sinclair

QL

User Guide

Introduction
Beginner's Guide
Reference Guide
- Keywords
- Concepts

Applications Software
- QL Quill
- QL Abacus
- QL Archive
- QL Easel

Information

FOR SERVICE MANUALS
CONTACT:
MAURITRON TECHNICAL SERVICES
www.mauritron.co.uk
TEL: 01844 - 351694
FAX: 01844 - 352554

sinclair

QL

User Guide

PLEASE READ THIS BEFORE UNPACKING THESE PAGES

Your QL User Guide is supplied unbound, to avoid damage in transit and to make rapid updating easy. In addition to this packet containing the pages of the Guide itself, you should also find a ring binder and then divider cards packed with your QL.

Insert the dividers into the binder first. The recommended order is as follows:

Position	Tab Label
Front	Introduction
	Beginniner's Guide
	Keywords
	Concepts
	QL Quill
	QL Abacus
	QL Archive
	QL Easel
Back	Information

1367

This will put the divider tabs in a logical order. If you wish, you may put the sections in a different order, perhaps to put often used sections near the front; or even miss out sections you do not expect to use.

Now look through the pages to identify the various sections; each begins with a title page with the Sinclair logo at the top. The pages within each section will be packed in the correct order, so be careful not to mix them up; the individual sections, however, may be in a different order to that shown above if a section or sections have recently been reprinted.

Once each section is placed in the binder as you like it, this sheet may be discarded; it does not form part of the Guide.

Sinclair Research has a policy of constant development and improvement of their products. Therefore, the right is reserved to change manuals, hardware, software and firmware at any time and without notice.

FOR SERVICE MANUALS
CONTACT:
MAURITRON TECHNICAL SERVICES
www.mauritron.co.uk
TEL: 01844 - 351694
FAX: 01844 - 352554

QL User Guide Second Edition
Published by Sinclair Research Limited 1984
25 Willis Road, Cambridge.
Edited by Stephen Berry (Sinclair Research Limited)

©Sinclair Research Limited
©Psion Limited

Printed and bound in Great Britain by
William Clowes Limited, Beccles and London

Designed and typeset by
Keywords, Manchester

No part of this User Guide may be reproduced in any form whatsoever without the written permission of Sinclair Research Limited.

QL, QLUB, QL Net, Qdos and **QL Microdrive** are trade marks of Sinclair Research Limited.

Quill, Archive, Easel and **Abacus** are trade marks of Psion Limited

sinclair

QL

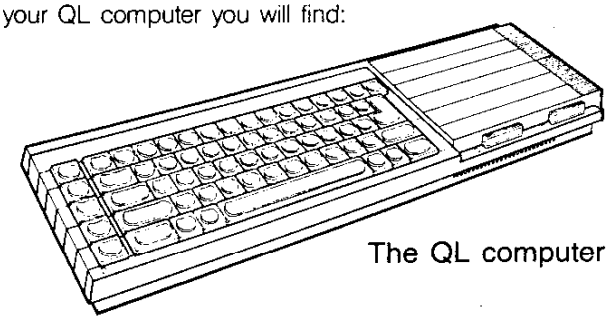
Introduction

FOR SERVICE MANUALS
CONTACT:
MAURITRON TECHNICAL SERVICES
www.mauritron.co.uk
TEL: 01844 - 351694
FAX: 01844 - 352554

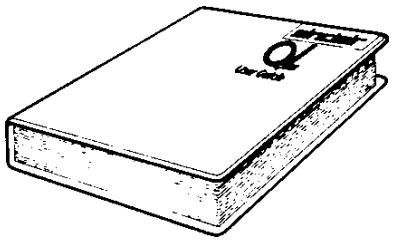
©1984 SINCLAIR RESEARCH LIMITED

INTRODUCTION TO THE QL

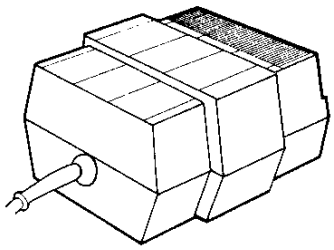
When you unpack your QL computer you will find:



The QL computer



The QL User Guide

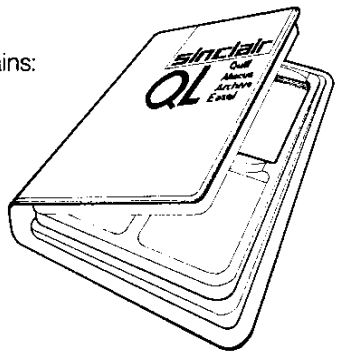


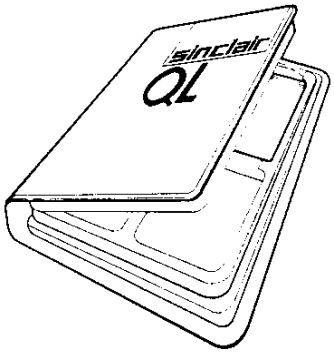
A power supply

Two wallets

one of which contains:

- QL Abacus
- QL Archive
- QL Easel
- QL Quill

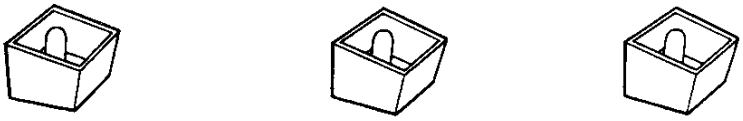




and the other contains:

four blank QL Microdrive cartridges.

Three plastic feet



these can be fitted under the QL to tilt the keyboard for more comfortable typing. The pips in the top of the legs should be fitted into the holes in the rubber feet, twisting them to make them fit securely.

An aerial lead

about two metres long with different connectors at either end. It is used for connecting your QL to your television's aerial socket.

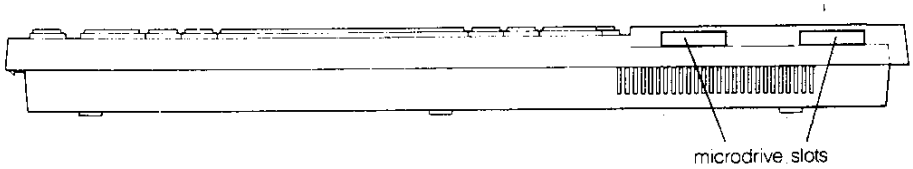
A network lead

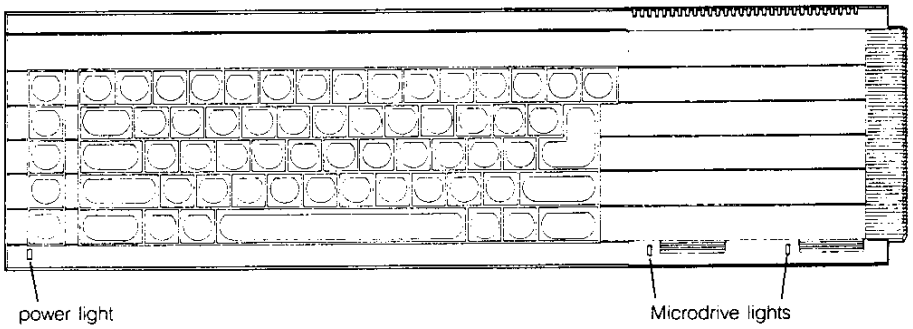
also about two metres long, with identical connectors at either end. It is used to connect your QL to other QLs so that data and messages can be sent between them.

A GUIDED TOUR

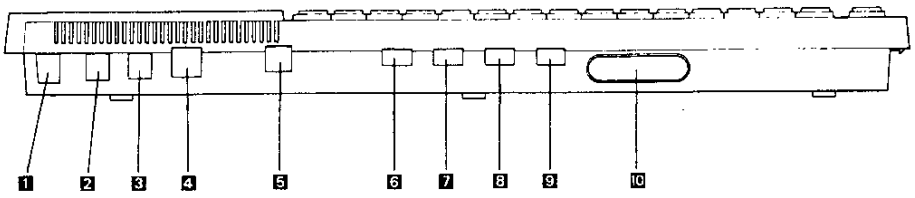
On the back and sides of the computer there are a series of connectors.

There are two slots on the right hand side of the computer - the two QL Microdrives. The cartridges for these Microdrives are used for storing programs and data on the QL. Next to each slot there is a small light. When the light is on the Microdrive is in use and the cartridge should not be removed. The yellow light on the front lefthand side indicates whether the QL is switched on.





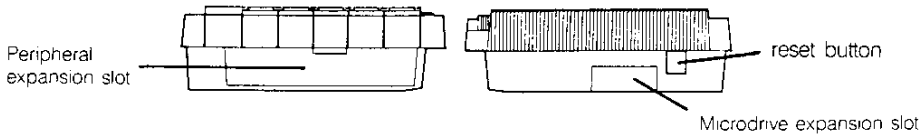
On the right-hand end of the QL there is a slot covered by a plastic strip. This slot is for attaching up to six more QL Microdrives. ZX Microdrives are not suitable for use with the QL but blank Microdrive cartridges can be used on either machine.



The connectors at the back of the computer are for attaching the following:

NET	– connector for the QL Network
NET	– connector for the QL Network
POWER	– power supply for the computer
RGB	– connection to a monochrome or colour monitor
UHF	– connection to the aerial socket of a television set
SER1	– RS-232-C serial port
SER2	– RS-232-C serial port
CTL1	– control port for joystick
CTL2	– control port for joystick
ROM	– QL ROM cartridge software (*use reversed one to 10*)

ZX ROM cartridges are not compatible with QL ROM cartridges and cannot be used in the QL.



The slot on the left-hand side of the QL is used for adding peripherals (equipment to expand the computer's capabilities) to the QL. One peripheral can be plugged directly into the expansion slot.

The reset button is on the right-hand end of the computer near the Microdrive expansion slot. It is used to 'reset' the QL to its original 'switch on' state. Any programs in the machine will be lost if reset is pressed and sometimes data already recorded on Microdrive cartridges can be corrupted. **Use reset with caution and always remove Microdrive cartridges before doing so.**

SETTING UP

To make the computer operate, various connections have to be made:

THE POWER SUPPLY

Your QL power supply has two leads. One is fitted with a small rectangular connector with three holes in it. The other is the mains lead and is supplied with bare ends to which a suitable mains plug must be fitted.

Please do not connect the power supply lead to the computer until all other leads and peripherals have been connected. Always connect the power supply lead to the computer last of all.

Connect the mains plug as follows:

- The blue wire goes to the terminal marked N or neutral, or coloured blue or black.
- The brown wire goes to the terminal marked L or live and coloured brown or red.
- The power supply is double insulated and does not need an earth connection.
- If you are using a fused plug, it must be fitted with a three amp fuse.
- Make sure all connections are sound.

If necessary, get someone with electrical experience to help you.

THE DISPLAY

Although the QL will work once the power supply is connected, you will not be able to see what it is doing until you add a television set or a monitor.

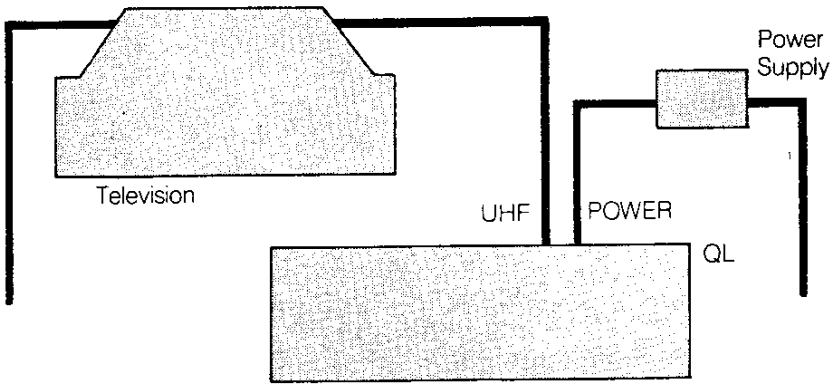
A monitor has a screen like a television, but it cannot receive television signals. It usually has better resolution than a television set and so can display more text and is therefore more expensive.

A colour television or monitor will of course be required to make use of the QL's colour display, but the computer will work perfectly well in black and white, representing colours as shades of grey.

Most television sets in current use will be suitable for the QL provided they are able to receive 625 line UHF transmissions, i.e. BBC2 and Channel 4.

Locate the aerial socket at the back of your TV and remove the aerial cable that may be plugged into it. If your set has more than one socket, use the one labelled UHF or 625. Plug in the QL's aerial lead. Use the end that looks similar to the original aerial plug, and plug the other end into the socket marked UHF on the back of the computer.

Plug the power supply into a mains socket and switch on. Remove any cartridges from the Microdrive slots and push the small power supply connector into the three pin plug marked POWER on the back of the QL. The yellow power light below the F5 key should now be glowing and your set up should look like this:

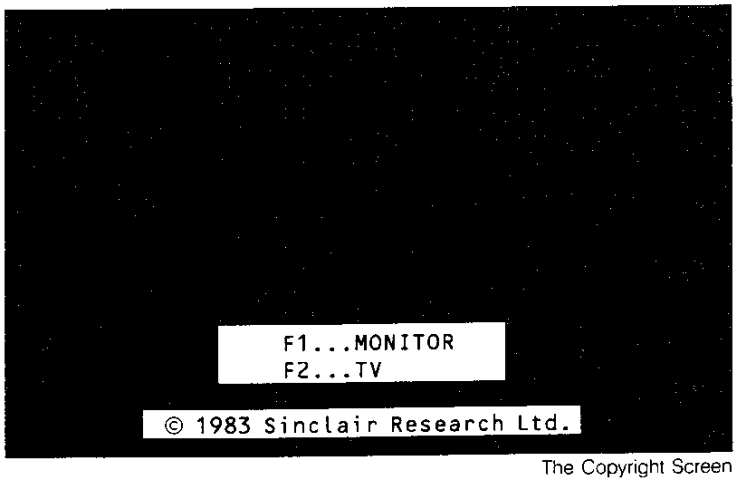


When the computer has been on for a while, the case above the Microdrives will feel warm; this is quite normal. The QL has no on/off switch but can be turned off by unplugging the power supply connector. **Remember that any program or data in the machine will be lost when it is turned off and should first be saved on a Microdrive cartridge** (for details of how to do this see the *Beginner's Guide* and *Concept* sections). If the QL is not going to be used for a while you should also switch the power supply off at the mains.

TUNING IN

The display signal to the television set is near channel 36. If your set has continuous tuning, tune to channel 36. If your television has push buttons, choose an unused button and tune this to the computer's signal. You may need to consult your dealer or the TV instruction manual to find out how to do this.

Once you are correctly tuned in you should see the copyright screen.



The Copyright Screen

The QL doesn't use television sound because it has its own internal loudspeaker. You can turn the television volume down if you wish.

A coloured pattern will appear after you switch on or reset the computer; this is the QL testing its memory. The pattern will disappear after a few seconds to be replaced by the copyright screen.

If you cannot get a picture at all, first check that your television can receive the normal broadcast stations. If it can then try the computer with another television set.

If you get a fuzzy or indistinct picture check that you are tuned in correctly; it may be possible to pick up the computer's signal in more than one place in the tuning range. Also check that the aerial lead is firmly plugged in, and that you are using the correct socket on your television set (if it has more than one).

If you wish to use a monitor instead of a television set, the connections will depend on whether it is colour or monochrome; details can be found in the *Concepts* section under the heading *Monitor*. A monitor lead with a plug to fit the QL's RGB socket is available from Sinclair Research. The order form is in the *Information* section of this guide.

The QL needs to know if you are using a monitor or a television set. Press

[F1] for a monitor

or

[F2] for a television.

Microdrive 1 will run briefly and the red Microdrive light will glow; the QL is looking for programs to load and run (this can be ignored for now). The computer will start up and display its cursor, a flashing coloured square, and the computer is now ready to accept commands.

USING THE QL

KEYBOARD

Unlike previous Sinclair computers there is no single keyword entry on the QL. However, various keys and groups of keys have special meanings:

Enter

The **ENTER** key is used to indicate to the computer that you want it to do something. Perhaps you have typed in a command and want the computer to execute it, or you may want to tell the computer that you have finished typing in data.

Shift

The keyboard has two **SHIFT** keys which perform the same function. Pressing **SHIFT** and an alphabetic key together will generate capital letters (upper case characters). On non-alphabetic keys **SHIFT** will cause the upper engraved character to be generated. For example:

[SHIFT] & [5] will give %

Caps Lock

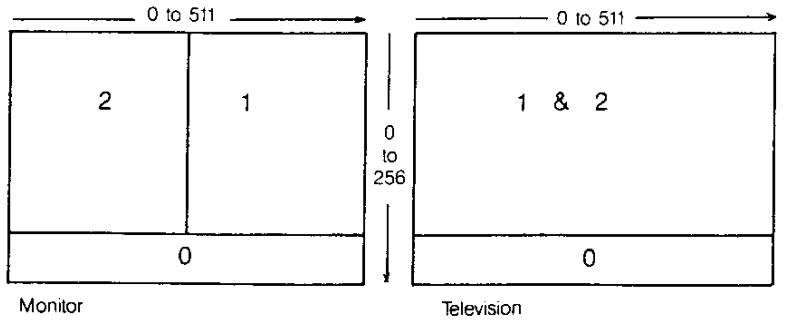
Pressing the **CAPS LOCK** key once will force alphabetic keys to generate capital letters regardless of whether the **SHIFT** key is pressed. This will remain in effect until **CAPS LOCK** is pressed again.

Delete

Hold down the **CTRL** key and then press the [←] key. The character to the left of the cursor will disappear and the cursor will move to the left. Hold down **CTRL** and press the [→] key. The cursor will not move; the character it was on will disappear and text to the right will move to fill the gap.

THE SCREEN

The QL screen may be divided into different areas, or windows, at will. Once you have switched on (or reset) and pressd **F1** or **F2**, the screen will look like this:



The long thin window at the bottom is used to display commands typed into the computer and initially will display the flashing cursor. When the cursor is visible the QL is ready to accept commands or data; it disappears when the computer is busy. As you type, the cursor will move along the line showing where the next character to be typed will appear.

If the machine ever fails to respond correctly or you want to force a SuperBASIC program to stop, hold down the **CTRL** key and press the space bar.

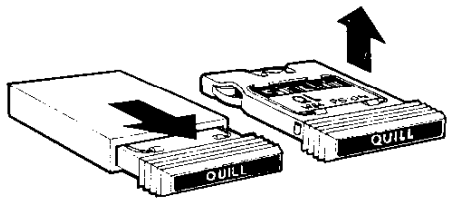
The computer should then display its cursor. If this doesn't work remove any Microdrive cartridges and then press reset.

The message **Bad Line** appearing in the command window means that the computer doesn't understand a command that you have typed in. Delete or correct the line using the cursor keys.

MICRODRIVES

The two QL Microdrives are called **mdv1_** on the left and **mdv2_** on the right.

Cartridges must be placed correctly into the Microdrives. Hold the cartridge by the ribbed plastic handle and remove it from its protective cover. The cartridge's name label, or the recess for its stick-on label, should face upwards.



Cartridges should always be treated with care. You should never turn the QL on or off with a cartridge in the Microdrives. Take care when inserting or removing cartridges; wait until the Microdrive lights have gone out before removing the cartridge, be gentle but firm. Never touch the tape in the cartridge and always return the cartridge to its protective cover.

Before a blank cartridge can be used it must go through a process called formatting. **This process erases any data or programs on a cartridge so always be sure that all cartridges are clearly labelled with their contents and check that cartridges to be formatted contain no useful data.** Instructions for formatting cartridges are contained in the *Information* section.

All magnetic storage media including Microdrive cartridges eventually suffer from wear. **Hence it is strongly recommended that all important programs and data should be stored on at least two cartridges, that is 'backed up'.** This means that if a cartridge is damaged and the data lost then at least part of the data can be recovered from the relevant back up cartridge. If you are continually adding data to a cartridge it must be backed up often; unless you do so you will lose everything that was added since the last backup if the main cartridge is damaged. Instructions for backing up cartridges are contained in the *Information* section.

STARTING WORK

There are several ways of using your computer and the User Guide. You can use ready made programs such as those supplied with the QL, or you can write your own programs in SuperBASIC.

To use the QL programs, first read the *Introduction to the QL Programs* later in this introduction and then the relevant section for each program concerned.

If you are a newcomer to computing and wish to write your own programs, you should read the *Beginner's Guide*. If you are familiar with BASIC programming, you may prefer to read from Chapter 8 in the *Beginner's Guide – From BASIC to SuperBASIC*. This chapter describes the major differences between BASICs you may already be familiar with and QL SuperBASIC. Alternatively, if you are feeling confident, the *Keywords* and *Concepts* sections should be useful.

IF YOU HAVE A PROBLEM

If you have a problem using your QL or QL programs, then:

1 Refer to the appropriate sections in the QL User Guide.

2 Consider joining the QL Users' Bureau for assistance on the QL programs. Full details of the services offered by QLUB and instructions for joining are contained in the *Information* section of the QL User Guide under the heading *QLUB*.

3 Refer to books published about the QL.

If your problems persist and you think they may be caused by a fault in either your QL or in the QL program cartridges then refer to the Guarantee details in the *Information* section of the QL User Guide.

FOR SERVICE MANUALS
CONTACT:
MAURITRON TECHNICAL SERVICES
www.mauritron.co.uk
TEL: 01844 - 351694
FAX: 01844 - 352554

INTRODUCTION TO THE QL PROGRAMS

This introduction outlines the four programs supplied with the QL and describes their common features.

The four programs are:

QL Quill – a wordprocessor
QL Abacus – A spreadsheet
QL Archive – a database
QL Easel – a graphics program

Individual sections in this guide describe each of the four programs in detail. Don't just read them – try out the examples and experiment with each new idea.

MICRODRIVES

Before you use any of the QL programs you should make at least one backup on a blank cartridge and use this copy only. Keep the original program cartridge in a safe place, and use it only for making copies. Any accidents will not then cause permanent loss of your programs.

Each QL program has a built in duplicating routine which is used as follows:

- Place the master cartridge in Microdrive 2
- Place the blank cartridge, or one containing nothing that you wish to keep, in Microdrive 1. Type

```
lrun mdv2_clone
```

- Press the **ENTER** key and the screen will display the message

```
FORMAT mdv1_ type space to continue
```

- Press the space bar only when you are sure that the cartridge contains nothing that you wish to keep, as everything on it will be erased. The computer will format the cartridge and will then copy the program in sections, displaying the name of each one as it does so.
- Wait until the Microdrive lights go out before removing the master cartridge from Microdrive 1.

LOADING

You should never use any of the original program cartridges except when making a copy onto a blank cartridge.

All the programs are loaded similarly. There are two ways of doing this:

Without cartridges in the Microdrives, press reset. Place your copy of the program cartridge in Microdrive 1, and then press either **F1** or **F2** as prompted. Microdrive 1 will automatically run and after a short pause a title display will appear on the screen to confirm that the program is being loaded. Once the program is loaded into the computer the program will start up by itself.

When you become more familiar with the programs and when using a printer or the network, you will sometimes find that commands need to be given to the computer before the programs start. You cannot switch off or reset the computer in this instance because your commands would be lost. Instead place the program cartridge in Microdrive 1 and type

```
lrun mdv1_boot
```

press **ENTER** and loading will proceed as before.

In both cases the program will occasionally need to load extra information from the Microdrive so keep the program cartridge in the Microdrive slot until the program has finished.

SCREEN LAYOUT

The *control area* at the top of the screen will guide you through each program by displaying the options that you will need most often and prompting you further if necessary. In many cases the program will suggest a suitable answer when it asks for information. Press **ENTER** to accept this suggestion or simply type in your own answer and the computer's suggestion will disappear.

Pressing F2 will remove this area and will make the central area larger. Pressing F2 again will restore the control area.

The central area of the screen shows the information that you are working on, for example, the text of a document, the contents of a card index, a graph, or financial forecast. It is shown in the style most suitable for the particular application.

The bottom of the screen shows the input line where, for example, commands that you type in are displayed.

Below this is the *status area* which reports on the current state of work. It displays things like the name of the data or document on which you are working, how much unused memory remains, etc.

FUNCTION KEYS

Three of the five function keys have the same meaning in all the QL programs. These are:

Key	Function
F1	request help
F2	remove or restore the control area
F3	call up the commands for selection

The remaining two function keys are used for actions particular to each program.

HELP

The first option, displayed at the top left of the control area, indicates that help is available by pressing F1.

When you ask for **HELP** there will be a short pause before the display changes to show the Help information.

Help will suggest other topics for which help is available. Type the name of the topic and press **ENTER**. **You do not need to type in the whole name, just enough characters for it to be distinguished from the other topics. You can repeat this as many times as necessary.**

Pressing **ENTER** without selecting a topic will take you out to the previous level. **ESC** will take you right out of **HELP** and back into the program.

Help is always available, provided that the program cartridge is in Microdrive 1. Press F1 and the most appropriate Help information will be displayed.

THE LINE EDITOR

You can use the line editor to change or correct a line of text that you have typed in. All the QL programs use the same line editor, but each program uses it in a way most suitable for that application. In QL Qill you use the line editor, for example, for editing the text in commands and QL Archive uses the editor extensively for editing database programs.

The line editor uses the four cursor keys, together with the **CTRL** and **SHIFT** keys.

Keys	Action
←	Move the cursor one character to the left
→	Move the cursor one character to the right
SHIFT & ←	Move the cursor one word to the left
SHIFT & →	Move the cursor one word to the right
CTRL & ←	Delete the character to the left of the cursor
CTRL & →	Delete the character under the cursor
CTRL & ↑	Delete the line to the left of the cursor
CTRL & ↓	Delete the line to the right of the cursor
SHIFT & CTRL & ←	Delete the word to the left of the cursor
SHIFT & CTRL & →	Delete the word to the right of the cursor

The & symbol indicates that the first key should be held down while the second is pressed. When **SHIFT** and **CTRL** are used together then hold them both down before pressing the cursor key.

MICRODRIVE USE

The program is loaded from the cartridge in Microdrive. You must always make sure that before using **Help** or using a print command that this cartridge is in Microdrive 1. Otherwise you can remove the cartridge at any time.

Use a cartridge in Microdrive 2 – and in additional Microdrives – for storing information, for example, Quill documents, Archive data files, etc.

FILE NAMES

Information can be stored on a cartridge in a 'file'. The file must be given a file name to distinguish it from others on the cartridge. Use a file name of not more than eight characters long, without spaces. It is a good idea to use a name which describes the contents of a file; for instance, 'sales' is obviously a better name for a file of sales figures than 'fred'.

File saving and loading will use a data cartridge which is assumed to be in Microdrive 2 unless a different drive number is given. The simplest way of replying to a file name request is just to type in the name by itself; for example:

```
sales
```

which automatically accesses Microdrive 2. If you wanted to access Microdrive 1, you would type:

```
mdv1_sales
```

There is a third component of a file name which you do not usually see, because it is automatically added by the program. This is an extension, three letters long which identifies which program saved the file. The extensions used are:

QL Quill	**__doc**
QL Abacus	**__aba**
QL Easel	**__grf**
QL Archive (data file)	**__dbf**
QL Archive (program file)	**__prg or __pro**
QL Archive (screen layout)	**__scn**

If you want to transfer information between programs, a special file is generated with the extension __exp (for export). All the programs will recognise this extension. More information on this process is contained in the *Information* section under the heading *QL Program—Import and Export*.

You can direct printer output to a file instead of to a printer, so that you can print the text later. This file has the extension **__lis**.

LISTING FILES

In all the programs except Archive you can request a list of the file names on a cartridge whenever a command needs a file name. This is useful if you cannot remember the exact name that you gave to the file when you first saved it.

Every time the program is waiting for you to type in a file name, you have the following options:

Press **ENTER** to accept the name the program suggests
Type in the file name followed by **ENTER**
Press ? followed by **ENTER** for a list of the files on Microdrive 2

If you type in a question mark (and **ENTER**) instead of the file name, the program displays

```
mdv2_
```

suggesting that it should list the files on Microdrive 2. You can accept this suggestion or you can edit the drive specifier to refer to a different Microdrive (**mdv1__**) and then press **ENTER** to list the files. When the list is complete the program asks you to type in the file name.

Archive does not use this method. Instead there is a command **(dir)** which lists the files. It allows you to type in **mdv1__, mdv2__** and so on, to specify the drive for which the list of files is needed.

ESCAPE

In general, ESC cancels the current action and will restore you to a sensible point in the program. You can also use ESC to cancel any numbers or text that you have typed into the input line or abort a partially completed command.

OTHER DEVICES

Data can be loaded and saved on other devices besides a Microdrive. The device is specified in the standard SuperBASIC way except that the device name is preceded by an underscore (_). See the devices entry in the *Concept Reference Guide*.

For example, to load and save via the network:

Before loading a QL program, each computer on the network must be given a station number. Switch the computer on but do not insert a program cartridges; press F1 or F2 when prompted.

To set the station number type the command NET followed by the station number of your choice. For example, to set the QL to station 5 type the command

```
NET 5 ENTER
```

Place the program cartridge in Microdrive 1 and load the program by typing

```
1run mdv1_boot ENTER
```

Once the program is running, you can receive data sent along the network by typing the **load** command in the normal way. If the data was being sent by station 12, you would enter

```
LOAD _neti_12
```

This must be done before station 12 starts sedning.

To send data, type in the **save** command. Assuming you were sending to station 23, you would enter

```
SAVE _neto_23
```

Station 23 must be ready to receive before you press ENTER

FOR SERVICE MANUALS
CONTACT:
MAURITRON TECHNICAL SERVICES
www.mauritron.co.uk
TEL: 01844 - 351694
FAX: 01844 - 352554

sinclair

QL

Beginner's Guide

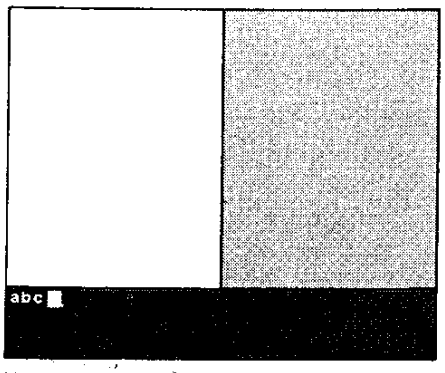
© SINCLAIR RESEARCH LIMITED
by Roy Atherton *(Bulmershe College Computer Centre)*

CHAPTER 1

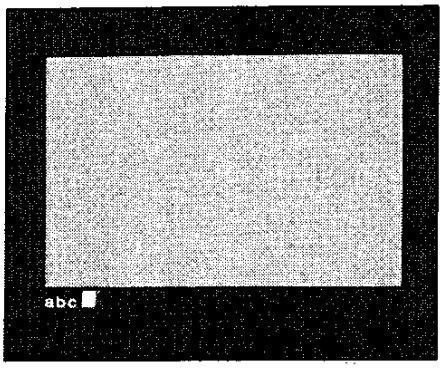
STARTING COMPUTING

THE SCREEN

Your QL should be connected to a monitor screen or TV set and switched on. Press a few keys, say abc, and the screen should appear as shown below. The small flashing light is called the cursor.



Monitor



Television

If your screen does not look like this read the section entitled Introduction. This should enable you to solve any difficulties.

THE KEYBOARD

The QL is a versatile and powerful computer, so there are features of the keyboard which you do not need yet. For the present we will explain just those items which you need for this and the next six chapters.

BREAK

This enables you to 'break' out of situations you do not like. For example;

a line which you have decided to abandon
something wrong which you do not understand
a running program which has ceased to be of interest
any other problem

Because BREAK is so powerful it has been made difficult to type accidentally.

Hold down CTRL and then press SPACE

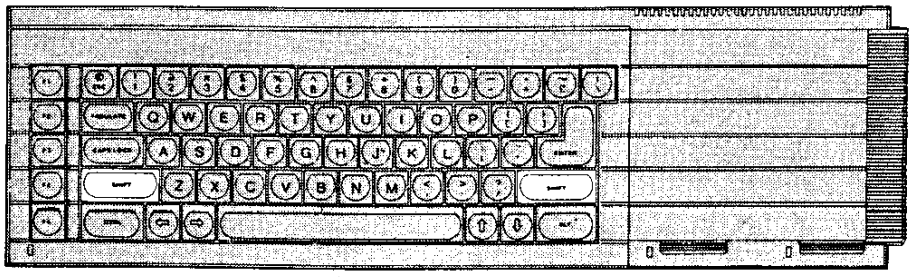
If nothing was added or removed from a program while it was halted with **BREAK** then it can be restarted by typing:

```
CONTINUE
```

RESET

This is not a key but a small push button on the right hand side of the QL. It is placed here deliberately, out of the way, because its effects are more dramatic than the break keys. If you cannot achieve what you need with the break keys then press the RESET button. This is almost the same as switching the computer off and on again. You get a clean re-start.

SHIFT



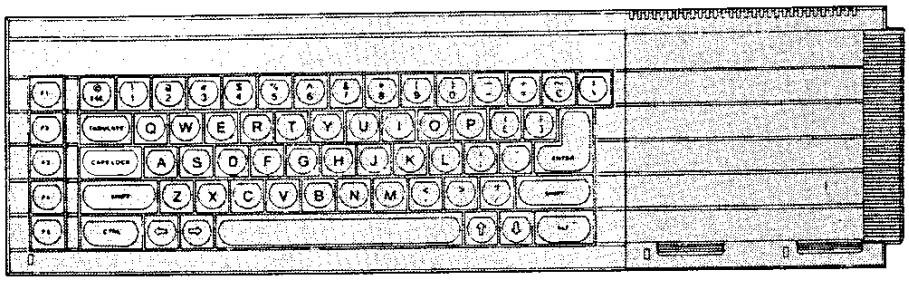
There are two SHIFT keys because they are used frequently and need to be available to either hand.

Hold down one SHIFT key and type some letter keys. You will get upper case (capital) letters.

Hold down one SHIFT key and type some other key, not a letter. You will get a symbol in an upper position on the key.

Without a SHIFT key you get lower case (small) letters or a symbol in a lower position on a key.

CAPITALS LOCK

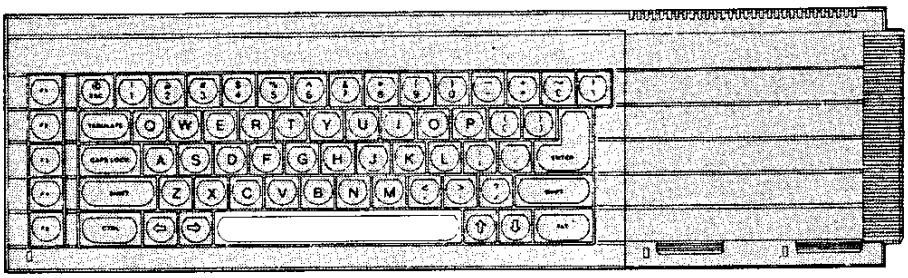


This key works like a switch. Just press it once and only the letter keys will be 'locked' into a particular mode – upper case or lower case.

Type some letter keys.
Type the CAPS LOCK key once.
Type some letter keys.

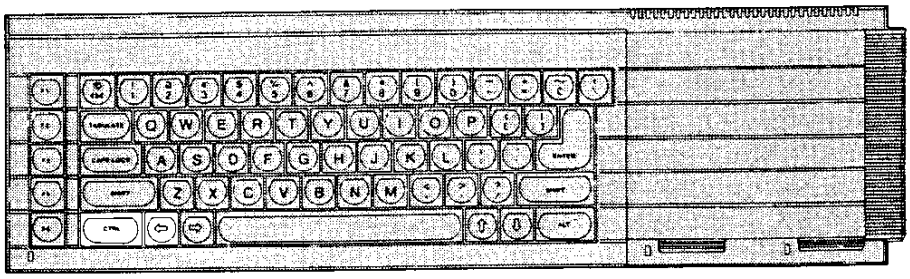
You will see that the mode changes and remains until you type the CAPS LOCK key again.

SPACE BAR



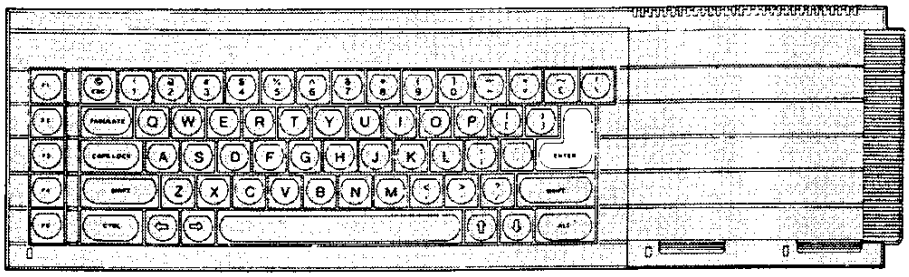
The long key at the bottom of the keyboard gives spaces. This is a very important key in SuperBASIC as you will see in chapter two.

RUBBING OUT



The left cursor together with the CTRL key acts like a rubber. You must hold down the CTRL key while you press the cursor key. Each time you then both together the previous character is deleted.

ENTER



The system needs to know when you have typed a complete message or instruction. When you have typed something complete such as **RUN** you type the **ENTER** key to enter it into the system for action.

Because this key is needed so often we have used a special symbol for it:

⏎

We shall use this for convenience, better presentation, and to save space. Test the ⏎ (ENTER) key by typing:

```
PRINT "Correct"⏎
```

If you made no mistakes the system will respond with:

```
Correct
```

OTHER KEYBOARD SYMBOLS OF IMMEDIATE USE

*	multiply	+	add
_	underscore	=	becomes equal to (used in LET)
"	quotes	'	apostrophe
,	comma	!	exclamation
;	semi colon	&	ampersand
:	colon	.	decimal point or full stop
\	backslash	$	dollar
(	left bracket	)	right bracket

UPPER AND LOWER CASE

SuperBASIC recognises commands (keywords) whether they are in upper or lower case. For example the SuperBASIC command to clear the screen is **CLS** and can be typed in as:

```
CLS⏎
cls⏎
clS⏎
```

These are all correct and have the same effect. Some keywords are displayed partly in upper case to show allowed abbreviations. Where a keyword cannot be abbreviated it is displayed completely in upper case.

USE OF QUOTES

The usual use of quotes is to define a word or sentence – a string of characters. Try:

```
PRINT "This works"⏎
```

The computer will respond with:

```
This works
```

The quotes are not printed but they indicate that some text is to be printed and they define exactly what it it is – everything between the opening and closing quote marks. If you wish to use the quote symbol itself in a string of characters then the apostrophe symbol can be used instead. For example:

```
PRINT 'The quote symbol is " '
```

will work and will print

```
The quote symbol is "
```

COMMON TYPING ERRORS

The zero key is with the other numeric digits at the top of the keyboard, and is slightly thinner.

The letter 'O' key is amongst the other letters. Be careful to use the right symbol.

Similarly avoid confusion between one, amongst the digits, and the letter 'l' amongst the letters.

KEEP SHIFT DOWN

When using a **SHIFT** key hold it down while you type the other key so that the **SHIFT** key makes contact before the other key and also remains in contact until after the other key has lifted.

The same rule applies to the control **CTRL** and alternate **ALT** keys which are used in conjunction with others but you do not need those at present.

Type the two simple instructions:

```
CLS↵
PRINT 'Hello'↵
```

Strictly speaking these constitute a computer program, however, it is the **stored program** that is important in computing. The above instructions are executed instantly as you type ↵ (ENTER).

Now type the program with line numbers:

```
10 CLS↵
20 PRINT 'HELLO'↵
```

This time nothing happens externally except that the program appears in the upper part of the screen. This means that it is accepted as correct grammar or syntax. It conforms to the rules of SuperBASIC, but it has not yet been executed, merely stored. To make it work, type:

```
RUN↵
```

The distinction between direct commands for immediate action and a stored sequence of instructions is discussed in the next chapter. For the present you can experiment with the above ideas and two more:

```
LIST↵
```

causes an internally stored program to be displayed (listed) on the screen or elsewhere.

```
NEW↵
```

causes an internally stored program to be deleted so that you can type in a **NEW** one.

SELF TEST ON CHAPTER 1

You can score a maximum of 16 points from the following test. Check your score with the answers on page 105.

1. In what circumstances might you use the BREAK sequence?
2. Where is the RESET button?
3. What is the effect of the RESET button?
4. Name two differences between a **SHIFT** key and the **CAPS LOCK** key.
5. How can you delete a wrong character which you have just typed?
6. What is the purpose of the **ENTER** key?
7. What symbol do we use for the **ENTER** key?

What is the effect of the commands in questions 8 to 11

8. `CLS`↵
9. `RUN`↵
10. `LIST`↵
11. `NEW`↵
12. Do keywords have the proper effect if you type them in lower case?
13. What is the significance of the parts of keywords which the QL displays in upper case?

FOR SERVICE MANUALS
CONTACT:
MAURITRON TECHNICAL SERVICES
www.mauritron.co.uk
TEL: 01844 - 351694
FAX: 01844 - 352554

CHAPTER 2
INSTRUCTING THE COMPUTER

Computers need to store data such as numbers. The storage can be imagined as pigeon holes.

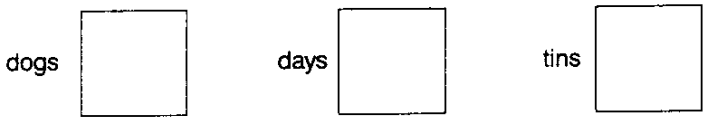
Though you cannot see them, you do need to give names to particular pigeon holes. Suppose you want to do the following simple calculation.

A dog breeder has 9 dogs to feed for 28 days, each at the rate of one tin of 'Beefo' per day. Make the computer print (display on the screen) the required number of tins.

One way of solving this problem would require three pigeon holes for:

number of *dogs*
number of *days*
total number of *tins*

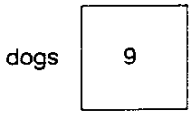
SuperBASIC allows you to choose sensible names for pigeon holes and you may choose as shown:



You can make the computer set up a pigeon hole, name it, and store a number in it with a single instruction or statement such as:

```
LET dogs = 9←
```

This will set up an internal pigeon hole, named *dogs*, and place in it the number 9 thus:



The word **LET** has a special meaning to SuperBASIC. It is called a **keyword**. SuperBASIC has many other keywords which you will see later. You must be careful about the space after **LET** and other keywords. Because SuperBASIC allows you to choose pigeon hole names with great freedom *LETdogs* would be a valid pigeon hole name.

The **LET** keyword is optional in SuperBASIC and because of this statements like:

```
LETdogs = 9←
```

are valid. This would refer to a pigeon hole called **LETdogs**.

Just as, in English, names, numbers and keywords should be separated from each other by spaces if they are not separated by special characters.

Even if it were not necessary, a program line without proper spacing is bad style. Machines with small memory size may force programmers into it, but that is not a problem with the QL.

You can check that a pigeon hole exists internally by typing:

```
PRINT dogs←
```

The screen should display what is in the pigeon hole:

```
9
```

Again, be careful to put a space after **PRINT**.

To solve the problem we can write a program which is a sequence of instructions or statements. You can now understand the first two:

```
LET dogs = 9←
LET days = 28←
```

These cause two pigeon holes to be set up, named, and given numbers or values.

The next instruction must perform a multiplication, for which the computer's symbol is *, and place the result in a new pigeon hole called *tins* thus:

```
LET tins = dogs * days←
```

1. The computer gets the values, 9 and 28, from the two pigeon holes named *dogs* and *days*.
2. The number 9 is multiplied by 28.
3. A new pigeon hole is set up and named *tins*.
4. The result of the multiplication becomes the value in the pigeon hole named *tins*.

All this may seem elaborate but you need to understand the ideas, which are very important.

The only remaining task is to make the computer print the result which can be done by typing:

```
PRINT tins ←
```

which will cause the output:

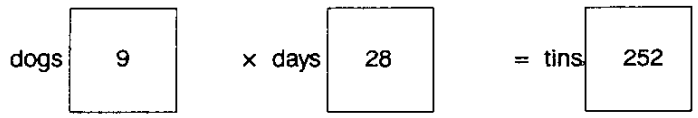
```
252
```

to be displayed on the screen.

In summary, the program:

```
LET dogs = 9←
LET days = 28←
LET tins = dogs * days←
PRINT tins←
```

causes the internal effects best imagined as three named pigeon holes containing numbers:



and the output on the screen:

```
252
```

Of course, you could achieve this result more easily with a calculator or a pencil and paper. You could do it quickly with the QL by typing:

```
PRINT 9 * 28←
```

which would give the answer on the screen. However, the ideas we have discussed are the essential starting points of programming in SuperBASIC. They are so essential that they occur in many computer languages and have been given special names.

1. Names such as *dogs*, *days* and *tins* are called **identifiers**.
2. A single instruction such as:

   ```
   LET dogs = 9←
   ```

   is called a **statement**.
3. The arrangement of name and associated pigeon hole is called a **variable**. The execution of the above statement stores the value 9 in the pigeon hole 'identified' by the identifier *dogs*.

A statement such as:

```
LET dogs = 9←
```

is an instruction for a highly dynamic internal process but the printed text is static and it uses the = sign borrowed from mathematics. It is better to think or say (but not type):

```
LET dogs become 9
```

and to think of the process having a right to left direction (do not type this):

```
dogs ⇦ 9
```

The use of = in a **LET** statement is not the same as the use of = in mathematics. For example, if another dog turns up you may wish to write:

```
LET dogs = dogs + 1←
```

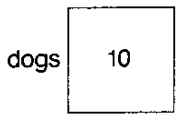
Mathematically, this is not very sensible but in terms of computer operations it is simple. If the value of dogs before the operation was 9 then the value after the operation would be 10. Test this by typing:

```
LET dogs = 9←
PRINT dogs←
LET dogs = dogs + 1←
PRINT dogs←
```

The output should be:

```
9
10
```

proving that the final value in the pigeon hole is as shown:



A good way to understand what is happening to the pigeon holes, or variables, is to do what is called a 'dry run'. You simply examine each instruction in turn and write down the values which result from each instruction to show how the pigeon holes are set up and given values, and how they retain their values as the program is executed.

```
LET dogs = 9←
LET days = 28←
LET tins = dogs * days←
PRINT tins←
```

The output should be:

```
252
```

You may notice that so far a variable name has always been used first on the left hand side of a **LET** statement. Once the pigeon hole is set up and has a value, the corresponding variable name can be used on the right hand side of a **LET** statement.

Now suppose you wish to encourage a small child to save money. You might give two bars of chocolate for every pound saved. Suppose you try to compute this as follows:

```
LET bars = pounds * 2←
PRINT bars←
```

You cannot do a dry run as the program stands because you do not know how many pounds have been saved.

We have made a deliberate error here in using *pounds* on the right of a **LET** statement without it having been set up and given some value. Your QL will search internally for the variable *pounds*. It will not find it, so it concludes that there is an error in the program and gives an error message. If we had tried to print out the value of *pounds*, the QL would have printed a * to indicate that *pounds* was undefined. We say that the variable *pounds* has not been **initialised** (given an initial value). The program works properly if you do this first:

```
LET pounds = 7
LET bars = pounds * 2
```

bars	pounds
7	
7	14

The program works properly and gives the output:

```
14
```

A STORED PROGRAM

Typing statements without line numbers may produce the desired result but there are two reasons why this method, as used so far, is not satisfactory except as a first introduction.

1. The program can only execute as fast as you can type. This is not very impressive for a machine that can do millions of operations per second.
2. The individual instructions are not stored after execution so you cannot run the program again or correct an error without re-typing the whole thing.

Charles Babbage, a nineteenth century computer pioneer, knew that a successful computer needed to store instructions as well as data in internal pigeon holes. These instructions would then be executed rapidly in sequence without further human intervention.

The program instructions will be stored but not executed if you use line numbers. Try this:

```
10 LET price = 15
20 LET pens = 7
30 LET cost = price * pens
40 PRINT cost
```

Nothing happens externally yet, but the whole program is stored internally. You make it work by typing:

```
RUN
```

and the output:

```
105
```

should appear.

The advantage of this arrangement is that you can edit or add to the program with minimal extra typing.

EDITING A PROGRAM

Later, you will see the full editing features of SuperBASIC but even at this early stage you can do three things easily:

replace a line
insert a new line
delete a line

Replace a line

Suppose you wish to alter the previous program because the price has changed to 20p for a pen. Simply re-type line 10.

```
10 LET price = 20
```

This line will replace the previous line 10. Assuming the other lines are still stored, test the program by typing:

```
RUN
```

and the new answer, 140, should appear.

Insert a new line

Suppose you wish to insert a line just before the last one, to print the words 'Total Cost'. This situation often arises so we usually choose line numbers 10, 20, 30 ... to allow space to insert extra lines.

To put in the extra line type:

```
35 PRINT "Total Cost"
```

and it will be inserted just before line 40. The system allows line numbers in the range 1 to 32768 to allow plenty of flexibility in choosing them. It is difficult to be quite sure in advance what changes may be needed.

FOR SERVICE MANUALS
CONTACT:
MAURITRON TECHNICAL SERVICES
www.mauritron.co.uk
TEL: 01844 - 351694
FAX: 01844 - 352554

Now type:

```
RUN↵
```

and the new output should be:

```
Total cost
140
```

Delete line

You can delete line 35 by typing:

```
35↵
```

It is as though an empty line has replaced the previous one.

OUTPUT-PRINT

Note how useful the **PRINT** statement is. You can **PRINT** text by using quotes or apostrophes:

```
PRINT "Chocolate bars"↵
```

You can print the values of variables (contents of pigeon holes) by typing statements such as:

```
PRINT bars↵
```

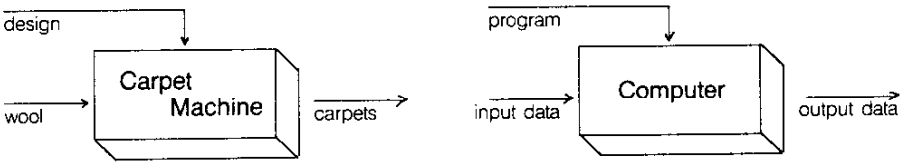
without using quotes.

You will see later how very versatile the **PRINT** statement can be in SuperBASIC. It will enable you to place text or other output on the screen exactly where you want it. But for the present these two facilities are useful enough:

printing of text
printing values of variables (contents of pigeon holes).

INPUT- INPUT, READ AND DATA

A carpet-making machine needs wool as input. It then makes carpets according to the current design.



If the wool is changed you may get a different carpet.

The same sort of relations exist in a computer.

However, if the data is input into pigeon holes by means of **LET** there are two disadvantages when you get beyond very trivial programs:

writing **LET** statements is laborious
changing such input is also laborious

You can arrange for data to be given to a program as it runs. The **INPUT** statement will cause the program to pause and wait for you to type in something at the keyboard. First type:

```
NEW↵
```

so that the previous stored program (if it is still there) will be erased ready for this new one. Now type:

```
10 LET price = 15↵
20 PRINT "How many pens?" ↵
30 INPUT pens ↵
40 LET cost = price * pens ↵
50 PRINT cost ↵
RUN ↵
```

The program pauses at line 30 and you should type the number of pens you want, say:

```
4←
```

Do not forget the ENTER key. The output will be:

```
60
```

The INPUT statement needs a variable name so that the system knows where to put the data which comes in from your typing at the keyboard. The effect of line 30 with your typing is the same as a LET statement's effect. It is more convenient for some purposes when interaction between computer and user is desirable. However, the LET statement and the INPUT statement are useful only for modest amounts of data. We need something else to handle larger amounts of data without pauses in the execution of the program.

SuperBASIC, like most BASICs, provides another method of input known as READing from DATA statements. We can retype the above program in a new form to give the same effects without any pauses. Try this:

```
NEW←
10 READ price, pens←
20 LET cost = price * pens←
30 PRINT cost←
40 DATA 15,4←
RUN←
```

The output should be:

```
60
```

as before.

Each time the program is run, SuperBASIC needs to be told where to start reading DATA from. This can either be done by typing RESTORE followed by the DATA line number or by typing CLEAR. Both these commands can also be inserted at the start of the programs.

When line 10 is executed the system searches the program for a DATA statement. It then uses the values in the DATA statement for the variables in the READ statement in exactly the same order. We usually place DATA statements at the end of a program. They are used by the program but they are not executed in the sense that every other line is executed in turn. DATA statements can go anywhere in a program but they are best at the end, out of the way. Think of them as necessary to, but not really part of, the active program. The rules about READ and DATA are as follows:

1. All DATA statements are considered to be a single long sequence of items. So far these items have been numbers but they could be words or letters.
2. Every time a READ statement is executed the necessary items are copied from the DATA statement into the variables named in the READ statement.
3. The system keeps track of which items have been READ by means of an internal record. If a program attempts to READ more items than exist in all the DATA statements an error will be signalled.

IDENTIFIERS (NAMES)

You have used names for 'pigeon holes' such as *dogs*, *bars*. You may choose words like these according to certain rules:

A name cannot include spaces.

A name must start with a letter.

A name must be made up from **letters**, **digits**, **$**, **%**, **_**(underscore)

The symbols **$**, **%** have special purposes, to be explained later, but you can use the **underscore** to make names such as:

dog_food
month_wage_total

more readable.

