic handles	61
Graphics macro parameters	50
Graphics macros	18
Groups	45

H

HIDECOLS(Block)	78
HIDEROWS(Block)	78

I

Inclusion in cell formulae	17
Indent (^F4)	22
INPUT	27
INPUTBLOCK(Block,Flag)	79
ISERROR(parameter)	74
Iteration	65

L

Labels and GOTOs	66
LOAD(Path)	79
Local Declarations	63
Local variables	60

M

MACROANGLE(Flag)	84
MacroEdit modes	15
MODIFIEDSHEETSO	87
MODIFYSTYLE	
(Spreadsheet, Style, Effect, Descriptor)	81
MOVE(Block1, Block2)	76

N

Names	59
NEWSTYLE (Spreadsheet, Style1 Style2)	81
NEWVIEW (Spreadsheet, Pos1, Pos2, Pos3, Pos4, Pos5, Pos6)	80

O

ON ERROR CONTINUE	68
ON_ERROR_EXIT	67
OPEN(Path, Flag)	86

P

Parameters	60, 72
PARTLOAD (Block, Path)	79
PARTSAVEFILE (Datablock, Path, File type)	79
PASTECLIP (Block, Flag1, Flag2, Flag3)	77
Path drawing functions	36
Possible Sources of error	67
PRINT(Block)	82
PUTBLOCK(String,Block)	77

Q

Q = FORMAT(value, format)	39
Q = PSSTEP(u)	53
Q = TEXTHEIGHT(w,text)	39
Q = TEXTWIDTH(w,text)	39
QUERY 26	
QUITPROGRAM(Int)	87

R

READ(File handle)	87
REPLICATECLIP (Block, Int1 Int2)	77
REPORTERROR(stringi, stringy int)	75
Res = BOX(w,x,y,width,height,colour)	33
Res = DRAWOBJECT(w, path, elements, fill colour)	37
Res = FONT(w, " font-name", height, width)	31
Res = GROUP(w), res = ENDGROUP(w)	40
Res = LINE(w,x,y,dx,dy)	33
Res = LINEATTRIBUTES (w, style, width, colour, path style)	31
Res = OPENPOLY(w,points,number)	34
Res = PICTURE(w,q,x,y)	35
Res = POLY(w,points,number,colour)	33
Res = POLYMARKER(w, points, number, type, size, colour)	34
Res = SAVEGRAPH(w, "Name")	31
Res = SQUEEZEDPICTURE(w, q, x, y, width, height)	36
Res = STRING(w,x,y,text,fore,back)	35
Res = TCO(w,x,y,width,height,text,colour)	39
Res =ARC(w,cx,cy,start_angle,end_angle, r,colour)	38
Res =TEXT(w,x,y,text,colour) (deprecated; use STRING)	39
Res =TEXTCOLUMNOBJECT(w, x, y, width, height, text, fore, back)	38
Reserved words	59
Results	73
Returning a graphic object	51
REVEALCOLS (Spreadsheet)	78
REVEALROWS (Spreadsheet)	78
ROWPAGEBREAK(Block)	82

S

Save as default	22
SAVEFILE (Spreadsheet, Path, File type)	79
SAVEWS (Path)	80
Search (F4)	20
SECOND (Array)	73
Selection	64
SELECTSHEET(String)	87
SETEFFECT(Block, Effect, Descriptor)	79
SETSHEETINFO(Spreadsheet, Sheet attribute, Descriptor) 85	
SETUPDRAFTPRINT(Spreadsheet, Int1, Flag, Int2)	82
SETUPIMAGEPRINT (Spreadsheet, Int,, Int2, Flag1, Pos1, Pos2, Pos3, Pos44, Flag2, Flag3, Flag4)	82
SHEETNAME(Spreadsheet)	85
SORT(Block, Flag, Int1, Int2, Int3, Int4, Int5, Int6, Int7, Int8, Int9, Int10)	87
Spaces	59
SYS(String, Array1, Array 2)	88

T

The Descriptor line	62
The Drawing functions	29
The Edit menu	20
The Edit support functions	29
The File menu structure	20
The graph description vector	52
The GRAPHMACRO line	50
The Header line	62
The RETURN statement	67
The Search Dialogue	21
The Text found dialogue	22
The Utilities Menu	20
TYPE(parameter)	74

U

U = PSREGISTER(v)	53
Using MacroEdit	15

V

VIEWHANDLE(Spreadsheet, Int)	81
------------------------------	----

W

W = FETCHGRAPH("identifier")	30
W = GSTART(x,y) Open a graph	30
W=GRAPHLOAD("name")	30
WHOLESHEET(String or Spreadsheet)	85
WIPE(Block)	77

Z

Z = PSCONTAINS(u, s)	53
Z = PSEXDEC(u, s, default)	53
Z = PSEXHEX(u, s, default)	53
Z = PSEXSTR(w, s, "default")	54
Z = PSEXSUB(u)	53

Here are just a few of the other products you will be able to
download from RISCOS.com free of charge
(You can make a donation to help support future product releases but this is entirely voluntary)



This Documentation is © 3QD Developments Ltd 2026
Some of the products shown above might not be available immediately.
Not all products will run on all versions of RISC OS, check the product information page for details.

Other Products list last updated 29/2/26