SuperBASIC does not distinguish between upper and lower case letters, so names like *TINS* and *tins* are the same.

The maximum number of characters in a name is 255.

Names which are constructed according to these rules are called **identifiers**. Identifiers are used for other purposes in SuperBASIC and you need to understand them. The rules allow great freedom in choice of names so you can make your programs easier to understand. Names like *total, count, pens* are more helpful than names like Z, P, Q.

SELF TEST ON CHAPTER 2

You can score a maximum of 21 points from this test. Check your score with the answers on page 106.

1. How should you imagine an internal number store?
2. State two ways of storing a value in an internal 'pigeon hole' to be created. (two points)
3. How can you find out the value of an internal 'pigeon hole?
4. What is the usual technical name for a 'pigeon hole?
5. When does a pigeon hole get its first value?
6. A variable is so called because its value can vary as a program is executed. What is the usual way of causing such a change?
7. The = sign in a **LET** statement does not mean 'equals' as in mathematics. What does it mean?
8. What happens when you **ENTER** an un-numbered statement?
9. What happens when you **ENTER** a numbered statement?
10. What is the purpose of quotes in a **PRINT** statement?
11. What happens when you do not use quotes in a **PRINT** statement?
12. What does an **INPUT** statement do which a **LET** statement does not?
13. What type of program statement is never executed?
14. What is the purpose of a **DATA** statement?
15. What is another word for the name of a pigeon hole (or variable)?
16. Write down three valid identifiers which use letters, letters and digits, letters and underscore (three points)
17. Why is the space bar especially important in SuperBASIC?
18. Why are freely chosen identifiers important in programming?

FOR SERVICE MANUALS
CONTACT:
MAURITRON TECHNICAL SERVICES
www.mauritron.co.uk
TEL: 01844 - 351694
FAX: 01844 - 352554

PROBLEMS ON CHAPTER 2

1. Carry out a dry run to show the values of all variables as each line of the following program is executed:

```
10 LET hours = 40
20 LET rate = 3
30 LET wage = hours * rate
40 PRINT hours, rate, wage
```

2. Write and test a program, similar to that of problem 1, which computes the area of a carpet which is 3 metres in width and 4 metres in length. Use the variable names: *width, length, area.*

3. Re-write the program of problem 1 so that it uses two **INPUT** statements instead of **LET** statements.

4. Re-write the program of problem 1 so that the input data (40 and 3) appears in a **DATA** statement instead of a **LET** statement.

5. Re-write the program of problem 2 using a different method of data input. Use **READ** and **DATA** if you originally used **LET** and vice-versa.

6. Bill and Ben agree to have a gamble. Each will take out of his wallet all the pound notes and give them to the other. Write a program to simulate this entirely with **LET** and **PRINT** statements. Use a third person, Sue, to hold Bill's money while he accepts Ben's.

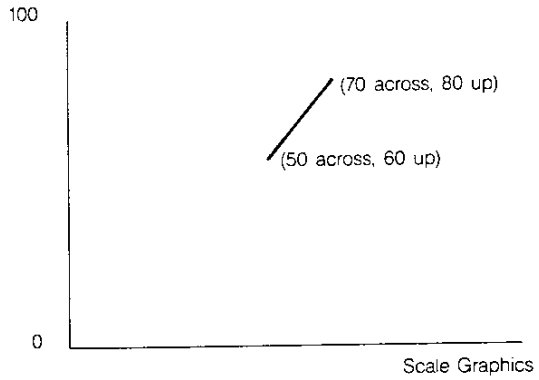
7. Re-write the program of problem 6 so that a **DATA** statement holds the two numbers to be exchanged.

CHAPTER 3

DRAWING ON THE SCREEN

In order to use either a television set or monitor with the QL, two different screen modes are available. MODE 8 permits eight colour displays with a graphics resolution of 256 by 256 pixels and large characters for display on a television set. MODE 4 allows four colours with a resolution of 512 by 256 pixels and a maximum of eighty character lines for which a monitor must be used for successful display. However, it would be unfortunate if a program was written to draw circles or squares in one mode and produced ellipses or rectangles in another mode (as some systems do). We therefore provide a system of scale graphics which avoids these problems. You simply choose a vertical scale and work to it. The other type of graphics (pixel oriented) is also available and is described fully in a later chapter.

Suppose, for example, that we choose a vertical scale of 100 and we wish to draw a line from position (50,60) to position (70,80).



A COLOURED LINE

We need to specify three things:

PAPER (background colour)
INK (drawing colour)
LINE (start and end points)

The following program will draw a line as shown in the above figure in red (colour code 2) on a white (colour code 7) background.

```
NEW ←
10 PAPER 7 : CLS ←
20 INK 2 ←
30 LINE 50,60 TO 70,80 ←
RUN ←
```

In line 10 the paper colour is selected first but it only comes into effect with a further command, such as **CLS**, meaning clear the screen to the current paper colour.

MODES AND COLOURS

So far it does not matter which screen mode you are using but the range of colours is affected by the choice of mode.

MODE 8 allows eight basic colours
MODE 4 allows four basic colours

Colours have codes as described below.

Code	Effect	
	8 colour	4 colour
0	black	black
1	blue	black
2	red	red
3	magenta	red
4	green	green
5	cyan	green
6	yellow	white
7	white	white

For example, INK 3 would give magenta in MODE 8 and red in MODE 4.

We will explain in a later chapter how the basic colours can be 'mixed' in various ways to produce a startling range of colours, shades and textures.

RANDOM EFFECTS

You can get some interesting effects with random numbers which can be generated with the RND function. For example:

```
PRINT RND(1 TO 6) ←
```

will print a whole number in the range 1 to 6, like throwing an ordinary six-sided dice. The following program will illustrate this:

```
NEW ←
10 LET die = RND(1 TO 6) ←
20 PRINT die ←
RUN ←
```

If you run the program several times you will get different numbers.

You can get random whole numbers in any range you like. For example:

```
RND(0 TO 100)
```

will produce a number which can be used in scale graphics. You can re-write the line program so that it produces a random colour. Where the range of random numbers starts from zero you can omit the first number and write:

```
RND(100)
```

```
NEW ←
10 PAPER 7 : CLS ←
20 INK RND(5) ←
30 LINE 50,60 TO RND(100), RND(100) ←
RUN ←
```

This produces a line starting somewhere near the centre of the screen and finishing at some random point. The range of possible colours depends on which mode is selected. You will find that a range of numbers 'something TO something' occurs often in SuperBASIC.

BORDERS

The part of the screen in which you have drawn lines and create other output is called a **window**. Later you will see how you can change the size and position of a window or create other windows. For the present we shall be content to draw a border round the current window. The smallest area of light or colour you can plot on the screen is called a pixel. In mode 8, called **low resolution mode**, there are 256 possible pixel positions across the screen and 256 down. In mode 4, called **high resolution mode**, there are 512 pixels across the screen and 256 down. Thus the size of a pixel depends on the mode.

You can make a border round the inside edge of a window by typing for example:

```
BORDER 4,2 ←
```

This will create a border 4 pixels wide in colour red (code 2). The effective size of the window is reduced by the border. This means that any subsequent printing or graphics will automatically fit within the new window size. The only exception to this is a further border which will overwrite the existing one.

A SIMPLE LOOP

Computers can do things very quickly but it would not be possible to exploit this great power if every action had to be written as an instruction. A building foreman has a similar problem. If he wants a workman to lay a hundred paving stones that is roughly what he says. He does not give a hundred separate instructions.

A traditional way of achieving looping or repetition in BASIC is to use a **GO TO** (or **GOTO**, they are the same) statement as follows:

```
NEW ←
10 PAPER 6 : CLS ←
20 BORDER 1,2 ←
30 INK RND(5) ←
40 LINE 50,60 TO RND(100), RND(100) ←
50 GOTO 0 ←
RUN ←
```

You may prefer not to type in this program because SuperBASIC allows a better way of doing repetition. Note certain things about each line.

```
10
20   Fixed part – not repeated
30
40   Changeable part – repeated
50   Controls program
```

You can re-write the above program by omitting the **GOTO** statement and, instead, putting **REPeat** and **END REPeat** around the part to be repeated.

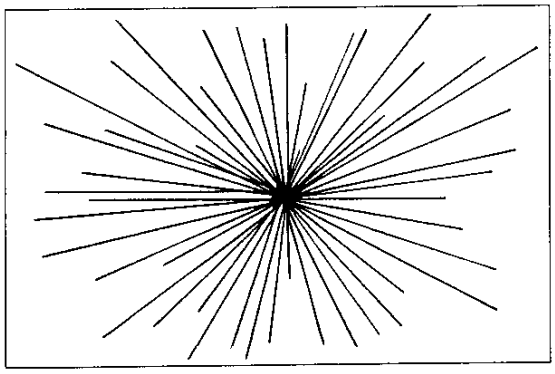
```
NEW ←
10 PAPER 6 : CLS ←
20 BORDER 1,2 ←
30 REPEAT star ←
40    INK RND(5) ←
50    LINE 50,60 TO RND(100), RND(100) ←
60 END REPEAT star ←
RUN ←
```

We have give the *repeat structure* a name, *star*. The structure consists of the two lines:

```
REPeat star
END REPeat star
```

and what lies between them is called the **content** of the structure. The use of upper case letters indicates that **REP** is a valid abbreviation of **REPeat**.

This program should produce coloured lines indefinitely to make a star as shown in the figure below.



The STAR program

You can stop it by pressing the break keys:

Hold down CTRL and then press SPACE

SuperBASIC provides a consistent and versatile method of stopping repetitive processes. Imagine running round and round inside the program activating statements. How can you escape? The answer is to use an **EXIT** statement. But there must be some reason for escaping. You might extend the choice of line colours by typing as an amendment to the program (do not type **NEW**):

```
40 INK RND(0 TO 6) ←
```

so that if **RND** produces 6 the ink is the same colour as the paper and you will not see it. This could be the reason for terminating the repetition. We can re-arrange the program as follows:

```
NEW ←
10 PAPER 6 : CLS ←
20 BORDER 1,2 ←
30 REPeat star ←
40    LET colour = RND(6) ←
50    IF colour = 6 THEN EXIT star ←
60    INK colour ←
70    LINE 50,60 TO RND(100), RND(100) ←
80 END REPeat star ←
```

The important thing to note here is that the program continues until *colour* becomes 6. Control then escapes from the loop to the point just after line 80. Since there are no program lines after 80 the program stops.

Another important concept has been introduced. It is the idea of a decision.

```
IF colour = 6 THEN EXIT star
```

This is another very useful structure because it is a choice of doing something or not; we call it a simple binary decision. Its general form is:

IF condition THEN statement(s)

You will see later how the two concepts of repetition (or looping) and decision-making (or selection) are the main structures for program control. You can stop the program by pressing the break keys: hold down **CTRL** and then press the space bar.

SELF TEST ON CHAPTER 3

You can score a maximum of 13 points from the following test. Check your score with the answers on page 107.

1. What is a pixel?
2. How many pixels fit across the screen in the low resolution mode?
3. How many pixels fit from bottom to top in low resolution mode?
4. What are the two numbers which determine the 'address' or position of a graphics point on the screen?
5. How many colours are available in the low resolution mode?
6. Name the keywords which do the following:
   i. draw a line
   ii. select a colour for drawing
   iii. select a background colour
   iv. draw a border (5 points)
7. What are the statements which open and close the **REPeat** loop?
8. When does an executing **REPeat** loop terminate?
9. Why do loops in SuperBASIC have names?

FOR SERVICE MANUALS
CONTACT:
MAURITRON TECHNICAL SERVICES
www.mauritron.co.uk
TEL: 01844 - 351694
FAX: 01844 - 352554

PROBLEMS ON CHAPTER 3

1. Write a program to draw straight lines all over the screen. The lines should be of random length and direction. Each should start where the previous one finished and each should have a randomly chosen colour.
2. Write a program to draw lines randomly with the restriction that each line has a random start on the left hand edge of the screen.
3. Write a program to draw lines randomly with the restriction that the lines start at the same point on the bottom edge of the screen.
4. Write a program to produce lines of random length, starting points and colour. All lines must be horizontal.
5. As problem 4 but make the lines vertical.
6. Write a program to produce a square 'spiral' in such a way that each line makes a random colour.

   HINT: First find the co-ordinates of some of the corners, then put them in groups of four. You should discover a pattern.

CHAPTER 4
CHARACTERS AND STRINGS

Teachers sometimes wish to assess the reading ability needed for particular books or classroom materials. Various tests are used and some of these compute the average lengths of words and sentences. We will introduce ideas about handling words or **character strings** by examining simple approaches to finding average word lengths.

We are talking about sequences of letters, digits or other symbols which may or may not be words. That is why the term 'character string' has been invented. It is usually abbreviated to **string**. Strings are handled in ways similar to number handling but, of course, we do not do the same operations on them. We do not multiply or subtract strings. We join them, separate them, search them and generally manipulate them as we need.

NAMES AND PIGEON HOLES FOR STRINGS

You can create pigeon holes for strings. You can put character strings into pigeon holes and use the information just as you do with numbers. If you intend to store (not all at once) words such as:

FIRST SECOND THIRD
and
JANUARY FEBRUARY MARCH

you may choose to name two pigeon holes:

Notice the dollar sign. Pigeon holes for strings are internally different from those for numbers and SuperBASIC needs to know which is which. All names of string pigeon holes must end with **$**. Otherwise the rules for choosing names are the same as the rules for the names of numeric pigeon holes.

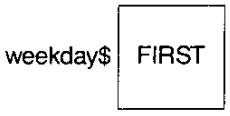
You may pronounce:

weekday$ as weekdaydollar
month$ as monthdollar

The **LET** statement works in the same way as for numbers. If you type:

```
LET weekday$ = "FIRST" ←
```

an internal pigeon hole, named *weekday$*, will be set up with the value FIRST in it thus:



The quote marks are not stored. They are used in the **LET** statement to make it absolutely clear what is to be stored in the pigeon hole. You can check by typing:

```
PRINT weekday$ ←
```

and the screen should display what is in the pigeon hole:

```
FIRST
```

You can use a pair of apostrophes instead of a pair of quote marks.

LENGTHS OF STRINGS

SuperBASIC makes it easy to find the length or number of characters of any string. You simply write, for example:

```
PRINT LEN(weekday$) ←
```

If the pigeon hole, *weekday$*, contains FIRST the number 5 will be displayed. You can see the effect in a simple program:

```
NEW ←
10 LET weekday$ = "FIRST" ←
20 PRINT LEN(weekday$) ←
RUN ←
```

The screen should display:

```
5
```

LEN is a keyword of SuperBASIC.

An alternative method of achieving the same result uses both a string pigeon hole and a numeric pigeon hole.

```
NEW←
10 LET weekday$ = "FIRST"←
20 LET length = LEN(weekday$)←
30 PRINT length←
RUN←
```

The screen should display:

```
5
```

as before, and two internal pigeon holes contain the values shown:

Let us return to the problem of average lengths of words.

Write a program to find the average length of the three words:

FIRST, OF, FEBRUARY

PROGRAM DESIGN

When problems get beyond what you regard as very trivial, it is a good idea to construct a **program design** before writing the program itself.

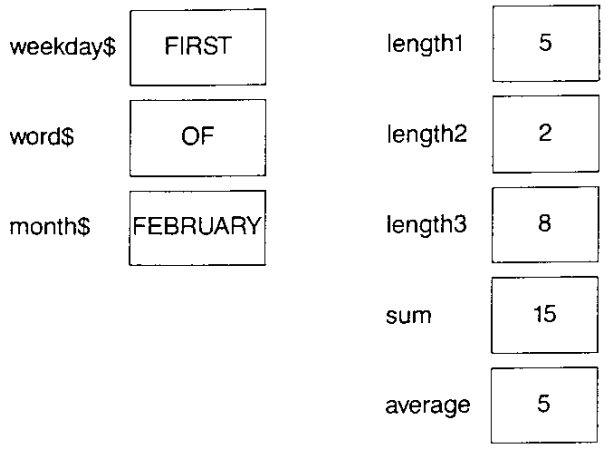
1. Store the three words in pigeon holes.
2. Compute the lengths and store them.
3. Compute the average.
4. Print the result.

```
NEW←
10 LET weekday$ = "FIRST"←
20 LET word$ = "OF"←
30 LET month$ = "FEBRUARY"←
40 LET length1 = LEN(weekday$)←
50 LET length2 = LEN(word$)←
60 LET length3 = LEN(month$)←
70 LET sum = length1 + length2 + length3←
80 LET average = sum/3←
90 PRINT average←
RUN←
```

The symbol / means **divided by**. The output or result of running the program is simply:

```
5
```

and there are eight internal pigeon holes involved:



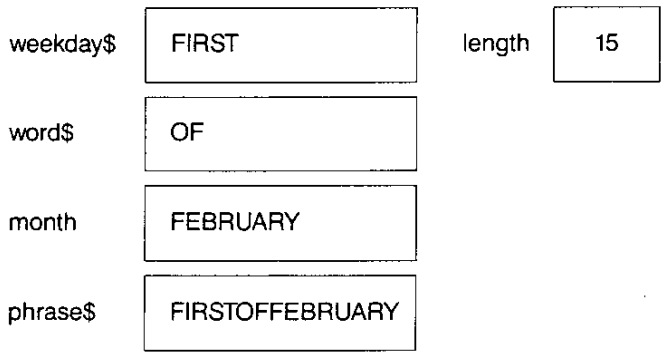
If you think that is a lot of fuss for a fairly simple problem you can certainly shorten it. The shortest version would be a single line but it would be less easy to read. A reasonable compromise uses the symbol & which stands for the operation:

Join two strings

Now type:

```
NEW
10 LET weekday$ = "FIRST"
20 LET word$ = "OF"
30 LET month$ = "FEBRUARY"
40 LET phrase$ = weekday$ & word$ & month$
50 LET length = LEN(phrase$)
60 PRINT length/3
RUN
```

The output is 5 as before but there are some different internal effects:



There is one more reasonable simplification which is to use **READ** and **DATA** instead of the first three **LET** statements. Type:

```
NEW
10 READ weekday$, word$, month$
20 LET phrase$ = weekday$ & word$ & month$
30 LET length = LEN(phrase$)
40 PRINT length/3
50 DATA "FIRST","OF","FEBRUARY"
RUN
```

The internal effects of this version are exactly the same as those of the previous one. **READ** causes the setting up of internal pigeon holes with values in them in a similar way to **LET**.

IDENTIFIERS AND STRING VARIABLES

Names of pigeon holes, such as:

weekday$
word$
month$
phrase$

are called string identifiers. The dollar signs imply that the pigeon holes are for character strings. The dollar must always be at the end.

Pigeon holes of this kind are called **string variables** because they contain only character strings which may vary as a program runs.

The contents of such pigeon holes are called values. Thus words like 'FIRST' and 'OF' may be values of string variables named *weekday$* and *+word$*.

RANDOM CHARACTERS

You can use character codes (see *Concept Reference Guide*) to generate random letters. The upper case letters A to Z have the codes 65 to 90. The function **CHR$** converts these codes into letters. The following program will print a letter B.

```
NEW
10 LET lettercode = 66
20 PRINT CHR$(lettercode)
RUN
```

The following program will generate trios of letters A, B, or C until the word CAB is spelled accidentally.

```
NEW
10 REPeat taxi
20 LET first$ = CHR$(RND(65 TO 67))
30 LET second$ = CHR$(RND(65 TO 67))
40 LET third$ = CHR$(RND(65 TO 67))
50 LET word$ = first$ & second$ & third$
60 PRINT ! word$ !
70 IF word$ = "CAB" THEN EXIT taxi
80 END REPeat taxi
```

Random characters, like random numbers or random points are useful for learning to program. You can easily get interesting effects for program examples and exercises.

Note the effect the ! ... ! have on the spacing of the output.

SELF TEST ON CHAPTER 4

You can score a maximum of 10 points from the following test. Check your score with the answers on page 107.

1. What is a character string?
2. What is the usual abbreviation of the term, 'character string'?
3. What distinguishes the name of a string variable?
4. How do some people pronounce a word such as 'word$'?
5. What keyword is used to find the number of characters in a string?
6. What symbol is used to join two strings?
7. Spaces can be part of a string. How are the limits of a string defined?
8. When a statement such as:

   LET meat$ = "steak"

   is executed, are the quotes stored?
9. What function will turn a suitable code number into a letter?
10. How can you generate random upper case letters?

PROBLEMS ON CHAPTER 4

1. Store the words 'Good' and 'day' in two separate variables. Use a LET statement to join the values of the two variables in a third variable. Print the result.

2. Store the following words in four separate pigeon holes:

   ```
   light let be there
   ```

   Join the words to make a sentence adding spaces and a full stop. Store the whole sentence in a variable, *sent$*, and print the sentence and the total number of characters it contains.

3. Write a program which uses the keywords:

   ```
   CHR$RND(65 TO 90))
   ```

   to generate one hundred random three letter words. See if you have accidentally generated any real English words. Test the effects of:

   a) ; at the end of a **PRINT** statement.
   b) ! on either side of item printed.

FOR SERVICE MANUALS
CONTACT:
MAURITRON TECHNICAL SERVICES
www.mauritron.co.uk
TEL: 01844 - 351694
FAX: 01844 - 352554

CHAPTER 5
KNOWN GOOD PRACTICE

You have already begun to work effectively with short programs. You may have found the following practices are helpful:

1. Use of lower case for identifiers: names of variables (pigeon holes) or repeat structures, etc.
2. Indenting of statements to show the content of a repeat structure.
3. Well chosen identifiers reflecting what a variable or repeat structure is used for.
4. Editing a program by:

   replacing a line
   inserting a line
   deleting a line

PROGRAMS AS EXAMPLES

You have reached the stage where it is helpful to be able to study programs to learn from them and to try to understand what they do. The mechanics of actually running them should now be well understood and in the following chapters we will dispense with the constant repetition of:

NEW before each program
← at the end of each line
RUN to start each program

You will understand that you should use all these features when you wish to enter and run a program. But their omission in the text will enable you to see the other details more clearly as you try to imagine what the program will do when it runs.

If we dispense with the above details we may use and understand programs more easily without the technical clutter. For example, the following program generates random upper case letters until a Z appears. It does not show the words **NEW** or **RUN** or the **ENTER** symbol but you still need to use these.

```
10 REPeat letters
20   LET lettercode = RND(65 TO 90)
30   cap$ = CHR$(lettercode)
40   PRINT cap$
50   IF cap$ = "Z" THEN EXIT letters
60 END REPeat letters
```

In this and subsequent chapters programs will be shown without **ENTER** symbols. Direct commands will also be shown without **ENTER** symbols. But you must use these keys as usual. You must also remember to use **NEW** and **RUN** as necessary.

AUTOMATIC LINE NUMBERING

It is tedious to enter line numbers manually. Instead you can type:

```
AUTO
```

before you start programming and the QL will reply with a line number:

```
100
```

Continue typing lines until you have finished your program when the screen will show:

```
100 PRINT "First"
110 PRINT "Second"
120 PRINT "End"
```

To finish the automatic production of line numbers use the BREAK sequence:

Hold down the **CTRL** and press the **SPACE** bar. This will produce the message:

```
130 not complete
```

and line 130 will not be included in your program.

If you make a mistake which does not cause a break from automatic numbering, you can continue and EDIT the line later. If you want to start at some particular line number, say 600, and use an increment other than 10 you can type, for an increment of 5:

```
AUTO 600,5
```

Lines will then be numbered 600, 605, 610, etc.

To cancel AUTO, press CTRL and the space bar at the same time.

EDITING A LINE

To edit a line simply type EDIT followed by the line number, for example:

```
EDIT 110
```

The line will then be displayed with the cursor at the end thus:

```
110 PRINT "Second"
```

You can move the cursor using:

⇦ one place left
⇨ one place right

To delete a character to the left use:

CTRL with ⇦

To delete the character in the cursor position type:

CTRL with ⇨

and the character to the right of the cursor will move up to close the gap.

USING MICRODRIVE CARTRIDGES

Before using a new Microdrive cartridge it must be formatted. Follow the instructions in the *Introduction*. The choice of name for the cartridge follows the same rules as SuperBASIC identifers, etc. but limited to only 10 characters. It is a good idea to write the name of the cartridge on the cartridge itself using one of the supplied sticky labels.

You should always keep at least one back-up copy of any program or data. Follow the instructions in the Information section of the User Guide.

WARNING

If you FORMAT a cartridge which holds programs and/or data, ALL the programs and/or data will be lost.

SAVING PROGRAMS

The following program sets borders, 8 pixels wide, in red (code 2), in three windows designated #0, #1, #2.

```
100 REMark Border
110 FOR k = 0 TO 2 : BORDER #k,8,2
```

You can save it on a microdrive by inserting a cartridge and typing:

```
SAVE mdv1_bord
```

The program will be saved in a Microdrive file called **bord**.

CHECKING A CARTRIDGE

If you want to know what programs or data files are on a particular cartridge place it in Microdrive 1 and type:

```
DIR mdv1_
```

The directory will be displayed on the screen. If the cartridge is in Microdrive 2 then type instead:

```
DIR mdv2_
```

COPYING PROGRAMS AND FILES

Once a program is stored as a file on a Microdrive cartridge it can be copied to other files. This is one way of making a backup copy of a Microdrive cartridge. You might copy all the previous programs, and similar commands for other programs, onto another cartridge in Microdrive 2 by typing:

```
COPY mdv1_bord TO mdv2_bord
```

DELETING A CARTRIDGE FILE

A file is anything, such as a program or data, stored on a cartridge. To delete a program called *prog* you type:

```
DELETE mdv1_prog
```

LOADING PROGRAMS

A program can be loaded from a Microdrive cartridge by typing:

```
LOAD mdv2_bord
```

If the program loads correctly it will prove that both copies are good. You can test the program by using:

LIST to list it.
RUN to run it.

Instead of using **LOAD** followed by **RUN** you can combine the two operations in one command.

```
LRUN mdv2_bord
```

The program will load and execute immediately.

MERGING PROGRAMS

Suppose that you have two programs saved on Microdrive 1 as *prog1* and *prog2*:

```
100 PRINT "First"
110 PRINT "Second"
```

If you type:

```
LOAD mdv1_prog1
```

followed by:

```
MERGE mdv1_prog2
```

The two programs will be merged into one. To verify this, type **LIST** and you should see:

```
100 PRINT "First"
110 PRINT "Second"
```

If you **MERGE** a program make sure that all its line numbers are different from the program already in main memory. Otherwise it will overwrite some of the lines of the first program. This facility becomes very valuable as you become proficient in handling procedures. It is then quite natural to build a program up by adding procedures or functions to it.

GENERAL

Be careful and methodical with cartridges. Always keep one back-up copy and if you suspect any problem with a cartridge or microdrive keep a second back-up copy. Computer professionals very rarely lose data. They know that even the best machines or devices will be occasional faults and they allow for this.

If you want to call a program by a particular name, say, *square*, it may be a good idea to use names like *sq1, sq2...* for preliminary versions. When the program is in its final form take at least two copies called *square* and the others may be deleted by re-formatting or by some more selective method.

SELF TEST ON CHAPTER 5

You can score a maximum of 14 points from the following test. Check your score with the answers on page 108.

1. Why are lower case letters preferred for program words which you choose?
2. What is the purpose of indenting?
3. What should normally guide your choice of identifiers for variables and loops?
4. Name three ways of editing a program in the computer's main memory (three points).
5. What should you remember to type at the end of every command or program line when you enter it?
6. What should you normally type before you enter a program at the keyboard?
7. What must be at the beginning of every line to be stored as part of a program?
8. What must you remember to type to make a program execute?
9. What keyword enables you to put into a program information which has no effect on the execution?
10. Which two keywords help you to store programs on and retrieve from cartridges? (two points).

PROBLEMS ON CHAPTER 5

1. Re-write the following program using lower case letters to give a better presentation. Add the words **NEW** and **RUN**. Use line numbers and the **ENTER** symbol just as you would to enter and run a program. Use **REMark** to give the program a name.

```
LET TWO$ = "TWO"
LET FOUR$ = "FOUR"
LET SIX$ = TWO$ & FOUR$
PRINT LEN(six$)
```

   Explain how two and four can produce 7.

2. Use indenting, lower case letters, **NEW**, **RUN**, line numbers and the **ENTER** symbol to show how you would actually enter and run the following program:

```
REPEAT LOOP
LETTER_CODE = RND(65 TO 90)
LET LETTERS$ = CHR$(LETTER_CODE)
PRINT LETTER$
IF LETTER$ = 'Z' THEN EXIT LOOP
END REPEAT LOOP
```

3. Re-write the following program in better style using meaningful variable names and good presentation. Write the program as you would enter it:

```
LET S = 0
REPeat TOTAL
LET N = RND(1 TO 6)
PRINT ! N !
LET S = S + N
IF n = 6 THEN EXIT TOTAL
END REPeat TOTAL
PRINT S
```

   Decide what the program does and then enter and run it to check your decision.

CHAPTER 6

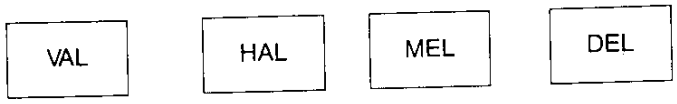
ARRAYS AND FOR LOOPS

WHAT IS AN ARRAY

You know that numbers or character strings can become values of variables. You can picture this as numbers or words going into internal pigeon holes or houses. Suppose for example that four employees of a company are to be sent to a small village, perhaps because oil has been discovered. The village is one of the few places where the houses only have names and there are four available for rent. All the house names end with a dollar symbol.

Westlea$ Lakeside$ Roselawn$ Oaktree$

The four employees are called:



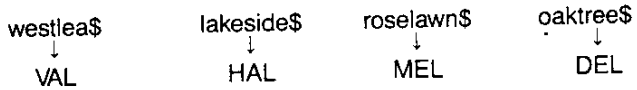
They can be placed in the houses by one of two methods:

Program 1

```
100 LET westlea$ = "VAL"
110 LET lakeside$ = "HAL"
120 LET roselawn$ = "MEL"
130 LET oaktree$ = "DEL"
140 PRINT ! westlea$ ! lakeside$ ! roselawn$ ! oaktree$
```

Program 2

```
100 READ westlea$, lakeside$, roselawn$, oaktree$
110 PRINT ! westlea$ ! lakeside$ ! roselawn$ ! oaktree$
120 DATA "VAL", "HAL", "MEL", "DEL"
```

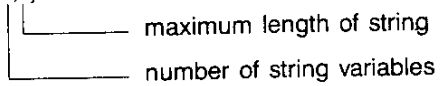


As the amount of data gets larger the advantages of **READ** and **DATA** over **LET** become greater. But when the data gets really numerous the problem of finding names for houses gets as difficult as finding vacant houses in a small village.

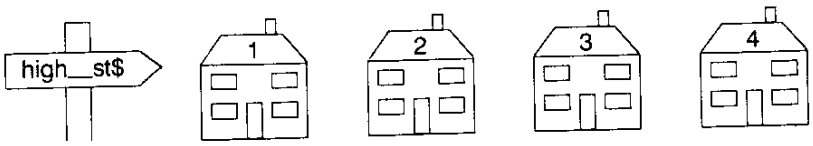
The solution to this and many other problems of handling data lies in a new type of pigeon hole or variable in which many may share a single name. However, they must be distinct so each variable also has a number like numbered houses in the same street. Suppose that you need four vacant houses in High Street numbered 1 to 4. In SuperBASIC we say there is an **array** of four houses. The name of the array is *high__st$* and the four houses are to be numbered 1 to 4.

But you cannot just use these array variables as you can ordinary (simple) variables. You have to declare the dimensions (or size) of the array first. The computer allocates space internally and it needs to know how many string variables there are in the array and also the maximum length of each string variable. You use a **DIM** statement thus:

```
DIM high__st$(4,3)
```



After the **DIM** statement has been executed the variables are available for use. It is as though the houses have been built but are still empty. The four 'houses' share a common name, *high__st$*, but each has its own number and each can hold up to three characters:



There are five programs below which all do the same thing: they cause the four 'houses' to be 'occupied' and they PRINT to show that the 'occupation' has really worked. The final method uses only four lines but the other four lead up to it in a way which moves all the time from known ideas to new ones or new uses of old ones. The movement is also towards greater economy.

If you understand the first two or three methods perfectly well you may prefer to move straight onto methods 4 and 5. But if you are in any doubt, methods 1, 2 and 3 will help to clarify things.

Program 1

```
100 DIM high_st$(4,3)
110 LET high_st$(1) = "VAL"
120 LET high_st$(2) = "HAL"
130 LET high_st$(3) = "MEL"
140 LET high_st$(4) = "DEL"
150 PRINT ! high_st$(1) ! high_st$(2) !
160 PRINT ! high_st$(3) ! high_st$(4) !
```

Program 2

```
100 DIM high_st$(4,3)
110 READ high_st$(1),high_st$(2),high_st$(3),high_st$(4)
120 PRINT ! high_st$(1) ! high_st$(2) !
130 PRINT ! high_st$(3) ! high_st(4) !
140 DATA "VAL", "HAL", "MEL", "DEL"
```

This shows how to economise on variable names but the constant repeating of *high__st$* is both tedious and the cause of the cluttered appearance of the programs. We can, again, use a known technique – the **REPeat** loop – to improve things further. We set up a counter, *number*, which increases by one as the **REPeat** loop proceeds.

Program 3

```
100 RESTORE 190
110 DIM high_st$(4,3)
120 LET number = 0
130 REPeat houses
140    LET number = number + 1
150    READ high_st$(number)
160    IF num = 4 THEN EXIT houses
170 END REPeat houses
180 PRINT high_st(1)! high_st(2)! high_st(3)! high_st(4)
190 DATA "VAL", "HAL", "MEL", "DEL"
```

This special type of loop, in which something has to be done a certain number of times, is well known. A special structure, called a **FOR** loop, has been invented for it. In such a loop the count from 1 to 4 is handled automatically. So is the exit when all four items have been handled.

Program 4

```
100 RESTORE 160
110 DIM high_st$(4,3)
120 FOR number = 1 TO 4
130    READ high_st$(number)
140    PRINT ! high_st$(number) !
150 END FOR number
160 DATA "VAL","HAL","MEL","DEL"
```

The output from all four programs is the same:

```
VAL HAL MEL DEL
```

Which proves that the data is properly stored internally in the four array variables:

Method 4 is clearly the best so far, because it can deal equally well with 4 or 40 or 400 items by just changing the number 4 and adding more DATA items. You can use as many DATA statements as you need.

In its simplest form the FOR loop is rather like the simplest form of REPeat loop. The two can be compared:

```
100 REPeat greeting                100 FOR greeting = 1 TO 40
110  PRINT "Hello"                 110  PRINT "Hello"
120 END REPeat greeting            120 END FOR greeting
```

Both these loops would work. The REPeat loop would print 'Hello' endlessly (stop it with the BREAK sequence) and the FOR loop would print 'Hello' just forty times.

Notice that the name of the FOR loop is also a variable, *greeting*, whose value varies from 1 to 40 in the course of running the program. This variable is sometimes called the **loop variable** or the **control variable** of the loop.

Note the structure of both loops takes the form:

Opening statement
    Content
Closing statement

However, certain structures have allowable short forms for use when there are only one or a few statements in the content of the loop. Short forms of the FOR loop are allowed so we could write the program in the most economical form of all.

Program 5

```
100 RESTORE 140 : CLS
110 DIM high_st$(4,3)
120 FOR number = 1 TO 4 : READ high_st$(number)
130 FOR number = 1 TO 4 : PRINT ! high_st$(number) !
140 DATA "VAL", "HAL", "MEL", "DEL"
```

Colons serve as end-of-statement symbols instead of ENTER and the ENTER symbols of lines 120 and 130 serve as END FOR statements.

There is an even shorter way of writing the above program. To print out the contents of the array *high_st$* we can replace line 130 by:

```
130 PRINT ! high_st$ !
```

This uses an array slicer which we will discuss later in chapter 13.

We have introduced the concept of an array of string variables so that the only numbers involved would be the subscripts in each variable name. Arrays may be string or numeric, and the following examples illustrate the numeric array.

Program 1

Simulate the throwing of a pair of dice four hundred times. Keep a record of the number of occurrences of each possible score from 2 to 12.

```
100 REMark DICE1
110 LET two = 0:three = 0:four = 0:five = 0:six = 0
120 LET seven = 0:eight = 0:nine = 0:ten = 0:eleven = 0:twelve = 0
130 FOR throw = 1 TO 400
140   LET die1 = RND(1 TO 6)
150   LET die2 = RND(1 TO 6)
160   LET score = die1 + die2
170   IF score = 2 THEN LET two = two + 1
180   IF score = 3 THEN LET three = three + 1
190   IF score = 4 THEN LET four = four + 1
200   IF score = 5 THEN LET five = five + 1
210   IF score = 6 THEN LET six = six + 1
220   IF score = 7 THEN LET seven = seven + 1
230   IF score = 8 THEN LET eight = eight + 1
240   IF score = 9 THEN LET nine = nine + 1
250   IF score = 10 THEN LET ten = ten + 1
260   IF score = 11 THEN LET eleven = eleven + 1
270   IF score = 12 THEN LET twelve = twelve + 1
280 END FOR throw
290 PRINT ! two ! three ! four ! five ! six
300 PRINT ! seven ! eight ! nine ! ten ! eleven ! twelve
```

In the above program we establish eleven simple variables to store the tally of the scores. If you plot the tallies printed at the end you find that the bar chart is roughly triangular. The higher tallies are for scores six, seven, eight and the lower tallies are for two and twelve. As every dice player knows this reflects the frequency of the middle range of scores (six,seven,eight) and the rarity of twos or twelves.

Program 2

```
100 REMark Dice2
110 DIM tally(12)
120 FOR throw = 1 TO 400
130   LET die_1 = RND(1 TO 6)
140   LET die_2 = RND(1 TO 6)
150   LET score = die_1 + die_2
160   LET tally(score) = tally(score) + 1
170 END FOR throw
180 FOR number = 2 to 12 : PRINT tally(number)
```

In the first **FOR** loop, using *throw*, the subscript of the array variable is *score*. This means that the correct array subscript is automatically chosen for an increase in the tally after each throw. You can think of the array, *tally*, as a set of pigeon-holes numbered 2 to 12. Each time a particular score occurs the tally of that score is increased by throwing a stone into the corresponding pigeon-hole.

In the second (short form) **FOR** loop the subscript is *number*. As the value of *number* changes from 2 to 12 all the values of the tallies are printed.

Notice that in the **DIM** statement for a numeric array you need only declare the number of variables required. There is no question of maximum length as there is in a string array.

If you have used other versions of BASIC you may wonder what has happened to the **NEXT** statement. All SuperBASIC structures end with **END** something. That is consistent and sensible but the **NEXT** statement has a part to play as you will see in later chapters.

SELF TEST ON CHAPTER 6

You can score a maximum of 16 points from the following test. Check your score with the answers on page 109.

1. Mention two difficulties which arise when the data needed for a program becomes numerous and you try to handle it without arrays. (two points)
2. If, in an array, ten variables have the same name then how do you know which is which?
3. What must you do normally in a program, before you can use an array variable?
4. What is another word for the number which distinguishes a particular variable of an array from the other variables which share its name?
5. Can you think of two ideas in ordinary life which correspond to the concept of an array in programming? (two points)
6. In a **REPeat** loop, the process ends when some condition causes an **EXIT** statement to be executed. What causes the process in a **FOR** loop to terminate?
7. A **REPeat** loop needs a name so that you can **EXIT** to its **END** properly. A **FOR** loop also has a name but what other function does a **FOR** loop's name have?
8. What are the two phrases which are used to describe the variable which is also the name of a **FOR** loop? (two points)
9. The values of a loop variable change automatically as a **FOR** loop is executed. Name one possible important use of these values.
10. Which of the following do the long form of **REPeat** loops and the long form of **FOR** loops have in common? For each of the four items either say that both have it or which type of loop has it.
    a An opening keyword or statement.
    b A closing keyword or statement.
    c A loop name.
    d A loop variable or control variable (four points)

PROBLEMS ON CHAPTER 6

1. Use a **FOR** loop to place one of four numbers 1,2,3,4 randomly in five array variables:

   card(1), card(2), card(3), card(4), card(5).

   It does not matter if some of the four numbers are repeated. Use a second **FOR** loop to output the values of the five card variables.

2. Imagine that the four numbers 1,2,3,4 represent 'Hearts', 'Clubs', 'Diamonds', 'Spades'. What extra program lines would need to be inserted to get output in the form of these words instead of numbers?

3. Use a **FOR** loop to place five random numbers in the range 1 to 13 in an array of five variables:

   card(1), card(2), card(3), card(4) and *card(5).*

   Use a second **FOR** loop to output the values of the five card variables.

4. Imagine that the random numbers generated in problem 1 represent cards. Write down the extra statements that would cause the following output:

Number	Output
1	the word 'Ace'
2 to 10	the actual number
11	the word 'Jack'
12	the word 'Queen'
13	the word 'King'

CHAPTER 7
SIMPLE PROCEDURES

If you were to try to write computer programs to solve complex problems you might find it difficult to keep track of things. A methodical problem solver therefore divides a large or complex job into smaller sections or **tasks**, and then divides these tasks again into smaller tasks, and so on until each can be be easily tackled.

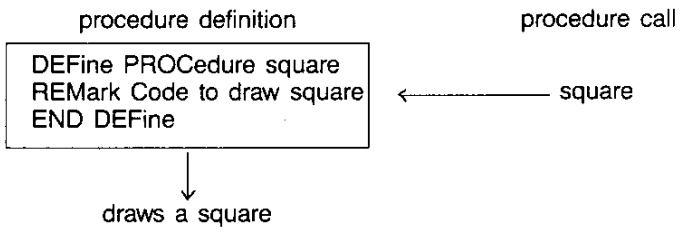
This is similar to the arrangement of complex human affairs. Successful government depends on a delegation of responsibility. The Prime Minister divides the work amongst ministers, who divide it further through the Civil Service until tasks can be done by individuals without further division. There are complicating features such as common services and interplay between the same and different levels, but the hierarchical structure is the dominant one.

A good programmer will also work in this way and a modern language like SuperBASIC which allows properly named, well defined procedures will be much more helpful than older versions which do not have such features.

The idea is that a separately named block of code should be written for a particular task. It doesn't matter where the block of code is in the program. If it is there somewhere, the use of its name will :

activate the code
return control to the point in the program immediately after that use.

If a procedure, *square*, draws a square the scheme is as shown below:



In practice the separate tasks within a job can be identified and named before the definition code is written. The name is all that is needed in calling the procedure so the main outline of the program can be written before all the tasks are defined.

Alternatively if it is preferred, the tasks can be written first and tested. If it works you can then forget the details and just remember the name and what the procedure does.

Example

The following example could quite easily be written without procedures but it shows how they can be used in a reasonably simple context. Almost any task can be broken down in a similar fashion which means that you never have to worry about more than, say, five to thirty lines at any one time. If you can write thirty-line programs well and handle procedures, then you have the capability to write three-hundred-line programs.

You can produce ready made buzz phrases for politicians or others who wish to give an impression of technological fluency without actually knowing anything. Store the following words in three arrays and then produce ten random buzz phrases.

adjec1$	adjec2$	noun$
Full	fifth-generation	systems
Systematic	knowledge-based	machines
Intelligent	compatible	computers
Controlled	cybernetic	feedback
Automated	user-friendly	transputers
Synchronised	parallel	micro-chips
Functional	learning	capability
Optional	adaptable	programming
Positive	modular	packages
Balanced	structured	databases
Integrated	logic-oriented	spreadsheets
Coordinated	file-oriented	word-processors
Sophisticated	standardised	objectives

ANALYSIS

We will write a program to produce ten buzzword phrases. The stages of the program are:

1 Store the words in three string arrays.

2 Choose three random numbers which will be the subscripts of the array variables.

3 Print the phrase.

4 Repeat 2 and 3 ten times.

DESIGN

VARIABLES

We identify three arrays of which the first two will contain adjectives or words used as adjectives – describing words. The third array will hold the nouns. There are 13 words in each section and the longest word has 16 characters including a hyphen.

Array	Purpose
adjec1$(13,12)	first adjectives
adjec2$(13,16)	second adjectives
noun$(13,15)	nouns

PROCEDURES

We use three procedures to match the jobs identified.

store_data stores the three sets of thirteen words
get_random gets three random numbers in range 1 to 13.
make_phrase prints a phrase.

MAIN PROGRAM

This is very simple because the main work is done by the procedures.

Declare (DIM) the arrays
Store_data
FOR ten phrases
get_random
make_phrase
END

Program

```
100 REMark ************
110 REMark * Buzzword *
120 REMark ************
130 DIM adjec1$(13,12), adjec2$(13,16),noun$(13,15)
140 store_data
150 FOR phrase = 1 TO 10
160   get_random
170   make_phrase
180 END FOR phrase
190 REMark ***************************
200 REMark * Procedure Definitions   *
210 REMark ***************************
220 DEFine PROCedure store_data
230   REMark *** procedure to store the buzzword data ***
240   RESTORE 420
250   FOR item = 1 TO 13
260     READ adjec1$(item), adjec2$(item),noun$(item)
270   END FOR item
280 END DEFine
290 DEFine PROCedure get_random
300   REMark *** procedure to select the phrase ***
310   LET ad1 = RND(1 TO 13)
320   LET ad2 = RND(1 TO 13)
330   LET n = RND(1 TO 13)
340 END DEFine
350 DEFine PROCedure make_phrase
360   REMark *** procedure to print out the phrase ***
370   PRINT ! adjec1$(ad1) ! adjec2$(ad2) ! noun$(n)
```

```
380 END DEFine
390 REMark ****************
400 REMark * Program Data *
410 REMark ****************
420 DATA "Full", "fifth-generation", "systems"
430 DATA "Systematic", "knowledge-based", "machines"
440 DATA "Intelligent", "compatible", "computers"
450 DATA "Controlled", "cybernetic", "feedback"
460 DATA "Automated", "user-friendly", "transputers"
470 DATA "Synchronised", "parallel", "micro-chips"
480 DATA "Functional", "learning", "capability"
490 DATA "Optional", "adaptable", "programming"
500 DATA "Positive", "modular", "packages"
510 DATA "Balanced", "structured", "databases"
520 DATA "Integrated", "logic-oriented", "spreadsheets"
530 DATA "Coordinated", "file-oriented", "word-processors"
540 DATA "Sophisticated", "standardised", "objectives"
```

```
Automated fifth-generation capability
Functional learning packages
Full parallel objectives
Positive user-friendly spreadsheets
Intelligent file-oriented capability
Synchronised cybernetic transputers
Functional logic-oriented micro-chips
Positive parallel feedback
Balanced learning databases
Controlled cybernetic objectives
```

FOR SERVICE MANUALS
CONTACT:
MAURITRON TECHNICAL SERVICES
www.mauritron.co.uk
TEL: 01844 - 351694
FAX: 01844 - 352554

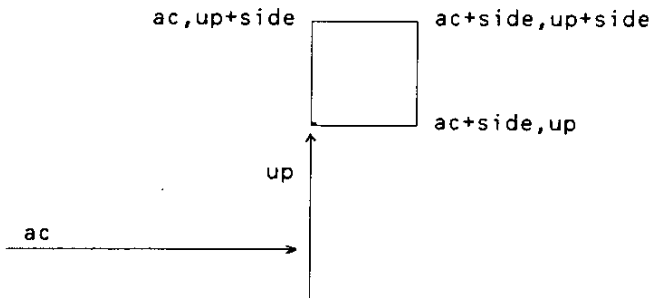
PASSING INFORMATION TO PROCEDURES

Suppose we wish to draw squares of various sizes and various colours in various positions on the scale graphics screen.

If we define a procedure, square, to do this it will require four items of information:

length of one side
colour (colour code)
position (across and up)

The square's position is determined by giving two values, across and up, which fix the bottom left hand corner of the square as shown below.



The colour of the square is easily fixed but the square itself uses the values of *side* and *ac* and *up* as follows.

```
200 DEFine PROCedure square(side,ac,up)
210    LINE ac,up TO ac+side,up
220    LINE TO ac+side,up+side
230    LINE TO ac,up+side TO ac,up
240 END DEFine
```

In order to make this procedure work values of *side,ac* and *up* must be provided. These values are provided when the procedure is called. For example you could add the following main program to get one green square of side 20.

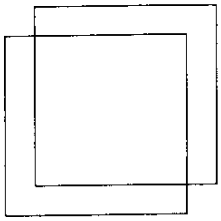
```
100 PAPER 7: CLS
110 INK 4
120 square 20,50,50
```

The numbers 20,50,50 are called parameters and they are passed to the variables named in the procedure definition thus:

```
                                    square 20,50,50
                                           ↓  ↓  ↓
DEFine PROCedure square(side,ac,up)
```

The numbers 20,50,50 are called actual parameters. They are numbers in this case but they could be variables or expressions. The variables *side,ac,up* are called formal parameters. They must be variables because the 'receive' values.

A more interesting main program uses the same procedure to create a random pattern of coloured pairs of squares. Each pair of squares is obtained by offsetting the second one across and up by one-fifth of the side length thus:



Assuming that the procedure **square** is still present at line 200 then the following program will have the classical effect.

```
100 REMark Squares Pattern
110 PAPER 7 : CLS
120 FOR pair = 1 TO 20
130    INK RND(5)
140    LET side = RND(10 TO 20)
150    LET ac = RND(50) : up = RND(70)
160    square side,ac,up
170    LET ac=ac+side/5 : up = up+side/5
180    square side,ac,up
190 END FOR pair
```

The advantage of procedures are:

1. You can use the same code more than once in the same program or in others.
2. You can break down a task into sub-tasks and write procedures for each sub-task. This helps the analysis and design.
3. Procedures can be tested separately. This helps the testing and debugging.
4. Meaningful procedure names and clearly defined beginnings and ends help to make a program readable.

When you get used to properly named procedures with good parameter facilities, you should find that your problem-solving and programming powers are greatly enhanced.

SELF TEST ON CHAPTER 7

You can score a maximum of 14 points from the following test. Check your score with the answers on page 110.

1. How do we normally tackle the problem of great size and complexity in human affairs?
2. How can this principle be applied in programming?
3. What are the two most obvious features of a simple procedure definition? (two points)
4. What are the two main effects of using a procedure name to 'call' the procedure? (two points)
5. What is the advantage of using procedure names in a main program before the procedure definitions are written?
6. What is the advantage of writing a procedure definition before using its name in a main program?
7. How can the use of procedures help a 'thirty-line-programmer' to write much bigger programs?
8. Some programs use more memory in defining procedures, but in what circumstances do procedures save memory space?
9. Name two ways by which information can be passed from a main program to a procedure. (two points)
10. What is an actual parameter?
11. What is a formal parameter?

FOR SERVICE MANUALS
CONTACT:
MAURITRON TECHNICAL SERVICES
www.mauritron.co.uk
TEL: 01844 - 351694
FAX: 01844 - 352554

PROBLEMS ON CHAPTER 7

1. Write a procedure which outputs one of the four suits : 'Hearts', 'Clubs', 'Diamonds', or 'Spades'. Call the procedure five times to get five random suits.
2. Write another program for problem 1 using a number in the range 1 to 4 as a parameter to determine the output word. If you have already done this, then try writing the program without parameters.
3. Write a procedure which will output the value of a card that is a number in the range 2 to 10 or one of the words 'Ace', 'Jack', 'Queen', 'King'.
4. Write a program which calls this procedure five times so that five random values are output.
5. Write the program of problem 3 again using a number in the range 1 to 13 as a parameter to be passed to the procedure. If this was the method you used first time, then try writing the program without parameters.
6. Write the most elegant program you can, using procedures, to output four hands of five cards each. Do not worry about duplicate cards. You can take elegance to mean an appropriate mixture of readability, shortness and efficiency. Different people and/or different circumstances will place different importance on these three qualities which sometimes work against each other.

CHAPTER 8
FROM BASIC TO SUPERBASIC

If you are familiar with one of the earlier versions of BASIC you may find it possible to omit the first seven chapters and use this chapter instead as a bridge between what you know already and the remaining chapters. If you do this and still find areas of difficulty it may be helpful to backtrack a little into some of the earlier chapters.

If you have worked through the earlier chapters this one should be easy reading. You may find that, as well as introducing some new ideas, it gives an interesting slant on the way BASIC is developing. Apart from its program structuring facilities SuperBASIC also pushes forward the frontiers of good screen presentation, editing, operating facilities and graphics. In short it is a combination of user-friendliness and computing power which has not existed before.

So, when you make the transition from BASIC to SuperBASIC you are moving not only to a more powerful, more helpful language, you are also moving into a remarkably advanced computing environment.

We will now discuss some of the main features of SuperBASIC and some of the features which distinguish it from other BASICs.

ALPHABETIC COMPARISONS

The usual simple arithmetic comparisons are possible. You can write:

```
LET pet1$ = "CAT"
LET pet2$ = "DOG"
IF pet1$ < pet2$ THEN PRINT "Meow"
```

The output will be Meow because in this context the symbol < means:

earlier (nearer to A in the alphabet)

SuperBASIC makes comparisons sensible. For example you would expect:

'cat' to come before 'DOG'

and

'ERD98L' to come before 'ERD746L'

A simplistic approach, blindly using internal character coding, would give the 'wrong' result in both the above cases but try the following program which finds the 'earliest' of two character strings.

```
100 INPUT item1$, item2$
110 IF item1$ < item2$ THEN PRINT item1$
120 IF item1$ = item2$ THEN PRINT "Equal"
130 IF item1$ > item 2$ THEN PRINT item2$
```

INPUT		OUTPUT
cat	dog	cat
cat	DOG	cat
ERD98L	ERD746L	ERD98L
ABC	abc	ABC

The *Concept Reference Guide* section will give full details about the way comparisons of strings are made in SuperBASIC.

VARIABLES AND NAMES - IDENTIFIERS

Most BASICs have numeric and string variables. As in other BASICs the distinguishing feature of a string variable name in SuperBASIC is the dollar sign on the end. Thus:

numeric:	string:
count	*word$*
sum	*high_st$*
total	*day_of_week$*

You may not have met such meaningful variable names before though some of the more recent BASICs do allow them. The rules for identifiers in SuperBASIC are given in the *Concept Reference Guide*. The maximum length of an identifier is 255 characters. Your choice of identifiers is a personal one. Sometimes the longer ones are more helpful in conveying to the human reader what a program should do. But they have to be typed and, as in ordinary English, *spade* is more sensible than *horticultural earth-turning implement*. Shorter words are preferred if they convey the meaning but very short words or single letters should be used sparingly. Variable names like *X,Z,P3,Q2* introduce a level of abstraction which most people find unhelpful.

INTEGER VARIABLES

SuperBASIC allows **integer** variables which take only whole-number values. We distinguish these with a percentage sign thus:

count%
number%
nearest_pound%

There are now two kinds of numeric variable. We call the other type, which can take whole or fractional values, **floating point**. Thus you can write:

```
LET price = 9
LET cost = 7.31
LET count% = 13
```

But if you write:

```
LET count% = 5.43
```

the value of *count%* will become 5. On the other hand:

```
LET count% = 5.73
```

will cause the value of *count%* to be 6. You can see that SuperBASIC does the best it can, rounding off to the nearest whole number.

COERCION

The principle of always trying to be intelligently helpful, rather than give an error message or do something obviously unwanted, is carried further. For example, if a string variable *mark$* has the value

```
'64'
```

then:

```
LET score = mark$
```

will produce a numeric value of 64 for score. Other versions of BASIC would be likely to halt and say something like:

'Type mis-match'
or 'Nonsense in BASIC'

If the string cannot be converted then an error is reported.

LOGICAL VARIABLES AND SIMPLE PROCEDURES

There is one other type of variable in SuperBASIC, or rather the SuperBASIC system makes it seem so. Consider the SuperBASIC statement:

```
IF windy THEN fly_kite
```

In other BASICs you might write:

```
IF w=1 THEN GOSUB 300
```

In this case *w=1* is a condition or logical expression which is either true or false. If it is true then a subroutine starting at line 300 would be executed. This subroutine may deal with kite flying but you cannot tell from the above line. A careful programmer would write:

```
IF w=1 THEN GOSUB 300 : REM fly_kite
```

to make it more readable. But the SuperBASIC statement is readable as it stands. The identifier *windy* is interpreted as true or false though it is actually a floating point variable. A value of 1 or any non-zero value is taken as *true*. Zero is taken as false. Thus the single word, *windy*, has the same effect as a condition of logical expression.

The other word, *fly_kite*, is a procedure. It does a job similar to, but rather better than, GOSUB 300.

The following program will convey the idea of logical variables and the simplest type of named procedure.

```
100 INPUT windy
110 IF windy THEN fly_kite
120 IF NOT windy THEN tidy_shed
130 DEFine PROCedure fly_kite
140   PRINT "See it in the air."
150 END DEFine
160 DEFine PROCedure tidy_shed
170   PRINT "Sort out rubbish."
180 END DEFine
```

INPUT	OUTPUT
0	Sort out rubbish.
1	See it in the air
2	See it in the air
-2	See it in the air

You can see that only zero is taken as meaning false. You would not normally write procedures with only one action statement, but the program illustrates the idea and syntax in a very simple context. More is said about procedures later in this chapter.

LET STATEMENTS

In SuperBASIC LET is optional but we use it in this manual so that there will be less chance of confusion caused by the two possible uses of =. The meanings of = in:

```
LET count = 3
```

and in

```
IF count = 3 THEN EXIT
```

are different and the LET helps to emphasise this. However, if there are two or a few LET statements doing some simple job such as setting initial values, an exception may be made.

For example:

```
100 LET first = 0
110 LET second = 0
120 LET third = 0
```

may be re-written as:

```
100 LET first = 0 : second = 0 : third = 0
```

without loss of clarity or style. It is also consistent with the general concept of allowing short forms of other constructions where they are used in simple ways.

The colon : is a valid statement terminator and may be used with other statements besides LET.

THE BASIC SCREEN

In a later chapter we will explain how other graphics facilities, such as drawing circles, can be handled but here we outline the pixel-oriented features. There are two modes which may be activated by any of the following:

Low resolution 8 Colour Mode 256 pixels across, 256 down	MODE 256 MODE 8
High resolution 4 Colour Mode 512 pixels across, 256 down	MODE 512 MODE 4

In both modes pixels are addressed by the range of numbers:

0 - 511 across
and 0 - 255 down

Since mode 8 has only half the number of pixels across the screen as mode 4, mode 8 pixels are twice as wide as mode 4 pixels and so in mode 8 each pixel can be specified by two coordinates. For example:

0 or 1 2 or 3 510 or 511

It also means that you use the same range of numbers for addressing pixels irrespective of the mode. Always think 0–511 across and 0–255 down.

If you are using a television then not all the pixels may be visible.

COLOURS

The colours available are:

MODE 256	Code	MODE 512
black	0	black
blue	1	
red	2	red
magenta	3	
green	4	green
cyan	5	
yellow	6	white
white	7	

You may find the following mnemonic helpful in remembering the codes:

Bonny Babies Really Make Good Children, You Wonder

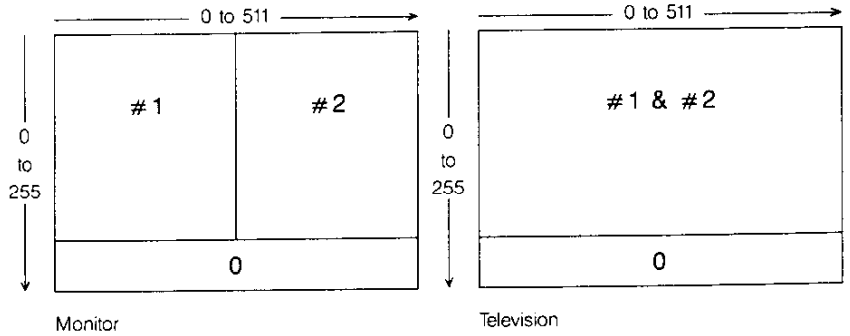
In the *high-resolution* mode each colour can be selected by one of two codes. You will see later how a startling range of colour and stipple (texture) effects can be produced if you have a good quality colour monitor.

Some of the screen presentation keywords are as follows:

INK *colour*	foreground colour
BORDER *width, colour*	draw border at edge of screen or window
PAPER *colour*	background colour
BLOCK *width, height, across, down, colour*	colour a rectangle which has its top left hand corner at position across, down

SCREEN ORGANISATION

When you switch on your QL the screen display is split into three areas called *windows* as shown below. Note than in order to fit these windows into the area covered by a television screen, some pixels around the border are not used in Television mode.



The windows are identified by #0, #1 and #2 so that you can relate various effects to particular windows. For example:

```
CLS
```

will clear window #1 (the system chooses) so if you want the left hand area cleared you must type:

```
CLS #2
```

If you want a different paper (background colour) type for green:

```
PAPER 4 : CLS
```

or

```
PAPER #2,4 : CLS #2
```

if you want to clear window #2 to the background colour green.

The numbers #0, #1, #2 are called *channel numbers*. In this particular case they enable you to direct certain effects to the window of your choice. You will discover later that channel numbers have many other uses but for the moment note that all of the following statements may have a channel number. The third column shows the default channel – the one chosen by the system if you do not specify one.

Note that windows may overlap. If you use a TV screen the system automatically overlaps windows #1 and #2 so that more character positions per line are available for program listings.

Keyword	Effect	Default
AT	Character Position	#1
BLOCK	Draws block	#1
BORDER	Draw border	#1
CLS	Clear screen	#1
CSIZE	Character size	#1
CURSOR	Position cursor	#1
FLASH	Causes/cancels flashing	#1
INK	Foreground colour	#1
OVER	Effect of printing and graphics	#1
PAN	Moves screen sideways	#1
PAPER	Background colour	#1
RECOL	Changes colour	#1
SCROLL	Moves screen vertically	#1
STRIP	Background for printing	#1
UNDER	Underlines	#1
WINDOW	Changes existing window	#1
LIST	Lists program	#2
DIR	Lists directory	#1
PRINT	Prints characters	#1
INPUT	Takes keyboard input	#1

Statements or direct commands appear in window #0.

For more detail about the syntax or use of these keywords see other parts of the manual.

RECTANGLES AND LINES

The program below draws a green rectangle in 256 mode on red paper with a yellow border one pixel wide. The rectangle has its top left corner at pixel co-ordinates 100,100 (see *QL Concepts*). Its width is 80 units across (40 pixels) and its height is 20 units down (20 pixels).

```
100 REMark Rectangle
110 MODE 256
120 BORDER 1,6
130 PAPER 2 : CLS
140 BLOCK 80,20,100,100,4
```

You have to be a bit careful in mode 256 because across values range from 0 to 511 even though there are only 256 pixels. We cannot say that the block produced by the above program is 80 pixels wide so we say 80 units.

INPUT AND OUTPUT

SuperBASIC has the usual **LET**, **INPUT**, **READ** and **DATA** statements for input. The **PRINT** statement handles most text output in the usual way with the separators:

, tabulates output

; just separates - no formatting effect

\ forces new line

! normally provides a space but not at the start of line. If an item will not fit at the end of a line it performs a new line operation.

TO Allows tabulation to a designated column position.

LOOPS

You will be familiar with two types of repetitive loop exemplified as follows:

(a) Simulate 6 throws of an ordinary six-sided die.

```
100 FOR throw = 1 TO 6
110 PRINT RND(1 TO 6)
120 NEXT throw
```

(b) Simulate throws of a die until a six appears.

```
100 die = RND(1 TO 6)
110 PRINT die
120 IF die <> 6 THEN GOTO 10
```

Both of these programs will work in SuperBASIC but we recommend the following instead. They do exactly the same jobs. Although program (b) is a little more complex there are good reasons for preferring it.

(a)
```
100 FOR throw = 1 TO 6
110   PRINT RND(1 TO 6)
120 END FOR throw
```

(b)
```
100 REPeat throws
110   die = RND(1 TO 6)
120   PRINT die
130   IF die = 6 THEN EXIT throws
140 END REPeat throws
```

FOR SERVICE MANUALS
CONTACT:
MAURITRON TECHNICAL SERVICES
www.mauritron.co.uk
TEL: 01844 - 351694
FAX: 01844 - 352554

It is logical to provide a structure for a loop which terminates on a condition (**REPeat** loops) as well as those which are controlled by a count.

The fundamental **REPeat** structure is:

REPeat *identifier*
  statements
END REPeat *identifier*

The **EXIT** statement can be placed anywhere in the structure but it must be followed by an identifier to tell SuperBASIC which loop to exit; for example:

EXIT throws

would transfer control to the statement after

END REPeat throws.

This may seem like a using a sledgehammer to crack the nut of the simple problem illustrated. However the **REPeat** structure is very powerful. It will take you a long way.

If you know other languages you may see that it will do the jobs of both **REPEAT** and **WHILE** structures and also cope with other, more awkward, situations.

The SuperBASIC **REPeat** loop is named so that a correct clear exit is made. The **FOR** loop, like all SuperBASIC structures, ends with **END**, and its name is given for reasons which will become clear later.

You will also see later how these loop structures can be used in simple or complex situations to match exactly what you need to do. We will mention only three more features of loops at this stage. They will be familiar if you are an experienced user of BASIC.

The increment of the control variable of a **FOR** loop is normally 1 but you can make it other values by using the **STEP** keyword. As the examples show.

i.
```
100 FOR even = 2 TO 10 STEP 2
110 PRINT ! even !
120 END FOR even
```
output is 2 4 6 8 10

ii.
```
100 FOR backwards = 9 TO 1 STEP -1
110 PRINT ! backwards !
120 END FOR backwards
```
output is 9 8 7 6 5 4 3 2 1

The second feature is that loops can be nested. You may be familiar with nested **FOR** loops. For example the following program outputs four rows of ten crosses.

```
100 REMark Crosses
110 FOR row = 1 TO 4
120   PRINT 'Row number'! row
130   FOR cross = 1 TO 10
140     PRINT !'X'!
150   END FOR cross
160   PRINT
170 PRINT \ 'End of row number'! row
180 END FOR row
```

output is

```
Row number 1
X X X X X X X X X X
End of row number 1
Row number 2
X X X X X X X X X X
End of row number 2
Row number 3
X X X X X X X X X X
End of row number 3
Row number 4
X X X X X X X X X X
End of row number 4
```

A big advantage of SuperBASIC is that it has structures for all purposes, not just **FOR** loops, and they can all be nested one inside the other reflecting the needs of a task. We can put a **REPeat** loop in a **FOR** loop. The program below produces scores of two dice in each row until a seven occurs, instead of crosses.

```
100 REMark Dice rows
110 FOR row = 1 TO 4
120   PRINT 'Row number '! row
130   REPeat throws
140     LET die1 = RND(1 TO 6)
150     LET die2 = RND(1 TO 6)
160     LET score = die 1 + die2
170     PRINT ! score !
180     IF score = 7 THEN EXIT throws
190   END REPeat throws
200   PRINT \'End of row' ! row
210 END FOR row
```

sample output:

```
Row number 1
8 11 6 3 7
End of row number 1
Row number 2
4 6 2 9 4 5 12 7
End of row number 2
Row number 3
7
End of row number 3
Row number 4
6 2 4 9 9 7
End of row number 4
```

The third feature of loops in SuperBASIC allows more flexibility in providing the range of values in a **FOR** loop. The following program illustrates this by printing all the divisible numbers from 1 to 20. (A divisible number is divisible evenly by a number other than itself or 1.)

```
100 REMark Divisible numbers
110 FOR num = 4,6,8, TO 10,12,14 TO 16,18,20
120   PRINT ! num !
130 END FOR num
```

More will be said about handling repetition in a later chapter but the features described above will handle all but a few uncommon or advanced situations.

DECISION MAKING

You will have noticed the simple type of decision:

```
IF die = 6 THEN EXIT throws
```

This is available in most BASICs but SuperBASIC offers extensions of this structure and a completely new one for handling situations with more than two alternative courses of action.

However, you may find the following long forms of **IF** ... **THEN** useful. They should explain themselves.

i.
```
100 REMark Long form IF...END IF
110 LET sunny = RND(0 TO 1)
120 IF sunny THEN
130   PRINT 'Wear sunglasses'
140   PRINT 'Go for walk'
150 END IF
```

ii.
```
100 REMark Long form IF...ELSE...END IF
110 LET sunny = RND(0 TO 1)
120 IF sunny THEN
130   PRINT 'Wear sunglasses'
140   PRINT 'Go for walk'
150 ELSE
160   PRINT 'Wear coat'
170   PRINT 'Go to cinema'
180 END IF
```

The separator, **THEN**, is optional in long forms or it can be replaced by a colon in short forms. The long decision structures have the same status as loops. You can nest them or put other structures into them. When a single variable appears where you expect a condition the value zero will be taken as false and other values as true.

SUBROUTINES AND PROCEDURES

Most BASICs have a **GOSUB** statement which may be used to activate particular blocks of code called subroutines. The **GOSUB** statement is unsatisfactory in a number of ways and SuperBASIC offers properly named procedures with some very useful features.

Consider the following programs both of which draw a green 'square' of side length 50 pixel screen units at a position 200 across 100 down on a red background.

(a) Using GOSUB

```
100 LET colour = 4 : background = 2
110 LET across = 20
120 LET down = 100
130 LET side = 50
140 GOSUB 170
150 PRINT 'END'
160 STOP
170 REMark Subroutine to draw square
180 PAPER background : CLS
190 BLOCK side, side, across, down, colour
200 RETurn
```

(b) Using a procedure with parameters

```
100 square 4, 50, 20, 100, 2
110 PRINT 'END'
120 DEFine PROCedure square(colour,side,across,down,background)
130   PAPER background : CLS
140   BLOCK side, side, across, down, colour
150 END DEFine
```

In the first program the values of *colour, across, down, side* are fixed by **LET** statements before the **GOSUB** statement activates lines 180 and 190. Control is then sent back by the **RETURN** statement.

In the second program the values are given in the first line as parameters in the procedure call, *square*, which activates the procedure and at the same time provides the values it needs.

In its simplest form a procedure has no parameters. It merely separates a particular piece of code, though even in this simpler use the procedure has the advantage over **GOSUB** because it is properly named and properly isolated into a self-contained unit.

The power and simplifying effects of procedures are more obvious as programs get larger. What procedures do, as programs get larger, is not so much make programming easier as prevent it from getting harder with increasing program size. The above example just illustrates the way they work in a simple context.

Examples

The following examples indicate the range of vocabulary and syntax of SuperBASIC which has been covered in this and earlier chapters, and will form a foundation on which the second part of this manual will build.

The letters of a palindrome are given as single items in **DATA** statements. The terminating item is an asterisk and you assume no knowledge of the number of letters in the palindrome. **READ** the letters into an array and print them backwards. Some palindromes such as 'MADAM I'M ADAM' only work if spaces and punctuation are ignored. The one used here works properly.

```
100 REMark Palindromes
110 DIM text$(30)
120 LET text$ = FILL$ (' ',30)
130 LET count = 30
140 REPeat get_letters
150   READ character$
160   IF character$ = '*' THEN EXIT get_letters
170   LET count = count-1
180 LET text$(count) = character$
190 END REPeat get_letters
200 PRINT text$
210 DATA 'A','B','L','E' ,'W','A','S' ,'I' ,'E','R'
220 DATA 'E' ,'I' ,'S','A','W' ,'E','L','B','A','*'
```

The following program accepts as input numbers in the range 1 to 3999 and converts them into the equivalent in Roman numerals. It does not generate the most elegant form. It produces IIII rather than IV.

```
100 REMark Roman numbers
110 INPUT number
120 RESTORE 210
130 FOR type = 1 TO 7
140 READ letter$, value
150   REPeat output
160     IF number < value : EXIT output
170     PRINT letter$;
180     LET number = number - value
190   END REPeat output
200 END FOR type
210 DATA 'M',1000,'D',500,'C',100,'L',50,'X',10,'V',5,'I',1
```

You should study the above examples carefully using dry runs if necessary until you are sure that you understand them.

CONCLUSION

In SuperBASIC full structuring features are provided so that program elements either follow in sequence or fit into one another neatly. All structures must be identified to the system and named. There are many unifying and simplifying features and many extra facilities.

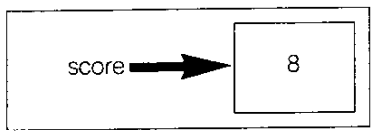
Most of these are explained and illustrated in the remaining chapters of this manual, which should be easier to read than the *Keyword* and *Concept Reference* sections. However, it is easier to read because it does not give every technical detail and exhaust every topic which it treats. There may, therefore, be a few occasions when you need to consult the reference sections. On the other hand some major advances are discussed in the following chapters. Few readers will need to use all of them and you may find it helpful to omit certain parts, at least on first reading.

CHAPTER 9

DATA TYPES VARIABLES AND IDENTIFIERS

You will have noticed that a program (a sequence of statements) usually gets some data to work on (input) and produces some kind of results (output). You will also have understood that there are internal arrangements for storing this data. In order to avoid unnecessary technical explanations we have suggested that you imagine pigeon holes and that you choose meaningful names for the pigeon holes. For example, if it is necessary to store a number which represents the score from simulated dice-throws you imagine a pigeon hole named score which might contain a number such as 8.

Internally the pigeon holes are numbered and the system maintains a dictionary which connects particular names with particular numbered pigeon holes. We say that the name, *score*, points to its particular pigeon-hole (by means of the internal dictionary).



The whole arrangement is called a **variable**.

What you see is the word *score*. We say that this word, *score* is an identifier. It is what we see and it identifies the concept we need, in this case the result, 8, of throwing a pair of dice. Because the identifier is what we see it becomes the thing we talk or write or think about. We write about score and its value at any particular moment.

There are four simple data types called floating point, integer, string and logical and these are explained below. We talk about data types rather than variable types because data can occur on its own, for example 3.4 or 'Blue hat' as the value of a variable. But if you understand the different types of variables, you must also understand the different types of data.

IDENTIFIERS AND VARIABLES

1. A SuperBASIC identifier must begin with a letter and is a sequence of:

   upper or lower case letters
   digits or underscore

2. An identifier may be up to 255 characters in length so there is no effective limit in practice.
3. An identifier cannot be the same as a keyword of SuperBASIC.
4. An integer variable name is an identifier with % on the end.
5. A string variable name is an identifier with $ on the end.
6. No other identifiers must use the symbols % and $.
7. An identifier should usually be chosen so that it means something to a human reader, but for SuperBASIC it does not have any particular meaning other than that it identifies certain things.

FLOATING POINT VARIABLES

Examples of the use of floating point variables are:

```
100 LET days = 24
110 LET sales = 3649.84
120 LET sales_per_day = sales/days
130 PRINT sales_per_day
```

The value of a floating point variable may be anything in the range:

$\pm 10^{-615}$ to $\pm 10^{+615}$ with 8 significant figures.

Suppose in the above program sales were, exceptionally, only 3p. Change line 110 to:

```
110 LET sales = 0.03
```

This system will change this to:

```
110 LET sales = 3E-2
```

To interpret this, start with 3 or 3.0 and move the decimal point −2 places, i.e. two places left. This shows that:

3E−2 is the same as 0.03

After running the program, the average daily sales are:

1.25E−3 which is the same as 0.00125

Numbers with an E are said to be in exponent form:

(mantissa) E (exponent) = (mantissa) x 10 to the power (exponent)

INTEGER VARIABLES

Integer variables can have only whole number values in the range -32678 to 32768 The following are examples of valid integer variable names which must end with %.

```
LET count% = 10
LET six_tally% = RND(10)
LET number_3% = 3
```

The only disadvantage of integer variables, when whole numbers are required, is the slightly misleading % symbol on the end of the identifier. It has nothing to do with the concept of percentage. It is just a convenient symbol tagged on to show that the variable is an integer.

NUMERIC FUNCTIONS

Using a function is a bit like making an omelette. You put in an egg which is processed according to certain rules (the recipe) and get out an omelette. For example the function INT takes any number as input and outputs the whole number part. Anything which is input to a function is called a parameter or argument. INT is a function which gives the integer part of an expression. You may write:

```
PRINT INT(5.6)
```

and 5 would be the output. We say that 5.6 is the parameter and the function returns the value 5. A function may have more than one parameter. You have already met:

```
RND(1 TO 6)
```

which is a function with two parameters. But functions always return exactly one value. This must be so because you can put functions into expressions. For example:

```
PRINT 2 * INT(5.6)
```

would produce the output 10. It is an important property of functions that you can use them in expressions. It follows that they must return a single value which is then used in the expression. INT and RND are system functions; they come with the system, but later you will see how to write your own.

The following examples show common uses of the INT function.

```
100 REMark Rounding
110 INPUT decimal
120 PRINT INT(decimal + 0.5)
```

In the example you input a decimal fraction and the output is rounded. Thus 4.7 would become 5 but 4.3 would become 4.

You can achieve the same result using an integer variable and coercion.

Trigonometrical functions will be dealt with in a later section but other common numeric functions are given in the list below.

Function	Effect	Examples	Returned values
ABS	Absolute or unsigned value	ABS(7) ABS(−4.3)	7 4.3
INT	Integer part of a floating point number	INT(2.4) INT(0.4) INT(−2.7)	2 0 −3
SQRT	Square root	SQRT(2) SQRT(16) SQRT(2.6)	1.414214 4 1.612452

There is a way of computing square roots which is easy to understand. To compute the square root of 8 first make a guess. It doesn't matter how bad the guess maybe. Suppose you simply take half of 8 as the first guess which is 4.

Because 4 is greater than the square root of 8 then 8 / 4 must be less than it. The reverse is also true. If you had guessed 2 which is less than the square root then 8 / 2 must be greater than it.

It follows that if we take any guess and compute number / guess we have two numbers, one too small and one too big. We take the average of these numbers as our next approximation and thus get closer to the correct answer.

We repeat this process until successive approximations are so close as to make little difference.

```
100 REMark Square Roots
110 LET number = 8
120 LET approx = number/2
130 REPeat root
140   LET newval = (approx + number/approx) /2
150   IF newval == approx THEN EXIT root
160   LET approx = newval
170 END REPeat root
180 PRINT 'Square root of' ! number ! 'is' ! newval
```

sample output

```
Square root of 8 is 2.828427
```

Notice that the conditional **EXIT** from the loop must be in the middle. The traditional structures do not cope with this situation as well as SuperBASIC does.

The == sign in line 150 means "approximately equal to", that is equal to within .0000001 of the values being compared.

NUMERIC OPERATIONS

SuperBASIC allows the usual mathematical operations. You may notice that they are like functions with exactly two operands each. It is also conventional in these cases to put an operand on each side of the symbol. Sometimes the operation is denoted by a familiar symbol such as + or *. Sometimes the operation is denoted by a keyword like **DIV** or **MOD** but there is no real difference. Numeric operations have an order of priority. For example, the result of:

```
PRINT 7 + 3*2
```

is 13 because the multiplication has a higher priority. However:

```
PRINT (7 + 3)*2
```

will output 20, because brackets over-ride the usual priority. As you will see later so many things can be done with SuperBASIC expressions that a full statement about priority cannot be made at this stage (see the *Concept Reference Guide* if you wish) but the operations we now deal with have the following order of priority:

highest – raising to a power
multiplication and division (including **DIV**, **MOD**)
lowest – add and subtract

The symbols + and – are also used with only one operand which simply denotes positive or negative. Symbols used in this way have the highest priority of all and can only be over-ridden by the use of brackets.

Finally if two symbols have equal priority the leftmost operation is performed first so that:

```
PRINT 7-2 + 5
```

will cause the subtraction before the addition. This might be important if you should ever deal with very large or very small numbers.

Operation	Symbol	Examples	Results	Note
Add	+	7+6.6	13.6	
Subtract	–	7–6.6	0.4	
Multiply	*	3*2.1 2.1*(–3)	6.3 –6.3	
Divide	/	7/2 –17/5	3.5 –3.4	Do not divide by zero
Raise to power	^	4^1.5	8	
Integer divide	DIV	–8 DIV 2 7 DIV 2	–4 3	Integers only Do not divide by zero
Modulus	MOD	13 MOD 5 21 MOD 7 –17 MOD 8	3 0 7	

Modulus returns the remainder part of a division. Any attempt to divide by zero will generate an error and terminate program exection.

NUMERIC EXPRESSIONS

Strictly speaking, a numeric expression is an expression which evaluates to a number and there are more possibilities than we need to discuss here. SuperBASIC allows you to do complex things if you want to but it also allows you to do simple things in simple ways. In this section we concentrate on those usual straightforward uses of mathematical features.

Basically numeric expressions in SuperBASIC are the same as those of mathematics but you must put the whole expression in the form of a sequence.

$$\frac{5 + 3}{6 - 4}$$

becomes in SuperBASIC (or other BASIC):

```
(5 + 3)/(6 - 4)
```

In secondary school algebra there is an expression for one solution of a quadratic equation:

$$ax^2 + bx + c = 0$$

One solution in mathematical notation is:

$$x = \frac{-b + \sqrt{b^2 - 4ac}}{2a}$$

If we start with the equation:

$$2x^2 - 3x + 1 = 0$$

The following program will find one solution.

Example 1

```
100 READ a,b,c
110 PRINT 'Root is' ! (-b +SQRT(b^2 - 4*a*c))/(2*a)
120 DATA 2,-3,1
```

Example 2

In problems which need to simulate the dealing of cards you can make cards correspond to the numbers 1 to 52 as follows:

1 to 13	Ace, two.....king of hearts
14 to 26	Ace, two.....king of clubs
27 to 39	Ace, two.....king of diamonds
40 to 52	Ace, two.....king of spades

A particular card can be identified as follows:

```
100 REM Card identification
110 LET card = 23
120 LET suit = (card-1) DIV 13
130 LET value = card MOD 13
140 IF value = 0 THEN LET value = 13
150 IF value = 1 THEN PRINT "Ace of ";
160 IF value >= 2 AND value <= 10 THEN PRINT value ! "of ";
170 IF value = 11 THEN PRINT "Jack of ";
180 IF value = 12 THEN PRINT "Queen of ";
190 IF value = 13 THEN PRINT "King of ";
200 IF suit = 0 THEN PRINT "hearts"
210 IF suit = 1 THEN PRINT "clubs"
220 IF suit = 2 THEN PRINT "diamonds"
230 IF suit = 3 THEN PRINT "spades"
```

There are new ideas in this program. They are in line 160. The meaning is clearly that the number is actually printed only if two logical statements are true. These are:

value is greater than or equal to 2 **AND** value is less than or equal to 10

Cards outside this range are either aces or 'court cards' and must be treated differently.

Note also the use of ! in the PRINT statement to provide a space and ; to ensure that output continues on the same line.

There are two groups of mathematical functions which we have not discussed here. They are the trigonometric and logarithmic. You may need the former in organising screen displays. Types of functions are also fully defined in the reference section.

LOGICAL VARIABLES

Strictly speaking, SuperBASIC does not allow logical variables but it allows you to use other variables as logical ones. For example you can run the following program:

```
100 REMark Logical Variable
110 LET hungry = 1
120 IF hungry THEN PRINT "Have a bun"
```

You expect a logical expression in line 120 but the numeric variable, hungry, is there on its own. The system interprets the value, 1, of hungry as true and the output is:

```
Have a bun
```

If line 110 read:

```
LET hungry = 0
```

there would be no output. The system interprets zero as false and all other values as true. That is useful but you can disguise the numeric quality of hungry by writing:

```
100 REMark Logical Variable
110 LET true = 1 : false = 0
120 LET hungry = true
130 IF hungry THEN PRINT "Have a bun"
```

STRING VARIABLES

There is much to be said about handling strings and string variables and this is left to a separate chapter.

PROBLEMS ON CHAPTER 9

1. A rich oil dealer gambles by tossing a coin in the following way. If it comes down heads he gets 1. If it comes down tails he throws again but the possible reward is doubled. This is repeated so that the rewards are as shown.

THROW	1	2	3	4	5	6	7
REWARDS	1	2	4	8	16	32	64

   By simulating the game try to decide what would be a fair initial payment for each such game:

   (a) if the player is limited to a maximum of seven throws per game.

   (b) if there is no maximum number of throws.

2. Bill and Ben agree to gamble as follows. At a given signal each divides his money into two halves and passes one half to the other player. Each then divides his new total and passes half to the other. Show what happens as the game proceeds if Bill starts with 16p and Ben starts with 64p.

3. What happens if the game is changed so that each hands over an amount equal to half of what the other possesses?

4. Write a program which forms random three-letter words chosen from A,B,C,D and prints them until 'BAD' appears.

5. Modify the last program so that it terminates when any real three letter word appears.

FOR SERVICE MANUALS
CONTACT:
MAURITRON TECHNICAL SERVICES
www.mauritron.co.uk
TEL: 01844 - 351694
FAX: 01844 - 352554

CHAPTER 10
LOGIC

If you have read previous chapters you will probably agree that repetition, decision making and breaking tasks into sub-tasks are major concepts in problem analysis, program design and encoding programs. Two of these concepts, repetition and decision making, need logical expressions such as those in the following program lines:

```
IF score = 7 THEN EXIT throws
IF suit = 3 THEN PRINT "spades"
```

The first enables **EXIT** from a **REPeat** loop. The second is simply a decision to do something or not. A mathematical expression evaluates to one of millions of possible numeric values. Similarly, a string expression can evaluate to millions of possible strings of characters. You may find it strange that logical expressions, for which great importance is claimed, can evaluate to one of only two possible values: *true or false*.

In the case of:

```
score = 7
```

this is obviously correct. Either score equals 7 or it doesn't ! The expression must be true or false – assuming that it's not meaningless. It may be that you do not know the value at some time, but that will be put right in due course.

You have to be a bit more careful of expressions involving words such as **OR**, **AND**, **NOT** but they are well worth investigating – indeed, they are essential to good programming. They will become even more important with the trend towards other kinds of languages based more on precise descriptions of what you require rather than what the computer must do.

AND

The word **AND** in SuperBASIC is like the word 'and' in ordinary English. Consider the following program.

```
100 REMark AND
110 PRINT "Enter two values" \ "1 for TRUE or 0 for FALSE"
120 INPUT raining, hole_in_roof
130 IF raining AND hole_in_roof THEN PRINT "Get wet"
```

As in real life, you will only get wet if it is raining and there is a hole in the roof. If one (or both) of the simple logical variables

raining
hole_in_roof

is false then the compound logical expression

raining **AND** *hole_in_roof*

is also false. It takes two true values to make the whole expression true. This can be seen from the rules below. Only when the compound expression is true do you get wet.

raining	hole_in_roof	raining AND hole_in_roof	effect
FALSE	FALSE	FALSE	DRY
FALSE	TRUE	FALSE	DRY
TRUE	FALSE	FALSE	DRY
TRUE	TRUE	TRUE	WET

Rules for AND

OR

In everyday life the word 'or' is used in two ways. We can illustrate the inclusive use of **OR** by thinking of a cricket captain looking for players. He might ask "Can you bat or bowl?" He would be pleased if a player could do just one thing well but he would also be pleased if someone could do both. So it is in programming: a compound expression using **OR** is true if either or both of the simple statements or variables are true. Try the following program.

```
100 REMark OR test
110 PRINT "Enter two values" \ "1 for TRUE or 0 for FALSE"
120 INPUT "Can you bat?", batsman
130 INPUT "Can you bowl?", bowler
140 IF batsman OR bowler THEN PRINT "In the team"
```

You can see the effects of different combinations if answers in the rules below:

batsman	bowler	batsman OR bowler	effect
FALSE	FALSE	FALSE	not in team
FALSE	TRUE	TRUE	in the team
TRUE	FALSE	TRUE	in the team
TRUE	TRUE	TRUE	in the team

Rules for OR

When the **inclusive OR** is used a true value in either of the simple statements will produce a true value in the compound expression. If Ian Botham, the England all-rounder, were to answer the questions both as a bowler and as a batsman, both simple statements would be true and so would the compound expression. He would be in the team.

If you write 0 for false and 1 for true you will get all the possible combinations by counting in binary numbers:

00
01
10
11

The word **NOT** has the obvious meaning.

NOT

NOT *true* is the same as *false*
NOT *false* is the same as *true*

However you need to be careful. Suppose you hold a red triangle and say that it is:

NOT *red* **AND** *square*

In English this may be ambiguous.

If you mean:

(**NOT** *red*) **AND** *square*

then for a red triangle the expression is false.

If you mean:

NOT (*red* **AND** *square*)

then for a red triangle the whole expression is true. There must be a rule in programming to make it clear what is meant. The rule is that **NOT** takes precedence over **AND** so the interpretation:

(**NOT** *red*) **AND** *square*

is the correct one. This is the same as:

NOT *red* **AND** *square*

To get the other interpretation you must use brackets. If you need to use a complex logical expression it is best to use brackets and **NOT** if their usage naturally reflects what you want. But you can if you wish always remove brackets by using the following laws (attributed to Augustus De Morgan)

NOT (a **AND** b)	is the same as	**NOT** a **OR NOT** b
NOT (a **OR** b)	is the same as	**NOT** a **AND NOT** b

For example:

NOT (*tall* **AND** *fair*) is the same as
NOT *tall* **OR NOT** *fair*

NOT (*hungry* **OR** *thirsty*) is the same as
NOT *hungry* **AND NOT** *thirsty*

Test this by entering:

```
100 REMark NOT and brackets
110 PRINT "Enter two values"\"1 for TRUE or 0 for FALSE"
120 INPUT "tall"; tall
130 INPUT "fair"; fair
140 IF NOT (tall AND fair) THEN PRINT "FIRST"
150 IF NOT tall OR NOT fair THEN PRINT "SECOND"
```

Whatever combination of numbers you give as input, the output will always be either two words or none, never one. This will suggest that the two compound logical expressions are equivalent.

XOR–Exclusive OR

Suppose a golf professional wanted an assistant who could either run the shop or give golf lessons. If an applicant turned up with both abilities he might not get the job because the golf professional might fear that such an able assistant would try to take over. He would accept a good golfer who could not run the shop. He would also accept a poor golfer who could run the shop. This is an exclusive OR situation: either is acceptable but not both. The following program would test applicants:

```
100 REMark XOR test
110 PRINT "Enter 1 for yes or 0 for no."
120 INPUT "Can you run a shop?", shop
130 INPUT "Can you teach golf?", golf
140 IF shop XOR golf THEN PRINT "Suitable"
```

The only combinations of answers that will cause the output "Suitable" are (0 and 1) or (1 and 0). The rules for **XOR** are given below.

Able to run shop	Able to teach	Shop XOR teach	effect
FALSE	FALSE	FALSE	no job
FALSE	TRUE	TRUE	gets the job
TRUE	FALSE	TRUE	gets the job
TRUE	TRUE	FALSE	no job

rules for XOR

PRIORITIES

The order of priority for the logical operators is (highest first):

NOT
AND
OR,XOR

For example the expression

rich OR *tall* AND fair

means the same as:

rich OR (*tall* AND *fair*)

The **AND** operation is performed first. To prove that the two logical expressions have identical effects run the following program:

```
100 REMark Priorities
110 PRINT "Enter three values"\"Type 1 for Yes and 0 for No"!
120 INPUT rich,tall,fair
130 IF rich OR tall AND fair THEN PRINT "YES"
140 IF rich OR (tall AND fair) THEN PRINT "AYE"
```

Whatever combination of three zeroes or ones you input at line 120 the output will be either nothing or:

```
YES
AYE
```

You can make sure that you test all possibilities by entering data which forms eight three-digit binary numbers 000 to 111:

```
000 001 010 011 100 101 110 111
```

PROBLEMS ON CHAPTER 10

1. Place ten numbers in a DATA statement. READ each number and if it is greater than 20 then print it.
2. Test all the numbers from 1 to 100 and print only those which are perfect squares or divisible by 7.
3. Toys are described as Safe (S), or Unsafe (U), Expensive (E) or Cheap (C), and either for Girls (G), Boys (B) or Anyone (A). A trio of letters encodes the qualities of each toy. Place five such trios in a DATA statement and then search it printing only those which are safe and suitable for girls.
4. Modify program 3 to print those which are expensive and not safe.
5. Modify program 3 to print those which are safe, not expensive and suitable for anyone.

CHAPTER 11
HANDLING TEXT - STRINGS

You have used string variables to store character strings and you know that the rules for manipulating string variables or string constants are not the same as those for numeric variables or numeric constants. SuperBASIC offers a full range of facilities for manipulating character strings effectively. In particular the concept of string-slicing both extends and simplifies the business of handling substrings or slices of a string.

ASSIGNING STRINGS

Storage for string variables is allocated as it is required by a program. For example, the lines:

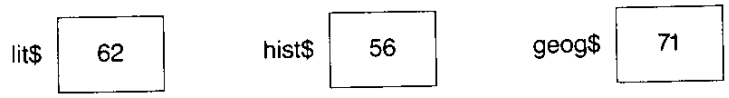
```
100 LET words$ = "LONG"
110 LET words$ = "LONGER"
120 PRINT words$
```

would cause the six letter word, LONGER, to be printed. The first line would cause space for four letters to be allocated but this allocation would be overruled by the second line which requires space for six characters.

It is, however, possible to dimension (i.e. reserve space for) a string variable, in which case the maximum length becomes defined, and the variable behaves as an array.

JOINING STRINGS

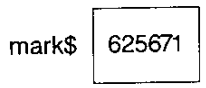
You may wish to construct records in data processing from a number of sources. Suppose, for example, that you are a teacher and you want to store a set of three marks for each student in Literature, History, and Geography. The marks are held in variables as shown:



As part of student record keeping you may wish to combine the three string values into one six-character string called *mark$*. You simply write:

```
LET mark$ = lit$ & hist$ & geog$
```

You have created a further variable as shown:

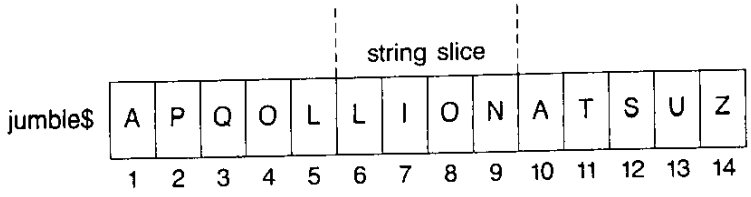


But remember that you are dealing with a character string which happens to contain number characters rather than an actual number. Note that in SuperBASIC the & symbol is used to join strings together, whereas in some other BASICs, the + symbol is used for that purpose.

COPY A STRING SLICE

A string slice is part of a string. It may be anything from a single character to the whole string. In order to identify the string slice you need to know the positions of the required characters.

Suppose you are constructing a children's game in which they have to recognise a word hidden in a jumble of letters. Each letter has an internal number – an index – corresponding to its position in the string. Suppose the whole string is stored in the variable *jumble$*, and the clue is Big cat.



You can see that the answer is defined by the numbers 6 to 9 which indicate where it is. You can abstract the answer as shown:

```
100 jumble$ = "APQOLLIONATSUZ"
110 LET an$ = jumble$(6 TO 9)
120 PRINT an$
```

REPLACE A STRING SLICE

Now suppose that you wish to change the hidden animal into a bull. You can write two extra lines:

```
130 LET jumble$(6 TO 9) = "BULL"
140 PRINT jumble$
```

The output from the whole five-line program is:

```
LION
APQOLBULLATSUZ
```

All string variables are initially empty, they have length zero. If you attempt to copy a string into a string-slice which has insufficient length then the assignment may not be recognised by SuperBASIC.

If you wish to copy a string into a string-slice then it is best to ensure the destination string is long enough by padding it first with spaces.

```
100 LET subject$ = "ENGLISH MATHS COMPUTING"
110 LET student$ = "                "
120 LET student$(9 TO 13) = subject$(9 TO 13)
```

We say that "BULL" is a slice of the string "APQOLBULLATSUZ". The defining phrase:

```
(6 TO 9)
```

is called a **slicer**. It has other uses. Notice how the same notation may be used on both sides of the **LET** statement. If you want to refer to a single character it would be clumsy to write:

```
jumble$(6 TO 6)
```

just to pick out the "B" (possibly as a clue) so you can write instead :

```
jumble$(6)
```

to refer to a single character.

COERCION

Suppose you have a variable, *mark$* holding a record of examination marks. The slice giving the history mark may be extracted and scaled up, perhaps because the history teacher has been too strict in the marking. The following lines will extract the history mark:

```
100 LET mark$ = "625671"
110 LET hist$ = mark$(3 TO 4)
```

The problem now is that the value "56" of the variable, *hist$* is a string of characters not numeric data. If you want to scale it up by multiplying by, say, 1.125, the value of *hist$* must be converted to numeric data first, SuperBASIC will do this conversion automatically when we type:

```
120 LET num = 1.125 * hist$
```

Line 120 converts the string "56" to the number 56 and multiplies it by 1.125 giving 63.

Now we should replace the old mark by the new mark but now the new mark is still the number 63 and before it can be inserted back into the original string it must be converted back to the string '63'. Again SuperBASIC will convert the number automatically when we type:

```
130 LET mark$(3 TO 4) = num
140 PRINT mark$
```

The output from the whole program is:

```
626371
```

which shows the history mark increased to 63.

Strictly speaking it is illegal to mix data types in a LET statement. It would be silly to write:

```
LET num = "LION"
```

and you would get an error message if you tried, but if you write:

```
LET num = "65"
```

the system will conclude that you want the number 65 to become the value of num and do that. The complete program is:

```
100 LET mark$ = "625671"
110 LET hist$ = mark$(3 TO 4)
120 LET num = 1.125 * hist$
130 LET mark$(3 TO 4) = num
140 PRINT mark$
```

Again the output is the same!

In line 120 a string value was converted into numeric form so that it could be multiplied; In line 130 a number was converted into string form. This converting of data types is known as **type coercion.**

You can write the program more economically if you understand both string-slicing and coercion now:

```
100 LET mark$ = "625671"
110 LET mark$(3 TO 4) = 1.125 * mark$(3 TO 4)
120 PRINT mark$
```

If you have worked with other BASICs you will appreciate the simplicity and power of string-slicing and coercion.

SEARCHING A STRING

You can search a string for a given substring. The following program displays a jumble of letters and invites you to spot the animal.

```
100 REM Animal Spotting
110 LET jumble$ = "SYNDICATE"
120 PRINT jumble$
130 INPUT "What is the animal?" ! an$
140 IF an$ INSTR jumble$ AND an$(1) = "C"
150    PRINT "Correct"
150 ELSE
170    PRINT "Not correct"
180 END IF
```

The operator **INSTR**, returns zero if the guess is incorrect. If the guess is correct **INSTR** returns the number which is the starting position of the string-slice, in this case 6.

Because the expression:

```
an$ INSTR jumble$
```

can be treated as a logical expression the position of the string in a successful search can be regarded as true, while in an unsuccessful search it can be regarded as *false*.

OTHER STRING FUNCTIONS

You have already met **LEN** which returns the length (number of characters) of a string.

You may wish to repeat a particular string or character several times. For example, if you wish to output a row of asterisks, rather than actually enter forty asterisks in a **PRINT** statement or organise a loop you can simply write:

```
PRINT FILL$ ("*",40)
```

Finally it is possible to use the function **CHR$** to convert internal codes into string characters. For example:

```
PRINT CHR$(65)
```

would output A.

COMPARING STRINGS

A great deal of computing is concerned with organising data so that it can be searched quickly. Sometimes it is necessary to sort it in to alphabetical order. The basis of various sorting processes is the facility for comparing two strings to see which comes first. Because the letters A,B,C... are internally coded as 65,66,67... it is natural to regard as correct the following statements:

A is less than B
B is less than C

and because internal character by character comparison is automatically provided:

CAT is less than DOG
CAN is less than CAT

You can write, for example:

```
IF "CAT" < "DOG" THEN PRINT "MEOW"
```

and the output would be:

```
MEOW
```

Similarly:

```
IF "DOG" > "CAT" THEN PRINT "WOOF"
```

would give the output:

```
WOOF
```

FOR SERVICE MANUALS
CONTACT:
MAURITRON TECHNICAL SERVICES
www.mauritron.co.uk
TEL: 01844 - 351694
FAX: 01844 - 352554

We use the comparison symbols of mathematics for string comparisons. All the following logical statements expressions are both permissible and *true*.

```
"ALF" < "BEN"
"KIT" > "BEN"
"KIT" <= "LEN"
"KIT" >= "KIT"
"PAT" >= "LEN"
"LEN" <= "LEN"
"PAT" <> "PET"
```

So far, comparisons based simply on internal codes make sense, but data is not always conveniently restricted to upper case letters. We would like, for example:

Cat to be less than COT
and K2N to be less than K27N

A simple character by character comparison based on internal codes would not give these results, so SuperBASIC behaves in a more intelligent way. The following program, with suggested input and the output that will result, illustrates the rules for comparison of strings.

```
100 REMark comparisons
110 REPeat comp
120   INPUT "input a string" ! first$
130   INPUT "input another string" ! second$
140   IF first$ < second$ THEN PRINT "Less"
150   IF first$ > second$ THEN PRINT "Greater"
160   IF first$ = second$ THEN PRINT "Equal"
170 END REPeat comp
```

input		output
CAT	COT	Greater
CAT	CAT	Equal
PET	PETE	Less
K6	K7	Less
K66	K7	Greater
K12N	K6N	Greater

> Greater than – Case dependent comparision, numbers compared in numerical order.

< Less than – Case dependent, numbers compared in numerical order

= Equals – Case dependent, strings must be the same

== Equivalent – String must be 'almost' the same, Case independent, numbers compared in numerical order.

>= Greater than or equal to – Case dependent, numbers compared in numerical order

<= Less than or equal to – Case dependent, numbers compared in numerical order.

PROBLEMS ON CHAPTER 11

1. Place 12 letters, all different, in a string variable and another six letters in a second string variable. Search the first string for each of the six letters in turn saying in each case whether it is found or not found.
2. Repeat using single character arrays instead of strings. Place twenty random upper case letters in a string and list those which are repeated.
3. Write a program to read a sample of text all in upper case letters. Count the frequency of each letter and print the results.

   "GOVERNMENT IS A TRUST, AND THE OFFICERS OF THE GOVERNMENT ARE TRUSTEES; AND BOTH THE TRUST AND THE TRUSTEES ARE CREATED FOR THE BENEFIT OF THE PEOPLE. – HENRY CLAY, 1829."

4. Write a program to count the number of words in the following text. A word is recognised because it starts with a letter and is followed by a space, full stop or other punctuation character.

   "THE REPORTS OF MY DEATH ARE GREATLY EXAGGERATED. – CABLE FROM MARK TWAIN TO THE ASSOCIATED PRESS, LONDON 1896."

5. Rewrite the last program illustrating the use of logical variables and procedures.

CHAPTER 12
SCREEN OUTPUT

SuperBASIC has so extended the scope and variety of facilities for screen presentation that we describe the features in two sections: *Simple Printing* and *Screen*.

The first section describes the output of ordinary text. Here we explain the minimal well established methods of displaying messages, text, or numerical output. Even in this mundane section there is innovation in the concept of the 'intelligent' space – an example of combining ease of use with very useful effects.

The second section is much bigger because it has a great deal to say. The wide range of features actually makes things easier. For example, you can draw a circle by simply writing the word **CIRCLE** followed by a few details to define such things as its position and size. Many other systems require you to understand some geometry and trigonometry in order to do what is, in concept, simple.

Each keyword has been carefully chosen to relect the effect it causes. **WINDOW** defines an area of the screen; **BORDER** puts a border round it; **PAPER** defines the background colour; **INK** determines the colour of what you put on the paper.

If you work through this chapter and get a little practice you will easily remember which keyword causes which effect. You will add that extra quality to your programming fairly easily. With experience you may see why computer graphics is becoming a new art form.

SIMPLE PRINTING

The keyword **PRINT** can be followed by a sequence of print items. A print item may be any of:

text such as: "This is text"
variables such as: *num*, *word$*
expressions such as: 3 * *num*, *day$* & *week$*

Print items may be mixed in any print statement but there must be one or more print separators between each pair. Print separators may be any of:

; No effect – it just separates print items.

! Normally inserts a space between output items. If an item will not fit on the current line it behaves as a new line symbol. If the item is at the start of line a space is not generated.

, A tabulator causes the output to be tabulated in columns of 8 characters

\ A new line symbol will force a new line.

TO Allows tabbing.

The numbers 1,2,3 are legitimate print items and are convenient for illustrating the effects of print separators

Statement	Effect
`100 PRINT 1,2,3`	`1 2 3`
`100 print 1!2!3!`	`1 2 3`

`100 PRINT 1\2\3`	`1`
	`2`
	`3`

`100 PRINT 1;2;3`	`123`
`100 PRINT "This is text"`	`This is text`
`100 LET word$ = " "` `110 PRINT word$`	moves print position
`100 LET num = 13` `110 PRINT num`	`13`
`100 LET an$ = "yes"` `110 PRINT "I say" ! an$`	`I say yes`
`110 PRINT "Sum is" ! 4 + 2`	`Sum is 6`

You can position print output anywhere on the screen with the AT command.

For example:

```
AT 10,15 : PRINT "This is on row 10 at column 15"
```

The CURSOR command can be used to position the print output anywhere on the screen's scale system. For example:

```
CURSOR 100,150 : PRINT "this is 100 pixel grid units across and
                        150 down"
```

If you read the *Keyword Reference Guide* you may find it difficult to reconcile the section on PRINT with the above description. Two of the difficulties disappear if you understand that:

Text in quotes, variables and numbers are all strictly speaking, expressions; they are the simplest (degenerate) forms of expressions.

Print separators are strictly classified as print items.

SCREEN

This section introduces general effects which apply whether you wish to output text or graphics. The statement:

```
MODE 8 or MODE 256
```

will select MODE 8 in which there are:

256 pixels across numbered 0–511 (two numbers per pixel)
256 pixels down numbered 0–255
8 colours

A pixel is the smallest area of colour which can be displayed. We use the term, **solid colour** because these start with ordinary solid-looking colours of which there are only eight. However, by using various effects a variety of shades and textures can be achieved. If you are using your QL with an ordinary television set then the television set will not be able to reproduce any of these extra effects.

The statement:

```
MODE 4 or MODE 512
```

will select MODE 4 in which there are:

512 pixels across numbered 0 to 511
256 pixels down numbered 0 to 255
4 colours

COLOUR

You can select a colour by using the following code in combination with suitable keywords such as PAPER, INK etc. Note that the numbers by themselves mean nothing. The numbers are only interpreted as colours when they are used with PAPER and INK, etc.

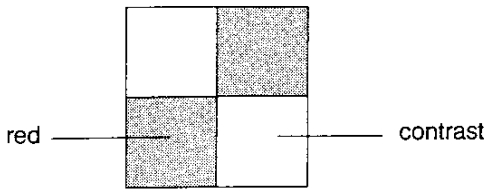
8 Colour Mode	Code	4 Colour Mode
black	0	black
blue	1	black
red	2	red
magenta	3	red
green	4	green
cyan	5	green
yellow	6	white
white	7	white

Colour Codes

For example INK 3 would give magenta in MODE 8.

STIPPLES

You can if you wish specify two colours in a suitable statement. For example 2,4 would give a chequerboard stipple as shown. In each group of four pixels two would be red (code 2) corresponding to the colour selected first. The other two pixels would be a contrast. It is not really possible to display this effect on a domestic television set.



If you write:

```
INK 2,4
```

the mix colour is formed from the two codes 2 and 4. We will call these choices colour and contrast!

```
INK colour, contrast
```

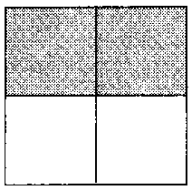
You can find out what the stipple effects are by trying them but we give more technical details below.

```
100 REMark Colour/Contrast
110 FOR colour = 0 TO 7 STEP 2
120   PAPER colour : CLS
140   FOR contrast = 0 TO 7 STEP 2
150     BLOCK 100,50,40,50,colour,contrast
160     PAUSE 50
170   END FOR contrast
180 END FOR colour
```

If you wish to try different stipples you can add a third code number to the colour specification. For example:

```
INK 2,4,1
```

would specify a red and green horizontal stripe effect. A block of four pixels would be:



The possible effects are shown using red and contrast .

Code	Name	Effect
0	Single pixel of contrast	
1	Horizontal Stripes	
2	Vertical Stripes	
3	Chequerboard	

Stipple Patterns

COLOUR PARAMETERS

You can specify a colour/stipple effect as described above by using three numbers. For example:

INK colour, contrast, stipple

could be used with:

colour in range 0 to 7
contrast in range 0 to 7
stipple in range 0 to 3

You could achieve the same effect with a single number if you wish though it is not so easy to construct. See the *Concept Reference Guide - colour*.

The following program will display all the possible colour effects:

```
100 REMark Colour Effects
110 FOR num = 0 TO 255
120   BLOCK 100,50,40,50,num
130   PAUSE 50
140 END FOR num
```

PAPER

PAPER followed by one, two or three numbers specifies the background. For example:

```
PAPER 2          {red}
PAPER 2,4        {red/green chequerboard}
PAPER 2,4,1      {red/green horizontal stripes}
```

The colour will not be visible until something else is done, for example, the screen is cleared by typing **CLS**.

INK

INK followed by one, two or three numbers specifies the colour for printing characters, lines or other graphics. The colour and stipple effects are the same as for **PAPER**. For example:

```
INK 2            {red ink}
INK 2,4          {red/green chequerboard ink 3}
INK 2,4,1        {red/green horizontal striped ink}
```

The ink will be changed for all subsequent output.

CLS

CLS means clear the window to the current paper colour – like a teacher cleaning a blackboard, except that it is electronic and multi-coloured.

FLASHING

You can make the ink colour flash in mode 8 only. To turn flash on you might type:

```
FLASH 1
```

and to turn it off:

```
FLASH 0
```

Allowing flashing characters to overlap can produce alarming results.

FILES

You will have used Microdrives for storing programs and you will have used the commands **LOAD** and **SAVE**. Cartridges can be used for storing data as well as programs. The word file usually means a sequence of data records, a record being some set of related information such as name, address and telephone number.

Two of the most widely used types of file are serial and direct access files. Items in a serial file are usually read in sequence starting with the first. If you want the fiftieth record you have to read the first forty-nine in order to find it. On the other hand the fiftieth record in a direct access file can be found quickly because the system does not need to work through the earlier records to get it. Pop music on a cassette is like a serial file but eight pieces on a long playing record form a direct access file. You can move the pick up arm directly onto any of the eight tracks.

The simplest possible type of file is just a sequence of numbers. To illustrate the idea we will place the numbers 1 to 100 in a file called *numbers*. However, the complete file name is made up of two parts:

device name
appended information

Suppose that we wish to create the file, *numbers*, on a cartridge in Microdrive 1. The device name is:

```
mdv1_
```

and the appended information is just the name of the file:

```
numbers
```

So the complete file name is:

```
mdv1_numbers
```

CHANNELS

It is possible for a program to use several files at once, but it is more convenient to refer to a file by an associated channel number. This can be any integer in the range 0 to 15. A file is associated with a channel number by using the **OPEN** statement or, if it is a new file, **OPEN_NEW**. For example you may choose channel 7 for the numbers file and write:

```
OPEN_NEW #7,mdv1_numbers
```

file
device
channel number
keyword

You can now refer to the file just by quoting the number #7. The complete program is:

```
100 REMark Simple file
110 OPEN_NEW #7, mdv1_numbers
120 FOR number = 1 to 100
130   PRINT #7, number
140 END FOR number
150 CLOSE #7
```

The **PRINT** statement causes the numbers to be 'printed' on the cartridge file because #7 has been associated with it. The **CLOSE #7** statement is necessary because the system has some internal work to do when the file has been used. It also releases channel 7 for other possible uses. After the program has executed type:

```
DIR mdv1_
```

and the directory should show that the file *numbers* exists on the cartridge in Microdrive mdv1_.

You also need to know that the file is correct and you can only be certain of this if the file is read and checked. The necessary keyword is **OPEN_IN**, otherwise the program for reading data from a file is similar to the previous one.

```
100 REMark Reading a file
110 OPEN_IN #6, mdv1_numbers
120 FOR item = 1 TO 100
130   INPUT #, number
140   PRINT ! number !
150 END FOR item
160 CLOSE #6
```

The program should output the numbers 1 to 100, but only if the cartridge containing the file *numbers* is still in Microdrive mdv1_.

DEVICES AND CHANNELS

You have seen one example of a device, a file of data on a Microdrive. We may say, loosely, that a file has been opened but strictly we mean that a device has been associated with a particular channel. Any further necessary information has also been provided. Certain devices have channels permanently associated with them by the system:

channel	use
#0	OUTPUT – command window INPUT – keyboard
#1	OUTPUT – print window
#2	LIST – list output

WINDOWS

You can create a window of any size anywhere on the screen. The device name for a window is:

```
scr
```

and the appended information is, for example:

```
scr_360_50a80_40
```

- down value
- across value
- height
- width

The following program creates a window with the channel number 5 and fills it with green (code 4) and then closes it:

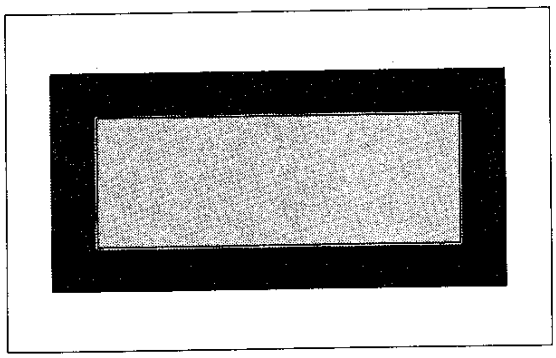
```
100 REMark Create a window
110 OPEN #5, scr_400x200a20x50
120 PAPER #5,4 : CLS #5
130 CLOSE #5
```

Notice that each window can have its own features such as paper, ink, etc. The fact that a window has been opened does not mean that it is the current default window.

You can change the position or shape of an opened window without closing it and reopening it. Try adding two lines to the previous program:

```
124 WINDOW #5,300,100,110,65
126 PAPER #5,2 : CLS #5
```

Re-run the program and you will find a red window within the original green one. This red window is now the one associated with channel 5, see figure.



BORDER

You can place a border round the edge of the screen or a window. For example:

```
BORDER #5,6
```

would create a border round the channel #5 window. It would be 6 units thick and the size of the window would be correspondingly reduced. The border would be transparent, protecting anything that was under it. You can specify a coloured border by the usual method.

```
BORDER #5,6,2
```

would produce a red border. You can make a border of other colours and textures by the usual methods. For example,

```
BORDER 10
```

will add a 10 pixel thick transparent border to the current window (transparent because no colour was specified) and

```
BORDER 2,0,7,0
```

will add a 2 pixel thick black and white stipple border.

BLOCK

You can specify a block's size, position and colour with a single statement. It is placed in the pixel co-ordinate system relative to the current window or screen. For example:

```
BLOCK #5,10,20,50,100,2
```

would create a block in the #5 window at a position 50 units across and 100 units down. It would be 10 units wide and 20 units high. Its colour would be red.

It is worth noting that **WINDOW** and **BLOCK** statements work without alteration in 4 and 8 colour mode (though the colours may vary) because the across values are always on a 0 to 511 scale and there are always 256 pixel positions down.

SPECIAL PRINTING

CSIZE

You can alter the size of characters. For example:

```
CSIZE 3,1
```

will give the largest possible characters and:

```
CSIZE 0,0
```

will give the smallest. The first number must be 0,1,2 or 3 and determines the width. The second must be 0 or 1 and determines the height. The normal sizes are:

```
MODE 4 CSIZE 0,0
MODE 8 CSIZE 2,0
```

The number of lines and columns available for each character size is dependent on whether the output is viewed on a monitor or on a television set; the row and column sizes given are for a monitor; those for a television set will be smaller and also will vary between different televisions.

If you are using low resolution mode the QL will not allow you to select a character size smaller than default size.

STRIP

You can provide a special background for characters to make them stand out. For example:

```
STRIP 7
```

will give a white strip while

```
STRIP 2,4,2
```

will give a red/green vertical striped strip. All the normal colour combinations are possible.

OVER

Normally printing occurs on the current paper colour. You can alter this by using strip.

You can make further effects by using:

```
OVER 1      1 prints in ink on a transparent strip
OVER -1     -1 prints in ink over existing display on screen
```

To revert to normal printing on current strip use:

```
OVER 0
```

UNDER

You can underline characters.

```
UNDER 1   underlines all subsequent output in the current ink
UNDER 0   switches off underling.
```

SCALE GRAPHICS

If you wish to draw reasonably true geometric figures on a TV or video screen you cannot easily use a pixel-based system. If you use **scale graphics** then the system will do the necessary work to ensure that you can fairly easily draw reasonable circles, squares and other shapes.

The default scale of the graphics coordinate system is 100 in the vertical direction and whatever is needed in the across direction to ensure that shapes drawn with the special graphics keywords (**PLOT**, **DRAW**, **CIRCLE**,) are true.

The **graphics origin** is not the same as the pixel origin which is used to define the position of windows and blocks. The graphics origin is at the bottom left hand corner of the current screen or window.

POINTS AND LINES

It is easy to draw points and lines using scale graphics. Using a vertical scale of 100 a point near the centre of the window can be plotted with:

```
POINT 60,50
```

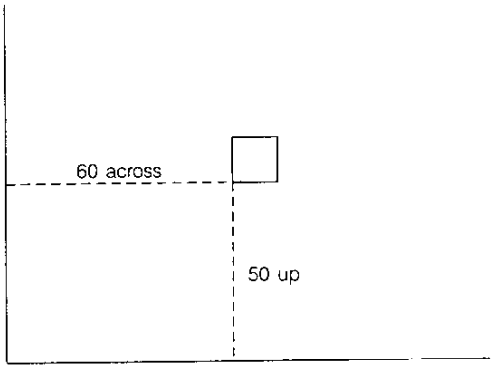
The point (60 units across and 50 units up) will be plotted in the current ink colour.

Similarly a line may be drawn with the statement:

```
LINE 60,50 TO 80,90
```

Further elements can be added. For example, the following will draw a square:

```
LINE 60,50 TO 70,50 TO 70,60 TO 60,60 TO 60,50
```



RELATIVE MODE

Pair of coordinates such as:

across, up

normally define a point relative to the origin 0,0 in the bottom left hand corner of a window (or elsewhere if you choose). It is sometimes more convenient to define points relative to the current cursor position. For example the square above may be plotted in another way using the LINE_R statement which means:

"Make all pairs of coordinates relative to the current cursor position."

```
POINT 60,50
LINE_R 0,0 TO 10,0 TO 0,10 TO -10,0 TO 0,-10
```

First the point 60,50 becomes the origin, then, as lines are drawn, the end of a line becomes the origin for the next one.

The following program will plot a pattern of randomly placed coloured squares.

```
100 REMark Coloured Squares
110 PAPER 7 : CLS
120 FOR sq = 1 TO 100
130    INK RND(1 TO 6)
140    POINT RND(90), RND(90)
150    LINE_R 0,0 TO 10,0 TO 0,10 TO -10,0 TO 0,-10
160 END FOR sq
```

The same result could be achieved entirely with absolute graphics but it would require a little more effort.

CIRCLES AND ELLIPSES

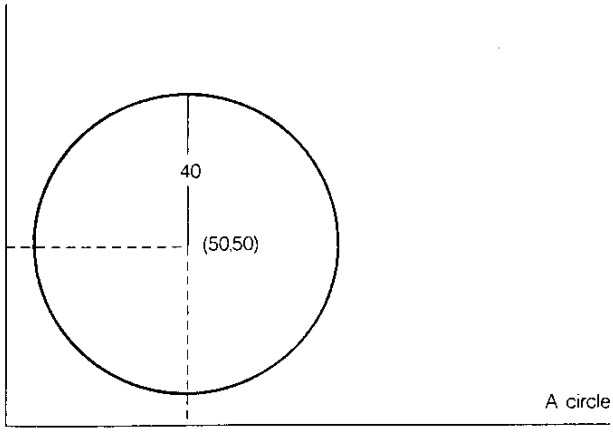
If you want to draw a circle you need to specify:

position say 50,50
radius say 40

The statement

```
CIRCLE 50,50,40
```

will draw a circle with the centre at position 50,50 and radius (or height) 40 units, see figure:

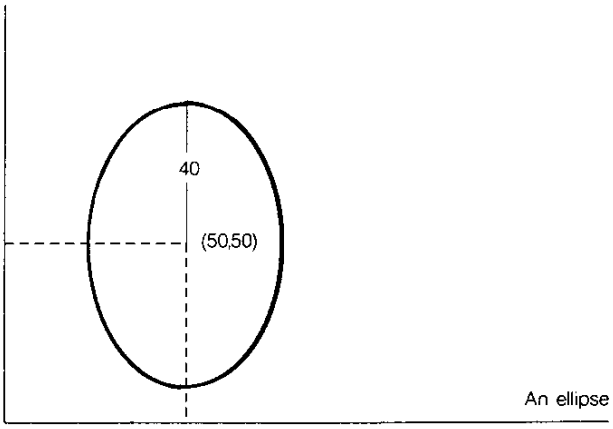


A circle

If you add two more parameters:

e.g. `CIRCLE 50,50,40,.5`

You will get an ellipse. The keywords **CIRCLE** and **ELLIPSE** are interchangeable.

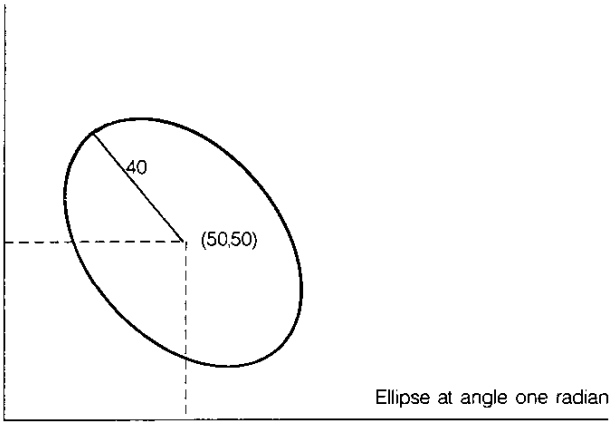


An ellipse

The height of the ellipse is 40 as before but the horizontal 'radius' is now only 0.5 of the height. The number 0.5 is called the eccentricity. If the eccentricity is 1 you get a circle if it is less than 1 and greater than zero you get an ellipse. If you want to tilt an ellipse you can change the fifth parameter, for example:

```
CIRCLE 50,50,40,.5,1
```

This will tilt the ellipse anti-clockwise by one radian, about 57 degrees, as shown in figure.



Ellipse at angle one radian

A straight angle is 180 degrees or PI radians, so you can make a pattern of ellipses with the program:

```
100 FOR rot = 0 TO 2*PI STEP PI/6
110    CIRCLE 50,50,40,0.5,rot
120 END FOR rot
```

The order of the parameters for a circle or ellipse is:

centre_across, centre_up, height, [eccentricity, angle]

The last two parameters are optional and this is indicated by putting them inside square brackets ([ ]).

Example Write a program which does the following:

1. Open a window 100x100 at (100,50)
2. Scale 100 in mode 8
3. Select black paper and clear window
4. Make green border 2 units wide
5. Draw a pattern of six coloured circles.
6. Close the window.

```
100 REMark pattern
110 MODE 8
120 OPEN #7, scr_100x100a 100x50
130 SCALE #7,100,0,0
140 PAPER #7,0 : CLS #7
150 BORDER #7,2,4
160 FOR colour = 1 TO 6
170   INK #7, colour
180   LET rot = 2*PI/colour
190   CIRCLE #7,50,50,30,0.5,rot
200 END FOR colour
210 CLOSE #7
```

You can get some interesting effects by altering the program. For example try the amendments:

```
160 FOR colour = 1 TO 100
180 LET rot = colour*PI/50
```

ARCS

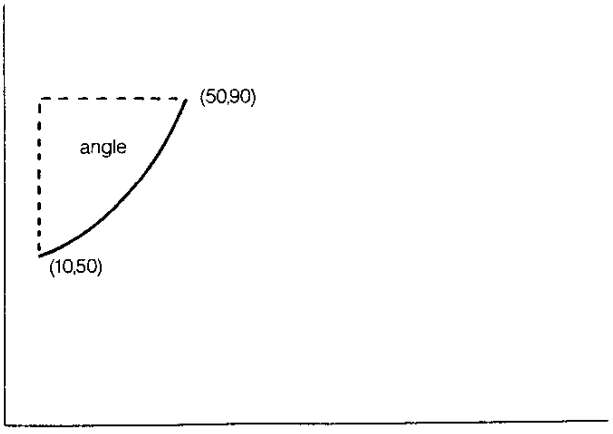
If you want to draw an arc you need to decide:

starting point
end point
amount of curvature.

The first two items are straightforward but the amount of curvature is not so easy. You can do it by drawing accurately or by trial and error but you must decide what angle the arc subtends and then specify the angle in radians. An angle of 1.5 radians would give a sharp bend and a small angle would give a very gentle curvature. Try, for example:

```
ARC 10,50 TO 50,90, 1
```

which gives a moderate curvature in the current INK colour.



FILL

You can fill a closed shape with the current **INK** colour by simply writing:

```
FILL 1
```

before the shape is drawn. The following program produces a green circle.

```
INK 4
FILL 1
CIRCLE 50,50,30
```

The **FILL** command works by drawing touching horizontal lines between suitable points.

The statement:

```
FILL 0
```

will turn off the **FILL** effect.

SCROLLING AND PANNING

You can scroll or pan the display in a window like a film cameraman. You arrange scrolling in terms of pixels. A positive number of pixels indicates upwards scrolling, thus

```
SCROLL 10
```

moves the display in the current window or screen 10 pixels downwards.

```
SCROLL -8
```

Moves the display 8 pixels up. You can add a second parameter to induce part-scrolling.

```
SCROLL -8, 1
```

will scroll the part above (not including) the cursor line and :

```
SCROLL -8,2
```

will scroll the part below (not including) the cursor line.

As scrolling occurs, the space left by movement of the display is filled with the current paper colour. A second parameter 0 has no effect.

You can **PAN** the display in the current window left or right. The **PAN** statement works in a similar manner to **SCROLL** but

```
PAN  40   moves display right
PAN -40   moves display left
```

A second parameter gives a partial **PAN**

0 – whole screen
3 – the whole of the line occupied by the cursor
4 – the right hand side of the line occupied by the cursor. The area of the cursor is also included.

If you are using stipples or are in 8 colour mode then windows must be panned or scrolled in multiples of 2 pixels.

PROBLEMS ON CHAPTER 12

1. Write a program which draws a 'Snakes and Ladders' grid of ten rows of ten squares.
2. Place the numbers 1 to 100 in the squares starting at the bottom left and place F for finish in the last square.
3. Draw a dartboard on the screen. It should consist of an outer ring which could hold numbers. A 'doubles' ring and 'triples' ring as shown and a centre consisting of a 'bull's eye' and a ring around it.

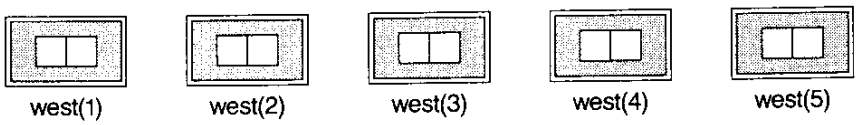
CHAPTER 13
ARRAYS

Suppose you are a prison governor and you have a new prison block which is called the West Block. It is ready to receive 50 new prisoners. You need to know which prisoner (known by his number) is in which cell. You could give each cell a name but it is simpler to give them numbers 1 to 50.

In a computing simulation we will imagine just 5 prisoners with numbers which we can put in a **DATA** statement:

```
DATA 50, 37, 86, 41, 32
```

We set up an array of variables which share the name, west, and are distinguished by a number appended in brackets.



It is necessary to declare an array and give its dimensions with a **DIM** statement

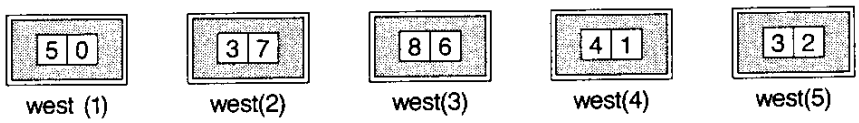
```
DIM west(5)
```

This enables SuperBASIC to allocate space, which might be a large amount. After the **DIM** statement has been executed the five variables can be used.

The convicts can be **READ** from the **DATA** statement into the five array variables:

```
FOR cell = 1 TO 5 : READ west(cell)
```

We can add another **FOR** loop with a **PRINT** statement to prove that the convicts are in the cells



The complete program is shown below:

```
100 REMark Prisoners
110 DIM west(5)
120 FOR cell = 1 TO 5 : READ west(cell)
130 FOR cell = 1 TO 5 : PRINT cell ! west(cell)
140 DATA 50, 37, 86, 41, 32
```

The output from the program is:

```
1 50
2 37
3 86
4 41
5 32
```

The numbers 1 to 5 are called *subscripts* of the array name, *west*. The array, *west*, is a numeric array consisting of five numeric array elements.

You can replace line 130 by:

```
130 PRINT west
```

This will output the values only:

```
 0
50
37
86
41
32
```

The zero at the top of the list appears because subscripts range from zero to the declared number. We will show later how useful the zero elements in arrays can be.

Note also that when a numeric array is DIMensioned its elements are all given the value zero.

STRING ARRAYS

String arrays are similar to numeric arrays but an extra dimension in the DIM statement specifies the length of each string variable in the array. Suppose that ten of the top players at Royal Birkdale for the 1982 British Golf Championship were denoted by their first names and placed in DATA statements.

```
DATA "Tom", "Graham", "Sevvy", "Jack", "Lee"
DATA "Nick", "Bernard", "Ben", "Gregg", "Hal"
```

You would need ten different variable names, but if there were a hundred or a thousand players the job would become impossibly tedious. An array is a set of variables designed to cope with problems of this kind. Each variable name consists of two parts:

a name according to the usual rules
a numeric part called a subscript

Write the variable names as:

```
flat$(1), flat$(2), flat$(3)...etc
```

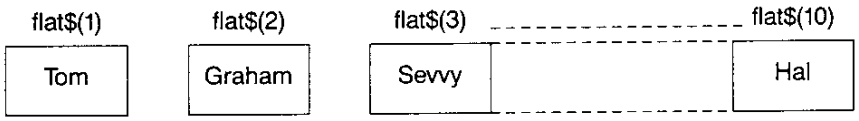
Before you can use the array variables you must tell the system about the array and its dimensions:

```
DIM flat$(10,8)
```

This causes eleven (0 to 10) variables to be reserved for use in the program. Each string variable in the array may have up to eight characters. DIM statements should usually be placed all together near the beginning of the program. Once the array has been declared in a DIM statement all the elements of the array can be used. One important advantage is that you can give the numeric part (the subscript) as a numeric variable. You can write:

```
FOR number = 1 TO 10 : READ flat$(number)
```

This would place the golfers in their 'flats'.



You can refer to the variables in the usual way but remember to use the right subscript. Suppose that Tom and Sevvy wished to exchange flats. In computing terms one of them, Tom say, would have to move into a temporary flat to allow Sevvy time to move. You can write:

```
LET temp$ = flat$(1): REMark Tom into temporary
LET flat$(1) = flat$(3): REMark Sevvy into flat$(1)
LET flat$(3) = temp$: REMark Tom into flat$(3)
```

The following program places the ten golfers in an array named flat$ and prints the names of the occupants with their 'flat numbers' (array subscripts) to prove that they are in residence. The occupants of flats 1 and 3 then change places. The list of occupants is then printed again to show that the exchange has occurred.

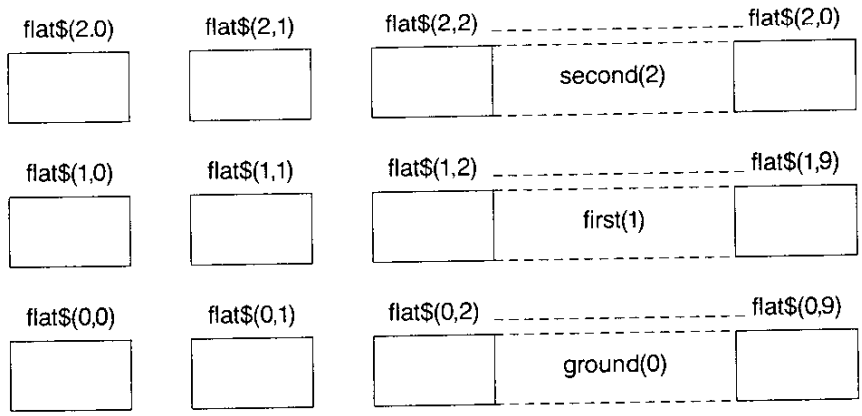
```
100 REMark Golfers' Flats
110 DIM flat$(10,8)
120 FOR number = 1 TO 10 : READ flat$(number)
130 printlist
140 exchange
150 printlist
160 REMark End of main program
170 DEFine PROCedure printlist
180   FOR num = 1 TO 10 : PRINT num, flat$(num)
190 END DEFine
200 DEFine PROCedure exchange
210   LET temp$ = flat$(1)
220   LET flat$(1) = flat$(3)
230   LET flat$(3) = temp$
240 END DEFine
250 DATA "Tom", "Graham", "Sevvy", "Jack", "Lee"
260 DATA "Nick", "Bernard", "Ben", "Greg", "Hal"
```

output (line 130)	output (line 150)
1 Tom	1 Sevvy
2 Graham	2 Graham
3 Sevvy	3 Tom
4 Jack	4 Jack
5 Lee	5 Lee
6 Nick	6 Nick
7 Bernard	7 Bernard
8 Ben	8 Ben
9 Gregg	9 Gregg
10 Hal	10 Hal

TWO DIMENSIONAL ARRAYS

Sometimes the nature of a problem suggests two dimensions such as 3 floors of 10 flats rather than just a single row of 30.

Suppose that 20 or more golfers need flats and there is a block of 30 flats divided into three floors of ten flats each. A realistic method of representing the block would be with a two-dimensional array. You can think of the thirty variables as shown below:



Assuming DATA statements with 30 names, a suitable way to place the names in the flats is:

```
120 FOR floor = 0 TO 2
130   FOR num = 0 TO 9
140     READ flats$(floor, num)
150   END FOR num
160 END FOR floor
```

You also need a DIM statement:

```
20 DIM flat$(2,9,8)
```

which shows that the first subscript can be from 0 to 2 (floor number) and the second subscript can be from 0 to 9 (room number). The third number states the maximum number of characters in each array element.

We add a print routine to show that the golfers are in the flats and we use letters to save space.

```
100 REMark 30 Golfers
110 DIM flat$(2,9,8)
120 FOR floor = 0 TO 2
130   FOR num = 0 TO 9
140     READ flat$(floor,num) : REMark Golfer goes in
150   END FOR num
160 END FOR floor
170 REMark End of input
180 FOR floor = 0 TO 2
190   PRINT "Floor number" ! floor
```

```
200    FOR num = 0 TO 9
210      PRINT 'Flat' ! num ! flat$(floor,num)
220    END FOR num
230 END FOR floor
240 DATA "A","B","C","D","E","F","G","H","I","J"
250 DATA "K","L","M","N","O","P","Q","R","S","T"
260 DATA "U","V","W","X","Y","Z","@","£","$","%"
```

The output starts:

```
Floor number 0
Flat 0 A
Flat 1 B
Flat 2 C
```

and continues giving the thirty occupants.

ARRAY SLICING

You may find this section hard to read though it is essentially the same concept as string-slicing. You will probably need string-slicing if you get beyond the learning stage of programming. The need for array-slicing is much rarer and you may wish to omit this section particularly on a first reading.

We now use the golfers' flats to illustrate the concept of array slicing. The flats will be numbered 0 to 9 to keep to single digits and names will be single characters for space reasons.

	2,0	2,1	2,2	2,2	3,4	2,5	2,6	2,7	2,8	2,9
flat$	U	V	W	X	Y	Z	@	£	$	%

	1,0	1,1	1,2	1,3	1,4	1,5	1,6	1,7	1,8	1,9
flat$	K	L	M	N	O	P	Q	R	S	T

	0,0	0,1	0,2	0,3	0,4	0,5	0,6	0,7	0,8	0,9
flat$	A	B	C	D	E	F	G	H	I	J

Given the above values the following are array slices:

```
flat$(1,3)          Means a single array element with value N
flat$(1,1 TO 6)     Means six elements with values L M N O P Q
```

Array Element	Value
flat$(1,1)	L
flat$(1,2)	M
flat$(1,3)	N
flat$(1,4)	O
flat$(1,5)	P
flat$(1,6)	Q

```
flat$(1)        Means flat$ (1,0 TO 9)
                ten elements with values K L M N O P Q R S T
```

In these examples a range of values of a subscript can be given instead of a single value. If a subscript is missing completely the complete range is assumed. In the third example the second subscript is missing and it is assumed by the system to be **0 TO 9**.

The techniques of array slicing and string slicing are similar though the latter is more widely applicable.

PROBLEMS ON CHAPTER 13

1. SORTING

Place ten numbers in an array by reading from a DATA statement. Search the array to find the lowest number. Make this lowest number the value of the first element of a new array. Replace it in the first array with a very large number. Repeat this process making the second lowest number the second value in the new array and so on until you have a sorted array of numbers which should then be printed.

2. SNAKES AND LADDERS

Represent a snakes and ladders game with a 100 element numeric array. Each element should contain either:

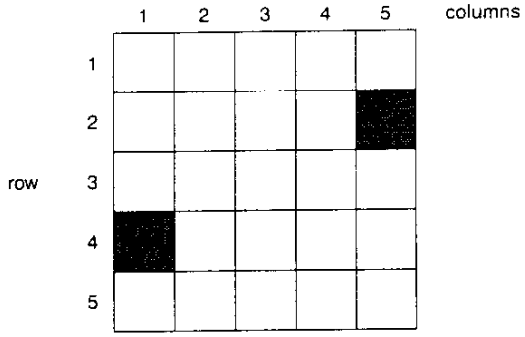
zero

or a number in the range 10 to 90 meaning that a player should transfer to that number by going 'up a ladder' or 'down a snake'.

or the digits 1, 2, 3, etc. to denote a particular player's position.

Set up six snakes and six ladders by placing numbers in the array and simulate one 'solo' run by a single player to test the game.

3. CROSSWORD BLANKS



Crosswords usually have an odd number of rows or columns in which the black squares have a symmetrical pattern. The pattern is said to have rotational symmetry because rotation through 180 degrees would not change it.

Note that after rotation through 180 degrees the square in row 4, column 1 could become the square in row 2, column 5. That is row 4, column 1 becomes row 2, column 5 in a 5 x 5 grid.

Write a program to generate and display a symmetrical pattern of this kind.

4. Modify the crossword pattern so that there are no sequences, across or down, of less than four white squares.

5. CARD SHUFFLE

Cards are denoted by the numbers 1–52 stored in an array. They can be converted easily to actual card values when necessary. The cards should be 'shuffled' as follows.

Choose any position in range 1–51 e.g. 17.

Place the card in this position in a temporary store.

Shunt all the cards in positions 52 to 18 down to positions 51 to 17.

Place the chosen card from the temporary store to position 52.

Deal similarly with the ranges 1–50, 1–49 .. down to 1–2 so that the pack is well shuffled.

Output the result of the shuffle.

6. Set up six **DATA** statements each containing a surname, initials and a telephone number (dialling code and local number). Decide on a suitable structure of arrays to store this information and **READ** it into the arrays.

PRINT the data using a separate **FOR** loop and explain how the input format (**DATA**), the internal format (arrays) and output format are not necessarily all the same.

CHAPTER 14
PROGRAM STRUCTURE

In this chapter we go again over the ground of program structure : loops and decisions or selection. We have tried to present things in as simple a way as possible but SuperBASIC is designed to cope properly with the simple and the complex and all levels in between. Some parts of this chapter are difficult and if you are new to programming you may wish to omit parts . The topics covered are:

Loops
Nested loops
Binary decisions
Multiple decisions

The latter parts of the first section, Loops, get difficult as we show how SuperBASIC copes with problems that other languages simply ignore. Skip these parts if you feel so inclined but the other sections are more straightforward.

LOOPS

In this section we attempt to illustrate the well-known problems of handling repetition with simulations of some Wild West scenes. The context may be contrived and trivial but it offers a simple basis for discussion and it illustrates difficulties which arise across the whole range of programming applications.

EXAMPLE 1

A bandit is holed up in the Old School House. The sheriff has six bullets in his gun. Simulate the firing of the six shots.

Program 1

```
100 REMark Western FOR
110 FOR bullets = 1 TO 6
120    PRINT "Take aim"
130    PRINT "Fire shot"
140 END FOR bullets
```

Program 2

```
100 REMark Western REPeat
110 LET bullets = 6
120 REPeat bandit
130    PRINT "Take aim"
140    PRINT "Fire shot"
150    LET bullets = bullets - 1
160    IF bullets = 0 THEN EXIT
170 END REPeat bandit
```

FOR SERVICE MANUALS
CONTACT:
MAURITRON TECHNICAL SERVICES
www.mauritron.co.uk
TEL: 01844 - 351694
FAX: 01844 - 352554

Both these programs produce the same output:

```
Take aim
Fire a shot
```

is printed six times.

If, in each program the 6 is changed to any number down to 1 both programs still work as you would expect. But what if the gun is empty before any shots have been fired?

EXAMPLE 2

Suppose that someone has secretly taken all the bullets out of the sheriff's gun. What happens if you simply change the 6 to 0 in each program?.

Program 1

```
100 REMark Western FOR Zero Case
110 FOR bullets = 1 to 0
120    PRINT "Take aim"
130    PRINT "Fire a shot"
140 END FOR bullets
```

This works correctly. There is no output. The 'zero case' behaves properly in SuperBASIC.

Program 2

```
100 REMark Western REPeat Fails
110 LET bullets = 0
120 REPeat bandit
130    PRINT "Take aim"
140    PRINT "Fire shot"
150    LET bullets = bullets - 1
160    IF bullets = 0 THEN EXIT bandit
170 END REPeat bandit
```

The program fails in two ways:

1. ```
   Take aim
   Fire a shot
   ```
   is printed though there were never any bullets.

2. By the time the variable, *bullets*, is tested in line 160 it has the value −1 and it never becomes zero afterwards. The program loops indefinitely. You can cure the infinite looping by re-writing line 160 :

   ```
   160 IF bullets < 1 THEN EXIT bandit
   ```

There is an inherent fault in the programming which does not allow for the possible zero case. This can be corrected by placing the conditional **EXIT** before the **PRINT** statements.

Program 3

```
100 REMark Western REPeat Zero Case
110 LET bullets = 0
120 REPeat Bandit
130    IF bullets = 0 THEN EXIT Bandit
140    PRINT "Take aim"
150    PRINT "Fire shot"
160    LET bullets = bullets -1
170 END REPeat Bandit
```

This program now works properly whatever the initial value of bullets as long as it is a positive whole number or zero. Method 2 corresponds to the **REPEAT** ... **UNTIL** loop of some languages. Method 3 corresponds to the **WHILE**....**ENDWHILE** loop of some languages. However, the **REPeat**...**END REPeat** with **EXIT** is more flexible than either or the combination of both.

If you have used other BASICs you may wonder what has happened to the **NEXT** statement. We will re-introduce it soon but you will see that both loops have a similar structure and both are named.

FOR *name* =	(opening keyword)	**REPeat** *name*
(statements)	(content)	(statements)
END FOR name	(closing keyword)	**END REPeat** *name*

In addition the **REPeat** loop must normally have an **EXIT** amongst the statements or it will never end.

Note also that the **EXIT** statement causes control to go to the statement which is immediately after the **END** of the loop.

A **NEXT** statement may be placed in a loop. It causes control to go to the statement which is just after the opening keyword **FOR** or **REPeat**. It should be considered as a kind of opposite to the **EXIT** statement. By a curious coincidence the two words, **NEXT** and **EXIT**, both contain EXT. Think of an **EXT**ension to loops and:

N means 'Now start again'
I means 'It's ended'

EXAMPLE 3 The situation is the same as in example 1. The sheriff has a gun loaded with six bullets and he is to fire at the bandit but two more conditions apply:

1. If he hits the bandit he stops firing and returns to Dodge City.

2. If he runs out of bullets before he hits the bandit, he tells his partner to watch the bandit while he (sheriff) returns to Dodge City.

Program 1

```
100 REMark Western FOR with Epilogue
110 FOR bullets = 1 TO 6
120    PRINT "Take aim"
130    PRINT "FIRE A SHOT"
140    LET hit = RND(9)
150    IF hit = 7 THEN EXIT bullets
160 NEXT bullets
170    PRINT "Watch Bandit"
180 END FOR bullets
190 PRINT "Return to Dodge City"
```
```

In this case, the content between NEXT and END FOR is a kind of epilogue which is only executed if the FOR loop runs its full course. If there is a premature EXIT the epilogue is not executed.

The same effect can be achieved with a REPeat loop though it is not necessarily the best way to do it. However, it is worth looking at (perhaps at a second reading) if you want to understand structures which are simple enough to use in simple ways and powerful enough to cope with awkward situations when they arise.

Program 2

```
100 REMark Western REPeat with Epilogue
110 LET bullets = 6
120 REPeat Bandit
130   PRINT "Take aim"
140   PRINT "Fire shot"
150   LET hit = RND(9)
160   IF hit = 7 THEN EXIT Bandit
170   LET bullets = bullets -1
180   IF bullets <> 0 THEN NEXT Bandit
190   PRINT "Watch Bandit"
200 END REPeat Bandit
210 PRINT "Return to Dodge City"
```

The program works properly as long as the sheriff has at least one bullet at the start. It fails if line 20 reads:

```
110 LET bullets = 0
```

You might think that the sheriff would be a fool to start an enterprise of this kind if he had no bullets at all, and you would be right. We are now discussing how to preserve good structure in the most complex type of situation. We have at least kept the problem context simple; we know what we are trying to do. Complex structural problems usually arise in contexts more difficult than Wild West simulations. But if you really want a solution to the problem which caters for a possible hit, running out of bullets and an epilogue, and also the zero case then add the following line to the above program:

```
125 IF bullets = 0 THEN PRINT "Watch Bandit" : EXIT bandit
```

We can conceive of no more complex type of problem than this with a single loop. SuperBASIC can easily handle it if you want it to.

NESTED LOOPS

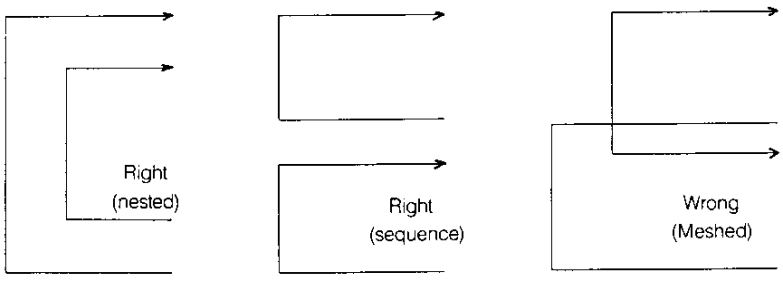
Consider the following FOR loop which PLOTS a row of points of various randomly chosen colours (not black).

```
100 REMark Row of pixels
110 PAPER 0 : CLS
120 LET up = 50
130 FOR across = 20 TO 60
140   INK RND(2 TO 7)
150   POINT across, up
160 END FOR across
```

This program plots a row of points thus:

```
.........................................
```

If you want to get say 51 rows of points you must plot a row for values up from 30 to 80. But you must always observe the rule that a structure can go completely within another or it can go properly around it. It can also follow in sequence, but it cannot 'mesh' with another structure. Books about programming often show how FOR loops can be related with a diagram like



In SuperBASIC the rule applies to all structures. You can solve all problems using them properly. We therefore treat the **FOR** loop as an entity and design a new program:

```
FOR up = 30 TO 80
  FOR across = 20 TO 60
    INK RND(2 To 7)
      POINT across, up
  END FOR across
END FOR up
```

When we translate this into a program we are entitled not only to expect it to work but to know what it will do. It will plot a rectangle made up of rows of pixels.

```
100 REMark Rows of pixels
110 PAPER 0 : CLS
120 FOR up = 30 TO 80
130    FOR across = 20 TO 60
140       INK RND(2 TO 7)
150       POINT across,up
160    END FOR across
170 END FOR up
```

Different structures may be nested. Suppose we replace the inner **FOR** loop of the above program by a **REPeat** loop. We will terminate the **REPeat** loop when the zero colour code appears for a selection in the range 0 to 7.

```
100 REMark REPeat in FOR
110 PAPER 0 : CLS
120 FOR up = 30 TO 80
130    LET across = 19
140    REPeat dots
150       LET colour = RND(7)
160       INK colour
170       LET across = across + 1
180       POINT across, up
190       IF colour = 0 then EXIT dots
200    END REPeat dots
210 END FOR up
```

Much of the wisdom about program control and structure can be expressed in two rules:

1. Construct your program using only the legitimate structures for loops and decision-making.
2. Each structure should be properly related in sequence or wholly within another.

BINARY DECISIONS

The three types of binary decision can be illustrated easily in terms of what to do when it rains.

i.
```
100 REMark Short form IF
110 LET rain = RND(0 TO 1)
120 IF rain THEN PRINT "Open brolly"
```

ii.
```
100 REMark Long form IF...END IF
110 LET rain = RND(0 TO 1)
120 IF rain THEN
130    PRINT "Wear coat"
140    PRINT "Open brolly"
150    PRINT "Walk fast"
160 END IF
```

iii.
```
100 REMark Long form IF ...ELSE...END IF
110 LET rain = RND(0 TO 1)
120 IF rain THEN
130    PRINT "Take a bus"
140 ELSE
150    PRINT "Walk"
160 END IF
```

All these are binary decisions. The first two examples are simple : either something happens or it does not. The third is a general binary decision with two distinct possible courses of action, both of which must be defined.

You can omit **THEN** in the long forms if you wish. In the short form you can substitute : for **THEN**.

EXAMPLE

Consider a more complex example in which it seems natural to nest binary decisions. This type of nesting can be confusing and you should only do it if it seems the most natural thing to do. Careful attention to layout, particularly indenting, is especially important.

Analyse a piece of text to count the number of vowels, consonants and other characters. Ignore spaces. For simplicity the text is all upper case.

Data

"COMPUTER HISTORY WAS MADE IN 1984"

Design

```
Read in the data
 FOR each character:
   IF letter THEN
     IF vowel
       increase vowel count
     ELSE
       increase consonant count
     END IF
   ELSE
     IF not space THEN increase other count
   END IF
 END FOR
 PRINT results
```

Program

```
100 REMark Character Counts
110 RESTORE 290
120 READ text$
130 LET vowels = 0 : cons = 0 : others = 0
140 FOR num = 1 TO LEN(text$)
150   LET ch$ = text$(num)
160   IF ch$ >= "A" AND ch$ <= "Z"
170     IF ch$ INSTR "AEIOU"
180       LET vowels = vowel + 1
190     ELSE
200       LET cons = cons + 1
210     END IF
220   ELSE
230     IF ch$ <> " " THEN others = others + 1
240   END IF
250 END FOR num
260 PRINT "Vowel count is" ! vowels
270 PRINT "Consonent count is" ! cons
280 PRINT "Other count is" ! others
290 DATA "COMPUTER HISTORY WAS MADE IN 1984"
```

Output

```
Vowel count is 9
Consonant count is 15
Other count is 4
```

MULTIPLE DECISIONS - SELect

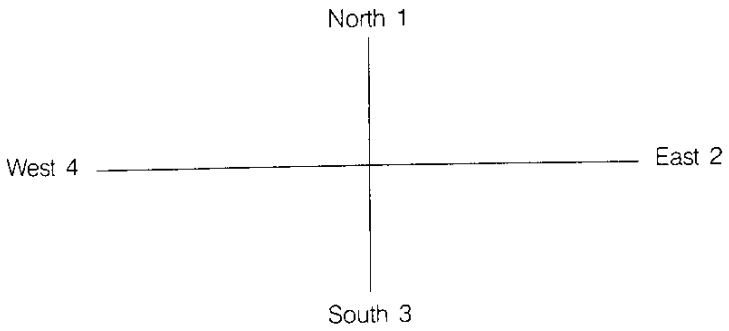
Where there are three or more possible actions and none is dependant on a previous choice the natural structure to use is **SELect** which enables selection from any number of possibilities.

EXAMPLE

A magic snake grows without limit by adding a section to its front. Each section may be up to twenty units long and may be a new colour or it may remain the same. Each new section must grow in one of the directions North, South East or West. The snake starts from the centre of the window.

Method

At any time while the snake is still on the screen you choose a random length and ink colour easily. The direction may be selected by a number 1,2,3 or 4 as shown:



Design

Select PAPER
Set snake to centre of window
REPeat
  Choose direction, colour, length of growth
  FOR unit = 1 to growth
    Make snake grow, north, south, east or west
    IF snake is off window THEN EXIT
  END FOR
END REpeat
PRINT end message

Program

```
100 REMark Magic Snake
110 PAPER 0 : CLS
120 LET across = 50 : up = 50
130 REPeat snake
140   LET direction = RND(1 TO 4) : colour = RND(2 TO 7)
150   LET growth = RND(2 TO 20)
160   INK colour
170   FOR unit = 1 TO growth
180     SELect ON direction
190       ON direction = 1
200         LET up = up + 1
210       ON direction = 2
220         LET across = across + 1
230       ON direction = 3
240         LET up = up - 1
250       ON direction = 4
260         LET across = across - 1
270     END SELect
280     IF across<1 OR across>99 OR up<1 OR up>99 THEN EXIT snake
290     POINT across,up
300   END FOR unit
310 END REPeat snake
320 PRINT "Snake off edge"
```

The syntax of the SELect ON structure also allows for the possibility of selecting on a list of values such as

```
5,6,8,10 TO 13
```

It is also possible to allow for an action to be executed if none of the stated values is found. The full structure is of the form given below.

LONG FORM

```
SELect ON num
ON num = list of values
   statements
ON num = list of values
   statements
   -
   -
   -
   -
ON num = REMAINDER
   statements
END SELect
```

where num is any numeric variable and the **REMAINDER** clause is optional.

SHORT FORM

There is a short form of the **SELect** structure. For example:

```
100 INPUT num
110 SELect ON num = 0 TO 9 : PRINT "digit"
```

will perform as you would expect.

PROBLEMS ON CHAPTER 14

1. Store 10 numbers in an array and perform a 'bubble-sort'. This is done by comparing the first pair and exchanging, if necessary, the second pair (second and third numbers), up to the ninth pair (ninth and tenth numbers). The first run of nine comparisons and possible exchanges guarantees that the highest number will reach its correct position. Another eight runs will guarantee eight more correct positions leaving only the lowest number which must be in the only (correct) position left. The simplest form of 'bubble sort' of ten numbers requires nine runs of nine comparisons.

2. Consider ways of speeding up bubblesort, but do not expect that it will ever be very efficient.

3. An auctioneer wishes to sell an old clock and he has instructions to invite a first bid of £50. If no-one bids he can come down to £40, £30, £20, but no lower, in an effort to start the bidding. If no-one bids, the clock is withdrawn from the sale. When the bidding starts, he takes only £5 increases until the final bid is made. If the final bid is £35 (the 'reserve price') or more, the clock is sold. Otherwise it is withdrawn.

   Simulate the auction using the equivalent of a six-sided die throw to start the bidding. A 'six' at any of the starting prices will start it off.

   When the bidding has started there should be a three out of four chance of a higher bid at each invitation.

4. In a wild west shoot-out the Sheriff has no ammunition and wishes to arrest a gunman camped in a forest. He rides amongst the trees tempting the gunman to fire. He hopes that when six shots have been fired he can rush in and overpower the gunman as he tries to re-load. Simulate the encounter giving the gunman a one-twentieth chance of hitting the Sheriff with each shot. If the Sheriff has not been hit after six shots he will arrest the gunman.

5. The Sheriff's instructions to his Deputy are:

   > "If the gun is empty then re-load it and if it ain't then keep on firing until you hit the bandit or he surrenders. If Mexico Pete turns up, get out fast."

   Write a program which caters properly for all these situations:

   > Whatever happens, return to Dodge City.
   > If Mexico Pete turns up, return immediately.
   > If the gun is empty, reload it.
   > If the gun is not empty, ask the bandit to surrender.
   > If the bandit surrenders, arrest him.
   > If he doesn't surrender, fire a shot.
   > If the bandit is hit, arrest him and fix his wound.

   Assume an unlimited supply of ammunition. Use a simulated 'twenty-sided die' and let a seven mean 'surrender' and a 'thirteen' mean the bandit is hit.

CHAPTER 15
PROCEDURES AND FUNCTIONS

In the first part of this chapter we explain the more straightforward features of SuperBASIC's procedures and functions. We do this with very simple examples so that you can understand the working of each feature as it is described. Though the examples are simple and contrived you will appreciate that, once understood, the ideas can be applied in more complex situations where they really matter.

After the first part there is a discussion which attempts to explain 'Why procedures' . If you understand, more or less, up to that point you will be doing well and you should be able to use procedures and functions with increasing effectiveness.

SuperBASIC first allows you to do the simpler things in simple ways and then offers you more if you want it. Extra facilities and some technical matters are explained in the second part of this chapter but you could omit these, certainly at a first reading, and still be in a stronger position than most users of older types of BASIC.

VALUE PARAMETERS

You have seen in previous chapters how a value can be passed to a procedure. Here is another example.

EXAMPLE

In "Chan's Chinese Take-Away" there are just six items on the menu.

Rice Dishes	Sweets
1 prawns	4 ice
2 chicken	5 fritter
3 special	6 lychees

Chan has a simple way of computing prices. He works in pence and the prices are:

for a rice dish 300 + 10 times menu number
for a sweet 12 times menu number

Thus a customer who ate special rice and an ice would pay:

```
300 + 10 * 3 + 12 * 4 = 378 pence
```

A procedure, *item*, accepts a menu number as a value parameter and prints the cost.

Program

```
100 REMark Cost of Dish
110 item 3
120 item 4
130 DEFine PROCedure item(num)
140   IF num <= 3 THEN LET price = 300 + 10*num
150   IF num >= 4 THEN LET price = 12*num
160   PRINT ! price !
170 END DEFine
```

Output

```
330 48
```

In the main program actual parameters 3 and 4 are used. The procedure definition has a formal parameter, num, which takes the value passed to it from the main program. Note that the formal parameters must be in brackets, but that actual parameters need not be.

EXAMPLE

Now suppose the working variable, *price*, was also used in the main program, meaning something else, say the price of a glass of lager, 70p. The following program fails to give the desired result.

Program

```
100 REMark Global price
110 LET price = 70
120 item 3
130 item 4
140 PRINT ! price !
150 DEFine PROCedure item(num)
160    IF num <= 3 THEN LET price = 300 + 10*num
170    IF num >= 4 THEN LET price = 12*num
180    PRINT ! price !
190 END DEFine
```

Output

```
330 48 48
```

The price of the lager has been altered by the procedure. We say that the variable, *price*, is **global** because it can be used anywhere in the program.

EXAMPLE

Make the procedure variable, *price*, **LOCAL** to the procedure. This means that SuperBASIC will treat it as a special variable accessible only within the procedure. The variable, *price*, in the main program will be a different thing even though it has the same name.

Program

```
100 REMark LOCAL price
110 LET price = 70
120 item 3
130 item 4
140 PRINT ! price !
150 DEFine PROCedure item(num)
160    LOCal price
170    IF num <= 3 THEN LET price = 300 + 10*num
180    IF num >= 4 THEN LET price = 12*num
190    PRINT ! price !
200 END DEFine
```

Output

```
330 48 70
```

This time everything works properly. Line 70 causes the procedure variable, *price* to be internally marked as 'belonging' only to the procedure, *item*. The other variable, *price* is not affected. You can see that local variables are useful things.

EXAMPLE

Local variables are so useful that we automatically make procedure formal parameters local. Though we have not mentioned it before parameters such as *num* in the above programs cannot interfere with main program variables. To prove this we drop the **LOCAL** statement from the above program and use *num* for the price of lager. Because *num* in the procedure is local everything works.

Program

```
100 REMark LOCAL parameter
110 LET num = 70
120 item 3
130 item 4
140 PRINT ! num !
150 DEFine PROCedure item(num)
160    IF num <= 3 THEN LET price = 300 + 10*num
170    IF num >= 4 THEN LET price = 12*num
180    PRINT ! price !
190 END DEFine
```

Output

```
330 48 70
```

VARIABLE PARAMETERS

So far we have only used procedure parameters for passing values to the procedure. But suppose the main program wants the cost of an item to be passed back so that it can compute the total bill. We can do this easily by providing another parameter in the procedure call. This must be a variable because it has to receive a value from the procedure. We therefore call it a variable parameter and it must be matched by a corresponding variable parameter in the procedure definition.

EXAMPLE Use actual variable parameters, *cost_1* and *cost_2* to receive the values of the variable price from the procedure. Make the main program compute and print the total bill.

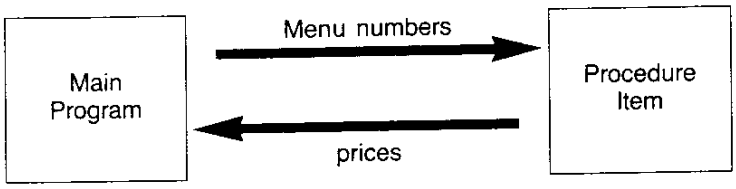
Program

```
100 REMark Variable parameter
110 LET num = 70
120 item 3,cost_1
130 item 4,cost_2
140 LET bill = num + cost_1 + cost_2
150 PRINT bill
160 DEFine PROCedure item(num, price)
170    IF num <= 3 THEN LET price = 300 + 10*num
180    IF num >= 4 THEN LET price = 12*num
190 END DEFine
```

Output

```
448
```

The parameters num and price are both automatically local so there can be no problems. The diagrams show how information passes from main program to procedure and back



That is enough about procedures and parameters for the present.

FUNCTIONS

You already know how a system function works. For example the function:

```
SQRT(9)
```

computes the value, 3, which is the square root of 9. We say the function returns the value 3. A function, like a procedure, can have one or more parameters, but the distinguishing feature of a function is that it returns exactly one value. This means that you can use it in expressions that you already have. You can type:

```
PRINT 2*SQRT(9)
```

and get the output 6. Thus a function behaves like a procedure with one or more value parameters and exactly one variable parameter holding the returned value; that variable parameter is the function name itself.

The parameters need not be numeric.

```
LEN("string")
```

has a string argument but it returns the numeric value 6.

EXAMPLE Re-write the program of the last section which used price as a variable parameter. Let price be the name of the function.

The value to be returned is defined by the RETurn statement as shown.

Program

```
100 REMark FuNction with RETurn
110 LET num = 70
120 LET bill = num + price(3) + price(4)
130 PRINT bill
140 DEFine FuNction price(num)
150    IF num <= 3 THEN RETurn 300 + 10*num
160    IF num >= 4 THEN RETurn 12*num
170 END DEFine
```

Output

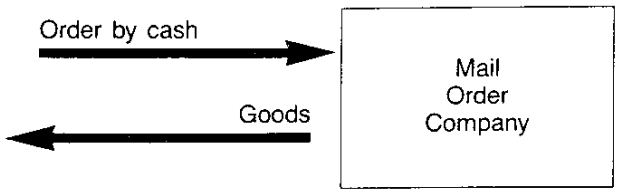
```
448
```

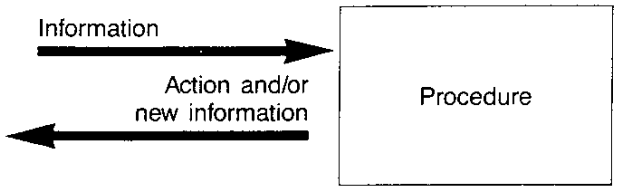
Notice the simplification in the calling of functions as compared with procedure calls.

WHY PROCEDURES?

The ultimate concept of a procedure is that it should be a 'black box' which receives specific information from 'outside' and performs certain operations which may include sending specific information back to the 'outside'. The 'outside' may be the main program or another procedure.

The term 'black box' implies that its internal workings are not important; you only think about what goes in and what comes out. If, for example, a procedure uses a variable, *count*, and changes its value, that might affect a variable of the same name in the main program. Think of a mail order company. You send them an order and cash; they send you goods. Information is sent to a procedure and it sends back action and/or new information.





You do not want the mail order company to use your name and address or other information for other purposes. That would be an unwanted side-effect. Similarly you do not want a procedure to cause unplanned changes to values of variables used in the main program.

Of course you could make sure that there are no double uses of variable names in a program. That will work up to a point but we have shown in this chapter how to avoid trouble even if you forget what variables have been used in any particular procedure.

A second aim in using procedures is to make a program modular. Rather than have one long main program you can break the job down into what Seymour Papert, the inventor of **LOGO**, calls 'Mind-sized bites'. These are the procedures, each one small enough to understand and control easily. They are linked together by the procedure calls in a sequence or hierarchy.

A third aim is to avoid writing the same code twice. Write it once as a procedure and call it twice if necessary. Functions and procedures written for one program can often be directly used, without change, by other programs, and one might create a library of commonly used procedures and functions.

We give below another example which shows how procedures make a program modular.

EXAMPLE

An order is placed for six dishes at Chan's Take Away, where the menu is:

Item Number	Dish	Price
1	Prawns	3.50
2	Chicken	2.80
3	Special	3.30

Write procedures for the following tasks.

1. Set up two three-element arrays showing menu, dishes and prices. Use a **DATA** statement.
2. Simulate an order for six randomly chosen dishes using a procedure, choose, and make a tally of the number of times each dish is chosen.

3. Pass the three numbers to a procedure, *waiter*, which passes back the cost of the order to the main program using a parameter *cost*. Procedure waiter calls two other procedures, *compute* and *cook*, which compute the cost and simulate "cooking".

4. The procedure, *cook*, simply prints the number required and the name of each dish.

The main program should call procedures as necessary, get the total cost from procedure, *waiter*, add 10% for a tip, and print the amount of the total bill.

Design

This program illustrates parameter passing in a fairly complex way and we will explain the program step by step before putting it together.

```
100 REMark Procedures
110 RESTORE 490
120 DIM item$(3,7), price(3), dish(3)
130 REMark *** PROGRAM ***
140 LET tip = 0.1
150 set_up
    -
    -
210 DEFine PROCedure set_up
220   FOR k = 1 TO 3
230     READ item$(k)
240     READ price(k)
250   END FOR k
260 END DEFine
    -
    -
    -
490 DATA "Prawns", 3.5, "Chicken", 2.8, "Special", 3.3
```

The names of menu items and their prices are placed in the arrays *item$* and *price*.

The next step is to choose a menu number for each of the six customers. The tally of the number of each dish required will be kept in the array *dish*.

```
160 choose dish
    -
    -
    -
270 DEFine PROCedure choose(dish)
280   FOR pick = 1 TO 6
290     LET number = RND(1 TO 3)
300     LET dish(number) = dish(number) + 1
310   END FOR pick
320 END DEFine
```

Note that the formal parameter *dish* is both:

local to procedure choose
an array in main program

The three values are passed back to the global array also called dish. These values are then passed to the procedure *waiter*.

```
170 waiter dish, bill
    -
    -
    -
    -
330 DEFine PROCedure waiter(dish, cost)
340   compute dish, cost
350   cook dish
360 END DEFine
```

The waiter passes the information about the number of each dish required to the procedure, *compute*, which computes the cost and returns it.

```
370 DEFine PROCedure compute(dish, total)
380   LET total = 0
390   FOR k = 1 to 3
400     LET total = total + dish(k)*price(k)
410   END FOR k
420 END DEFine
```

The waiter also passes information to the cook who simply prints the number required for each menu item.

```
430 DEFine PROCedure cook(dish)
440   FOR c = 1 TO 3
450     PRINT ! dish(c) ! item$(c) !
460   END FOR c
470 END DEFine
```

Again, the array, *dish* in the procedure *cook* is local. It receives the information which the procedure uses in its PRINT statement.

The complete program is listed below.

Program

```
100 REMark Procedures
110 RESTORE 490
120 DIM item$(3,7), price(3), dish(3)
130 REMark *** PROGRAM ***
140 LET tip = 0.1
150 set_up
160 choose dish
170 waiter dish, bill
180 LET bill = bill + tip*bill
190 PRINT "Total cost is £" ; bill
200 REMark *** PROCEDURE DEFINITIONS ***
210 DEFine PROCedure set_up
220   FOR k = 1 TO 3
230     READ item$(k)
240     READ price(k)
250   END FOR k
260 END DEFine
270 DEFine PROCedure choose(dish)
280   FOR pick = 1 TO 6
290     LET number = RND(1 TO 3)
300     LET dish(number) = dish(number) + 1
310   END FOR pick
320 END DEFine
330 DEFine PROCedure waiter(dish, cost)
340   compute dish,cost
350   cook dish
360 END DEFine
370 DEFine PROCedure compute(dish, total)
380   LET total = 0
390   FOR k = 1 TO 3
400     LET total = total + dish(k)*price(k)
410   END FOR k
420 END DEFine
430 DEFine PROCedure cook(dish)
440   FOR c = 1 TO 3
450     PRINT ! dish(c) ! item$(c)
460   END FOR c
470 END DEFine
480 REMark *** PROGRAM DATA ***
490 DATA "Prawns", 3.5, "Chicken",2.8,"Special",3.3
```

Output

The output depends on the random choice of dishes but the following choice illustrates the pattern, and gives a sample of output.

```
3 Prawns
1 Chicken
2 Special
Total cost is £20.40
```

COMMENT Obviously the use of procedures and parameters in such a simple program is not necessary but imagine that each sub-task might be much more complex. In such a situation the use of procedures would allow a modular build-up of the program with testing at each stage. The above example merely illustrates the main notations and relationships of procedures.

Similarly the next example illustrates the use of functions.

Note that in the previous example the procedures *waiter* and *compute* both return exactly one value. Rewrite the procedures as functions and show any other changes necessary as a consequence

```
DEFine FuNction waiter(dish)
  cook dish
  RETurn compute(dish)
END DEFine

DEFine FuNction compute(dish)
  LET total = 0
  FOR k = 1 TO 3
    LET total = total + dish(k)*price(k)
  END FOR k
RETurn total
END DEFine
```

The function call to *waiter* also takes a different form

```
LET bill = waiter(dish)
```

This program works as before. Notice that there are fewer parameters though the program structure is similar. That is because the function names are also serving as parameters retuning information to the source of the function call.

EXAMPLE All the variables used as formal parameters in procedures or functions are 'safe' because they are automatically local. Which variables used in the procedures or functions are not local? What additional statements would be needed to make them local?

Program Changes The variables *k*, *pick* and *num* are not local. The necessary changes to make them so are:

```
LOCAL k
LOCAL pick, num
```

TYPELESS PARAMETERS

Formal parameters do not have any type. You may prefer that a variable which handles numbers has the appearance of a numeric variable and which handles strings looks like a string variable, but however you write your parameters they are typeless. To prove it, try the following program.

Program

```
100 REMark Number or word
110 waiter 2
120 waiter "Chicken"
130 DEFine PROCedure waiter(item)
140   PRINT ! item !
150 END DEFine
```

Output

```
2 Chicken
```

The type of the parameter is determined only when the procedure is called and an actual parameter 'arrives'.

SCOPE OF VARIABLES

Consider the following program and try to consider what two numbers will be output.

```
100 REMark scope
110 LET number = 1
120 test
130 DEFine PROCedure test
140   LOCal number
150   LET number = 2
160   PRINT number
170   try
```

```
180 END DEFine
190 DEFine PROCedure try
200    PRINT number
210 END DEFine
```

Obviously the first number to be printed will be 2 but is the variable *number* in line 200 global?

The answer is that the value of *number* in line 160 will be carried into the procedure *try*. A variable which is local to a procedure will be the same variable in a second procedure called by the first.

Equally if the procedure *try* is called by the main program, the variable *number* will be the same number in both the main program and procedure, *try*. The implications may seem strange at first but they are logical.

1. The variable *number* in line 110 is global.
2. The variable *number* in procedure *test* is definitely local to the procedure.
3. The variable *number* in procedure *try* 'belongs' to the part of the program which was the last call to it.

We have covered many concepts in this chapter because SuperBASIC functions and procedures are very powerful. However, you should not expect to use all these features immediately. Use procedures and functions in simple ways at first. They can be very effective and the power is there if you need it.

PROBLEMS ON CHAPTER 15

1. Six employees are identified by their surnames only. Each employee has a particular pension fund rate expressed as a percentage. The following data represent the total salaries and pension fund rates of the six employees.

Benson	13,800	6.25
Hanson	8,700	6.00
Johnson	10,300	6.25
Robson	15,000	7.00
Thomson	6,200	6.00
Watson	5,100	5.75

   Write procedures to:

   input the data into arrays
   compute the actual pension fund contributions
   output the lists of names and computed contributions.

   Link the procedures with a main program calling them in sequence.

2. Write a function *select* with two arguments *range* and *miss*. The function should return a random whole number in the given *range* but it should not be the value of *miss*.

   Use the function in a program which chooses a random **PAPER** colour and then draws random circles in random **INK** colours so that none is in the colour of **PAPER**.

3. Re-write the solution to exercise 1 so that a function *pension* takes salary and contribution rate as arguments and returns the computed pension contribution. Use two procedures, one to input the data and one to output the required information using the function *pension*.

4. Write the following:

   a procedure which sets up a 'pack of cards'.
   a procedure which shuffles the cards.
   a function which takes a number as an argument and returns a string value describing the card.
   a procedure which 'deals' and displays four poker hands of five cards each.
   a main program which calls the above procedures.
   (see chapter 16 for discussion of a similar problem)

CHAPTER 16
SOME TECHNIQUES

In this final chapter we present some applications of concepts and facilities already discussed and we show how some further ideas may be applied.

SIMULATION OF CARD PLAYING

It is easy to store and manipulate "playing cards" by representing them with the numbers 1 to 52. This is how you might convert such a number to the equivalent card. Suppose, for example, that the number 29 appears. You may decide that:

cards 1–13 are hearts
cards 14–26 are clubs
cards 27–39 are diamonds
cards 40–52 are spades

and you will know that 29 means that you have a "diamond". You can program the QL to do this with:

```
LET suit = (card-1) DIV 13
```

This will produce a value in the range 0 to 3 which you can use to cause the appropriate suit to be printed. The value can be reduced to the range 1 to 13 by writing:

```
LET value = card MOD 13
IF value = 0 THEN LET value = 13
```

Program

The numbers 1 to 13 can be made to print Ace, 2, 3.... Jack, Queen, King, or, if you prefer it, such phrases as "two of hearts" can be printed. The following program will print the name of the card corresponding to your input number.

```
100 REMark Cards
110 DIM suitname$(4,8),cardval$(13,5),
120 LET f$ = " of"
130 set_up
140 REPeat cards
150   INPUT "Enter a card number 1-52:" ! card
160   IF card <1 OR card> 52 THEN EXIT cards
170   LET suit = (card-1) DIV 13
180   LET value = card MOD 13
190   IF value = 0 THEN LET value = 13
200   PRINT cardval$(value) ! f$ ! suitname$(suit)
210 END REPeat cards
220 DEFine PROCedure set_up
230   FOR s = 1 TO 4 : READ suitname$(s)
240   FOR v = 1 TO 13 : READ cardval$(v)
250 END DEFine
260 DATA "hearts","clubs","diamonds","spades"
270 DATA "Ace","Two","Three","Four","Five","Six","Seven"
280 DATA "Eight","Nine","Ten","Jack","Queen","King"
```

Input and Output

```
13
King of hearts
49
Ten of spades
27
Ace of diamonds
0
```

COMMENT

Notice the use of DATA statements to hold a permanent file of data which the program always uses. The other data which changes each time the program runs is entered through an INPUT statement. If the input data was known before running the program it would be equally correct to use another READ and more DATA statements. This would give better control.

SEQUENTIAL DATA FILES

Numeric File

The following program will establish a file of one hundred numbers.

```
100 REMark Number File
110 OPEN_NEW #6,mdv1_numbers
120 FOR num = 1 TO 100
130    PRINT #6,num
140 END FOR num
150 CLOSE #6
```

After running the program check that the filename 'numbers' is in the directory by typing:

```
DIR mdv1_numbers
```

You can get a view of the file without any special formatting by copying from Microdrive to screen:

```
COPY mdv1_numbers to scr
```

You can also use the following program to read the file and display its records on the screen.

```
100 REMark Read File
110 OPEN_IN #6,mdv1_numbers
120 FOR num = 1 TO 100
130    INPUT #6,item
140    PRINT ! item !
150 END FOR num
160 CLOSE #6
```

If you wish you can alter the program to get the output in a different form.

Character File

In a similar fashion the following programs will set up a file of one hundred randomly selected letters and read them back.

```
100 REMark Letter File
110 OPEN_NEW #6,mdv1_chfile
120 FOR num = 1 TO 100
130    LET ch$ = CHR$(RND(65 TO 90))
140    PRINT #6,ch$
150 END FOR num
160 CLOSE #6
```

```
100 REMark Get Letters
110 OPEN_IN #6,mdv1_chfile
120 FOR num = 1 TO 100
130    INPUT #6,item$
140    PRINT ! item$ !
150 END FOR num
160 CLOSE #6
```

FOR SERVICE MANUALS
CONTACT:
MAURITRON TECHNICAL SERVICES
www.mauritron.co.uk
TEL: 01844 - 351694
FAX: 01844 - 352554

SETTING UP A DATA FILE

Suppose that you wish to set up a simple file of names and telephone numbers.

RON	678462
GEOFF	896487
ZOE	249386
BEN	584621
MEG	482349
CATH	438975
WENDY	982387

The following program will do it.

```
100 REMark Phone numbers
110 OPEN_NEW #6,mdv1_phone
120 FOR record = 1 TO 7
130    INPUT name$,num$
140    PRINT #6;name$;num$
150 END FOR record
160 CLOSE #6
```

Type RUN and enter a name followed by the ENTER key and a number followed by the ENTER key. Repeat this seven times.

Notice that the data is 'buffered'. It is stored internally until the system is ready to transfer a batch to the Microdrive. The Microdrive is only accessed once, as you can tell from looking and listening.

COPY A FILE

Once a file is established, it should be copied immediately as a back-up. To do this type:

```
COPY mdv1_phone TO mdv2_phone
```

READ A FILE

You need to be certain that the file exists in a correct form so you should read it back from a Microdrive and display it on the screen. You can do this easily using:

```
COPY mdv2_phone TO scr
```

The output to the screen will not provide spaces automatically between the name and the number but it will provide a 'newline' at the end of each record. The output will be:

```
RON678462
GEOFF896487
ZOE249386
BEN584621
MEG482349
CATH438975
WENDY982387
```

You can get a more controlled presentation of the data with the following program.

```
100 REMark Read Phone NUmbers
110 OPEN_IN #5,mdv1_phone
120 FOR record = 1 TO 7
130   INPUT #5,rec$
140   PRINT,rec$
150 END FOR record
160 CLOSE #5
```

The data is printed, as before, but this time each pair of fields is held in the variable *rec$* before being printed on the screen. You have the opportunity to manipulate it into any desired form.

Note that more than one string variable may be used at the file creation stage with **INPUT** and **PRINT** but the whole record so created may be retrieved from the Microdrive file with a single string variable (rec$ in the above example).

AN INSERTION SORT

The following colours are available in the low resolution screen mode (in code number order, 0–7).

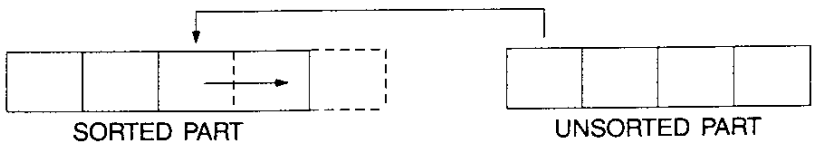
black blue red magenta green cyan yellow white

EXAMPLE

Write a program to sort the colours into alphabetical order using an *insertion* sort.

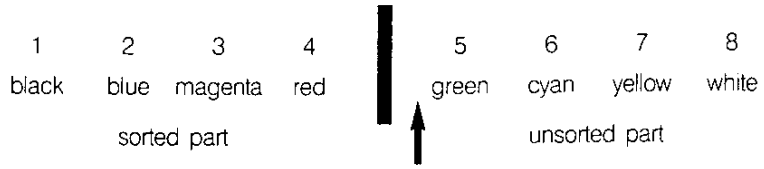
Method

We place the eight colours in an array, *colour$*, which we divide into two parts:



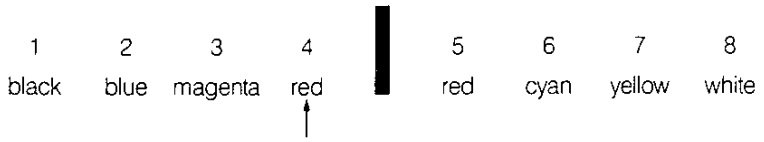
We take the leftmost item of the unsorted part and compare it with each item, from right to left, in the sorted part until we find its right place. As we compare we shuffle the sorted items to the right so that when we find the right place to insert we can do so immediately, without further shuffling.

Suppose we have reached the point where four items are sorted and we now focus on green, the leftmost item in the unsorted part.

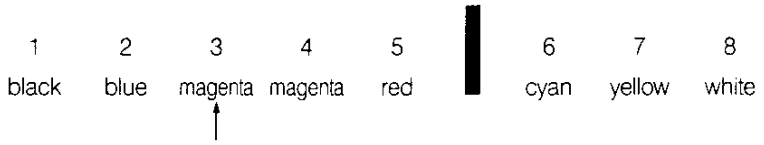


1. We place green in the variable, *comp$*, and set a variable, *p*, to 5.
2. The variable, *p*, will eventually indicate where we think green should go. When we know that green should move left, then we decrease the value of *p*.
3. We compare green with red. If green is greater than (nearer to Z) or equal to red we exit and green stays where it is.

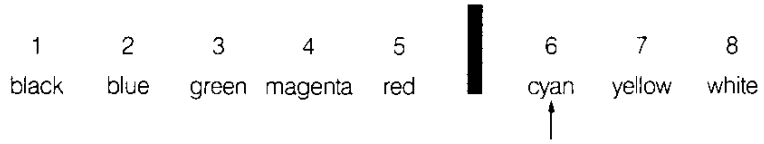
   Otherwise we copy red in to position 5 thus and decrease the value of *p* thus:



4. We now repeat the process but this time we are comparing green with magenta and we get:



5. Finally we move left again comparing green with blue. This time there is no need to move or change anything. We exit from the loop and place green in position 3. We are then ready to focus on the sixth item, cyan.



PROBLEM ANALYSIS

1. We will first store the *colour$* in an array *colour$(8)* and use:

   comp$ the current colour being compared
   p to point at the position where we think the colour in **comp$** might go.
2. A **FOR** loop will focus attention on positions 2 to 8 in turn (a single item is already sorted).
3. A **REPeat** loop will allow comparisons until we find where the *comp$* value actually goes.

```
REPeat compare
  IF comp$ need go no further left EXIT
  copy a colour into the position on its right
  and decrease p
END REPeat compare
```

4. After **EXIT** from the **REPeat** loop the colour in *comp$* is placed in position *p* and the **FOR** loop continues.

Program Design

```
1    Declare array colour$
2    Read colours into the array
3    FOR item = 2 TO 8
       LET p = item
       LET comp$ = colour$(p)
         REPEAT compare
         IF comp$ >= colour$(p-1) : EXIT compare
         LET colour$(p) = colour$(p-1)
         LET p = p-1
       END REPeat compare
       LET colour$(p) = comp$
     END FOR item
4    PRINT sorted array colour$
5    DATA
```

Further testing reveals a fault. It arises very easily if we have data in which the first item is not in its correct position at the start. A simple change of initial data to:

red black blue magenta green cyan yellow white

reveals the problem. We compare black with red and decrease *p* to the value, 1. We come round again and try to compare black with a variable *colour$(p-1)* which is colour$(0) which does not exist.

This is a well-known problem in computing and the solution is to "post a sentinel" on the end of the array. Just before entering the **REPeat** loop we need:

LET colour$(0) = comp$

Fortunately, SuperBASIC allows zero subscripts, otherwise the problem would have to be solved at the expense of readability.

MODIFIED PROGRAM

```
100 REM Insertion Sort
110 DIM colour$(8,7)
120 FOR item = 1 TO 8 : READ colour$(item)
130 FOR item = 2 TO 8
140   LET p = item
150   LET comp$ = colour$(p)
160   LET colour$(0) = comp$
170   REPeat compare
180     IF comp$ >= colour$(p-1) : EXIT compare
190     LET colour$(p) = colour$(p-1)
200     LET p = p-1
210   END REPeat compare
220   LET colour$(p) = comp$
230 END FOR item
240 PRINT "Sorted..." ! colour$
250 DATA "black", "blue", "magenta", "red"
260 DATA "green", "cyan", "yellow", "white"
```

COMMENT

1. The program works well. It has been tested with awkward data:

   A A A A A A A
   B A B A B A B
   A B A B A B A
   B C D E F G H
   G F E D C B A

2. An insertion sort is not particularly fast, but it can be useful for adding a few items to an already sorted list. It is sometimes convenient to allow modest amounts of time frequently to keep items in order rather than a substantial amount of time less frequently to do a complete re-sorting.

You now have enough background knowledge to follow a development of the handling of the file of seven names and telephone numbers.

SORTING A MICRODRIVE FILE

In order to sort the file 'phone' into alphabetical order of names we must read it into an internal array, sort it, and then create a new file which will be in alphabetical order of names.

It is never good practice to delete a file before its replacement is clearly established and proven correct. You should therefore copy the file first, as security, using a different name. The required processes are as follows:

1. Copy the file 'phone' to 'phone_temp'
2. Read the file 'phone' into an array.
3. Sort the array.
4. Pause to check that everything is in order.
5. Delete file 'phone'.
6. Create new file 'phone'.

This is all the program needs to do but the new file should be immediately checked using:

```
COPY mdv1_phone TO scr
```

Any further necessary checks should be carried out then:

```
DELETE mdv2_phone
COPY mdv1_phone TO mdv2_phone
COPY mdv1_phone TO scr
DELETE mdv1_phone_temp
```

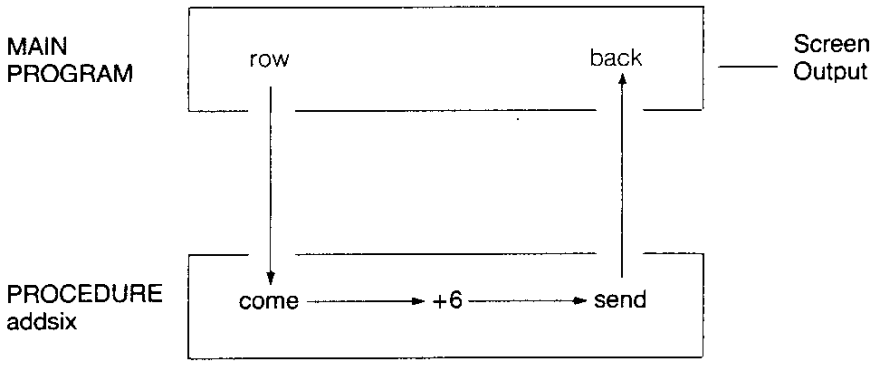
The above operations complete the process of substituting a sorted file for the original unsorted one in both master and back-up files.

ARRAY PARAMETERS

In the following program we illustrate the passing of complete arrays between main program and procedure. The data passes in both directions.

In line 40 the array, *row*, holding the numbers 1,2,3 is passed to the procedure, *addsix*. The parameter, *come*, receives the incoming data and the procedure adds six to each element. The array parameter, *send*, at this point holds the numbers 7,8,9.

These numbers are passed back to the main program to become the values of array, *black*. The values are printed to prove that the data has moved as required.



Program

```
100 REMark Pass Arrays
110 DIM row(3),back(3)
120 FOR k = 1 TO 3 : LET row(k) = k
130 addsix row, back
140 FOR k = 1 TO 3 : PRINT ! back(k) !
150 DEFine PROCedure addsix(come,send)
160    FOR k = 1 TO 3 : LET send(k)=come(k)+6
170 END DEFine
```

Output

```
7 8 9
```

The following procedure receives an array containing data to be sorted. The zero element will contain the number of items. Note that it does not matter whether the array is numeric or string. The principle of coercion will change string to numeric data if necessary.

A second point of interest is that the array element, *come(0)*, is used for two purposes:

it carries the number of items to be sorted
it is used to hold the item currently being placed.

```
100 DEFine PROCedure sort(come,send)
110   LET num = come(0)
120   FOR item = 2 TO num
130     LET p = item
140     LET come(0) = come(p)
150     REPeat compare
160       IF come(0) >= come(p-1) : EXIT compare
170       LET come(p) = come(p-1)
180       LET p = p-1
190     END REPeat compare
200     LET come(p) = come(0)
210   END FOR item
220   FOR k=1 TO 7 : send(k) = come(k)
230 END DEFine
```

The following additional lines will test the sort procedure. First type **AUTO 10** to start the line numbers from 10 onwards.

```
10 REMark Test Sort
20 DIM row$(7,3),back$(7,3)
30 LET row$(0) = 7
40 FOR k = 1 TO 7 : READ row$(k)
50 sort row$,back$
60 PRINT ! back$ !
70 DATA "EEL", "DOG", "ANT", "GNU", "CAT", "BUG", "FOX"
```

Output

```
ANT BUG CAT DOG EEL FOX GNU
```

COMMENT

This program illustrates how easily you can handle arrays in SuperBASIC. All you have to do is use the array names for passing them as parameters or for printing the whole array. Once the procedure is saved you can use **MERGE mdv1_sort** to add it to a program in main memory.

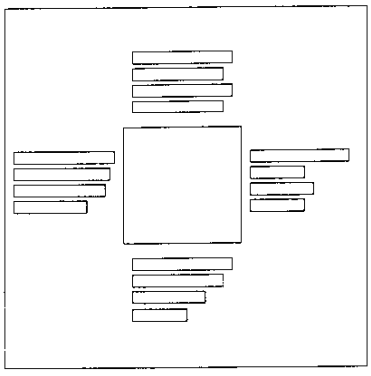
You now have enough understanding of techniques and syntax to handle a more complex screen layout. Suppose you wish to represent the hands of four card players. A hand can be represented by something like:

H: A 3 7 Q
C: 5 9 J
D: 6 10 K
S: 2 4 Q

To help the presentation the Hearts and Diamonds will be printed in red and the Clubs and Spades in black. A suitable **STRIP** colour might be white. The general background could be green and a table may be a colour obtained by mixing two colours.

METHOD

Since a substantial amount of character printing is involved it is best to start planning in terms of the pixel screen. You can see that you need to provide for twelve lines of characters with some space between lines and a total screen height of 256 pixels.



It is useful to recall that the possible character heights are 10 pixels or 20 pixels. It is obvious that the 10 pixel height must be used to allow space for a proper layout.

The number of character positions across the screen must be estimated. If we adopt the convention of "T" for ten instead of "10" all card values can be represented as a single character. Suppose that we also allow a maximum of eight cards of the same suit as a first approach. We can reconsider the problem again if necessary. That would require a total of 10 characters for each hand. The across requirement is therefore:

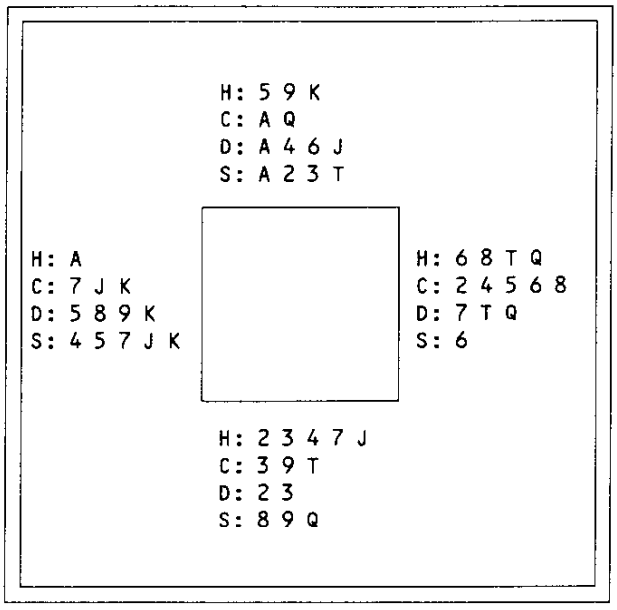
west hand + table width + east hand

Allowing a space between characters that would be:

20 + table width + 20

The decision now depends on which screen mode you choose. The 256 mode will cope with the problem, as you will see later, but first we will work in 512 pixel mode. The smallest character width is six pixels which would give a total of 240 pixels + table width. The diagram will have some balance if we have a table width of about half of 240.

We should therefore experiment with a table width of about 120 pixels which may be adjusted. A little testing produced the layout shown.



WINDOW 440 x 220 at 35,15
Green with black border of 10 units

TABLE 100 x 60 at 150,60
Chequerboard stipple of red and green

HANDS Room for at least eight card symbols.
Initial cursor positions are:

north	150,10
east	260,60
south	150,130
west	30,60

CHARACTER SIZE Standard for 512 mode
NUMBER OF PIXELS between lines is 12

CHARACTER COLOUR White
CHARACTER STRIP Red for Hearts and Diamonds
Black for Clubs and Spades.

FOR SERVICE MANUALS
CONTACT:
MAURITRON TECHNICAL SERVICES
www.mauritron.co.uk
TEL: 01844 - 351694
FAX: 01844 - 352554

VARIABLES

card(52)	stores card numbers
sort(13)	used to sort each hand
tok$(4,2)	stores tokens H:, C:, D:, S:
kmcmh	working loop variables
ran	random position for card exchange
temp	used in card exchange
item	card to be inserted in sort
dart	pointer to find position in sort
comp	hold card number in sort
inc	pixel increment in card rows
seat	current 'deal' position
ac,dn,	cursor position for characters
row	current row for characters
lin$	builds up row of characters
max	highest card number
p	points to card position
n	current number of card

PROCEDURES

shuffle	shuffles 52 cards
split	splits cards into four hands and calls *sortem* to sort each hand
sortem	sorts 13 cards in ascending order
layout	provides background colour, border and table
printem	prints each line of card symbols
getline	gets one row of cards and converts numbers into the symbols A,2,3,4,5,6,7,8,9,T,J,Q,K

PROGRAM DESIGN OUTLINE

1. Declare arrays, pick up 'tokens' and place 52 numbers in array *card*.
2. Shuffle cards.
3. Split into 4 hands and sort each.
4. OPEN screen window.
5. Fix the screen layout.
6. Print the four hands.
7. CLOSE the screen window.

```
100 DIM card(52),sort(13),tok$(4,2)
110 FOR k = 1 TO 4 : READ tok$(k)
120 FOR k = 1 TO 52 : LET card(k) = k
130 shuffle
140 split
150 OPEN #6,scr_440x220a35x15
160 layout
170 printem
180 CLOSE #6
190 DEFine PROCedure shuffle
200   FOR c = 52 TO 3 STEP -1
210     LET ran = RND(1 TO c-1)
220     LET temp = card(c)
230     LET card(c) = card(ran)
240     LET card(ran) = temp
250   END FOR c
260 END DEFine
270 DEFine PROCedure split
280   FOR h = 1 TO 4
290     FOR c = 1 TO 13
300       LET sort(c) = card((h-1)*13+c)
310     END FOR c
320     sortem
330     FOR c = 1 TO 13
340       LET card((h-1)*13+c) = sort(c)
350     END FOR c
360   END FOR h
370 END DEFine
380 DEFine PROCedure sortem
390   FOR item = 2 TO 13
400     LET dart = item
```

```
410      LET comp = sort(dart)
420      LET sort(0) = comp
430      REPeat compare
440        IF comp >= sort(dart-1) : EXIT compare
450        LET sort(dart) = sort(dart-1)
460        LET dart = dart -1
470      END REPeat compare
480      LET sort(dart) = comp
490    END FOR item
500 END DEFine
510 DEFine PROCedure layout
520    PAPER #6,4 : CLS #6
530    BORDER #6,10,0
540    BLOCK #6,100,60,150,60,2,4
550 END DEFine
560 DEFine PROCedure printem
570    LET inc = 12 : INK #6,7
580    LET p = 0
590    FOR seat = 1 TO 4
600      READ ac,dn
610      FOR row = 1 TO 4
620        getline
630        CURSOR #6,ac,dn
640        PRINT #6,lin$
650        LET dn = dn + inc
660      END FOR row
670    END FOR seat
680 END DEFine
690 DEFine PROCedure getline
700    IF row MOD 2 = 0 THEN STRIP #6,0
710    IF row MOD 2 = 1 THEN STRIP #6,2
720    LET lin$ = tok$(row)
730    LET max = row*13
740    REPeat one_suit
750      LET p = p + 1
760      LET n = card(p)
770      IF n >max THEN p = p-1 : EXIT one_suit
780      LET n = n MOD 13
790      IF n = 0 THEN n = 13
800      IF n = 1 : LET ch$ = "A"
810      IF n >= 2 AND n <= 9 : LET ch$ = n
820      IF n = 10 : LET ch$ = "T"
830      IF n = 11 : LET ch$ = "J"
840      IF n = 12 : LET ch$ = "Q"
850      IF n = 13 : LET ch$ = "K"
860      LET lin$ = lin$ & " " & ch$
870      IF p = 52 : EXIT one=suit
880      IF card(p)>card(p+1) : EXIT one_suit
890    END REPeat one_suit
900 END DEFine
910 DATA "H:","C:","D:","S:"
920 DATA 150,10,260,60,150,130,30,60
```

COMMENT

The program works in the 256 mode But the various lines of card symbols may overlap the "table" or overflow at the edge of the window. A simple change in procedure *getline* from:

```
860 LET lin$ = lin$ & " " & ch$
```

to:

```
860 LET lin$ = lin$ & ch$
```

will correct this. The spaces between characters disappear but the larger sized characters enable the rows to be easily readable. The program thus works well in either graphics mode.

CONCLUSION

We have tried to show how you can use SuperBASIC to solve problems. We have shown how simple tasks can be performed in simple ways. When the task is inherently complex, like manipulating arrays or designing screen graphics, SuperBASIC enables it to be handled efficiently with maximum possible clarity.

If you were a beginner and you have worked through a fair proportion of this guide you will have started well on the road to good programming. If you were already experienced, we hope that you will appreciate and exploit the extra features offered by SuperBASIC.

So enormous is the range of tasks which can be done with SuperBASIC that we have only been able to touch a fraction of them in this guide. We cannot guess at which of the thousands of possibilities you will attempt, but we hope that you will find them fruitful, stimulating and fun.

ANSWERS TO SELF TEST ON CHAPTER 1

1. Use the BREAK sequence to abandon a running program because:
   a) something is wrong and you do not understand it
   b) it is longer of interest
   c) any other problem (three points)
2. The RESET button is on the right hand side of the computer
3. The effect of the RESET button is rather like switching the computer off and on again.
4. The **SHIFT** key:
   a) is only effective while you hold it down whereas the **CAPS LOCK** key stays effective after you have pressed it. (one point)
   b) The **SHIFT** key affects all the letter, digit and symbol keys, but the **CAPS LOCK** key affects only letters. (one point)
5. The **CTRL** ⇦ keys delete the previous character just left of the cursor.
6. The ⬅ (ENTER) key causes a message or instruction to be entered for action by the computer.
7. We use ⬅ for the **ENTER** key.
8. **CLS** ⬅ causes part of the screen to be cleared.
9. **RUN** ⬅ causes a stored program to be executed.
10. **LIST** ⬅ causes a stored program to be displayed on the screen.
11. **NEW** ⬅ clears the main memory ready for a new program.
12. Keywords of SuperBASIC are recognised in upper or lower case.
13. The part of a keyword displayed in upper case is the allowed abbreviation.

CHECK YOUR SCORE

14 to 16 is very good. Carry on reading.

12 or 13 is good, but re-read some parts of chapter one.

10 or 11 is fair, but re-read some parts of chapter one and do the test again.

Under 10. You should work carefully through chapter one again and repeat the test.

FOR SERVICE MANUALS
CONTACT:
MAURITRON TECHNICAL SERVICES
www.mauritron.co.uk
TEL: 01844 - 351694
FAX: 01844 - 352554

ANSWERS TO SELF TEST ON CHAPTER 2

1. An internal number store is like a pigeon hole which you can name and put numbers into.
2. A **LET** statement which uses a particular name for the first time will cause a pigeon hole to be created and named, for example

   LET count = 1 ⬅ (1 point)

   A **READ** statement which uses a name for the first time will have the same effect, for example

   READ count ⬅ (1 point)
3. You can find the value of a pigeon hole with a **PRINT** statement.
4. The technical name for a pigeon hole is 'variable' because its values can vary as a program runs.
5. A variable gets its first value when it is first used in a **LET** statement, **INPUT** statement or **READ** statement.
6. A change in the value of a variable is usually caused by the execution of a **LET** statement.
7. The = sign in a **LET** statement represents an operation:

   'Evaluate whatever is on the right hand side and place it in the pigeon hole named on the left hand side', that is 'Let the left hand side become equal to the right hand side'.
8. An un-numbered statement is executed immediately.
9. A numbered statement is not executed immediately. It is stored.
10. The quotes in a **PRINT** statement enclose text which is to be printed.
11. When quotes are not used you are printing out the value of a variable.
12. An **INPUT** statement makes the program pause so that you can type data at the keyboard.
13. **DATA** statements are never executed.
14. They are used to provide values for the variables in **READ** statements.
15. The technical word for the name of a pigeon hole is 'identifier'.
16. Example answers:

    i. day
    ii. day__23
    iii. day__of__week (3 points)
17. The space bar is especially important for putting spaces after or before keywords so that they cannot be taken as identifiers (names) chosen by the user.
18. Freely chosen identifiers are important because they help you to make programs easier to understand. Such programs are less prone to errors and more adaptable.

CHECK YOUR SCORE

18 to 21 is very good. Carry on reading.

16 or 17 good but re-read some parts of chapter two.

14 or 15 fair, but re-read some parts of chapter two and do the test again.

Under 14 you should work carefuly through chapter two again and repeat the test.

ANSWERS TO SELF TEST ON CHAPTER 3

1. A pixel is the smallest area of light that can be displayed on the screen.
2. There are 256 pixel positions across the low resolution mode.
3. There are 256 pixel positions from top to bottom in the low resolution mode.
4. An address is determined by:
   the up value, 0 to 100
   the across value, 0 to a number computed by the system
5. There are eight colours available in the low resolution mode including black and white.
6. i. LINE draws a line, e.g. LINE a,b TO x,y
   ii. INK selects a colour for drawing, e.g. INK 5
   iii. PAPER selects a background colour, e.g. PAPER 7
   iv. BORDER draws a border, e.g. BORDER 1,5
7. REPeat *name*....END REPeat *name*.
8. A REPeat loop terminates when an 'EXIT name' statement is executed.
9. Loops in SuperBASIC have names so that it is possible to EXIT from them in a straightforward way. It is not necessary to work out line numbers in advance.

CHECK YOUR SCORE

11 to 13 is very good. Carry on reading.
8 to 10 is good but re-read some parts of chapter three.
6 or 7 is fair but re-read some parts of chapter three and do the test again.
Under 6. You should work carefully through chapter three again and repeat the test.

ANSWERS TO SELF TEST ON CHAPTER 4

1. A character string is a sequence of characters such as letters, digits or other symbols.
2. The term, 'character string', is often abbreviated to 'string'.
3. A string variable name always ends with $.
4. Names such as *word$* are sometimes pronounced 'worddollar'.
5. The keyword LEN will find the length or number of characters in a string. For example, if the variable *meat$* has the value 'steak' then the statement:

   ```
   PRINT LEN(meat$)
   ```

   will output 5.
6. The symbol for joining two strings is &.
7. The limits of a string may be defined by quotes or apostrophes.
8. The quotes are not part of the actual string and are not stored.
9. The function is CHR$. You must use it with brackets as in CHR$(66) or with brackets as in CHR$(RND(65 TO 67)).
10. You generate random letters with statements like:

    ```
    lettercode = RND(65 TO 90)
    PRINT CHR$(lettercode)
    ```

CHECK YOUR SCORE

9 or 10 is very good. Carry on reading.
7 or 8 is good but re-read some parts of chapter four.
5 or 6 is fair but re-read some parts of chapter four and do the test again.
Under 5 You should work carefully through chapter four again and repeat the test.

ANSWERS TO SELF TEST ON CHAPTER 5

1. Lower case letters for variable names or loop names contrast with the keywords which are at least partly displayed in upper case.
2. Indenting reveals clearly what is the extent and content of loops (and other structures).
3. Identifiers (names) should normally be chosen so that they mean something, for example, *count* or *word$* rather than *C* or *W$*.
4. You can edit a stored program by:

   replacing a line
   inserting a line
   deleting a line (three points)
5. The **ENTER** key must be used to enter a command or program line.
6. The word **NEW** will wipe out the previous SuperBASIC program in the QL and will ensure that a new program which you enter will not be merged with an old one.
7. If you wish a line to be stored as part of a program then you must use a line number.
8. The word **RUN** followed by ⏎ will cause a program to execute.
9. The word **REMark** enables you to put into a program information which is ignored at execution time.
10. The keywords **SAVE** and **LOAD** enable programs to be stored on and retrieved from cartridges. (two points).

CHECK YOUR SCORE

12 to 14 is very good. Carry on reading.

10 or 11 is good but re-read some parts of chapter five.

8 or 9 is fair but re-read some parts of chapter five and do the test again.

Under 8 You should re-read chapter five carefully and do the test again.

ANSWERS TO SELF TEST ON CHAPTER 6

1. It is not easy to think of many different variable names for storing the data. If you can think of enough names, every one has to be written in a **LET** statement or a **READ** statement if you do not use arrays.
2. A number, called the subscript, is part of an array variable name. All the variables in an array share one name but each has a different subscript.
3. You must 'declare' an array giving its size (dimension) in a **DIM** statement usually placed near the beginning of a program before the declared array is used.
4. The distinguishing number of an array variable is called the subscript.
5. Houses in a street share the same street name but each has its own number.

   Beds in a hospital ward may share the name of the ward but each bed may be numbered.

   Cells in a prison block may have a common block name but a different number.

   Holes on a golf course, e.g. the fifth hole at Royal Birkdale.
6. A **FOR** loop terminates when the process corresponding to the last value of the loop variable has been completed.
7. A **FOR** loop's name is also the name of the variable which controls the loop.
8. The two phrases for this variable are 'loop variable' or 'control variable'.
9. The values of a loop variable may be used as subscripts for array variable names. Thus, as the loop proceeds, each array variable is 'visited' once.
10. Both **FOR** loops and **REPeat** loops:

    a. have an opening keyword:

       REPeat , FOR

    b. have a closing statement:

       END REPeat *name*, **END FOR** *name*

    c have a loop name.

    Only the **FOR** loop has

    d. a loop variable or control variable. (four points)

CHECK YOUR SCORE

This test is more searching than the previous ones.

15 or 16 is excellent. Carry on reading.

13 or 14 is very good but think a bit more about some of the ideas. Look at programs to see how they work.

11 or 12 is good but re-read some parts of chapter six.

8 to 10 is fair but re-read some parts of chapter six and do the test again.

Under 8 You should re-read chapter six carefully and do the test again.

ANSWERS TO SELF TEST ON CHAPTER 7

1. We normally break down large or complex jobs into smaller tasks until they are small enough to be completed.
2. This principle can be applied in programming by breaking the total job down and writing a procedure for each task.
3. A simple procedure is:

   a separate block of code
   properly named. (two points)
4. A procedure call ensures that:

   the procedure is activated
   control returns to just after the calling point. (two points)
5. Procedure names can be used in a main program before the procedures have been written. This enables you to think about the whole job and get an overview without worrying about the detail.
6. If you write a procedure definition before using its name you can test it and then when it works properly forget the details. You need only remember its name and roughly what it does.
7. A programmer who can write up to thirty line programs can break down a complex task into procedures in such a way that none is more than thirty lines and most are much less. In this way he need only worry about one bit of the job at a time.
8. The use of a procedure would save memory space if it is necessary to call it more than once from different parts of a program. The definition of a procedure only occurs once but it can be called as often as necessary.
9. A main program can place information in 'pigeon-holes' by means of LET or READ statements. These 'pigeon-holes' can be accessed by the procedure. Thus the procedure uses information originally set up by the main program.

   A second method is to use parameters in the procedure call. These values are passed to variables in the procedure definition which then uses them as necessary.
10. An actual parameter is the actual value passed from a procedure call in a main program to a procedure.
11. A formal parameter is a variable in a procedure definition which receives the value passed to the procedure by the main program.

FOR SERVICE MANUALS
CONTACT:
MAURITRON TECHNICAL SERVICES
www.mauritron.co.uk
TEL: 01844 - 351694
FAX: 01844 - 352554

CHECK YOUR SCORE

This is a searching test. You may need more experience of using procedures before the ideas can be fully appreciated. But they are very powerful and, when understood, extremely helpful ideas. They are worth whatever effort is necessary.

12 to 14 excellent. Read on with confidence.

10 or 11 very good. Just check again on certain points.

8 or 9 good but re-read some parts of chapter seven.

6 or 7 fair but re-read some parts of chapter seven. Work carefully through the programs writing down all changes in variable values. Then do the test again.

Under 6 read chapter seven again. Take it slowly working all the programs. These ideas may not be easy but they are worth the effort. When you are ready, take the test again.

sinclair

QL Keywords

The Keyword Reference Guide lists all SuperBASIC keywords in alphabetical order. A brief explanation of the keywords function is given followed by loose definition of the syntax and examples of usage. An explanation of the syntax definition is given in the *Concept Reference Guide* under the entry *syntax*.

Each keyword entry indicates to which, if any, group of operations it relates, i.e. **DRAW** is a *graphics operation* and further information can be obtained from the *graphics* section of the *Concept Reference Guide*.

Sometimes it is necessary to deal with more than one keyword at a time, i.e. **IF, ELSE, THEN, END, IF,** these are all listed under **IF**.

An index is provided which attempts to cover all possible ways you might describe a SuperBASIC keyword. For example the clear screen command, **CLS**, is also listed under *clear screen* and *screen clear*.

©1984 SINCLAIR RESEARCH LIMITED
by Stephen Berry *(Sinclair Research Limited)*

ABS
maths functions

ABS returns the absolute value of the parameter. It will return the value of the parameter if the parameter is positive and will return zero minus the value of the parameter if the parameter is negative.

syntax: ABS(*numeric_expression*)

example:
i. `PRINT ABS(0.5)`
ii. `PRINT ABS(a-b)`

ACOS, ASIN ACOT, ATAN
maths functions

ACOS and ASIN will compute the arc cosine and the arc sine respectively. ACOT will calculate the arc cotangent and ATAN will calculate the arc tangent. There is no effective limit to the size of the parameter.

syntax: *angle*:= *numeric_expression* [in radians]

ACOS(*angle*) ASIN(*angle*)
ACOT(*angle*) ATAN(*angle*)

example:
i. `PRINT ATAN(angle)`
ii. `PRINT ASIN(1)`
iii. `PRINT ACOT(3.6574)`
iv. `PRINT ATAN(a-b)`

ADATE

clock

ADATE allows the *clock* to be adjusted.

syntax: *seconds:= numeric__expression*

ADATE *seconds*

example:
i. `ADATE 3600` {will advance the clock 1 hour}
ii. `ADATE -60` {will move the clock back 1 minute}

ARC
ARC__R

graphics

ARC will draw an arc of a circle between two specified points in the *window* attached to the default or specified **channel**. The end points of the arc are specified using the *graphics co-ordinate* system.

Multiple arcs can be drawn with a single ARC command.

The end points of the arc can be specified in absolute coordinates (relative to the *graphics origin* or in relative coordinates (relative to the *graphics cursor*). If the first point is omitted then the arc is drawn from the graphics cursor to the specified point through the specified angle.

ARC will always draw with absolute coordinates, while ARC__R will always draw relative to the graphics cursor.

syntax:
x:= numeric__expression
y:= numeric__expression
angle:= numeric__expression {in radians}
point:= x,y

parameter__2:= | TO *point, angle* 1
| *,point* TO *point,angle* 2

parameter__1:= | *point* TO *point,angle* 1
| TO *point,angle* 2

ARC [*channel,*] *parameter__1* *[*parameter__2*] *
ARC__R [*channel,*] *parameter__1* *[*parameter__2*] *

where 1 will draw from the specified point to the next specified point turning through the specified angle
2 will draw from the the last point plotted to the specified point turning through the specified angle

example:
i. `ARC 15,10 TO 40,40, PI/2`
{draw an arc from 15,10 to 40,40 turning through $\pi/2$ radians}

ii. `ARC TO 50,50,PI/2`
{draw an arc from the last point plotted to 50,50 turning through $\pi/2$ radians}

iii. `ARC_R 10,10 TO 55,45,0.5`
{draw an arc, starting 10,10 from the last point plotted to 55,45 from the start of the arc, turning through 0.5 radians}

AT
windows

AT allows the print position to be modified on an imaginary row/column grid based on the current character size. AT uses a modified form of the *pixel coordinate system* where (row 0, column 0) is in the top left hand corner of the window. AT affects the print position in the window attached to the specified or default channel.

syntax: *line*:= *numeric_expression*
column:= *numeric_expression*

AT [*channel*,] *line* , *column*

example:
```
AT 10,20 : PRINT "This is at line 10 column 20"
```

AUTO

AUTO allows line numbers to be generated automatically when entering programs directly into the computer. AUTO will generate the next number in sequence and will then enter the SuperBASIC line editor while the line is typed in. If the line already exists then a copy of the line is presented along with the line number. Pressing ENTER at any point in the line will check the syntax of the whole line and will enter it into the program.

AUTO is terminated by pressing

CTRL space

syntax: *first_line*:= *line_number*
gap:= *numeric_expression*

AUTO [*first_line*] [,*gap*]

example:
i. `AUTO` {start at line 100 with intervals of 10}
ii. `AUTO 10,5` {start at line 10 with intervals of 5}
iii. `AUTO ,7` {start at line 100 with intervals of 7}

BAUD

communications

BAUD sets the baud rate for communication via both serial channels. The speed of the channels cannot be set independently.

syntax: *rate*:= *numeric_expression*

BAUD *rate*

The value of the numeric expression must be one of the supported baud rates on the QL:

75
300
600
1200
2400
4800
9600
19200 (transmit only)

If the selected baud rate is not supported, then an error will be generated.

example:
i. `BAUD 9600`
ii. `BAUD print_speed`

BEEP

sound

BEEP activates the inbuilt sound functions on the QL. BEEP can accept a variable number of parameters to give various levels of control over the sound produced. The minimum specification requires only a duration and pitch to be specified. BEEP used with no parameters will kill any sound being generated.

syntax:

duration:= *numeric_expression* {range −32768 .. 32767}
pitch:= *numeric_expression* {range 0 .. 255}
grad_x:= *numeric_expression* {range −32768 .. 32767}
grad_y:= *numeric_expression* {range −8 .. 7}
wrap:= *numeric_expression* {range 0 .. 15}
fuzzy:= *numeric_expression* {range 0 .. 15}
random:= *numeric_expression* {range 0 .. 15}

BEEP [*duration, pitch*
[, *pitch_2* , *grad_x*, *grad_y*
[, *wrap*
[, *fuzzy*
[, *random*]]]]]

duration — specifies the duration of the sound in units of 72 microseconds. A duration of zero will run the sound until terminated by another BEEP command.

pitch — specifies the pitch of the sound. A pitch of 1 is high and 255 is low.

pitch_2 — specifies an second pitch level between which the sound will 'bounce'.

grad_x — defines the time interval between pitch steps.

grad_y — defines the size of each step. *grad_x* and *grad_y* control the rate at which the pitch bounces between levels.

wrap — will force the sound to wrap around the specified number of times. If wrap is equal to 15 the sound will wrap around forever.

fuzzy — defines the amount of fuzziness to be added to the sound.

random — defines the amount of randomness to be added to the sound.

BEEPING
sound

BEEPING is a function which will return zero (false) if the QL is currently not beeping and a value of one (true) if it is beeping.

syntax: BEEPING

example:
```
100 DEFine PROCedure be_quiet
110   BEEP
120 END DEFine
130 IF BEEPING THEN be_quiet
```

BLOCK
windows

BLOCK will fill a block of the specified size and shape, at the specified position relative to the origin of the *window* attached to the specified, or default *channel*.

BLOCK uses the *pixel coordinate system*.

syntax:
width:= *numeric_expression*
height:= *numeric_expression*
x:= *numeric_expression*
y:= *numeric_expression*

BLOCK [*channel*,] *width*, *height*, *x*, *y* , *colour*

example: i.
```
BLOCK 10, 10, 5, 5, 7
```
{a 10x10 pixel white block at 5,5}

ii.
```
100 REMark "bar chart"
110 CSIZE 3,1
120 PRINT "bar chart"
130 LET bottom = 100 : size = 20 : left = 10
140 FOR bar = 1 to 10
150   LET colour = RND(0 TO 255)
160   LET height = RND(2 TO 20)
170   BLOCK size, height, left+bar*size,
        bottom-height,0
180   BLOCK size-2, height-2, left+bar*size+1,
        bottom-height+1,colour
190 END FOR bar
```
{use `LET colour = RND(0 TO 7)` for televisions}

BORDER

windows

BORDER will add a border to the *window* attached to the specified *channel*, or default channel.

For all subsequent operations except BORDER the window size is reduced to allow space for the BORDER. If another BORDER command is used then the full size of the original window is restored prior to the border being added; thus multiple BORDER commands have the effect of changing the size and colour of a single border. Multiple borders are not created unless specific action is taken.

If BORDER is used without specifying a colour then a transparent border of the specified width is created.

syntax: *width*:= *numeric_expression*

BORDER [*channel*,] *size* [, *colour*]

example: i.
```
BORDER 10,0,7
```
{black and white stipple border}

ii.
```
100 REMark Lurid Borders
110 FOR thickness = 50 to 2 STEP -2
120   BORDER thickness, RND(0 TO 255)
130 END FOR thickness
140 BORDER 50
```
{use RND(0 TO 7) for televisions}

CALL

Qdos

Machine code can be accessed directly from SuperBASIC by using the CALL command. CALL can accept up to 13 long word parameters which will be placed into the 68008 data and address registers (D1 to D7, A0 to A5) in sequence.

No data is returned from CALL.

syntax: *address*:= *numeric_expression*
data:= *numeric_expression*

CALL *address*, *[*data*]* {13 data parameters maximum}

example: i.
```
CALL 262144,0,0,0
```
ii.
```
CALL 262500,12,3,4,1212,6
```

warning

Address register A6 should not be used in routines *called using* this command. To return to SuperBASIC use the instructions:

```
MOVEQ #0,D0
RTS
```

CHR$
BASIC

CHR$ is a function which will return the character whose value is specified as a parameter.

CHR$ is the inverse of CODE.

syntax: CHR$ (*numeric__expression*)

example:
i. `PRINT CHR$(27)` {print ASCII escape character}
ii. `PRINT CHR$(65)` {print A}

FOR SERVICE MANUALS
CONTACT:
MAURITRON TECHNICAL SERVICES
www.mauritron.co.uk
TEL: 01844 - 351694
FAX: 01844 - 352554

CIRCLE
CIRCLE__R
graphics

CIRCLE will draw a circle (or an ellipse at a specified angle) on the screen at a specified position and size. The circle will be drawn in the *window* attached to the specified or default channel.

CIRCLE uses the *graphics coordinate system* and can use absolute coordinates (i.e. relative to the *graphics origin*), and relative coordinates (i.e. relative to the *graphics cursor*). For relative coordinates use CIRCLE__R.

Multiple circles or ellipses can be plotted with a single call to CIRCLE. Each set of parameters must be separated from each other with a semi colon (;)

The word ELLIPSE can be substituted for CIRCLE if required.

syntax:

x:= *numeric__expression*
y:= *numeric__expression*
radius:= *numeric__expression*
eccentricity:= *numeric__expression*
angle:= *numeric__expression* {range 0..2π}

parameters:= | *x*, *y*, 1
| *radius*, eccentricity, angle 2

where 1 will draw a circle
2 will draw an ellipse of specified eccentricity and angle

CIRCLE [*channel*,] parameters *[; *parameters*]*

x	horizontal offset from the graphics origin or graphics cursor
y	vertical offset from the graphics origin or graphics cursor
radius	radius of the circle
eccentricity	the ratio between the major and minor axes of an ellipse.
angle	the orientation of the major axis of the ellipse relative to the screen vertical. The angle must be specified in radians.

example:
i. `CIRCLE 50,50,20` {a circle at 50,50 radius 20}
ii. `CIRCLE 50,50,20,0.5,0` {an ellipse at 50,50 major axis 20 eccentricity 0.5 and aligned with the vertical axis}

CLEAR

CLEAR will clear out the SuperBASIC variable area for the current program and will release the space for Qdos.

syntax: CLEAR

example:
```
CLEAR
```

comment

CLEAR can be used to restore to a known state the SuperBASIC system. For example, if a program is broken into (or stops due to an error) while it is in a procedure then SuperBASIC is still in the procedure even after the program has stopped. CLEAR will reset the SuperBASIC. (See CONTINUE, RETRY.)

CLOSE

devices

CLOSE will close the specified *channel*. Any *window* associated with the channel will be deactivated.

syntax: *channel*:= #*numeric_expression*

CLOSE *channel*

example:

i.
```
CLOSE #4
```

ii.
```
CLOSE #input_channel
```

CLS
windows

Will clear the *window* attached to the specified or default *channel* to current PAPER colour, excluding the border if one has been specified. CLS will accept an optional parameter which specifies if only a part of the window must be cleared.

syntax: *part*:= *numeric_expression*

CLS [*channel*,] [*part*]

where: *part* = 0 - whole screen (default if no parameter)
part = 1 - top excluding the cursor line
part = 2 - bottom excluding the cursor line
part = 3 - whole of the cursor line
part = 4 - right end of cursor line including the cursor position

example:
i. `CLS` {the whole window}
ii. `CLS 3` {clear the cursor line}
iii. `CLS #2,2` {clear the bottom of the window on channel 2}

CODE

CODE is a function which returns the internal code used to represent the specified character. If a string is specified then CODE will return the internal representation of the first character of the string.

CODE is the inverse of CHR$.

syntax: CODE (*string_expression*)

example:
i. `PRINT CODE("A")` {prints 65}
ii. `PRINT CODE("SuperBASIC")` {prints 83}

CONTINUE
RETRY

error handling

CONTINUE allows a program which has been halted to be continued. RETRY allows a program statement which has reported an error to be re-executed.

syntax: CONTINUE
RETRY

example:
```
CONTINUE
RETRY
```

warning

A program can only continue if:

1. No new lines have been added to the program
2. No new variables have been added to the program
3. No lines have been changed

The value of variables may be set or changed.

COPY
COPY_N

devices

COPY will copy a file from an input device to an output device until an end of file marker is detected. COPY_N will not copy the header (if it exists) associated with a file and will allow Microdrive files to be correctly copied to another type of device.

Headers are associated with directory-type devices and should be removed using COPY_N when copying to non-directory devices, e.g. **mdv1** is a directory device; **ser1** is a non-directory device.

syntax: COPY *device* TO *device*
COPY_N *device* TO *device*

It must be possible to input from the source device and it must be possible to output to the destination device.

example:

i. `COPY mdv1_data_file TO con_` {copy to default window}

ii `COPY neti_3 TO mdv1_data` {copy data from network station to mdv_data.}

iii. `COPY_N mdv1_test_data TO ser1` {copy mdv1_test_data to serial port 1 removing header information}

COS
maths functions

COS will compute the cosine of the specified argument.

syntax: *angle*:= *numeric_expression* {range −10000..10000 in radians}

COS (angle)

example:
i. `PRINT COS(theta)`
ii. `PRINT COS(3.141592654/2)`

COT
maths functions

COT will compute the cotangent of the specified argument.

syntax: *angle*:= *numeric_expression* {range −30000..30000 in radians}

COT (*angle*)

example:
i. `PRINT COT(3)`
ii. `PRINT COT(3.141592654/2)`

CSIZE

windows

Sets a new character size for the *window* attached to the specified or default *channel*. The standard size is 0,0 in *512 mode* and 2,0 in *256 mode*.

Width defines the horizontal size of the character space. Height defines the vertical size of the character space. The character size is adjusted to fill the space available.

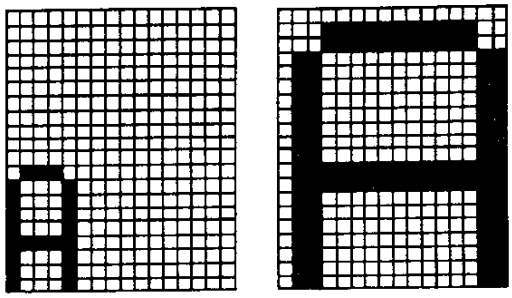


Figure A Character Square

width	size	height	size
0	6 pixels	0	10 pixels
1	8 pixels	1	20 pixels
2	12 pixels		
3	16 pixels		

syntax: *width*:= *numeric_expression* {range 0..3}
height:= *numeric_expression* {range 0..1}

CSIZE [*channel*,] *width*, *height*

example:
i. `CSIZE 3,0`
ii. `CSIZE 3,1`

CURSOR

windows

CURSOR allows the screen cursor to be positioned anywhere in the window attached to the specified or default *channel*.

CURSOR uses the *pixel coordinate system* relative to the window origin and defines the position for the top left hand corner of the cursor. The size of the cursor is dependent on the character size in use.

If CURSOR is used with four parameters then the first pair is interpreted as graphics coordinates (using the graphics coordinate system) and the second pair as the position of the cursor (in the pixel coordinate system) relative to the first point.

This allows diagrams to be annotated relatively easily.

syntax: *x*:= *numeric_expression*
y:= *numeric_expression*

CURSOR [*channel*,] *x*, *y* [,*x*, *y*]

example:
i. `CURSOR 0,0`
ii. `CURSOR 20,30`
iii. `CURSOR 50,50,10,10`

DATA
READ
RESTORE

BASIC

READ, DATA and RESTORE allow embedded data, contained in a SuperBASIC program, to be assigned to variables at run time.

DATA is used to mark and define the data, READ accesses the data and assigns it to variables and RESTORE allows specific data to be selected.

DATA: allows data to be defined within a program. The data can be read by a READ statement and the data assigned to variables. A DATA statement is ignored by SuperBASIC when it is encountered during normal processing.

syntax: DATA *[*expression*,]*

READ: reads data contained in DATA statements and assigns it to a list of variables. Initially the data pointer is set to the first DATA statement in the program and is incremented after each READ. Re-running the program will not reset the data pointer and so in general a program should contain an explicit RESTORE.

An error is reported if a READ is attempted for which there is no DATA.

syntax: READ *[*identifier*,]*

RESTORE: restores the data pointer, i.e. the position from which subsequent READs will read their data. If RESTORE is followed by a line number then the data pointer is set to that line. If no parameter is specified then the data pointer is reset to the start of the program.

syntax: RESTORE [*line_number*]

example: i.

```
100 REMark Data statement example
110 DIM weekdays$(7,4)
120 RESTORE
130 FOR count= 1 TO 7 :
      READ weekdays$(count)
140 PRINT weekday$
150 DATA "MON","TUE","WED","THUR","FRI"
160 DATA "SAT","SUN"
```

ii.

```
100 DIM month$(12,9)
110 RESTORE
120 REMark Data statement example
130 FOR count= 1 TO 12 :
      READ month$(count)
140 PRINT month$
150 DATA "January","February","March"
160 DATA "April","May","June"
170 DATA "July","August","September"
180 DATA "October","November","December"
```

warning

An implicit RESTORE is not performed before running a program. This allows a single program to run with different sets of data. Either include a RESTORE in the program or perform an explicit RESTORE or CLEAR before running the program.

FOR SERVICE MANUALS
CONTACT:
MAURITRON TECHNICAL SERVICES
www.mauritron.co.uk
TEL: 01844 - 351694
FAX: 01844 - 352554

DATE$
DATE

clock

DATE$ is a function which will return the date and time contained in the QL's *clock*.

The format of the string returned by DATE$ is:

"yyyy mmm dd hh:mm:ss"

where	*yyyy*	is the year 1984, 1985, etc
	mmm	is the month Jan, Feb etc
	dd	is the day 01 to 28, 29, 30, 31
	hh	is the hour 00 to 23
	mm	are the minutes 00 to 59
	ss	are the seconds 00 to 59

DATE will return the date as a floating point number which can be used to store dates and times in a compact form.

If DATE$ is used with a numeric parameter then the parameter will be interpreted as a date in floating point form and will be converted to a date string.

syntax:	DATE$	{get the time from the clock}
	DATE$(*numeric_expression*)	{get time from supplied parameter}
example:	i. `PRINT DATE$`	{output the date and time}
	ii. `PRINT DATE$(234567)`	{convert 234567 to a date}

DAY$

clock

DAY$ is a function which will return the current day of the week. If a parameter is specified then DAY$ will interpret the parameter as a date and will return the corresponding day of the week.

syntax:	DAY$	{get day from clock}
	DAY$ (*numeric_expression*)	{get day from supplied parameter}
example:	i. `PRINT DAY$`	{output the day}
	ii. `PRINT DAY$(234567)`	{output the day represented by 234567 (seconds)}

DEFine FuNction END DEFine

functions and procedures

DEFine FuNction defines a SuperBASIC function. The sequence of statements between the **DEFine** function and the **END DEFine** constitute the function. The function definition may also include a list of *formal parameters* which will supply data for the function. Both the formal and *actual parameters* must be enclosed in brackets. If the function requires no parameters then there is no need to specify an empty set of brackets.

Formal parameters take their type and characteristics from the corresponding *actual parameters*. The type of data returned by the function is indicated by the type appended to the function identifier. The type of the data returned in the **RETURN** statement must match.

An answer is returned from a function by appending an expression to a **RETurn** statement. The type of the returned data is the same as type of this expression.

A function is activated by including its name in a SuperBASIC *expression*.

Function calls in SuperBASIC can be recursive; that is, a function may call itself directly or indirectly via a sequence of other calls.

syntax:

formal_parameters:= (*expression* *[, *expression*]*)
actual_parameters:= (*expression* *[, *expression*]*)

type:= | $
| %
|

DEF FuNction *identifier type* [*formal_parameters*]
  [**LOCal** *identifier* *[, *identifier*]*]
  statements
  RETurn *expression*
END DEFine

RETurn can be at any position within the procedure body. **LOCal** statements must preceed the first executable statement in the function.

example:

```
10 DEFine FuNction mean(a, b, c)
20    LOCal answer
30    LET answer = (a + b + c)/3
40    RETurn answer
50 END DEFine
60 PRINT mean(1,2,3)
```

comment

To improve legibility of programs the name of the function can be appended to the **END DEFine** statement. However, the name will not be checked by SuperBASIC.

DEFine PROCedure
END DEFine

functions and procedures

DEFine PROCedure defines a SuperBASIC procedure. The sequence of statements between the DEFine PROCedure statement and the END DEFine statement constitutes the procedure. The procedure definition may also include a list of *formal parameters* which will supply data for the procedure. The *formal parameters* must be enclosed in brackets for the procedure definition, but the brackets are not necessary when the procedure is called. If the procedure requires no parameters then there is no need to include an empty set of brackets in the procedure definition.

Formal parameters take their type and characteristics from the corresponding *actual parameters*.

Variables may be defined to be LOCal to a procedure. Local variables have no effect on similarly named variables outside the procedure. If required, local arrays should be dimensioned within the LOCal statement.

The procedure is called by entering its name as the first item in a SuperBASIC statement together with a list of actual parameters. Procedure calls in SuperBASIC are recursive; that is, a procedure may call itself directly or indirectly via a sequence of other calls.

It is possible to regard a procedure definition as a command definition in SuperBASIC; many of the system commands are themselves defined as procedures.

syntax:

formal_parameters:= (*expression* *[, *expression*]*)
actual_parameters:= *expression* *[, *expression*]*

DEFine PROCedure *identifier* [*formal_parameters*]
  [LOCal *identifier* *[, *identifier*]*]
  statements
  [RETurn]
END DEFine

RETURN can appear at any position within the procedure body. If present the LOCal statement must be before the first executable statement in the procedure. The END DEFine statement will act as an automatic return.

example:

i.
```
100 DEFine PROCedure start_screen
110   WINDOW 100,100,10,10
120   PAPER 7 : INK 0 : CLS
130   BORDER 4,255
140   PRINT "Hello Everybody"
150 END DEFine
160 start_screen
```

ii.
```
100 DEFine PROCedure slow_scroll(scroll_limit)
110   LOCal count
120   FOR count = 1 TO scroll_limit
130     SCROLL 2
140   END FOR count
150 END DEFine
160 slow_scroll 20
```

comment

To improve legibility of programs the name of the procedure can be appended to the END DEFine statement. However, the name will not be checked by SuperBASIC.

DEG
maths functions

DEG is a function which will convert an angle expressed in radians to an angle expressed in degrees.

syntax: DEG(*numeric_expression*)

example:
```
PRINT DEG(PI/2)
```
{will print 90}

DELETE
Microdrives

DELETE will remove a file from the directory of the cartridge in the specified Microdrive.

syntax: DELETE *device*

The device specification must be a Microdrive device

example:
i.
```
DELETE mdv1_old_data
```
ii.
```
DELETE mdv1_letter_file
```

DIM

arrays

Defines an array to SuperBASIC. *String*, *integer* and *floating point* arrays can be defined. String arrays handle fixed length strings and the final *index* is taken to be the string length.

Array indices run from 0 up to the maximum index specified in the DIM statement; thus DIM will generate an array with one more element in each dimension than is actually specified.

When an array is specified it is initialised to zero for a numeric array and zero length strings for a string array.

syntax: *index*:= *numeric_expression*
array:= *indentifier*(*index* *[, *index*] *)

DIM *array* *[, *array*] *

example:
i. `DIM string_array$(10,10,50)`
ii. `DIM matrix(100,100)`

DIMN

arrays

DIMN is a function which will return the maximum size of a specified dimension of a specified array. If a dimension is not specified then the first dimension is assumed. If the specified dimension does not exist or the identifier is not an array then zero is returned.

syntax: *array*:= *identifier*
index:= *numeric_expression* {1 for dimension 1, etc.}

DIMN(*array* [,dimension])

example: consider the array defined by: `DIM a(2,3,4)`

i. `PRINT DIMN(A,1)` {will print 2}
ii. `PRINT DIMN(A,2)` {will print 3}
iii. `PRINT DIMN(A,3)` {will print 4}
iv. `PRINT DIMN(A)` {will print 2}
v. `PRINT DIMN(A,4)` {will print 0}

DIR
Microdrives

DIR will obtain and display in the *window* attached to the specified or default *channel* the directory of the cartridge in the specified Microdrive.

syntax: DIR *device*

The device specification must be a valid *Microdrive device*

The directory format output by DIR is as follows:

free__sectors:= the number of free sectors
available__sectors:= the maximum number of sectors on this cartridge
file__name:= a SuperBASIC file name

screen format:
Volume name
free__sectors / *available__sectors* sectors
file__name
.
.
file__name

example:

i.
```
DIR mdv1_
```
ii.
```
DIR "mdv2_"
```
iii.
```
DIR "mdv" & microdrive_number$ & "_"
```

screen format:
```
BASIC_
183 / 221 sectors
demo_1
demo_1_old
demo_2
```

DIV
operator

DIV is an operator which will perform an integer divide.

syntax: *numeric__expression* DIV *numeric__expression*

example:

i. `PRINT 5 DIV 2` {will output 2}
ii. `PRINT -5 DIV 2` {will output −3}

DLINE

BASIC DLINE will delete a single line or a range of lines from a SuperBASIC program.

syntax: *range*:= | *line_number* TO *line_number* 1
| *line_number* TO 2
| TO *line_number* 3
| *line_number* 4

DLINE *range* * [,*range*] *

where 1 will delete a range of lines
2 will delete from the specified line to the end
3 will delete from the start to the specified line
4 will delete the specified line

example: i. `DLINE 10 TO 70, 80, 200 TO 400`
{will delete lines 10 to 70 inclusive, line 80 and lines 200 to 400 inclusive}

ii. `DLINE`
{will delete nothing}

EDIT

The EDIT command enters the SuperBASIC line editor.

The EDIT command is closely related to the AUTO command, the only difference being in their defaults. EDIT defaults to a line increment of zero and thus will edit a single line unless a second parameter is specified to define a line increment.

If the specified line already exists then the line is displayed and editing can be started. If the line does not exist then the line number is displayed and the line can be entered.

The cursor can be manipulated within the edit line using the standard QL keystrokes.

[→]	cursor right	
[←]	cursor left	
[↑]	cursor up	same as ENTER but automatically gives previous existing line to edit next
[↓]	cursor down	same as ENTER but automatically gives next existing line to edit next
[CTRL] [→]		delete character right
[CTRL] [←]		delete character left

When the line is correct pressing ENTER will enter the line into the program.

If an *increment* was specified then the next line in the sequence will be edited otherwise edit will terminate.

syntax: *increment*:= *numeric_expression*

EDIT *line_number* [,*increment*]

example: i. `EDIT 10` {edit line 10 only}
ii. `EDIT 20,10` {edit lines 20, 30 etc.}

EOF
devices

EOF is a function which will determine if an end of file condition has been reached on a specified channel. If EOF is used without a channel specification then EOF will determine if the end of a program's embedded data statements has been reached.

syntax: EOF [(*channel*)]

example:
i.
```
IF EOF(#6) THEN STOP
```
ii.
```
IF EOF THEN PRINT "Out of data"
```

EXEC
EXEC_W
Qdos

EXEC and EXEC_W will load a sequence of programs and execute them in parallel.

EXEC will return to the command processor after all processes have started execution, EXEC_W will wait until all the processes have terminated before returning.

syntax: *program*:=device {used to specify a Microdrive file containing the program}

EXEC *program*

example:
i.
```
EXEC mdv1_communcations
```
ii.
```
EXEC_W mdv1_printer_process
```

EXIT

repetition

EXIT will continue processing after the END of the named FOR or REPeat structure.

syntax: EXIT *identifier*

example: i.

```
100 REM start looping
110   LET count = 0
120 REPeat loop
130   LET count = count + 1
140   PRINT count
150   IF count = 20 THEN EXIT loop
160 END REPeat loop
```

{the loop will be exited when count becomes equal to 20}

ii.

```
100 FOR n = 1 TO 1000
110   REM program statements
120   REM program statements
130   IF RND >.5 THEN EXIT n
140 END FOR n
```

{the loop will be exited when a random number greater than 0.5 is generated}

EXP

maths functions

EXP will return the value of e raised to the power of the specified parameter.

syntax: EXP(*numeric_expression*) {range −500..500}

example: i.

```
PRINT EXP(3)
```

ii.

```
PRINT EXP(3.141592654)
```

FILL
graphics

FILL will turn *graphics fill* on or off. FILL will fill any non-re-entrant shape drawn with the *graphics* or *turtle graphics* procedures as the shape is being drawn. Re-entrant shapes must be split into smaller non-re-entrant shapes.

When you have finished filling, FILL 0 should be called.

syntax: *switch*:= *numeric_expression* {range 0..1}

FILL [*channel*,] *switch*

example: i.
```
FILL 1:LINE 10,10 TO 50,50 TO 30,90 TO 10,10:FILL 0
```
{will draw a filled triangle}

ii.
```
FILL 1:CIRCLE 50,50,20:FILL 0
```
{will draw a filled circle}

FILL$
string arrays

FILL$ is a function which will return a string of a specified length filled with a repetition of one or two characters.

syntax: FILL$ (*string_expression,numeric_expression*)

The string expression supplied to FILL$ must be either one or two characters long.

example:
```
i.   PRINT FILL$("a",5)                    {will print aaaaa}
ii.  PRINT FILL$("oO",7)                   {will print oOoOoOo}
iii. LET a$ = a$ & FILL$(" ", 10)
```

FLASH

windows

FLASH turns the flash state on and off. FLASH is only effective in *low resolution mode.*

FLASH will be effective in the *window* attached to the specified or default *channel.*

syntax: *switch*:= *numeric_expression* {range 0..1}

FLASH [*channel*,] *switch*

where: *switch* = 0 will turn the flash off
switch = 1 will turn the flash on

example:

```
100 PRINT "A ";
110 FLASH 1
120 PRINT "flashing ";
130 FLASH 0
140 PRINT "word"
```

warning

Writing over part of a flashing character can produce spurious results and should be avoided.

FOR
END FOR

repetition

The FOR statement allows a group of SuperBASIC statements to be repeated a controlled number of times. The FOR statement can be used in both a long and a short form.

NEXT and END FOR can be used together within the same FOR loop to provide a *loop epilogue*, ie. a group of SuperBASIC statements which will not be executed if a loop is exited via an EXIT statement but which will be executed if the FOR loop terminated normally.

define: *for_item*:= | *numeric_expression*
| *numeric_exp* TO *numeric_exp*
| *numeric_exp* TO *numeric_exp* STEP *numeric_exp*

for_list:= *for_item* *[, *for_item*] *

short

The FOR statement is followed on the same logical line by a sequence of SuperBASIC statements. The sequence of statements is then repeatedly executed under the control of the FOR statement. When the FOR statement is exhausted, processing continues on the next line. The FOR statement does not require its terminating NEXT or END FOR. Single line FOR loops must not be nested.

syntax: FOR *variable* = *for_list* : *statement* *[: *statement*] *

example:

i.
```
FOR i = 1, 2, 3, 4 TO 7 STEP 2 : PRINT i
```

ii.
```
FOR element = first TO last : LET buffer(element) = 0
```

long

The FOR statement is the last statement on the line. Subsequent lines contain a series of SuperBASIC statements terminated by an END FOR statement. The statements enclosed between the FOR statement and the END FOR are processed under the control of the FOR statement.

syntax: FOR *variable* = *for_list*
statements
END FOR *variable*

example:

```
100 INPUT "data please" ! x
110 LET factorial = 1
120 FOR value = x TO 1 STEP -1
130   LET factorial = factorial * value
140   PRINT x !!!! factorial
150   IF factorial>1E20 THEN
160     PRINT "Very large number"
170     EXIT value
180   END IF
190 END FOR value
```

warning

A floating point variable must be used to control a FOR loop.

FORMAT

Microdrives

FORMAT will format and make ready for use the cartridge contained in the specified Microdrive.

syntax: FORMAT [channel,] *device*

Device specifies the Microdrive to be used for formatting and the identifier part of the specification is used as the medium or volume name for that cartridge. FORMAT will write the number of good sectors and the total number of sectors available on the cartridge on the default or on the specified channel.

It is helpful to format a new cartridge several times before use. This conditions the surface of the tape and gives greater capacity.

example: i. `FORMAT mdv1_data_cartridge`
ii. `FORMAT mdv2_wp_letters`

FORMAT can be used to reinitialise a used cartridge. However, all data contained on that cartridge will be lost.

warning

FOR SERVICE MANUALS
CONTACT:
MAURITRON TECHNICAL SERVICES
www.mauritron.co.uk
TEL: 01844 - 351694
FAX: 01844 - 352554

GOSUB

For compatibility with other BASICs, SuperBASIC supports the GOSUB statement. GOSUB transfers processing to the specified line number; a RETurn statement will transfer processing back to the statement following GOSUB.

The line number specification can be an expression.

syntax: GOSUB *line_number*

example i. `GOSUB 100`
ii. `GOSUB 4*select_variable`

The control structures available in SuperBASIC make the GOSUB statement redundant.

comment

GOTO

For compatibility with other BASICs, SuperBASIC supports the GOTO statement. GOTO will unconditionally transfer processing to the statement number specified. The statement number specification can be an expression.

syntax: GOTO *line_number*

example:
```
i.   GOTO program_start
ii.  GOTO 9999
```

comment The control structures available in SuperBASIC make the GOTO statement redundant.

IF
THEN
ELSE
END IF

The IF statement allows conditions to be tested and the outcome of that test to control subsequent program flow.

The IF statement can be used in both a long and a short form:

short The THEN keyword is followed on the same logical line by a sequence of SuperBASIC keyword. This sequence of SuperBASIC statements may contain an ELSE keyword. If the expression in the IF statement is true (evaluates to be non-zero), then the statements between the THEN and the ELSE keywords are processed. If the condition is false (evaluates to be zero) then the statements between the ELSE and the end of the line are processed.

If the sequence of SuperBASIC statements does not contain an ELSE keyword and if the expression in the IF statement is true, then the statements between the THEN keyword and the end of the line are processed. If the expression is false then processing continues at the next line.

syntax: *statements*:= *statement* *[: *statement*]*
IF *expression* THEN *statements* [:ELSE *statements*]

example:
```
i.   IF a=32 THEN PRINT "Limit" : ELSE PRINT "OK"
ii.  IF test >maximum THEN LET maximum = test
iii. IF "1"+1=2 THEN PRINT "coercion OK"
```

long 1 The THEN keyword is the last entry on the logical line. A sequence of SuperBASIC statements is written following the IF statements. The sequence is terminated by the END IF statement. The sequence of SuperBASIC statements is executed if the expression contained in the IF statement evaluates to be non zero. The ELSE keyword and second sequence of SuperBASIC statements are optional.

long 2 The THEN keyword is the last entry on the logical line. A sequence of SuperBASIC statements follows on subsequent lines, terminated by the ELSE keyword. IF the expression contained in the IF statement evaluates to be non zero then this first sequence of SuperBASIC statements is processed. After the ELSE keyword a second sequence of SuperBASIC statements is entered, terminated by the END IF keyword. If the expression evaluated by the IF statement is zero then this second sequence of SuperBASIC statements is processed.

syntax: IF *expression* THEN
statements
[ELSE
statements]
END IF

example:
```
100 LET limit = 10
110 INPUT "Type in a number" ! number
120 IF number > limit THEN
130   PRINT "Range error"
140 ELSE
150   PRINT "Inside limit"
160 END IF
```

comment

In all three forms of the IF statement the THEN is optional. In the short form it must be replaced by a colon to distinguish the end of the IF and the start of the next statement. In the long form it can be removed completely.

nesting

IF statements may be nested as deeply as the user requires (subject to available memory). However, confusion may arise as to which ELSE, END IF etc matches which IF. SuperBASIC will match nested ELSE statements etc to the closest IF statement, for example:

```
100 IF a = b THEN
110   IF c = d THEN
120     PRINT "error"
130   ELSE
140     PRINT "no error"
150   END IF
160 ELSE
170   PRINT "not checked"
180 END IF
```

The ELSE at line 130 is matched to the second IF. The ELSE at line 160 is matched with the first IF (at line 100).

INK

windows

This sets the current ink colour, i.e. the colour in which the output is written. INK will be effective for the *window* attached to the specified or default *channel*.

syntax: INK [*channel*,] *colour*

example:
```
i.   INK 5
ii.  INK 6,2
iii. INK #2,255
```

INKEY$

INKEY$ is a function which returns a single character input from either the specified or default *channel*.

An optional timeout can be specified which can wait for a specified time before returning, can return immediately or can wait for ever. If no parameter is specified then INKEY$ will return immediately.

syntax: INKEY$ [|(*channel*)
|(*channel*, *time*)
|(*time*)|]

where: *time* = 1..32767 {wait for specified number of frames}
time = −1 {wait forever}
time = 0 {return immediately}

example:

i. `PRINT INKEY$` {input from the default channel}
ii. `PRINT INKEY$(#4)` {input from channel 4}
iii. `PRINT INKEY$(50)` {wait for 50 frames then return anyway}
iv. `PRINT INKEY$(0)` {return immediatly (poll the keyboard)}
v. `PRINT INKEY$(#3,100)` {wait for 100 frames for an input from channel 3 then return anyway}

INPUT

INPUT allows data to be entered into a SuperBASIC program directly from the QL keyboard by the user. SuperBASIC halts the program until the specified amount of data has been input; the program will then continue. Each item of data must be terminated by the ENTER key.

INPUT will input data from either the specified or the default *channel*.

If input is required from a particular console channel the cursor for the window connected to that channel will appear and start to flash.

syntax: *separator*:= | !
| ,
| \
| ;
| TO

prompt:= [*channel*,] *expression separator*

INPUT [*prompt*] [*channel*] *variable* *|,*variable*]*

example:

i.
```
INPUT ("Last guess "& guess & "New guess?") !
guess
```

ii.
```
INPUT "What is your guess?"; guess
```

iii.
```
100 INPUT "array size?" ! limit
110 DIM array(limit-1)
120 FOR element = 0 to limit-1
130   INPUT ("data for element" & element) !
          array(element)
140 END FOR element
150 PRINT array
```

INSTR
operator

INSTR is an operator which will determine if a given substring is contained within a specified string. If the string is found then the substring's position is returned. If the string is not found then INSTR returns zero.

Zero can be interpreted as false, i.e. the substring was not contained in the given string. A non zero value, the substrings position, can be intepreted as true, i.e. the substring was contained in the specified string.

syntax: *string_expression* INSTR *string expression*

example:

```
i.   PRINT "a" INSTR "cat"                  {will print 2}
ii.  PRINT "CAT" INSTR "concatenate"        {will print 4}
iii. PRINT "x" INSTR "eggs"                 {will print 0}
```

INT
maths functions

INT will return the integer part of the specified floating point expression.

syntax: INT (*numeric_expression*)

example:

```
i.   PRINT INT(X)
ii.  PRINT INT(3.141592654/2)
```

KEYROW

KEYROW is a function which looks at the instantaneous state of a row of keys (the table below shows how the keys are mapped onto a matrix of 8 rows by 8 columns). **KEYROW** takes one parameter, which must be an integer in the range 0 to 7: this number selects which row is to be looked at. The value returned by **KEYROW** is an integer between 0 and 255 which gives a binary representation indicating which keys have been depressed in the selected row.

Since **KEYROW** is used as an alternative to the normal keyboard input mechanism using **INKEY$** or **INPUT**, any character in the keyboard type-ahead buffer are cleared by **KEYROW**: thus key depressions which have been made before a call to **KEYROW** will not be read by a subsequent **INKEY$** or **INPUT**.

Note that multiple key depressions can cause surprising results. In particular, if three keys at the corner of a rectangle in the matrix are depressed simultaneously, it will appear as if the key at the fourth corner has also been depressed. The three special keys **CTRL**, **SHIFT** and ALT are an exception to this rule, and do not interact with other keys in this way.

syntax: *row*:= *numeric_expression* {range 0..7}

KEYROW (*row*)

example:

```
100 REMark run this program and press a few keys
110 REPeat loop
120   CURSOR 0,0
130   FOR row = 0 to 7
140     PRINT row !!! KEYROW(row) ; " "
150   END FOR row
160 END REPeat loop
```

KEYBOARD MATRIX

COLUMN / ROW	1	2	4	8	16	32	64	128
7	SHIFT	CTRL	ALT	X	V	/	N	,
6	8	2	6	Q	E	O	T	U
5	9	W	I	TAB	R	–	Y	
4	L	3	H	1	A	P	D	J
3	↑	CAPS LOCK	K	S	F	=	G	;
2	↓	Z	.	C	B	£	M	~
1	ENTER	←	up	ESC	→	\	SPACE	down
0	F4	F1	5	F2	F3	F5	4	7

LBYTES
devices
Microdrives

LBYTES will load a data file into memory at the specified start address.

syntax: *start_address:= numeric_expression*

LBYTES *device ,start_address*

example: i.
```
LBYTES mdv1_screen, 131072
```
{load a screen image}

ii.
```
LBYTES mdv1_program, start_address
```
{load a program at a specified address}

LEN
string arrays

LEN is a function which will return the length of the specified *string expression.*

syntax: LEN (*string_expression*)

example: i.
```
PRINT LEN( "LEN will find the length of this
string")
```

ii.
```
PRINT LEN(output_string$)
```

LET

LET starts a SuperBASIC assignment statement. The use of the LET keyword is optional. The assignment may be used for both string and numeric assignments. SuperBASIC will automatically convert unsuitable data types to a suitable form wherever possible.

syntax: [LET] *variable* = *expression*

example:

```
i.    LET a = 1 + 2
ii.   LET a$ = "12345"
iii.  LET a$ = 6789
iv.   b$ = test_data
```

LINE
LINE_R
graphics

LINE allows a straight line to be drawn between two points in the *window* attached to the default or specified *channel*. The ends of the line are specified using the *graphics coordinate system*.

Multiple lines can be drawn with a single LINE command.

The normal specification requires specifying the two end points for a line. These end points can be specified either in absolute coordinates (relative to the *graphics origin*) or in relative coordinates (relative to the *graphics cursor*). If the first point is omitted then a line is drawn from the graphics cursor to the specified point. If the second point is omitted then the graphics cursor is moved but no line is drawn.

LINE will always draw with absolute coordinates, i.e. relative to the *graphics origin*, while LINE_R will always draw relative to the graphics cursor.

syntax:

x:= *numeric_expression*
y:= *numeric_expression*
point:= *x* , *y*

parameter_2:= | TO *point* 1
| ,*point* TO *point* 2

parameter_1:= | TO *point, angle* 1
| TO *point* 2
| *point* 3

LINE [*channel,*] *parameter_1* *[, *parameter_2*]*

LINE_R [*channel,*] *parameter_1* *[,*parameter_2*]*

where 1 will draw from the specified point to the next specified point
2 will draw from the the last point plotted to the specified point
3 will move to the specified point - no line will be drawn

example:

```
i.    LINE 0,0 TO 0, 50 TO 50,0 TO 50,0 TO 0,0 {a square}
ii.   LINE TO 0.75, 0.5 {a line}
iii.  LINE 25,25 {move the graphics cursor}
```

LIST

LIST allows a SuperBASIC line or group of lines to be listed on a specific or default *channel*.

LIST is terminated by CTRL space

syntax:

line:=	*line_number* TO *line_number*	1
	line_number TO	2
	TO *line_number*	3
	line_number	4
		5

LIST [*channel*,] *line* *[,*line*]*

where 1 will list from the specified line to the specified line
2 will list from the specified line to the end
3 will list from the start to the specified line
4 will list the specified line
5 will list the whole program

example:

```
i.   LIST                {list all lines}
ii.  LIST 10 to 300      {list lines 10 to 300}
iii. LIST 12,20,50       {list lines 12,20 and 50 only}
```

comment

If LIST output is directed to a *channel* opened as a printer channel then LIST will provide hard copy.

LOAD

devices
Microdrives

LOAD will load a SuperBASIC program from any QL device. LOAD automatically performs a NEW before loading another *program*, and so any previously loaded program will be cleared by LOAD.

If a line input during a load has incorrect SuperBASIC syntax, the word MISTAKE is inserted between the line number and the body of the line. Upon execution, a line of this sort will generate an error

syntax: LOAD *device*

example:

```
i.   LOAD "mdv1_test_program"
ii.  LOAD mdv1_games
iii. LOAD neti_3
iv.  LOAD ser1_e
```

LN
LOG10

maths functions

LN will return the natural logarithm of the specified argument. LOG10 will return the common logarithm. There is no upper limit on the parameter other than the maximum number the computer can store.

syntax: LOG10 (*numeric_expression*) {range greater than zero}
LN (*numeric_expression*) {range greater than zero}

example:
i. `PRINT LOG10(20)`
ii. `PRINT LN(3.141592654)`

FOR SERVICE MANUALS
CONTACT:
MAURITRON TECHNICAL SERVICES
www.mauritron.co.uk
TEL: 01844 - 351694
FAX: 01844 - 352554

LOCal

functions and procedures

LOCal allows *identifiers* to be defined to be LOCal to a *function or procedure*. Local identifiers only exist within the function or procedure in which they are defined, or in procedures and functions called from the function or procedure in which they are defined. They are lost when the function or procedure terminates. Local identifiers are independent of similarly named identifiers outside the defining function or procedure. *Arrays* can be defined to be local by dimensioning them within the LOCal statement.

The LOCal statement must precede the first executable statement in the function or procedure in which it is used.

syntax: LOCal *identifier* *[, *identifier*]*

example:
i. `LOCal a, b, c(10,10)`
ii. `LOCal temp_data`

comment

Defining variables to be LOCal allows variable names to be used within functions and procedures without corrupting meaningful variables of the same name outside the function or procedure.

LRUN

devices
Microdrives

LRUN will load and run a SuperBASIC *program* from a specified *device*. LRUN will perform NEW before loading another program and so any previously stored SuperBASIC program will be cleared by LRUN.

If a line input during a loading has incorrect SuperBASIC syntax, the word MISTAKE is inserted between the line number and the body of the line. Upon execution, a line of this sort will generate an error.

syntax: LRUN *device*

example:
i. `LRUN mdv2_TEST`
ii. `LRUN mdv1_game`

MERGE

devices
Microdrives

MERGE will load a *file* from the specified *device* and interpret it as a SuperBASIC *program*. If the new file contains a *line number* which doesn't appear in the program already in the QL then the line will be added. If the new file contains a replacement line for one that already exists then the line will be replaced. All other old program lines are left undisturbed.

If a line input during a MERGE has incorrect SuperBASIC syntax, the word MISTAKE is inserted between the line number and the body of the line. Upon execution, a line of this sort will generate an error.

syntax: MERGE *device*

example:
i. `MERGE mdv1_overlay_program`
ii. `MERGE mdv1_new_data`

MOD

operators

MOD is an operator which gives the modulus, or remainder, when one integer is divided by another.

syntax: *numeric_expression* MOD *numeric_expression*

example:
i. `PRINT 5 MOD 2` {will print 1}
ii. `PRINT 5 MOD 3` {will print 2}

MODE

screen

MODE sets the resolution of the screen and the number of solid colours which it can display. MODE will clear all *windows* currently on the screen, but will preserve their position and shape. Changing to low resolution mode (8 colour) will set the minimum character size to 2,0.

syntax: MODE *numeric_expression*

where: 8 or 256 will select low resolution mode
4 or 512 will select high resolution mode

example:
i. `MODE 256`
ii. `MODE 4`

MOVE
turtle graphics

MOVE will move the graphics turtle in the *window* attached to the default or specified *channel* a specified distance in the current direction. The direction can be specified using the TURN and TURNTO commands. The graphics scale factor is used in determining how far the turtle actually moves. Specifying a negative distance will move the turtle backwards.

The turtle is moved in the *window* attached to the specified or default *channel*.

syntax: *distance*:= *numeric_expression*

MOVE [*channel*,] *distance*

example:

i.
```
MOVE #2,20
```
{move the turtle in channel 2 20 units forwards}

ii.
```
MOVE -50
```
{move the turtle in the default channel 50 units backwards}

MRUN
devices
Microdrives

MRUN will interpret a *file* as a SuperBASIC *program* and merge it with the currently loaded program.

If used as *direct command* MRUN will run the new program from the start. If used as a program *statement* MRUN will continue processing on the line following MRUN.

If a line input during a merge has incorrect SuperBASIC syntax, the word MISTAKE is inserted between the line number and the body of the line. Upon execution, a line of this sort will generate an error.

syntax: MRUN *device*

example:

i.
```
MRUN mdv1_chain_program
```

ii.
```
MRUN mdv1_new_data
```

NET

network NET allows the *network* station number to be set. If a station number is not explicitly set then the QL assumes station number 1.

syntax: *station*:= *numeric_expression* {range 1..127}

NET *station*

example: i. `NET 63`
ii. `NET 1`

comment Confusion may arise if more than one station on the network has the same station number.

NEW

NEW will clear out the old *program*, *variables* and *channels* other than 0,1 and 2.

syntax: NEW

example: `NEW`

NEXT

repetition

NEXT is used to terminate, or create a loop *epilogue* in, REPeat and FOR loops.

syntax: NEXT *identifier*

The identifier must match that of the loop which the NEXT is to control

example: i.

```
10 REMark this loop must repeat forever
11 REPeat infinite_loop
12    PRINT "still looping"
13 NEXT infinite_loop
```

ii.

```
10 REMark this loop will repeat 20 times
11 LET limit = 20
12 FOR index=1 TO limit
13    PRINT index
14 NEXT index
```

iii.

```
10 REMark this loop will tell you when a 30 is found
11 REPeat loop
12    LET number = RND(1 TO 100)
13    IF number <> 30 THEN NEXT loop
14    PRINT number; " is 30"
15 EXIT LOOP
16 END REPeat loop
```

in REPeat

If NEXT is used inside a REPeat - END REPeat construct it will force processing to continue at the statement following the matching REPeat statement.

in FOR

The NEXT statement can be used to repeat the FOR loop with the control variable set at its next value. If the FOR loop is exhausted then processing will continue at the statement following the NEXT; otherwise processing will continue at the statement after the FOR.

ON...GOTO
ON...GOSUB

To provide compatibility with other BASICs, SuperBASIC supports the ON GOTO and ON GOSUB statements. These statements allow a variable to select from a list of possible *line numbers* a line to process in a GOTO or GOSUB statement. If too few line numbers are specified in the list then an error is generated.

syntax: ON *variable* GOTO *expression* *[, *expression*]*
ON *variable* GOSUB *expression* *[, *expression*]*

example: i.

```
ON x GOTO 10, 20, 30, 40
```

ii.

```
ON select_variable GOSUB 1000, 2000, 3000, 4000
```

comment

SELect can be used to replace these two BASIC commands.

OPEN
OPEN_IN
OPEN_NEW

devices
Microdrives

OPEN allows the user to link a logical *channel* to a physical QL *device* for I/O purposes.

If the channel is to a Microdrive then the Microdrive file can be an existing file or a new file. In which case OPEN_IN will open an already existing Microdrive file for input and OPEN_NEW will create a new Microdrive file for output.

syntax: *channel*:= #*numeric_expression*

OPEN *channel*, *device*

example:

i. `OPEN #5, f_name$`

ii. `OPEN_IN #9, "mdv1_file_name"`
{open file mdv1_file_name}

iii. `OPEN_NEW #7, mdv1_data_file`
{open file mdv1_data_file}

iv. `OPEN #6, con_10x20a20x20_32`
{Open channel 6 to the console device creating a window size 10x20 pixels at position 20,20 with a 32 byte keyboard type ahead buffer.}

v. `OPEN #8, mdv1_read_write_file.`

OVER

windows

OVER selects the type of over printing required in the window attached to the specified or default channel. The selected type remains in effect until the next use of OVER.

syntax: *switch*:= *numeric_expression* {range −1..1}

OVER {*channel*,} *switch*

where *switch* = 0 – print *ink* on *strip*
switch = 1 – print in *ink* on transparent *strip*
switch = −1 – XORs the data on the screen

example:

i. `OVER 1` {set "overprinting"}

ii.
```
10 REMark Shadow Writing
11 PAPER 7 : INK 0 : OVER 1 : CLS
12 CSIZE 3,1
13 FOR i = 0 TO 10
14    CURSOR i,i
15    IF i=10 THEN INK 2
16    PRINT "Shadow"
17 END FOR i
```

PAN
windows

PAN the entire current window the specified number of pixels to the left or the right. PAPER is scrolled in to fill the clear area.

An optional second parameter can be specified which will allow only part of the *screen* to be panned.

syntax:

distance:= *numeric_expression*
part:= *numeric_expression*

PAN [*channel*,] *distance* [, *part*]

where *part* = 0 – whole screen (or no parameter)
part = 3 – whole of the cursor line
part = 4 – right end of cursor line including the cursor position

If the expression evaluates to a positive value then the contents of the screen will be shifted to the right.

example:

i. `PAN #2,50` {pan left 50 pixels}
ii. `PAN -100` {pan right 100 pixels}
iii. `PAN 50,3` {pan the whole of the current cursor line 50 pixels to the right}

warning

If *stipples* are being used or the screen is in low resolution mode then, to maintain the stipple pattern, the screen must be panned in multiples of two pixels.

PAPER
windows

PAPER sets a new paper colour (ie. the colour which will be used by CLS, PAN, SCROLL, etc). The selected paper colour remains in effect until the next use of PAPER. PAPER will also set the STRIP colour

PAPER will change the paper colour in the *window* attached to the specified or default *channel*

syntax: PAPER [*channel*,] *colour*

example:

i. `PAPER #3,7` {White paper on channel 3}
ii. `PAPER 7,2` {White and red stipple}
iii. `PAPER 255` {Black and white stipple}
iv.

```
10 REMark Show colours and stipples
11 FOR colour = 0 TO 7
12   FOR contrast = 0 TO 7
13     FOR stipple = 0 TO 3
14       PAPER colour, contrast, stipple
15       SCROLL 6
16     END FOR stipple
17   END FOR contrast
18 END FOR colour
```

{not suitable for televisions}

PAUSE

PAUSE will cause a *program* to wait a specified period of time. Delays are specified in units of 20ms in the UK only, otherwise 16.67ms. If no delay is specified then the program will pause indefinitely. Keyboard input will terminate the PAUSE and restart program execution.

syntax: *delay*:= *numeric_expression*

PAUSE [*delay*]

example:
i. `PAUSE 50` {wait 1 second}
ii. `PAUSE 500` {wait 10 seconds}

PEEK
PEEK_W
PEEK_L

BASIC

PEEK is a function which returns the contents of the specified memory location.

PEEK has three forms which will access a byte (8 bits), a word (16 bits), or a long word (32 bits).

syntax: *address*:= *numeric_expression*

PEEK(*address*) {byte access}
PEEK_W(*address*) {word access}
PEEK_L(*address*) {long word access}

example:
i. `PRINT PEEK(12245)` {byte contents of location 12245}
ii. `PRINT PEEK_W(12)` {word contents of locations 12 and 13}
iii. `PRINT PEEK_L(1000)` {long word contents of location 1000}

warning For word and long word access the specified address must be an even address.

PENUP
PENDOWN
turtle graphics

Operates the 'pen' in turtle graphics. If the pen is up then nothing will be drawn. If the pen is down then lines will be drawn as the turtle moves across the screen.

The line will be drawn in the *window* attached to the specified or default *channel*. The line will be drawn in the current ink colour for the channel to which the output is directed.

syntax: PENUP [*channel*]
PENDOWN [*channel*]

example:
i. `PENUP` {will raise the pen in the default channel}
ii. `PENDOWN #2` {will lower the pen in the window attached to channel 2}

PI
maths functions

PI is a function which returns the value of π.

syntax: PI

example:
```
PRINT PI
```

POINT
POINT_R

graphics

POINT plots a point at the specified position in the *window* attached to the specified or default *channel*. The point is plotted using the *graphics coordinates system* relative to the *graphics origin*. If POINT_R is used then all points are specified relative to the graphics cursor and are plotted relative to each other.

Multiple points can be plotted with a single call to POINT.

syntax:

x:= *numeric_expression*
y:= *numeric_expression*

parameters:= *x* , *y*

POINT [*channel*,] *parameters* *[,*parameters*]*

example:

i.
```
POINT 256,128
```
{plot a point at (256,128)}

ii.
```
POINT x, x*x
```
{plot a point at (x,x*x)}

iii.
```
10 REPeat example
20   INK RND(255)
30   POINT RND(100),RND(100)
40 END REPeat example
```

POKE
POKE_W
POKE_L

BASIC

POKE allows a memory location to be changed. For word and long word accesses the specified address must be an even address.

POKE has three forms which will access a byte (8 bits), a word (16 bits), a long word (32 bits).

syntax:

address:= *numeric_expression*
data:= *numeric_expression*

POKE *address*, *data* {byte access}
POKE_W *address*, *data* {word access}
POKE_L *address*, *data* {long word access}

example:

i.
```
POKE 12235,0
```
{set byte at 12235 to 0}

ii.
```
POKE_L 131072, 12345
```
{set long word at 131072 to 12345}

warning

Poking data into areas of memory used by Qdos can cause the system to crash and data to be lost. Poking into such areas is not recommended.

PRINT

devices
Microdrives

Allows output to be sent to the specified or default *channel*. The normal use of PRINT is to send data to the QL *screen*.

syntax: *separator*:= | !
| ,
| \
| ;
| TO numeric_expression

item:= | *expression*
| *channel*
| *separator*

PRINT *[*item*] *

Multiple print *separators* are allowed. At least one separator must separate *channel* specifications and *expressions*.

example:

i. `PRINT "Hello World"`
{will output Hello World on the default output device (channel 1)}

ii. `PRINT #5, "data", 1,2,3,4`
{will output the supplied data to channel 5 (which must have been previously opened)}

iii. `PRINT TO 20 ; "This is in column 20"`

separators

! Normal action is to insert a space between items output on the screen. If the item will not fit on the current line a line feed will be generated. If the current print position is at the start of a line then a space will not be output. ! affects the next item to be printed and therefore must be placed in front of the print item being printed. Also a ; or a ! must be placed at the end of a print list if the spacing is to be continued over a series of PRINT statements.

, Normal separator, SuperBASIC will tabulate output every 8 columns.

\ Will force a new line.

; Will leave the print position immediately after the last item to be printed. Output will be printed in one continuous stream.

TO Will perform a tabbing operation. TO followed by a *numeric_expression* will advance the print position to the column specified by the *numeric_expression*. If the requested column is meaningless or the current print position is beyond the specified position then no action will be taken.

RAD

maths functions

RAD is a function which will convert an angle specified in degrees to an angle specified in radians.

syntax: RAD (*numeric_expression*)

example: `PRINT RAD(180)` {will print 3.141593}

RANDOMISE

maths functions

RANDOMISE allows the random number generator to be reseeded. If a parameter is specified the parameter is taken to be the new seed. If no parameter is specified then the generator is reseeded from internal information.

syntax: RANDOMISE [*numeric_expression*]

example:

i. `RANDOMISE` {set seed to internal data}

ii. `RANDOMISE 3.2235` {set seed to 3.2235}

RECOL

windows

RECOL will recolour individual pixels in the window attached to the specified or default *channel* according to some preset pattern. Each parameter is assumed to specify, in order, the colour in which each pixel is recoloured, i.e. the first parameter specifies the colour with which to recolour all black pixels, the second parameter blue pixels, etc.

The colour specification must be a solid colour, i.e. it must be in the range 0 to 7.

syntax:

c0:= new colour for black
c1:= new colour for blue
c2:= new colour for red
c3:= new colour for magenta
c4:= new colour for green
c5:= new colour for cyan
c6:= new colour for yellow
c7:= new colour for white

RECOL [*channel* ,] *c0*, *c1*, *c2*, *c3*, *c4*, *c5*, *c6*, *c7*

example: `RECOL 2,3,4,5,6,7,1,0` {recolour blue to magenta, red to green, magenta to cyan etc.}

REMark

REMark allows explanatory text to be inserted into a program. The remainder of the line is ignored by SuperBASIC.

syntax: **REMark** *text*

example:
```
REMark This is a comment in a program
```

REMark is used to add comments to a program to aid clarity.

comment

RENUM

RENUM allows a group or a series of groups of SuperBASIC line numbers to be changed. If no parameters are specified then **RENUM** will renumber the entire program. The new listing will begin at line 100 and proceed in steps of 10.

If a start line is specified then line numbers prior to the start line will be unchanged. If an end line is specified then line numbers following the end line will be unchanged.

If a start number and stop are specified then the lines to be renumbered will be numbered from the start number and proceed in steps of the specified size.

If a **GOTO** or **GOSUB** statement contains an expression starting with a number then this number is treated as a line number and is renumbered.

syntax:

start_line:=	*numeric_expression*	{start renumber}
end_line:=	*numeric_expression*	{stop renumber}
start_number:=	*numeric_expression*	{base line number}
step:=	*numeric_expression*	{step}

RENUM [*start_line* [**TO** *end_line*];] [*start_number*] [,*step*]

example:

i.
```
RENUM                    {renumber whole program from 100 by 10}
```
ii.
```
RENUM 100 TO 200         {renumber from 100 to 200 by 10}
```

warning

No attempt must be made to use **RENUM** to renumber program lines out of sequence, ie to move lines about the program. **RENUM** should not be used in a program.

REPeat
END REPeat

repetition

REPeat allows general repeat loops to be constructed. REPeat should be used with EXIT for maximum effect. REPeat can be used in both long and short forms:

short

The REPEAT keyword and loop identifer are followed on the same logical line by a colon and a sequence of SuperBASIC statements. EXIT will resume normal processing at the next logical line.

syntax: REPeat *identifier* : *statements*

example:
```
REPeat wait : IF INKEY$ <> "" THEN EXIT wait
```

long

The REPEAT keyword and the loop identifier are the only statements on the logical line. Subsequent lines contain a series of SuperBASIC *statements* terminated by an END REPeat statement.

The statements between the REPeat and the END REPeat are repeatedly processed by SuperBASIC.

syntax: REPeat *identifier*
statements
END REPeat *identifier*

example:
```
10 LET number = RND(1 TO 50)
11 REPeat guess
12    INPUT "What is your guess?", guess
13    IF guess = number THEN
14       PRINT "You have guessed correctly"
15       EXIT guess
16    ELSE
17       PRINT "You have guessed incorrectly"
18    END IF
19 END REPeat guess
```

comment

Normally at least one statement in a REPeat loop will be an EXIT statement.

RESPR

Qdos

RESPR is a function which will reserve some of the resident procedure space. (For example to expand the SuperBASIC procedure list.)

syntax: *space* := *numeric_expression*
RESPR (*space*)

example:
```
PRINT RESPR(1024)
```
{will print the base address of a 1024 byte block}

FOR SERVICE MANUALS
CONTACT:
MAURITRON TECHNICAL SERVICES
www.mauritron.co.uk
TEL: 01844 - 351694
FAX: 01844 - 352554

RETurn

functions and procedures

RETurn is used to force a *function or procedure* to terminate and resume processing at the statement after the procedure or function call. When used within a function definition theRETurn statement is used to return the function's value.

syntax: RETurn [*expression*]

example: i.

```
100 PRINT ack (3,3)
110 DEFine FuNction ack(m,n)
120   IF m=0 THEN RETurn n+1
130   IF n=0 THEN RETurn ack (m-1,1)
140   RETurn ack(m-1,ack(m,n-1))
150 END DEFine
```

ii.

```
10 LET warning_flag = 1
11 LET error_number = RND(0 TO 10)
12 warning error_number
13 DEFine PROCedure warning(n)
14   IF warning_flag THEN
15     PRINT "WARNING:";
16     SELect ON n
17       ON n = 1
18         PRINT "Microdrive full"
19       ON n = 2
20         PRINT "Data space full"
21       ON n = REMAINDER
22         PRINT "Program error"
23     END SELect
24   ELSE
25     RETurn
26   END IF
27 END DEFine
```

comment

It is not compulsory to have a RETurn in a procedure. If processing reaches the END DEFine of a procedure then the procedure will return automatically.

RETurn by itself is used to return from a GOSUB.

RND

maths functions

RND generates a random number. Up to two parameters may be specified for RND. If no parameters are specified then RND returns a pseudo random *floating point* number in the exclusive range 0 to 1. If a single parameter is specified then RND returns an integer in the inclusive range 0 to the specified parameter. If two parameters are specified then RND returns an integer in the inclusive range specified by the two parameters.

syntax: RND ([*numeric_expression*] [TO *numeric_expression*])

example:

```
i.   PRINT RND                  {floating point number between 0 and 1}
ii.  PRINT RND(10 TO 20)        {integer between 10 and 20}
iii. PRINT RND(1 TO 6)          {integer between 1 and 6}
iv.  PRINT RND(10)              {integer between 0 and 10}
```

FOR SERVICE MANUALS
CONTACT:
MAURITRON TECHNICAL SERVICES
www.mauritron.co.uk
TEL: 01844 - 351694
FAX: 01844 - 352554

RUN

program

RUN allows a SuperBASIC program to be started. If a line number is specified in the **RUN** command then the program will be started at that point, otherwise the program will start at the lowest line number.

syntax: **RUN** [*numeric_expression*]

example:
```
i.   RUN          {run from start}
ii.  RUN 10       {run from line 10}
iii. RUN 2*20     {run from line 40}
```

comment

Although **RUN** can be used within a program its normal use is to start program execution by typing it in as a *direct command*.

SAVE

devices

Microdrives

SAVE will save a SuperBASIC program onto any QL device.

syntax:

line:=

numeric_expression **TO** *numeric_expression*	1
numeric_expression **TO**	2
TO *numeric_expression*	3
numeric_expression	4
	5

SAVE *device* *[,*line*]*

where 1 will save from the specified line to the specified line
2 will save from the specified line to the end
3 will save from the start to the specified line
4 will save the specified line
5 will save the whole program

example:
```
i.   SAVE mdv1_program, 20 TO 70
                {save lines 20 to 70 on mdv1_program}
ii.  SAVE mdv2_test_program, 10,20,40
                {save lines 10,20,40 on mdv1_test_program}
iii. SAVE net3
                {save the entire program on the network}
iv.  SAVE ser1
                {save the entire program on serial channel 1}
```

SBYTES
devices
Microdrives

SBYTES allows areas of the QL memory to be saved on a QL *device*.

syntax:

start_address:= *numeric_expression*
length:= *numeric_expression*

SBYTES *device*, *start_address*, *length*

example:

i. `SBYTES mdv1_screen_data, 131072,32768`
{save memory 50000 length 10000 bytes on mdv1_test_program}

ii. `SBYTES mdv1_test_program, 50000,10000`
{save memory 50000 length 1000 bytes on mdv1_test_program}

iii. `SBYTES neto_3, 32768,32678`
{save memory 32768 length 32768 bytes on the network}

iv. `SBYTES ser1, 0,32768`
{save memory 0 length 32768 bytes on serial channel 1}

SCALE
graphics

SCALE allows the scale factor used by the *graphics* procedures to be altered. A scale of 'x' implies that a vertical line of length 'x' will fill the vertical axis of the *window* in which the figure is drawn. A scale of 100 is the default. SCALE also allows the origin of the coordinate system to be specified. This effectively allows the window being used for the graphics to be moved around a much larger graphics space.

syntax:

x:= *numeric_expression*
y:= *numeric_expression*

origin:= *x*,*y*
scale:= *numeric_expression*

SCALE [*channel*,] *scale*, *origin*

example:

i. `SCALE 0.5,0.1,0.1` {set scale to 0.5 with the origin at 0.1,0.1}
ii. `SCALE 10,0,0` {set scale to 10 with the origin at 0,0}
iii. `CALE 100,50,50` {set scale to 100 with the origin at 50,50}

SCROLL

windows

SCROLL scrolls the *window* attached to the specified or default *channel* up or down by the given number of pixels. *Paper* is scrolled in at the top or the bottom to fill the clear space.

An optional third parameter can be specified to obtain a part screen scroll.

syntax:

part:= *numeric_expression*
distance:= *numeric_expression*

where *part* = 0 - whole screen (default is no parameter)
part = 1 - top excluding the cursor line
part = 2 - bottom excluding the cursor line

SCROLL [*channel*,] *distance* [, *part*]

If the distance is positive then the contents of the screen will be shifted down.

example:

```
i.   SCROLL 10          {scroll down 10 pixels}
ii.  SCROLL -70         {scroll up 70 pixels}
iii. SCROLL -10, 2      {scroll the lower part of the window up 10
                         pixels}
```

SDATE

clock

The SDATE command allows the QL's clock to be reset.

syntax:

year:= *numeric_expression*
month:= *numeric_expression*
day:= *numeric_expression*
hours:= *numeric_expression*
minutes:= *numeric_expression*
seconds:= *numeric_expression*

SDATE *year*, *month*, *day*, *hours*, *minutes*, *seconds*

example:

```
i.   SDATE 1984,4,2,0,0,0
ii.  SDATE 1984,1,12,9,30,0
iii. SDATE 1984,3,21,0,0,0
```

SELect
END SELect

conditions

SELect allows various courses of action to be taken depending on the value of a variable.

define:

select_variable := *numeric_variable*

select_item := | *expression*
| *expression* TO *expression*

select_list := | *select_item* *[, *select_item*]*

long

Allows multiple actions to be selected depending on the value of a *select_variable*. The select variable is the last item on the logical line. A series of SuperBASIC *statements* follows, which is terminated by the next **ON** statement or by the **END SELect** statement. If the select item is an expression then a check is made within approximately 1 part in 10^{-7}, otherwise for expression **TO** expression the range is tested exactly and is inclusive. The **ON REMAINDER** statement allows a, "catch-all" which will respond if no other select conditions are satisfied.

syntax:

SELect ON *select_variable*
*{[**ON** *select_variable*] = *select_list*
statements]*
[**ON** *select_variable*] = **REMAINDER**
statements
END SELect

example:

```
100 LET error_number = RND(1 TO 10)
110 SELect ON error_number
120   ON error_number = 1
130     PRINT "Divide by zero"
140     LET error_number = 0
150   ON error_number = 2
160     PRINT "File not found"
170     LET error_number = 0
180   ON error_number = 3 TO 5
190     PRINT "Microdrive file not found"
200     LET error_number = 0
210   ON error_number = REMAINDER
220     PRINT "Unknown error"
230 END SELect
```

If the select variable is used in the body of the **SELect** statement then it must match the select variable given in the select header.

short

The short form of the **SELect** statement allows simple single line selections to be made. A sequence of SuperBASIC statements follows on the same logical line as the **SELect** statement. If the condition defined in the select statement is satisfied then the sequence of SuperBASIC statements is processed.

syntax:

SELect ON *select_variable* = *select_list* : *statement* *[: *statement*]*

example:

i.
```
SELect ON test_data = 1 TO 10 :
PRINT "Answer within range"
```

ii.
```
SELect ON answer = 0.00001 TO 0.00005 :
PRINT "Accuracy OK"
```

iii.
```
SELect ON a = 1 TO 10 : PRINT a ! "in range"
```

comment

The short form of the **SELect** statement allows ranges to be tested more easily than with an **IF** statement. Compare example ii. above with the corresponding **IF** statement.

SEXEC

Qdos

Will save an area of memory in a form which is suitable for loading and executing with the EXEC command.

The data saved should constitute a machine code program.

syntax:

start_address:= *numeric_expression* {start of area}
length:= *numeric_expression* {length of area}
data_space:= *numeric_expression* {length of data area which will be required by the program}

SEXEC *device*, *start_address*, *length*, *data_space*

example:

```
SEXEC mdv1_program, 262144,3000,500
```

comment

The Qdos system documentation should be read before attempting to use this command.

SIN

maths functions

SIN will compute the sine of the specified parameter.

syntax:

angle:= *numeric_expression* {range −10000..10000 in radians}

SIN(*angle*)

example:

i.
```
PRINT SIN(3)
```
ii.
```
PRINT SIN(3.141592654/2)
```

SQRT
maths functions

will compute the square root of the specified argument. The argument must be greater than or equal to zero.

syntax: SQRT (*numeric_expression*) {range >= 0}

example:

i. `PRINT SQRT(3)` {print square root of 3}

ii. `LET C = SQRT(a^2 + b^2)` {let c become equal to the square root of a^2 + b^2}

STOP
BASIC

STOP will terminate execution of a program and will return SuperBASIC to the *command interpreter*.

syntax: STOP

example:

i. `STOP`

ii. `IF n = 100 THEN STOP`

You may CONTINUE after STOP.

The last executable line of a program will act as an automatic stop.

comment

STRIP

windows

STRIP will set the current strip colour in the *window* attached to the specified or default *channel*. The strip colour is the background colour which is used when OVER 1 is selected. Setting PAPER will automatically set the strip colour to the new PAPER colour.

syntax: STRIP [*channel*,] *colour*

example:

i. `STRIP 7` {set a white strip}

ii. `STRIP 0,4,2` {set a black and green stipple strip}

comment

The effect of STRIP is rather like using a highlighting pen.

TAN

maths functions

TAN will compute the tangent of the specified argument. The argument must be in the range −30000 to 30000 and must be specified in radians.

syntax: TAN (*numeric_expression*) {range −30000..30000}

example:

i. `TAN(3)` {print tan 3}

ii. `TAN(3.141592654/2)` {print tan π/2}

TURN
TURNTO
turtle graphics

TURN allows the heading of the 'turtle' to be turned through a specified angle while **TURNTO** allows the turtle to be turned to a specific heading.

The turtle is turned in the *window* attached to the specified or default *channel*.

The angle is specified in degrees. A positive number of degrees will turn the turtle anti-clockwise and a negative number will turn it clockwise.

Initially the turtle is point at 0°, that is to the right hand side of the window.

syntax: *angle*:= *numeric_expression* {angle in degrees}

TURN [*channel*,] *angle*
TURNTO [*channel*,] *angle*

example:
i. `TURN 90` {turn through 90°}
ii. `TURNTO 0` {turn to heading 0°}

FOR SERVICE MANUALS
CONTACT:
MAURITRON TECHNICAL SERVICES
www.mauritron.co.uk
TEL: 01844 - 351694
FAX: 01844 - 352554

UNDER
windows

Turns underline either on or off for subsequent output lines. Underlining is in the current **INK** colour in the *window* attached to the specified or default *channel*.

syntax: *switch*:= *numeric_expression* {range 0..1}

UNDER [*channel*,] *switch*

example:
i. `UNDER 1` {underlining on}
ii. `UNDER 0` {underlining off}

WIDTH

devices

WIDTH allows the default width for non-console based devices to be specified, for example printers.

syntax: *line_width*:= *numeric_expression*

WIDTH [*channel*,] *line_width*

example:

i. `WIDTH 80` {set the device width to 80}

ii. `WIDTH #6, 72` {set the width of the device attached to channel 6 to 72}

WINDOW

windows

Allows the user to change the position and size of the *window* attached to the specified or default channel. Any borders are removed when the window is redefined.

Coordinates are specified using the *pixel system* relative to the screen origin.

syntax: *width*:= *numeric_expression*
depth:= *numeric_expression*
x:= *numeric_expression*
y:= *numeric_expression*

WINDOW [*channel*,] *width*, *depth*, *x*, *y*

example: `WINDOW 30, 40, 10, 10` {window 30x40 pixels at 10,10}

A

ABS 1
Absolute values 1
ACOT 1
ADATE 2
ARC
  cotangent 2
  tangent 2
ARC_R 2
Arctangent 2
Arguments 14, 15
Arrays
  DIM 17
  DIMN 17
Assignment 32
AT 3
ATAN 1
AUTO 3

B

BAUD 4
Baudrates 4
BEEP 4
BEEPING 5
BLOCK 5
BORDER 6

C

CALL 6
Channel
  CLOSE 8
Character
  CODE 9
  repetition 28
  size 12
CHR$ 7
CIRCLE 7
CIRCLE_R 7
CLEAR 8
  BASIC 8
  screen 9
  window 9
Clock
  ADATE 2
  DATE 14
  DATE$ 14
  DAY$ 14
  SDATE 52
CLOSE 8
Closing
  channels 8
CLS 9
CODE 9
Colour
  INK 27
  MODE 36
  PAPER 41
  RECOL 46
  recolour 46
Comments 47
Communications
  baud rates 4
  networks 38
Conditions
  IF 26
  SELect 53
CONTINUE 10
COPY 10
COPY_N 10
COS 11
cosine 11
COT 11
cotangent 11
CSIZE 12
CURSOR 12

D

DATA 13
  structures 18
DATE 14
DATE$ 14
DEFine 15, 16
DAY$ 14
DEFine
  FuNction 15
  PROCedure 16
DEG 17
Degrees 17
Delay 42
DELETE 17
  files 17
  lines 20
Devices
  CLOSE 8
  directory 19
  LBYTES 31
  LOAD 33
  load and run 35
  LRUN 35
  MERGE 35
  merge and run 37
  MRUN 37
  NET 38
  network station 38
  OPEN 40
  OPEN_IN 40
  open for input 40
  OPEN_NEW 40
  open new 40
  RUN 50
  SAVE 50
  SBYTES 51
DIM 18
Dimension arrays 18
DIMN 18
DIR 20
Directory 19
Display directory 19
DIV 19
DLINE 20
Documentation 47
Dots 44

E

EDIT 20
ELLIPSE 7
ELLIPSE_R 7

END
DEFINE ... 15, 16
FOR ... 24
IF ... 26
REPeat ... 48
SELect ... 53
EOF ... 21
Equals ... 32
Errors
CONTINUE ... 10
RETRY ... 10
EXEC ... 21
EXEC_W ... 21
EXIT ... 22
with FOR ... 24
with REPeat ... 48
EXP ... 22
Exponentiation ... 22

F

Files
COPY ... 10
COPY_N ... 10
DELETE ... 17
DIR ... 19
directory ... 19
LBYTES ... 31
LOAD ... 33
load and run ... 33
LRUN ... 35
MERGE ... 35
merge and run ... 37
MRUN ... 37
OPEN ... 40
open for input ... 40
OPEN_IN ... 40
open new ... 40
OPEN_NEW ... 40
PRINT ... 45
RUN ... 50
SAVE ... 50
FILL ... 23
FILL$ ... 23
FLASH ... 24
FN ... 15
FOR ... 24
with EXIT ... 24
with NEXT ... 24
FuNction ... 15
DEFine ... 15
RETurn ... 49

G

GOSUB ... 25
GOTO ... 26
Graphics
ARC ... 2
ARC_R ... 2
CIRCLE ... 7
CIRCLE_R ... 7
ELLIPSE ... 7
ELLIPSE_R ... 7
FILL ... 23
fill shape ... 23
LINE ... 32
LINE_R ... 32
POINT ... 44
POINT_R ... 44
SCALE ... 51
Turtle
FILL ... 23
MOVE ... 37
TURN ... 57
TURNTO ... 57
SCALE ... 51
PENDOWN ... 43
PENUP ... 43

H

Highlighting ... 56

I

I/O
INKEY$ ... 28
keyboard input ... 30
KEYROW ... 30
printing ... 45
IF ... 26
nesting ... 27
INK ... 27
INKEY$ ... 28
INPUT ... 28
INSTR ... 29
INT ... 29
Integer divide ... 18

J

Jump ... 26

K

Keyboard input ... 28, 30
KEYROW ... 30

L

LBYTES ... 31
LEN ... 31
Length of strings ... 31
LET ... 32
LINE ... 32
delete ... 20
DLINE ... 20
editor ... 20
numbering ... 3
renumbering ... 47
RENUM ... 47
LINE_R ... 32
LIST ... 33
LN ... 34
LOAD ... 33
load and run ... 35
LRUN ... 35
Local variables ... 34
in functions ... 15
in procedures ... 16

Logarithm ... 34
LOG10 ... 34
Loop epilogue ... 24
Loop repetition
  EXIT ... 22
  FOR ... 24
  NEXT ... 39
  REPeat ... 48

M

Machine code ... 6
  CALL ... 6
  SEXEC ... 54
  saving ... 54
  EXEC ... 21
  EXEC_W ... 21
  loading ... 21
  respr ... 48
Maths functions
  ABS ... 1
  absolute value ... 1
  ACOT ... 1
  arc cotangent ... 1
  ATAN ... 1
  arc tangent ... 1
  common logarithm ... 34
  COS ... 11
  cosine ... 11
  COT ... 11
  EXP ... 22
  exponentiation ... 22
  INT ... 29
  integer part ... 29
  LOG ... 34
  LN ... 34
  natural logarithm ... 34
  RAD ... 45
  radians conversion ... 45
  SIN ... 54
  sine ... 54
  SQR ... 55
  square root ... 55
  TAN ... 56
  tangent ... 56
Merge and run ... 37
Microdrives
  COPY ... 10
  copying ... 10
  DELETE ... 17
  deleting files ... 17
  FORMAT ... 25
  formatting cartridges ... 25
  LOAD ... 33
  loading SuperBASIC programs ... 33
  SAVE ... 50
  saving SuperBASIC programs ... 50
MOD ... 36
  MODE ... 36
modulus ... 36
MOVE ... 37
MRUN ... 37
Multitasking
  PAUSE ... 42
  SEXEC ... 54

N

NET ... 38
Networks ... 38
NEW ... 38
NEXT ... 39
  with FOR ... 24
  with REPeat ... 48
Restarting SuperBASIC ... 38

O

ON GOSUB ... 39
ON GOTO ... 39
OPEN ... 40
  channel ... 40
  serial port ... 40
  window ... 40
OPEN_IN ... 40
open existing file ... 40
OPEN_NEW ... 40
open new file ... 40
Operators
  INSTR ... 29
  MOD ... 36
OVER ... 40
overprinting ... 40

P

PAN ... 40
PAPER ... 40
Parameters ... 15, 16
PAUSE ... 42
PEEK ... 42
PEEK_L ... 42
PEEK_W ... 42
PENDOWN ... 43
PENUP ... 43
PI ... 43
Plotting points ... 44
POINT ... 44
POINT_R ... 44
POKE ... 44
POKE_L ... 44
POKE_W ... 44
PRINT ... 45
  OVER ... 40
  UNDER ... 57
Printout ... 45
Procedures
  DEFine ... 15, 16
  LOCal ... 34
  RETurn ... 49
Programs
  CONTINUE ... 10
  RETRY ... 10
  RUN ... 50
  SAVE ... 50

R

RAD ... 45
Random numbers ... 49
RANDOMISE ... 46
READ ... 13
RECOL ... 47

REM 47
REMark 47
RENUM 47
Renumber lines 47
REPeat 48
  EXIT 22
  NEXT 39
Repetition
  FOR 24
  NEXT 39
Reset clock 2, 52
Resolution 36
RESPR 48
RESTORE 13
RETRY 10
RETurn 49
  with FuNction 15
  with PROCedures 16
RND 49
Routines 16
RS-232-C 4
RUN 50
  LRUN 35
  load and run 35

S

SAVE 50
  machine code 51
  programs 50
SBYTES 51
SCALE 51
Screen
  BLOCK 5
  BORDER 6
  character size 12
  clear 9
  FLASH 24
  INK 27
  MODE 36
  output 45
  OVER 40
  overprinting 40
  PAN 41
  PAPER 41
  PRINT 45
  RECOL 46
  recolouring 46
  SCROLL 52
  STRIP 56
  UNDER 57
  underlining 57
  WINDOW 58
SCROLL 52
SDATE 52
SELect 53
Setting clock 52
Setting station number 38
Shapes
  ARC 2
  CIRCLE 7
  ELLIPSE 7
  FILL 23
  LINE 34
SIN 54
sine 54
Size of characters 12
Sound
  BEEP 4
  BEEPING 5
SQRT 55
Square root 55
Starting programs 35, 50
Station number 38
STOP 55
Strings
  CHR$ 7
  FILL$ 23
  INSTR 29
  LEN 31
  length 31
STRIP 56
subroutines 15, 16

T

TAN 56
Tangent 56
THEN 26
Time
  clock adjust 2
  clock reset 52
  date 14
  PAUSE 42
TURN 57
TURNTO 57
Turtle graphics
  FILL 23
  MOVE 37
  PENUP 43
  PENDOWN 43
  TURN 57
  TURNTO 57

U

Unconditional jump 26
UNDER 57
Underlining 57

V

Value absolutes 1

W

Windows
  AT 3
  BLOCK 5
  BORDER 6
  CSIZE 12
  Character size 12
  clear 9
  cursor control 3, 12
  FILL 23
  FLASH 24
  INK 27
  MODE 36
  OVER 40
  overprinting 40
  PAN 41
  PAPER 41

print position 3
SCROLL 52
STRIP 56
UNDER 57
underlining 57
WINDOW 58

sinclair

QL

Concepts

The Concept Reference Guide describes concepts relating to SuperBASIC and the QL hardware. It is best to think of the Concept Guide as a source of information. If there are any questions about SuperBASIC or the QL itself which arise out of using the computer or other sections of the manual then the Concept Guide may have the answer. Concepts are listed in alphabetical order using the most likely term for that concept. If the subject cannot be found then consult the index which should be able to tell you which page to turn to.

Where an example is listed with line numbers, then it is a complete program and can be entered and run. Examples listed without numbers are usually simple commands and it may not always be sensible to enter them into the computer in isolation. Examples which demonstrate stipples will not work properly on a television set.

©1984 SINCLAIR RESEARCH LIMITED
by Stephen Berry *(Sinclair Research Limited)*

arrays

Arrays must be DIMensioned before they are used. When an array is dimensioned the value of each of its elements is set to zero or a zero length string if it is a string array. An array dimension runs from zero up to the specified value. There is no limit on the number of dimensions which can be defined other than the total memory capacity of the computer. An array of data is stored such that the last index defined cycles round most rapidly:

example

the array defined by

```
DIM array(2,4)
```

will be stored as

0,0 low address
0,1
0,2
0,3
0,4
1,0
1,1
1,2
1,3
1,4
2,0
2,1
2,2
2,3
2,4 high address

The element referred to by **array(a,b,c)** is equivalent to the element referred to by **array(a)(b)(c)**.

Command	Function
DIM	dimension an array
DIMN	find out about the dimensions of an array

BASIC

SuperBASIC includes most of the functions, procedures and constructs found in other dialects of BASIC. Many of these functions are superfluous in SuperBASIC but are included for compatibility reasons:

GOTO	use IF, REPeat, etc
GOSUB	use DEFine PROCedure
ON GOTO	use SELect
ON GOSUB	use SELect

Some commands appear not to be present. They can always be obtained by using a more general function. For example, there are no LPRINT or LLIST statements in SuperBASIC but output can be directed to a printer by opening the relevant *channel* and using PRINT or LIST.

LPRINT	use PRINT#
LLIST	use LIST#
VAL	not required in SuperBASIC
STR$	not required in SuperBASIC
IN	not applicable to 68008
OUT	not applicable to 68008

comment Almost all forms of BASIC require the VAL(x$) and STR$(x) functions in order to be able to convert the internal codified form of the value of a string expression to or from the internal codified form of the value of a numeric expression.

These functions are redundant in SuperBASIC because of the provision of a unique facility referred to as "coercion". The VAL and STR$ functions are therefore not provided.

break

If at any time the computer fails to respond or you wish to stop a SuperBASIC program or command then

hold down

CTRL

and then press

SPACE

A program broken into in this way can be restarted by using the CONTINUE command.

channels

A *channel* is a means by which data can be output to or input from a QL *device*. Before a channel can be used it must first be activated (or opened) with the **OPEN** command. Certain channels should always be kept open: these are the default channels and allow simple communication with the QL via the keyboard and screen. When a channel is no longer in use it can be deactivated (closed) with the **CLOSE** command.

A channel is identified by a channel number. A channel number is a numeric expression preceded by a **#**. When the channel is opened a *device* is linked to a channel number and the channel is initialised. Thereafter the channel is identified only by its channel number. For example:

```
OPEN #5,SER1
```

Will link serial port 1 to the channel number 5. When a channel is closed only the channel number need be specified. For example:

```
CLOSE #5
```

Opening a channel requires that the *device driver* for that channel be activated. Usually there is more than one way in which the device driver can be activacted, for example the network requires a *station number*. This extra information is appended to the device name and passed to the **OPEN** command as a parameter, see concept *device* and *peripheral expansion*.

Data can be output to a channel by **PRINT**ing to that channel; this is the same mechanism by which output appears on the QL screen. **PRINT** without a parameter outputs to the default channel #1. For example

```
10 OPEN #5, mdv1_test_file
20 PRINT #5, "this text is in file test_file"
30 CLOSE #5
```

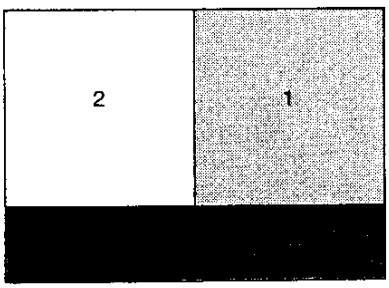
will output the text 'this text is in file test_file' to the file test_file. It is important to close the file after all the accesses have been completed to ensure that all the data is written.

Data can be input from a file in an analogous way using **INPUT**. Data can be input from a channel a character at a time using **INKEY$**.

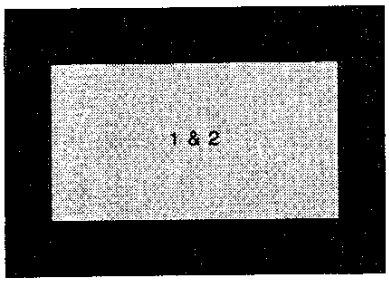
A channel can be opened as a console channel; output is directed to a specified *window* on the QL screen and input is taken from the QL keyboard. When a console channel is opened the size and shape of the initial window is specified. If more than one console channel is active then it is possible for more than one channel to be requesting input at the same time. In this case, the required channel can be selected by pressing CTRL C to cycle round the waiting channels. The cursor in the window of the selected channel will flash.

The QL has three default channels which are opened automatically. Each of these channels is linked to a window on the QL screen:

channel 0 – command and error channel
channel 1 – output and graphics channel
channel 2 – program listing channel



Monitor



Television

Command	Function
OPEN	open a channel for I/O
CLOSE	close a previously opened channel
PRINT	output to a channel
INPUT	input from a channel
INKEY$	input a character from a channel

The cursor controls are not built in to the operating system: however, if these functions are to be provided by applications software, they should use the keys specified; also the specified keys should not normally be used for any other purpose.

Decimal	Hex	Keying	Display/Function
0	00	CTRL £	NULL
1	01	CTRL A	
2	02	CTRL B	
3	03	CTRL C	change input channel (see note)
4	04	CTRL D	
5	05	CTRL E	
6	06	CTRL F	
7	07	CTRL G	
8	08	CTRL H	
9	09	TAB (CTRL I)	Next field
10	0A	ENTER (CTRL J)	New line/Command entry
11	0B	CTRL K	
12	0C	CTRL L	
13	0D	CTRL M	Enter
14	0E	CTRL N	
15	0F	CTRL O	
16	10	CTRL P	
17	11	CTRL Q	
18	12	CTRL R	
19	13	CTRL S	
20	14	CTRL T	
21	15	CTRL U	
22	16	CTRL V	
23	17	CTRL W	
24	18	CTRL X	
25	19	CTRL Y	
26	1A	CTRL Z	
27	1B	ESC (CTRL SHIFT I)	Abort current level of command
28	1C	CTRL SHIFT \	
29	1D	CTRL SHIFT]	
30	1E	CTRL SHIFT £	
31	1F	CTRL SHIFT ESC	
32	20	Space	Space
33	21	SHIFT 1	!
34	22	SHIFT '	"
35	23	SHIFT 3	#
36	24	SHIFT 4	$
37	25	SHIFT 5	%
38	26	SHIFT 7	&
39	27	'	'
40	28	SHIFT 9	(
41	29	SHIFT 0	)
42	2A	SHIFT 8	*
43	2B	SHIFT =	+
44	2C	,	,
45	2D	-	-
46	2E	.	.
47	2F	/	/

Decimal	Hex	Keying	Display/Function
48	30	0	0
49	31	1	1
50	32	2	2
51	33	3	3
52	34	4	4
53	35	5	5
54	36	6	6
55	37	7	7
56	38	8	8
57	39	9	9
58	3A	SHIFT ;	:
59	3B	;	;
60	3C	SHIFT ,	<
61	3D	=	=
62	3E	SHIFT .	>
63	3F	SHIFT /	?
64	40	SHIFT 2	@
65	41	SHIFT A	A
66	42	SHIFT B	B
67	43	SHIFT C	C
68	44	SHIFT D	D
69	45	SHIFT E	E
70	46	SHIFT F	F
71	47	SHIFT G	G
72	48	SHIFT H	H
73	49	SHIFT I	I
74	4A	SHIFT J	J
75	4B	SHIFT K	K
76	4C	SHIFT L	L
77	4D	SHIFT M	M
78	4E	SHIFT N	N
79	4F	SHIFT O	O
80	50	SHIFT P	P
81	51	SHIFT Q	Q
82	52	SHIFT R	R
83	53	SHIFT S	S
84	54	SHIFT T	T
85	55	SHIFT U	U
86	56	SHIFT V	V
87	57	SHIFT W	W
88	58	SHIFT X	X
89	59	SHIFT Y	Y
90	5A	SHIFT Z	Z
91	5B	[	[
92	5C	\	\
93	5D	]	]
94	5E	SHIFT 6	^
95	5F	SHIFT -	—
96	60	£	£
97	61	A	a
98	62	B	b
99	63	C	c
100	64	D	d
101	65	E	e
102	66	F	f
103	67	G	g
104	68	H	h
105	69	I	i
106	6A	J	j
107	6B	K	k
108	6C	L	l
109	6D	M	m
110	6E	N	n
111	6F	O	o

Decimal	Hex	Keying	Display/Function
112	70	P	p
113	71	Q	q
114	72	R	r
115	73	S	s
116	74	T	t
117	75	U	u
118	76	V	v
119	77	W	w
120	78	X	x
121	79	Y	y
122	7A	Z	z
123	7B	SHIFT |	¦
124	7C	SHIFT \	|
125	7D	SHIFT ¦	¦
126	7E	SHIFT £	~
127	7F	SHIFT ESC	©
128	80	CTRL ESC	ä
129	81	CTRL SHIFT 1	ã
130	82	CTRL SHIFT '	å
131	83	CTRL SHIFT 3	ë
132	84	CTRL SHIFT 4	õ
133	85	CTRL SHIFT 5	õ
134	86	CTRL SHIFT 7	ç
135	87	CTRL `	ü
136	88	CTRL SHIFT 9	å
137	89	CTRL SHIFT 0	ñ
138	8A	CTRL SHIFT 8	œ
139	8B	CTRL SHIFT =	¿
140	8C	CTRL ,	á
141	8D	CTRL -	à
142	8E	CTRL .	â
143	8F	CTRL /	ë
144	90	CTRL 0	è
145	91	CTRL 1	ê
146	92	CTRL 2	ï
147	93	CTRL 3	í
148	94	CTRL 4	ì
149	95	CTRL 5	î
150	96	CTRL 6	ö
151	97	CTRL 7	ò
152	98	CTRL 8	ô
153	99	CTRL 9	ú
154	9A	CTRL SHIFT ;	ù
155	9B	CTRL ;	û
156	9C	CTRL SHIFT ,	ß
157	9D	CTRL =	¢
158	9E	CTRL SHIFT .	¥
159	9F	CTRL SHIFT /	·
160	A0	CTRL SHIFT 2	Ä
161	A1	CTRL SHIFT A	Ã
162	A2	CTRL SHIFT B	Å
163	A3	CTRL SHIFT C	Ë
164	A4	CTRL SHIFT D	Õ
165	A5	CTRL SHIFT E	Õ
166	A6	CTRL SHIFT F	Ç
167	A7	CTRL SHIFT G	Ü
168	A8	CTRL SHIFT H	Å
169	A9	CTRL SHIFT I	Ñ
170	AA	CTRL SHIFT J	Œ
171	AB	CTRL SHIFT K	¡
172	AC	CTRL SHIFT L	alpha
173	AD	CTRL SHIFT M	delta
174	AE	CTRL SHIFT N	theta
175	AF	CTRL SHIFT O	lambda

Decimal	Hex	Keying	Display/Function
176	B0	CTRL SHIFT P	mu
177	B1	CTRL SHIFT Q	pi
178	B2	CTRL SHIFT R	phi
179	B3	CTRL SHIFT S	¡
180	B4	CTRL SHIFT T	¿
181	B5	CTRL SHIFT U	
182	B6	CTRL SHIFT V	§
183	B7	CTRL SHIFT W	¤
184	B8	CTRL SHIFT X	<<
185	B9	CTRL SHIFT Y	>>
186	BA	CTRL SHIFT Z	°
187	BB	CTRL [	–
188	BC	CTRL \	←
189	BD	CTRL]	→
190	BE	CTRL SHIFT 6	↑
191	BF	CTRL SHIFT -	↓
192	C0	Left	Cursor left one character
193	C1	ALT Left	Cursor to start of line
194	C2	CTRL Left	Delete left one character
195	C3	CTRL ALT Left	Delete line
196	C4	SHIFT Left	Cursor left one word
197	C5	SHIFT ALT Left	Pan left
198	C6	SHIFT CTRL Left	Delete left one word
199	C7	SHIFT CTRL ALT Left	
200	C8	Right	Cursor right one character
201	C9	ALT Right	Cursor to end of line
202	CA	CTRL Right	Delete character under cursor
203	CB	CTRL ALT Right	Delete to end of line
204	CC	SHIFT Right	Cursor right one word
205	CD	SHIFT ALT Right	Pan right
206	CE	SHIFT CTRL Right	Delete word under & right of cursor
207	CF	SHIFT CTRL ALT Right	
208	D0	Up	Cursor up
209	D1	ALT Up	Scroll up
210	D2	CTRL Up	Search backwards
211	D3	ALT CTRL UP	
212	D4	SHIFT Up	Top of screen
213	D5	SHIFT ALT Up	
214	D6	SHIFT CTRL Up	
215	D7	SHIFT CTRL ALT Up	
216	D8	Down	Cursor down
217	D9	ALT Down	Scroll down
218	DA	CTRL Down	Search forwards
219	DB	ALT CTRL Down	
220	DC	SHIFT Down	Bottom of screen
221	DD	SHIFT ALT Down	
222	DE	SHIFT CTRL Down	
223	DF	SHIFT CTRL ALT Down	
224	E0	CAPSLOCK	Toggle CAPSLOCK function
225	E1	ALT CAPSLOCK	
226	E2	CTRL CAPSLOCK	
227	E3	ALT CTRL CAPSLOCK	
228	E4	SHIFT CAPSLOCK	
229	E5	SHIFT ALT CAPSLOCK	
230	E6	SHIFT CTRL CAPSLOCK	
231	E7	SHIFT CTRL ALT CAPSLOCK	
232	E8	F1	
233	E9	CTRL F1	
234	EA	SHIFT F1	
235	EB	CTRL SHIFT F1	
236	EC	F2	
237	ED	CTRL F2	
238	EE	SHIFT F2	
239	EF	CTRL SHIFT F2	

Decimal	Hex	Keying	Display/Function
240	F0	F3	
241	F1	CTRL F3	
242	F2	SHIFT F3	
243	F3	CTRL SHIFT F3	
244	F4	F4	
245	F5	CTRL F4	
246	F6	SHIFT F4	
247	F7	CTRL SHIFT F4	
248	F8	F5	
249	F9	CTRL F5	
250	FA	SHIFT F5	
251	FB	CTRL SHIFT F5	
252	FC	SHIFT space	"Special" space
253	FD	SHIFT TAB	Back tab (CTRL ignored)
254	FE	SHIFT ENTER	"Special" newline (CTRL ignored)
255	FF	See below	

Codes up to 20 hex are either control characters or non-printing characters. Alternative keyings are shown in brackets after the main keying.

Note that CTRL-C is trapped by Qdos and cannot be detected without changes to the system variables.

Note that codes C0-DF are cursor control commands.

The ALT key depressed with any key combination other than cursor keys or CAPSLOCK generates the code FF, followed by a byte indicating what the keycode would have been if ALT had not been depressed.

Note that CAPSLOCK and CTRL-F5 are trapped by Qdos and cannot be detected without special software.

clock

The QL contains a real time clock which runs when the computer is switched on.

The format used for the date and time is standard ISO format:

```
1983 JAN 01 12:09:10
```

Individual year, month, day, and time can all be obtained by assigning the string returned by **DATE** to a *string variable* and *slicing* it. The clock will run from 1961 JAN 01 00:00:00.

comment For a description of the format see BS5249: PART 1: 1976 and as modified in Appendix D.2.1 Table 5 Serial 5 and Appendix E.2 Table 6 Serials 1 and 2.

Command	Function
SDATE	set the clock
ADATE	adjust the clock
DATE	return the date as a number
DATE$	return the date as a string
DAY$	return day of the week

FOR SERVICE MANUALS
CONTACT:
MAURITRON TECHNICAL SERVICES
www.mauritron.co.uk
TEL: 01844 - 351694
FAX: 01844 - 352554

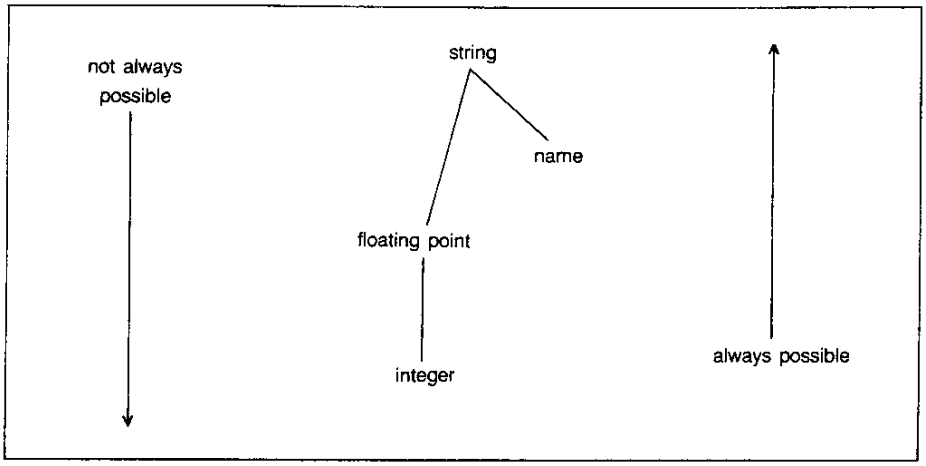
coercion

If necessary SuperBASIC will convert the type of unsuitable data to a type which will allow the specified operation to proceed.

The *operators* used determine the conversion required. For example, if an operation requires a *string* parameter and a numeric parameter is supplied then SuperBASIC will first convert the parameter to type string. It is not always possible to convert data to the required form and if the data cannot be converted an error is reported.

The type of a *function or procedure* parameter can also be converted to the correct type. For example, the SuperBASIC **LOAD** command requires a parameter of type *name* but can accept a parameter of type *string* and which will be converted to the correct type by the procedure itself. Coercion of this form is always dependent on the way the function or procedure was implemented.

There is a a natural ordering of data types on the QL, see figure. *String* is the most general type since it can represent names, floating point and *integer* numbers. *Floating point* is not as general as string but it is more general than integer since floating point data can represent integer data (almost exactly). The figure below shows the ordering diagramatically. Data can always be converted moving up the diagram but it is not always possible moving down.



example

`a = b + c`	(no conversion is necessary before performing the addition. Conversion is not necessary before assigning the result to a.)
`a% = b + c`	(no conversion is necessary before performing the addition but the result is converted to integer before assigning)
`a$ = b$ + c$`	(**b$** and **c$** are converted to floating point, if possible, before being added together. The result is converted to string before assigning.)
`LOAD "mdv1_data"`	(the string **"mdv1_data"** is converted to type name by the load procedure before it is used.)

comment

Statements can be written in SuperBASIC which would generate errors in most other computer languages. In general it is possible to mix data types in a very flexible manner:

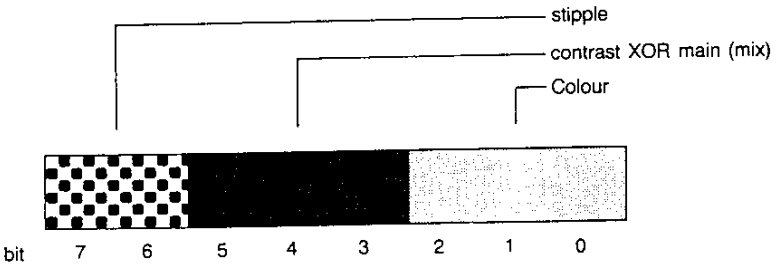
i. `PRINT "1" + 2 + "3"`
ii. `LET a$ = 1 + 2 + a$ + "4"`

colour

Colours on the QL can be either a **solid colour** or a **stipple** – a mixture of two colours to some predefined pattern. Colour specification on the QL can be up to three items; a colour, a contrast colour and a stipple pattern.

single *colour*:= *composite_colour*

The single argument specifies the three parts of the colour specification. The main colour is contained in the bottom three bits of the colour byte. The next three bits contain the exclusive or (XOR) of the main colour and the contrast colour. The top two bits indicate the *stipple* pattern.



By specifying only the bottom three bits (i.e. the required colour) no *stipple* will be requested and a single solid colour will be used for display.

double *colour*:= *background, contrast*

The *colour* is a *stipple* of the two specified colours. The default checkerboard stipple is assumed (stipple 3).

triple *colour*:= *background, contrast, stipple*

Background and *contrast* colours and *stipple* are each defined separately.

colours The codes for colour selection depend on the screen mode in use:

code	bit pattern	composition	colour	
			8 colour	4 colour
0	0 0 0		black	black
1	0 0 1	blue	blue	black
2	0 1 0	red	red	red
3	0 1 1	red + blue	magenta	red
4	1 0 0	green	green	green
5	1 0 1	green + blue	cyan	green
6	1 1 0	green + red	yellow	white
7	1 1 1	green + red + blue	white	white

Colour Composition and Codes

stipples Stipples mix a backgound and a contrast colour in a fine stipple pattern. Stipples can be used on the QL in the same manner as ordinary solid colours although stipples may not be reproduced correctly on an ordinary domestic television. There are four stipple patterns:

Stipple 0 Stipple 1 Stipple 2 Stipple 3

Stipple 3 is the default.

example

i. `PAPER 255 : CLS`
ii. `PAPER 2,4 : CLS`
iii. `PAPER 0,2,0 : CLS`

warning Stipples may not reproduce correctly on a domestic television set which is fed via the UHF socket.

RS-232-C

The QL has two serial ports (called SER1 and SER2) for connecting it to equipment which uses serial communications obeying EIA standard RS-232-C or a compatible standard.

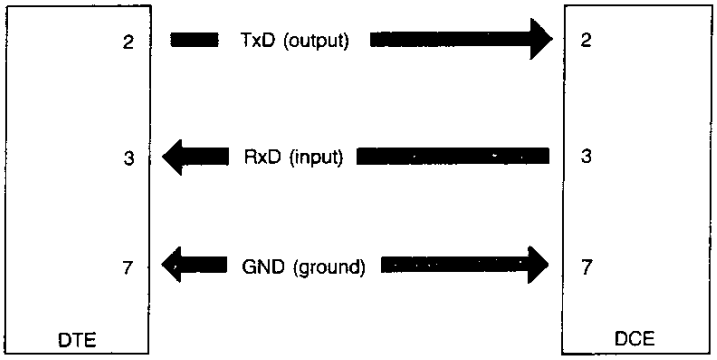
The RS-232-C 'standard' was originally designed to enable computers to send and receive data via telephone lines using a modem. However, it is now frequently used to connect computers directly with each other and to various items of peripheral equipment, e.g. printers, plotters, etc.

As the RS-232-C 'standard' manifests itself in many different forms on different pieces of equipment, it can be an extremely difficult job, even for an expert, to connect together for the first time two pieces of supposedly standard RS-232-C equipment. This section will attempt to cover most of the basic problems that you may encounter.

The RS-232-C 'standard' refers to two types of equipment:

1 Data Terminal Equipment (DTE)
2 Data Communication Equipment (DCE)

The standard envisaged that the terminal (usually the DTE) and the modem (usually the DCE) would both have the same type of connector.



The diagram above illustrates how the DTE transmits data on pin 2 whilst the DCE must receive data on its pin 2 (which is still called transmit data!). Likewise, the DTE receives data on pin 3 whilst the DCE must transmit data on its pin 3 (which is still called receive data!). Although this is confusing in itself, it can lead to far greater problems when there is disagreement as to whether a certain device should be configured as DCE or DTE.

Unfortunately, some people decide that their computers should be configured as DCE devices whilst others configure equivalent computers as DTE devices. This obviously leads to difficulties in the configuration of the serial ports on each piece of equipment.

SER1 on the QL is configured as DCE, while SER2 is configured as DTE. Therefore it should be possible to connect at least one of the serial ports to a given device simply by using whichever port is wired the 'correct' way. The pin-out for the serial ports is given below. A cable for connecting the QL to a standard 25-way 'D' type connector is available from Sinclair Research Limited.

SER1

pin	name	function
1	GND	signal ground
2	TxD	input
3	RxD	output
4	DTR	ready input
5	CTS	ready output
6	-	+12V

TxD Transmit Data
RxD Receive Data

SER2

pin	name	function
1	GND	signal ground
2	TxD	output
3	RxD	input
4	DTR	ready output
5	CTS	ready input
6	-	+12V

DTR Data Terminal Ready
CTS Clear To Send

Once the equipment has been connected to the 'correct' port, the baud rate, (the speed of transmission of data) must be set so that the baud rates for both the QL and the connected equipment are the same. The QL can be set to operate at:

75
300
600
1200
2400
4800
9600
19200 (transmit only) baud.

The QL baud rate is set by the **BAUD** command and is set for both channels. The baud rates cannot be set independently.

The **parity** to be used by the QL must also be set to match that expected by the peripheral equipment. Parity is usually used to detect simple transmission errors and may be set to be even, odd, mark, space or no parity, i.e. all 8 bits of the byte are used for data.

Stop bits mark the end of transmission of a byte or character. The QL will receive data with one, one and a half, or two stop bits, and will always transmit data with at least two stop bits. Note that if the QL is set up to 9600 baud it will not receive data with only one stop bit: at least 1½ stop bits are required.

It may be necessary to connect the **handshake** lines between the QL and a piece of equipment connected to it. This allows the QL and its peripheral to monitor and control their rate of communication. They may need to do this if one of them cannot cope with the speed at which data is being transmitted. The QL uses two handshaking lines:

CTS Clear to Send
DTR Data Terminal Ready.

If the DTE cannot cope with the rate of transmission of data then it can negate the DTR line which tells the DCE to stop sending data. Obviously when the DTE has caught up it tells the DCE, via the DTR line, to start transmitting again. In the same way, the DCE can stop the DTE sending data by negating the CTS line. If additional control signals are required they can be wired up using the 12V supply available on both serial ports.

Although transmission from the QL is often possible without any handshaking at all, the QL will not receive correctly under any circumstances without the use of CTS on SER1 and DTR on SER2.

Communications on the QL are 'full duplex', that is both receive and transmit can operate concurrently.

The parity and handshaking are selected when the serial channel is opened.

command	function
BAUD	set transmission speed
OPEN	open serial channels *
CLOSE	close serial channels

* see concept *device* for a full specification

data types variables

integer

Integers are whole numbers in the range −32768 to +32767. *Variables* are assumed to be integer if the variable identifier is suffixed with a percent %. There are no integer constants in SuperBASIC, so all constants are stored as *floating point* numbers.

syntax: *identifier*%

example:
i. `counter%`
ii. `size_limit%`
iii. `this_is_an_integer_variable%`

floating point

Floating point numbers are in the range $\pm(10^{-615}$ to $10^{+615})$, with 8 significant digits. Floating point is the default data type in SuperBASIC. All constants are held in floating point form and can be entered using exponent notation.

syntax: *identifier* | *constant*

example:
i. `current_accumulation`
ii. `76.2356`
iii. `354E25`

string

A string is a sequence of characters up to 32766 characters long. *Variables* are assumed to be type string if the variable name is suffixed by a $. String data is represented by enclosing the required characters in either single or double quotation marks.

syntax: *identifier*$ | *"text"*

example:
i. `string_variables$`
ii. `"this is string data"`
iii. `"this is another string"`

name

Type name has the same form as a standard SuperBASIC *identifier* and is used by the system to name *Microdrive files* etc.

syntax: *identifier*

example:
i. `mdv1_data_file`
ii. `ser1e`

devices

A **device** is a piece of equipment on the QL to which data can be sent (input) and from which data can be output.

Since the system makes no assumptions about the ultimate I/O (input/output) device which will be used, the I/O device can be easily changed and the data diverted between devices. For example, a *program* may have to output to a printer at some point during its run. If the printer is not available then the output can be diverted to a *Microdrive file* and stored. The file can then be printed at a later date. I/O on the QL can be thought of as being written to and read from a **logical file** which is in a standard device-independent form.

All device specific operations are performed by individual **device drivers** specially written for each device on the QL. The system can automatically find and include drivers for peripheral devices which are fitted. These should be written in the standard QL device driver format; see the concept *peripheral expansion*

When a device is activated a *channel* is opened and linked to the device. To correctly open a channel device basic information must sometimes be supplied. This extra information is appended to the device name.

The file name should conform to the rules for a SuperBASIC *type name* though it is also possible to build up the file name (device name) as a SuperBASIC *string expression*.

In summary the general form of a file name is:

identifier [*information*]

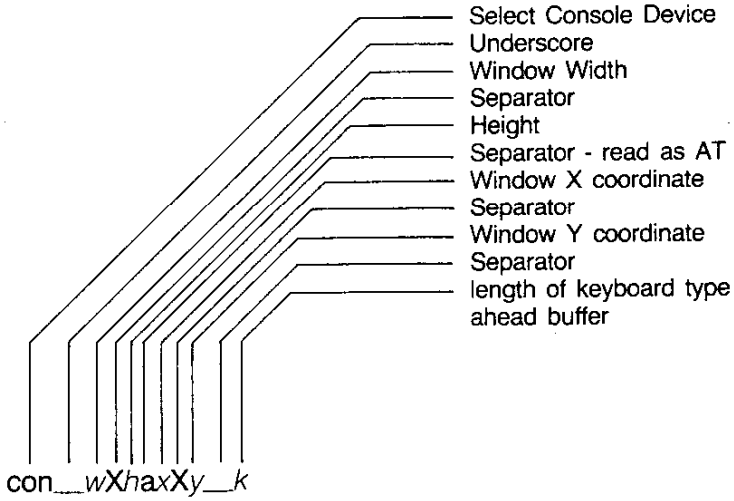
where the complete file name (including the extra information) conforms to the rules for a SuperBASIC identifier.

Each logical device on the system requires its own particular 'extra information' although default parameters will be assumed in each case where possible:

define *device*:= *name*

where the form of the device name is outlined below.

example for console device



CON__*wXhaxXy__k*

Console I/O

[*wXh*]	- window *width, height*
[*AxXy*]	- window *X,Y* coordinate of upper left-hand corner
[*k*]	- keyboard type ahead buffer length (bytes)

default: con__448x180a32x16__128

example:

```
OPEN #4,con_20x50a0x0_32
OPEN #8,con_20x50
OPEN #7,con_20x50a10x10
```

Screen Output — SCR__*wXhaxXy*

[*wXh*] - window *width, height*
[A*xXy*] - window *X,Y* coordinate

default: scr__448x180a32x16

example:
```
OPEN #4, scr_10x10a20x50
OPEN #5, scr_10x10
```

Serial (RS-232-C) — SER*nphz*

n port number (1 or 2)

[*p*] parity
- e – even
- o – odd
- m – mark
- s – space

[*h*] handshaking
- i – ignore
- h – handshake

[*z*] protocol
- r – raw data no EOF
- z – control Z is EOF
- c – as z but converts ASCII 10 (Qdos newline character) to ASCII 13 <CR>)

default: **ser1rh** (8 bit no parity with handshake)

example:
```
OPEN #3, ser1e
OPEN #4, serc
COPY mdv1_test_file TO ser1c
```

Serial Network I/O — NET*d__s*

[*d*] indicates direction
- i - input
- o - output

s station number
- 0 - for broadcast
- **own station** - for general listen (input only)

default: no default

example:
```
OPEN #7, neti_32
OPEN #4, neto_0
COPY ser1 TO neto_21
```

Microdrive File Access — MDVn__*name*

n Microdrive number
name Microdrive file name

default: no default

example:
```
OPEN #9, mdv1_data_file
OPEN #9, mdv1_test_program
COPY mdv1_test_file TO scr_
```

Keyword	Function
OPEN	initialise a device and activate it for use
CLOSE	deactivate a device
COPY	copy data between devices
COPY__N	copy data between devices, but do not copy a file's header information
EOF	test for end of file
WIDTH	set width

direct command

SuperBASIC makes a distinction between a statement typed in preceded by a line number and a statement typed in without a line number. Without a line number the statement is a **direct command** and is processed immediately by the SuperBASIC **command interpreter**. For example, **RUN** is typed in on the command line and is processed, the effect being that the program starts to run. If a statement is typed in with a line number then the syntax of the line is checked and any detectable syntax errors reported. A correct line is entered into the SuperBASIC program and stored. These statements constitute a SuperBASIC *program* and will only be executed when the program is started with the **RUN** or **GOTO** command.

Not all SuperBASIC statements make sense when entered as a direct command, for example, **END FOR**, **END DEFine**, etc.

error handling

Errors are reported by SuperBASIC in a standard form:

At line *line_number error_text*

Where the line number is the number of the line where the error was detected and the error text is listed below.

(1) **Not complete**
An operation has been prematurely terminated (or break has been pressed).

(2) **Invalid job**
An error return from Qdos relating to system calls controlling multitasking or I/O.

(3) **Out of memory**
Qdos and/or SuperBASIC has insufficient free memory.

(4) **Out of range**
Usually results from attempts to write outside a window or an incorrect array index.

(5) **Buffer full**
An I/O operation to fetch a buffer full of characters filled the buffer before a record terminator was found.

(6) **Channel not open**
Attempt to read, write or close a channel which has not been opened.
Can also occur if an attempt to open a channel fails.

(7) **Not found**
File system, device, medium or file cannot be found.
SuperBASIC cannot find an identifier. This can result from incorrectly nested structures.

(8) **Already exists**
The file system has found an already existing file with the same name as a new file to be opened for writing.

(9) **In use**
The file system has found that a file or device is already exclusively used.

(10) **End of file**
End of file detected during input.

(11) **Drive full**
A device has been filled (usually Microdrive).

(12) **Bad name**
The file system has recognised the name but there is a syntax or parameter value error.

In SuperBASIC it means a name has been used out of context. For example, a variable has been used as a procedure.

(13) **Xmit error**
RS-232-C parity error.

(14) **Format failed**
Attempted format operation has failed, the medium is possibly faulty (usually a Microdrive cartridge).

(15) **Bad parameter**
There is an error in the parameter list of a system or SuperBASIC procedure or function call.
An attempt was made to read data from a write only device.

(16) **Bad or changed medium**
The medium (usually a Microdrive cartridge) is possibly faulty.

(17) **Error in expression**
An error was detected while evaluating an expression.

(18) **Overflow**
Arithmetic overflow, division by zero, square root of a negative number, etc.

(19) **Not Implemented**

(20) Read only
There has been an attempt to write data to a shared file.

(21) Bad line
A SuperBASIC syntax error has occurred.

(22) PROC/FN cleared
This is a message which is for information only and is not reporting an error. It is reporting that the program has been stopped and subsequently changed forcing SuperBASIC to reset its internal state to the outer program level and so losing any procedure environment which may have been in effect.

error recovery

After an error has occurred the program can be restarted at the next statement by typing

```
CONTINUE
```

If the error condition can be corrected, without changing the program, the program can be restarted at the statement which triggered the error. Type

```
RETRY
```

FOR SERVICE MANUALS
CONTACT:
MAURITRON TECHNICAL SERVICES
www.mauritron.co.uk
TEL: 01844 - 351694
FAX: 01844 - 352554

expressions

SuperBASIC expressions can be *string*, *numeric*, *logical* or a mixture; unsuitable data types are automatically converted to a suitable form by the system wherever this is possible.

define

monop:= | +
| –
| NOT

expression:= | [*monop*] *expression operator expression*
| (*expression*)
| *atom*

atom:= | *variable*
| *constant*
| *function* [(*expression* *[, *expression*] *)]
| *array__element*

variable:= | *identifier*
| *identifier* %
| *identifier* $

function:= | *identifier*
| *identifier* %
| *identifier* $

constant:= | *digit* *[*digit*] *
| *[*digit*] * . *[*digit*] *
| *[*digit*] * [.] *[*digit*] * E *[*digit*] *

The final value returned by the evaluation of the expression can be integer giving an **integer__expression**, string giving a **string__expression** or floating point giving a **floating__expression**. Often floating point and integer expressions are equivalent and the term **numeric__expression** is then used.

Logical operators can be included in an expression. If the specified operation is true then a one is returned as the value of the operation. If the operation is false then a zero is returned. Though logical operators can be used in any expression they are usually used in the expression part of an **IF** statement.

example:

i. `test_data + 23.3 + 5`
ii. `"abcdefghijklmnopqrstuvwxyz"(2 TO 4)`
iii. `32.1 * (colour=1)`
iv. `count = -limit`

file types

files

All I/O on the QL is to or from a *logical file*. Various file types exist:

data SuperBASIC programs, text files. Created using **PRINT**, **SAVE**, accessed using **INPUT**, **INKEY$**, **LOAD** etc.

exec An executable transient program. Saved using **SEXEC**, loaded using **EXEC**, **EXEC_W** etc.

code Raw memory data, screen images, etc. Saved using **SBYTES**, loaded using **LBYTES**.

functions and procedures

SuperBASIC *functions and procedures* are defined with the **DEFine FuNction** and **DEFine PROCedure** statements. A function is activated (or called) by typing its name at the appropriate point in a SuperBASIC expression. The function must be included in an expression because it is returning a value and the value must be used. A procedure is activated (or called) by typing its name as the first item in a SuperBASIC statement.

Data can be passed into a function or procedure by appending a list of **actual parameters** after the function or procedure name. This list is compared to a similar list appended after the name of the function or procedure when it was defined. This second list is called the **formal parameters** of the function or procedure. The formal parameters must be SuperBASIC variables. The actual parameters must be an *array*, an *array slice* or a SuperBASIC *expression* of which a single *variable* or constant is the simplest form.

Since the actual parameters are actual expressions, they must have an actual type associated with them. The formal parameters are merely used to indicate how the actual parameters must be processed and so have no type associated with them. The items in each list of parameters are paired off in order when the function or procedure is called and the formal parameters become equivalent to the actual parameters. There are three distinct ways of using parameters.

If the actual parameter is a single variable and if data is assigned to the formal parameter in the function or procedure then the data is also assigned to the corresponding actual parameter.

If the actual parameter is an expression then assigning data to the corresponding formal parameter will have no effect outside the procedure. Note that a variable can be turned into an expression by enclosing it within brackets.

If the actual parameter is a variable but has not previously been set then assigning data to the corresponding formal parameter will set the variable specified as the actual parameter.

Variables can be defined to be local to a function or procedure with the **LOCal** statement. Local variables have no effect on similarly named variables outside the function or procedure in which they are defined and so allow greater freedom in choosing sensible variable names without the risk of corrupting external variables. A local variable is available to any inside function or procedure called from the procedure function in which it is declared to be local unless the function or procedure called contains a further local declaration of the same variable name.

Functions and procedures in SuperBASIC can be used recursively. That is a function or procedure can call itself either directly or indirectly.

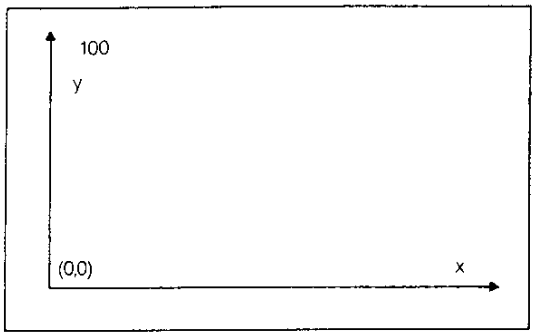
Command	Function
DEFine FuNction	define a function
DEFine PROCedure	define a procedure
RETurn	leave a function or procedure (return data from a function)
LOCal	define local data in a function or procedure

graphics

It is important to realise that the QL screen has non square pixels and that changing mode will change the shape of the pixels. Thus if the graphics procedures were simply pixel based they would draw different shapes in the two modes. For example, in one mode we would have a circle while the same figure in the other mode would be an ellipse.

The graphics procedures ensure that whatever screen mode is in use, consistent figures are produced. It is not possible to use a simple pixel count to indicate sizes of figures, so instead the graphics procedures use an arbitrary scale and coordinate system to specify sizes and positions of figures.

The graphics procedures use the **graphics co-ordinate system**, i.e. draw relative to the **graphics origin** which is in the bottom left hand corner of the specified or default window. Note that this is not the same as the *pixel origin* used to define the position of *windows* and *blocks*, etc. The graphics origin allows a standard Cartesian coordinate system to be used. A graphics cursor is updated after each graphics operation; subsequent operations can either be relative to this cursor or can be absolute, i.e. relative to the graphics origin.



The Graphics Coordinate System

The **scaling factor** is such that the full distance in the vertical direction in the specified or default window has length 100 by default and can be changed with the SCALE command. The scale in the x direction is equal to the scale in the y direction. However, the length of line which can be drawn in the x direction is dependent on the shape of the window. Increasing the scale factor increases the maximum size of the figure which can be drawn before the window size is exceeded. If the graphics output is switched to a different size of window then the subsequent size of the output is adjusted to fit the new window. If a figure exceeds its output window then the figure is clipped.

It is useful to consider the window to be a window onto a larger graphics space in which the figures are drawn. The **SCALE** command allows the graphics origin to be set so allowing the window to be moved around the graphics space.

The graphics procedures are output to the window attached to the specified or default *channel* and the output is drawn in the **INK** colour for that channel.

Command	Function	
CIRCLE	draw an ellipse or a circle	absolute
LINE	draw a line	
ARC	draw an arc of a circle	
POINT	plot a point	
CIRCLE_R	draw an ellipse or a circle	relative
LINE_R	draw a line	
ARC_R	draw an arc of a circle	
POINT_R	plot a point	
SCALE	set scale and move origin	
FILL	fill in a shape	
CURSOR	position text	

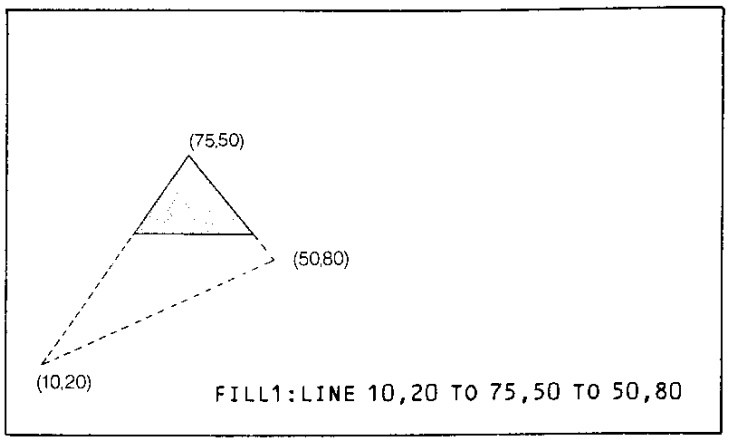
graphics fill

Figures drawn with the graphics and turtle graphics procedures can be optionally 'filled' with a specified stipple or colour. If **FILL** is selected then the figure is filled as it is drawn.

The **FILL** algorithm stores a list of points to plot rather than actually plotting them. When the figure closes there are two points on the same horizontal line. These two points are

connected by a line in the current ink colour and the process repeats. Fill must always be reselected before drawing a new figure to ensure that the buffer used to store the list of points is reset.

The following diagram illustrates **FILL**:



warning

There is an implementation restriction on **FILL**. **FILL** must not be used for re-entrant shapes (i.e. a shape which is concave). Re-entrant shapes must be split into smaller shapes which are not re-entrant and each sub-shape filled independently.

identifer

A SuperBASIC identifier is a sequence of letters, numbers and underscores.

define: *letter*:= | a..Z
| A..Z

number:= | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 0 |

identifier:= *letter* *|| *letter* | *number* | _ | | *

example:
i. `a`
ii. `limit_1`
iii. `current_guess`
iv. `counter`

An identifier must begin with a letter followed by a sequence of letters, numbers and underscores and can be up to 255 characters long. Upper and lower case characters are equivalent.

Identifiers are used in the SuperBASIC system to identify *variables*, *procedures*, *functions*, *repetition* loops, etc.

warning NO meaning can be attributed to an identifier other than its ability to 'identify' constructs to SuperBASIC. SuperBASIC cannot infer the intended use of an identifier from the identifier's name!

joystick

The joystick ports, marked CTL1 and CTL2, allow two joysticks to be attached to the QL.

The joysticks are arranged to generate specific key depressions when moved in a specific way and any program which uses a joystick must be able to adapt to these keys. The QL keyboard can be read directly using the KEYROW function.

	CTL1	CTL2
mode	key	key
up	cursor up	F4
down	cursor down	F2
left	cursor left	F1
right	cursor right	F3
fire	space	F5

comment

The joystick ports can be used for adding other more general purpose control devices to the QL.

Joysticks for other computers using a 9-way 'D' connector require an adaptor to be used with the QL. Such an adaptor is available from Sinclair Research.

keyword

SuperBASIC keywords are identifiers which are defined in the SuperBASIC *Keyword Reference Guide*. Keywords have the same form as a SuperBASIC standard *identifier*. The case of the keyword is not significant. Keywords are echoed as a mixture of upper and lower case letters and are always reproduced in full. The upper case portion indicates the minimum required to be typed in for SuperBASIC to recognise the keyword.

The set of SuperBASIC keywords may be extended by adding *procedures* to the QL. It is a good idea to define these with their names in upper case, procedure names defined this way will always be reproduced by SuperBASIC in upper case, and this will indicate their special function in the SuperBASIC system. Conversely, ordinary procedures should be defined with their names in lower case.

warning Existing keywords cannot be used as ordinary identifiers within a SuperBASIC program. SuperBASIC keywords are:

List of Keywords

ABS	DEFine PROCedure	LEN	RANDOMISE
ACOS, ASIN	END DEFine	LET	RND
ACOT, ATAN	DEG	LIST	RECOL
ADATE	DELETE	LOAD	REMark
ARC, ARC_R	DIM	LOCal	RENUM
AT	DIMN	LN, LOG10	REPeat,
AUTO	DIR	LRUN	END REPeat
BAUD	DIV	MERGE	RESPR
BEEP	DLINE	MOD	RETurn
BEEPING	EDIT	MODE	RETRY
BLOCK	ELLIPSE,	MOVE	RUN
BORDER	ELLIPSE_R	MRUN	SAVE
CALL	EOF	NET	SIN
CHR$	EXEC, EXEC_W	NEW	SCALE
CIRCLE	EXIT	NEXT	SCROLL
CIRCLE_R	EXP	ON GO TO	SDATE
CLEAR	FILL	ON GO SUB	SELect
CLOSE	FILL$	OPEN, OPEN_IN	END SELect
CLS	FLASH	OPEN_NEW	SEXEC
CODE	FOR	OVER	SQRT
CONTINUE	END FOR	PAN	STOP
RETRY	FORMAT	PAPER	STRIP
COPY, COPY_N	GO SUB	PAUSE	TAN
COS	GO TO	PEEK, PEEK_W	TO
COT	IF, THEN, ELSE	PEEK_L	TURN
CSIZE	END IF	PENUP	TURN TO
CURSOR	INK	PENDOWN	UNDER
DATA, READ,	INKEY$	PI	VER$
RESTORE	INPUT	POINT, POINT_R	WIDTH
DATE$, DATE	INSTR	POKE, POKE_W	WINDOW
DAY$	INT	POKE_L	
DEFine FuNction,	KEYROW	PRINT	
END DEFine	LBYTES	RAD	

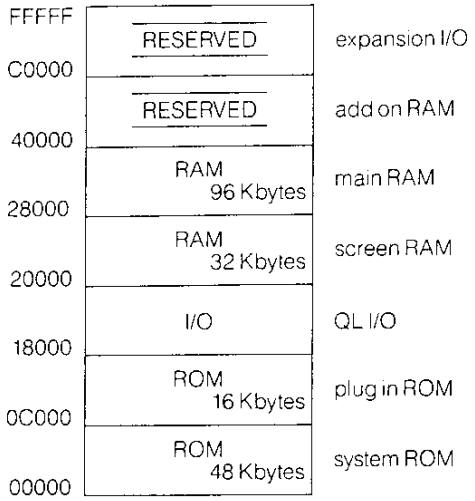
maths functions

SuperBASIC has the standard trigonometrical and mathematical functions

Function	Name
COS	cosine
SIN	sin
TAN	tangent
ATAN	arctangent
ACOT	arcotangent
ACOS	arcosine
ASIN	arcsine
COT	cotangent
EXP	exponential
LN	natural logarithm
LOG10	common logarithm
INT	integer
ABS	absolute value
RAD	convert to radians
DEG	convert to degrees
PI	return the value of π
RND	generate a random number
RANDOMISE	reseed the random number. generator

memory map

The QL contains a Motorola 68008 microprocessor, which can address 1 Megabyte of memory, i.e. from 00000 to FFFFF Hex. The use of addresses within this range are defined by Sinclair Research to be as follows:



Physical Memory Map

The screen RAM is organised as a series of sixteen bit words starting at address Hex 20000 and progressing in the order of the raster scan, i.e. from left to right with each display line and then from the top to the bottom of the picture. The bits within each word are organised so that a pixel to the left is always more significant than a pixel to the right (i.e. the pixel pattern on the screen looks the same as the binary pattern). However, the organisation of the colour information in the two screen modes is different:

high byte AO=0	low byte AO=1	mode
GGGGGGGG	RRRRRRRR	512 mode (high res)
GFGFGFGF	RBRBRBRB	256 mode (low res)

G—green B—blue R—red F—flash

Setting the Flash bit toggles the flash state and freezes the backgound colour for the flash to the value given by R, G, and B for that pixel. Flashing is always reset at the beginning of each display line.

In high resolution mode, red and green specified together is interpreted by the hardware as white.

warning Use of reserved areas in the memory map may cause incompatibility with future Sinclair products. Spurious output to addresses defined to be peripheral I/O addresses can cause unpredictable behaviour. It is recommended that these areas are NOT written to and not used for any other purpose. Poking areas in use as Microdrive buffers can corrupt Microdrive data and can result in a loss of information. Poking areas in use such as system tables can cause the system to crash and can result in the loss of data and programs.

All I/O should be performed using either the relevant SuperBASIC commands or the *Qdos operating system* traps.

Microdrives

Microdrives provide the main permanent storage on the QL. Each Microdrive cartridge has a capacity of at least 100 Kbytes. Available free memory space is allocated by Qdos as Microdrive buffers when necessary to improve performance.

Each blank cartridge must be **formatted** before use and can hold up to 255 sectors of 512 bytes per sector. Qdos keeps a directory of *files* stored on the cartridge. Each Microdrive file is identified using a standard SuperBASIC *file* or *device* name.

A cartridge can be write-protected by removing the small lug on the right hand side.

On receiving new blank QL Microdrive cartridges, format them a few times to condition the tape.

general care

Physically each Microdrive cartridge contains a 200 inch loop of high quality video tape which is moved at 28 inches per second. The tape completes one circuit every 7½ seconds.

NEVER touch the tape with your fingers or insert anything into the cartridge

NEVER turn the computer on or off with cartridges in place.

ALWAYS store cartridges in their sleeves when not in use.

ALWAYS insert or remove cartridges from the Microdrive slowly and carefully.

ALWAYS ensure the cartridge is firmly installed before starting the Microdrive.

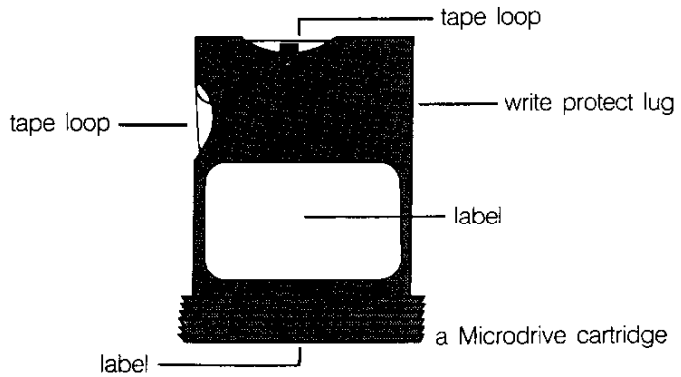
NEVER move the QL with cartridges installed - even if not in operation.

NEVER touch the cartridge while the Microdrive is in operation.

DO NOT repeatedly insert and remove the cartridge without running the Microdrive.

tape loops

If a tape loop appears at either of the two places shown in figure 1 then gently ease it back into the cartridge. Use a non-fibrous instrument for this, e.g. the side of a pen or pencil. NEVER touch the tape with your fingers for this or any other reason.



CONTINUED.....

FOR SERVICE MANUALS
CONTACT:
MAURITRON TECHNICAL SERVICES
www.mauritron.co.uk
TEL: 01844 - 351694
FAX: 01844 - 352554

Command	Function
FORMAT	prepare a new cartridge for use
DELETE	delete a file from a cartridge
DIR	lists the files on a cartridge
SAVE SBYTES SEXEC	saves data from a cartridge
LOAD LBYTES EXEC MERGE	loads data from a cartridge
OPEN_IN OPEN_NEW OPEN CLOSE	opens and closes files
PRINT INPUT INKEY$	SuperBASIC file I/O

warning If you attempt to write to a cartridge which is write protected then the QL will repeatedly attempt to write the data but will eventually give up and give a "bad medium" error.

A monitor may be connected to the QL via the RGB socket on the back of the computer. Connection is via an 8-way DIN plug plus cable for colour monitors, or a 3-way DIN plug plus cable for monochrome. The RGB socket connections are as in the following table, and the column indicating wire colour refers to the colour coding used on the 8-way cable and connector available from Sinclair Research Limited. Pin designation is as shown in the diagram below.

pin	function	signal		sleeve colour on QL RGB colour lead
1	PAL	composite PAL	(4)	orange
2	GND	ground		green
3	VIDEO	composite monocrhome video	(3)	brown
4	CSYNC	composite sync	(2)	yellow
5	VSYNC	vertical sync	(1)	blue
6	GREEN	green	(1)	red
7	RED	red	(1)	white
8	BLUE	blue	(1)	purple

A monochrome monitor can be connected using a screened lead with a 3-way or as 8-way DIN plug at the QL end. Only pins 2 (ground) and 3 (composite video) need to be connected via the cable to the monitor. The connection at the monitor end will vary according to the monitor but is usually a phono plug. The monitor must have a 75 ohm 1V pk-pk composite video non-inverting input (which is the industry standard). Both 3-way DIN plugs and phono plugs are commonly available from audio shops.

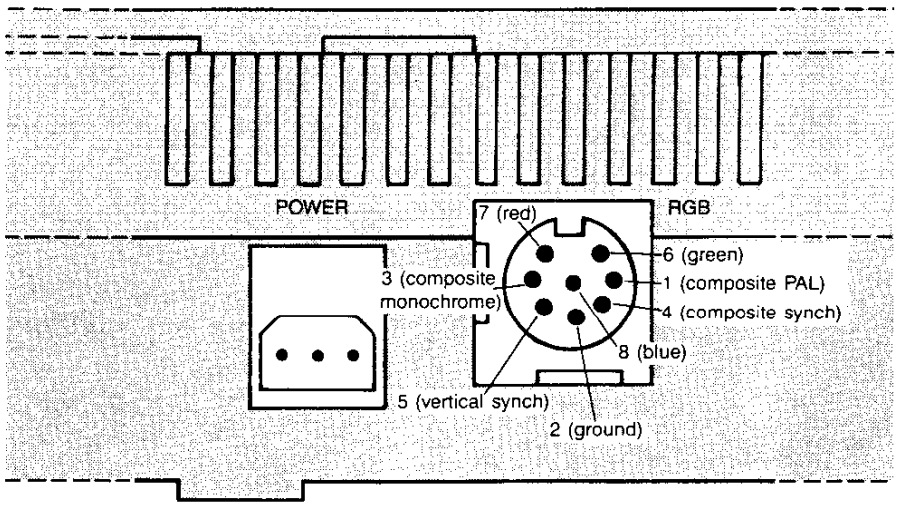


Diagram of Monitor Connector as Viewed from rear of QL. Showing pin numbers and functions.

An RGB (colour) monitor can be connected using a lead with an 8-way DIN Plug at the QL end. The connection at the monitor end will vary according to the monitor (there is no industry standard) and will often be supplied with it. A suitable cable with an 8-way DIN plug at one end and bare wires at the other end is available from Sinclair Research Limited.

A composite PAL monitor, or the composite video input on some VCR's, may work with the QL. Only pins 2 (ground) and 1 (composite PAL) need to be connected via a cable, to the monitor or VCR.

network

The QL can be connected with up to 63 other QLs. If there are more than two computers on the network then each computer (or station) must be assigned a unique station number. On the QL this can be done using the NET command.

Information is transmitted over the network in blocks. For normal communication between two stations the receiving station must acknowledge correct reception of the block. If a block is corrupted then the receiving station will request retransmission.

Using a network station number of zero has a special meaning. Sending to **neto__0** is called broadcasting: any message sent in this way can be read by any station which is listening to **neti__0**. Note that the normal verification that a message has been received is disabled for broadcasts, so that broadcasting messages of length more than one block (255 bytes) is unreliable.

A network station which listens to its own station number (e.g. NET3:LOAD neti__3) can receive data from any station sending to it.

Command	Function
NET	assign a network station number
OPEN CLOSE	open a network channel close a network channel
PRINT INPUT INKEY$	network I/O
LOAD SAVE LBYTES SBYTES EXEC SEXEC LRUN MRUN MERGE	load and save via network

comment

If you are planning to connect several QLs on the network, or use a long piece of cable, then you should wire it up with low-capacitance twin core cable, such as 3 amp light-flex or bell-wire. Take care to connect the centres of each jack to each other, and the outsides to each other. You will find that although the software can handle 63 stations, the hardware will not drive more than about 100m of cable, depending on what type it is.

If you are only connecting a few machines with the leads supplied, you need not worry.

operators

Operator	Type	Function
=	floating string	logical type 2 comparison
==	numeric string	almost equal** (type 3 comparison)
+	numeric	addition
-	numeric	subtraction
/	numeric	division
*	numeric	multiplication
<	numeric string	less than (type 2 comparison)
>	numeric string	greater than (type 2 comparison)
<=	numeric string	less than or equal to (type 2 comparison)
>=	numeric string	greater than or equal (type 2 comparison)
<>	numeric string	not equal to (type 3 comparison)
&	string	concatenation
&&	bitwise	AND
| |	bitwise	OR
^^	bitwise	XOR
~	bitwise	NOT
OR	logical	OR
AND	logical	AND
XOR	logical	XOR
NOT	logical	NOT
MOD	integer	modulus
DIV	integer	divide
INSTR	string	type 1 string comparison
^	floating	raise to the power
-	floating	unary minus
+	floating	unary plus

FOR SERVICE MANUALS
CONTACT:
MAURITRON TECHNICAL SERVICES
www.mauritron.co.uk
TEL: 01844 - 351694
FAX: 01844 - 352554

**almost equal – equal to 1 part in 10^7

If the specified logical operation is true then a value not equal to zero will be returned. If the operation is false then a value of zero will be returned.

precedence

The precedence of SuperBASIC operators is defined in the table above. If the order of evaluation in an expression cannot be deduced from this table then the relevant operations are performed from left to right. The inbuilt precedence of SuperBASIC operators can be overriden by enclosing the relevant sections of the expression in parentheses.

highest unary plus and minus
string concatenation
INSTR
exponentiation
multiply, divide, modulus and integer divide
add and subtract
logical comparison
NOT (bitwise or logical)
AND (bitwise or logical)
lowest OR and XOR (bitwise or logical)

peripheral expansion

The expansion connector allows extra peripherals to be plugged into the QL. The connections available at the connector are:

GND	a	1	b	GND
D3	a	2	b	D2
D4	a	3	b	D1
D5	a	4	b	D0
D6	a	5	b	ASL
D7	a	6	b	DSL
A19	a	7	b	RDWL
A18	a	8	b	DTACKL
A17	a	9	b	BGL
A16	a	10	b	BRL
CLKCPU	a	11	b	A15
RED	a	12	b	RESETCPUL
A14	a	13	b	CSYNCL
A13	a	14	b	E
A12	a	15	b	VSYNCH
A11	a	16	b	VPAL
A10	a	17	b	GREEN
A9	a	18	b	BLUE
A8	a	19	b	FC2
A7	a	20	b	FC1
A6	a	21	b	FC0
A5	a	22	b	A0
A4	a	23	b	ROMOEH
A3	a	24	b	A1
DBGL	a	25	b	A2
SP2	a	26	b	SP3
DSMCL	a	27	b	IPLOL
SP1	a	28	b	BERRL
SP0	a	29	b	IPL1L
VP12	a	30	b	EXTINTL
VM12	a	31	b	VIN
VIN	a	32	b	VIN

The connector on the QL is a 64-way (male) DIN-41612 indirect edge connector

An 'L' appdended to a signal name indicates that the signal is active low.

Signal	Function
A0..A19	68008 address lines
RDWL	Read / Write
ASL	Address Strobe
DSL	Data Strobe
BGL	Bus Grant
DSMCL	Data Strobe - Master Chip
CLKCPU	CPU Clock
E	6800 peripherals clock
RED	Red
BLUE	Blue
GREEN	Green
CSYNCL	Composite Sync
VSYNCH	Vertical Sync
ROMOEH	ROM Output Enable
FC0	Processor Status
FC1	Processor Status
FC2	Processor Status
RESETCPUL	Reset CPU

QL Peripheral Output Signals

Signal	Function
DTACKL	Data acknowledge
BRL	Bus request
VPAL	Valid Peripheral Address
IPL0L	Interrupt Priority Level 5
IPL1L	Interrupt Priority Level 2
BERRL	Bus Error
EXTINTL	External Interrupt
DBGL	Data bus grab

QL Peripheral Input Signals

Signal	Function
D0..D7	Data Lines

QL Peripheral Bi-directional Signals

Signal	Function
SP0..SP3	Select peripheral 0 to 3
VIN	9V DC (nominal) - 500mA maximum
VM12	−12V
VP12	+12V
GND	ground

Miscellaneous

It is not intended that the following description of the QL peripheral expansion mechanism be sufficient to implement an actual expansion device, but rather be read to gain a basic understanding of the expansion mechanism.

Single or multiple peripherals may be added to the QL up to a maximum of 16 devices. A single peripheral can be plugged directly into the QL Expansion Slot while multiple peripherals must be plugged into the QL Expansion Module, which in turn is plugged into the QL Expansion Slot via a buffer card.

In this context the term device also includes expansion memory. Although the areas of the QL memory map allocated to expansion memory are different from those allocated to expansion devices, the basic mechanism is the same. Only one expansion memory peripheral can be plugged into the QL at any one time. The address space allocated for peripheral expansion in the QL Physical memory map allows 16 Kbytes per peripheral. This area must contain the memory mapped I/O required for the driver and the code for the driver itself.

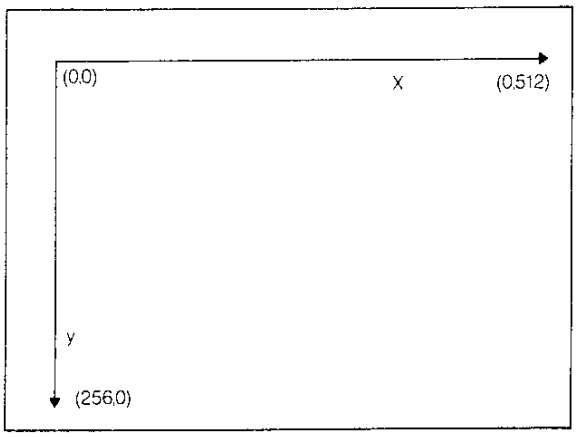
Qdos includes facilities for queue management and simple serial I/O which may be of use when writing device drivers.

The position of each peripheral device in the overall memory map of the QL is determined by the select peripheral lines: SP0, SP1, SP2 and SP3. These select lines generate a signal corresponding to the slot position in the QL expansion module, thus for a device to be selected the address input from address lines: A14, A15, A16 and A17 must be the same as the signals from SP0, SP1, SP2 and SP3 respectively.

pixel coordinate system

The pixel coordinate system is used to define the positions and sizes of *windows*, *blocks* and *cursor* positions on the QL screen. The coordinate system has its origin in the top left hand corner of the default window (or screen) and always assumes that positions are specified as though the screen were in *512 mode* (high resolution mode). The system will use the nearest pixel available for the particular mode set making the coordinate system independent of the screen mode in use.

Some commands are always relative to the default window origin, e.g. **WINDOW**, while some are always relative to the current window origin, e.g. **BLOCK**.



The Pixel Coordinate System

program

A SuperBASIC program consists of a sequence of SuperBASIC *statements*, where each statement is preceded by a *line number*. Line numbers are in the range of 1 to 32767.

Command	Function
RUN	start a loaded program
LRUN	load a program from a device and start it
CTRL SPACE	force a program to stop

syntax: *line_number* = *{*digit*}* {range 1..32767}

*{*line_number statement *{*:statement*}**}*

example: i.

```
100 PRINT "This is a valid line number"
RUN
```

ii.

```
100 REM a small program
110 FOR foreground = 0 TO 7
120   FOR contrast = 0 TO 7
130     FOR stipple = 0 TO 3
140       PAPER foreground, contrast, stipple
150       CURSOR 0,70
160       FOR n = 0 TO 2
170         SCROLL 2,1
180         SCROLL -2, 2
190       END FOR n
200     END FOR stipple
210   END FOR contrast
220 END FOR foreground
RUN
```

Qdos

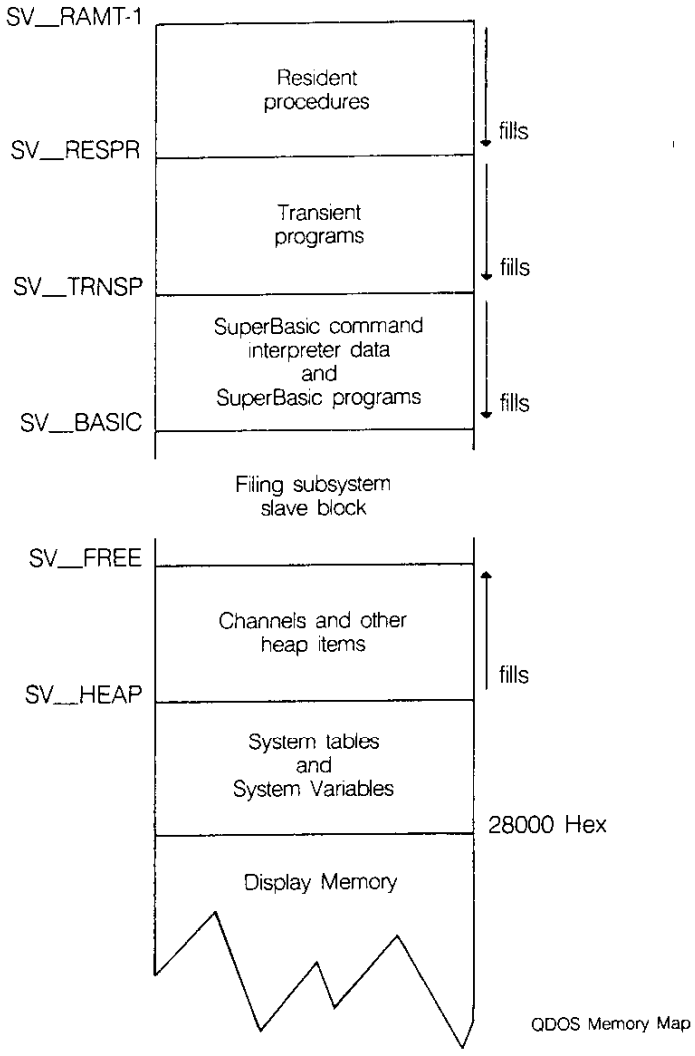
Qdos is the QL Operating System and supervises:

Task Scheduling and resource allocation
Screen I/O (including windowing)
Microdrive I/O
Network and serial channel communication
Keyboard input
Memory management

memory map

A full description of Qdos is beyond the scope of this guide but a brief description is included.

The system RAM has an organisation imposed by the Qdos operating system and is defined as follows:



QDOS Memory Map

The terms SV__RAMT, SV__RESPR, SV__TRNSP, SV__BASIC, SV__FREE, SV__HEAP are used to represent addresses inside the QL. These terms are not recognised by SuperBASIC or the Qdos operating system. Furthermore the addresses represented are liable to change as the system is running.

sv__ramt RAM Top
This will vary according to the memory expansion boards attached to the system.

sv__respr Resident Procedures
Resident procedures are loaded into the top of RAM. Space can be allocated in the resident procedure area using the RESPR function, but this space cannot be released except by resetting the QL. Resident procedures written in machine code can be added to the SuperBASIC name list and so become extensions to the SuperBASIC system.

sv__trnsp — Transient Programs

Transient programs are loaded immediately below the resident procedures. Each program must be self contained, i.e. it must contain space for its own data and its own stack. It must be position independent or must be loaded by a specially written linking loader. A transient program is executed from BASIC by using the **EXEC** command or from Qdos by activating it as a *job*.

The transient program area may be used for storing data only but this data will still be treated by Qdos as a job and therefore must not be activated.

sv__basic — SuperBASIC Area

This area contains all loaded SuperBASIC programs and related data. This area expands and contracts using up the free space as required.

sv__free — Free Space

Free space is used by the Qdos file subsystem to create Microdrive Slave Blocks, i.e. copies of Microdrive blocks which can be held in RAM.

sv__heap — System Heap

This is used by the system to store data channel definitions etc. and also provides working storage for the I/O subsystem. Transient programs may allocate working space for themselves on the heap via Qdos system calls.

System Tables / System Variables

This area is directly above the screen memory. The System Tables and supervisor stack are resident above the system variables.

system calls

System calls are processed by Qdos in **supervisor mode**. When in supervisor mode Qdos will not allow any other job to take over the processor. System calls processed in this way are said to be **atomic**, i.e. the system call will process to completion before relinquishing the processor. Some system calls are only **partially atomic**, i.e. once they have completed their primary function they will relinquish the processor if necessary. Unless specifically requested all the I/O system calls are partially atomic.

The standard mechanism for making a system call is by making a trap to one of the Qdos system vectors with appropriate parameters in the processor registers. The action taken by Qdos following a system call is dependent on the particular call and the overall state of the system at the time the call was made.

input/output

Qdos supports a multitasking environment and therefore a file can be accessed by more than one process at a time. The Qdos filing sub-system can handle files which have been opened as **exclusive** files or as **shared** files. A shared file can not be written to. QL devices are processed by the **serial I/O sub-system**. The filing sub-system and the serial I/O sub-system together make up the **redirectable I/O system**. As its name suggests any data output via this system can be redirected to any other device also supported by the redirectable I/O system.

The device names required by Qdos are the same as the device names required by SuperBASIC and are discussed in the concept section *devices*. The collection of standard devices supplied with the QL can be expanded.

devices

The standard devices included in the system are discussed in this guide in the section **devices**. Further devices may be added to the system, given a name (e.g. SER1, NET) and then accessed in the same way as any other QL device.

multitasking

Jobs will be allowed a share of the CPU in line with their priority and competition with other jobs in the system. Jobs running under the control of Qdos can be in one of three states:

active: Capable of running and sharing system resources. A job in this state may not be running continuously but will obtain a share of the CPU in line with its priority.

suspended: The job is capable of running but is waiting for another job or I/O. A job may be suspended indefinitely or for a specific period of time.

inactive: The job is incapable of running, its priority is 0 and so it can never obtain a share of the CPU.

Qdos will reschedule the system automatically at a rate related to the 50 Hz frame rate. The system will also be rescheduled after certain system calls.

example: This program generates an on-screen readout of the real-time clock, running as an independent job.

First **RUN** this program with a formatted cartridge in microdrive 2. This generates a machine code title called "clock". Wait for the Microdrive to stop. Next, set the clock using the **SDATE** command.

Then type:

```
EXEC mdv2_clock
```

and a continuous time display will appear at the top right of the command window.

```
100  c=RESPR(100)
110  FOR i=0 TO 68 STEP 2
120    READ x:POKE_W i+c,x
130  END FOR i
140  SEXEC mdv2_clock,c,100,256
1000 DATA 29439,29697,28683,20033,17402
1010 DATA 48,13944,200,20115,12040
1020 DATA 28691,20033,17402,74,-27698
1030 DATA 13944,236,20115,8279,-11314
1040 DATA 13944,208,20115,16961,16962
1050 DATA 30463,28688,20035,24794
1060 DATA 0,7,240,10,272,200
```

N.B. Line 1060 governs the position and colour of the clock window – the data items are, in order:

border colour/width, paper/ink colour, window width, height, x-origin, y-origin

These are pairs of bytes, entered by **POKE_W** as words.

The x-origin and the y-origin (the last data item) should be 272 and 202 in monitor mode, or 240 and 216 in TV mode.

Generate the paper and ink word, for example, as 256 * paper + ink. Thus white paper, red ink is 256 * 7 + 2 = 1794.

repetition

Repetition in SuperBASIC is controlled by two basic program constructs. Each construct must be identified to SuperBASIC:

```
REPEAT identifier                FOR identifier = range
  statements                       statements
END REPEAT identifier            END FOR identifier
```

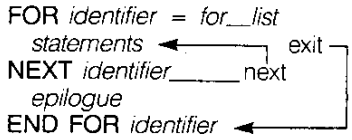
These two constructs are used in conjunction with two other SuperBASIC statements:

```
NEXT identifier                  EXIT identifier
```

Processing a **NEXT** statement will either pass control to the statement following the appropriate **FOR** or **REPeat** statement, or if a **FOR** range has been exhausted, to the statement following the **NEXT**.

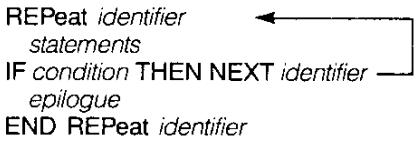
Processing an **EXIT** will pass control to the statement after the **END FOR** or **END REPeat** selected by the **EXIT** statement. **EXIT** can be used to exit through many levels of nested repeat structures. **EXIT** should always be used in **REPeat** loops to terminate the loop on some condition.

A combination of **NEXT**, **EXIT** and **END** statements allows **FOR** and **REPeat** loops to have a **loop epilogue** added. A loop epilogue is a series of SuperBASIC statements which are executed on some special condition arising within the loop:



The loop epilogue is only processed if the **FOR** loop terminates normally. If the loop terminates via an **EXIT** statement then processing will continue at the **END FOR** and the epilogue will not be processed.

It is possible to have a similar construction in a **REPeat** loop:



This time entry into the loop epilogue is controlled by the **IF** statement. The epilogue will or will not be processed depending on the condition in the **IF** statement. A **SELect** statement can also be used to control entry into the epilogue.

ROM cartridge slot

Allows software to be used in the QL system from a Sinclair QL ROM cartridge. The ROM cartridge can contain software to directly change the behaviour of the SuperBASIC system. The cartridge can contain:

i. Software to be used instead of or with the SuperBASIC system. For example:

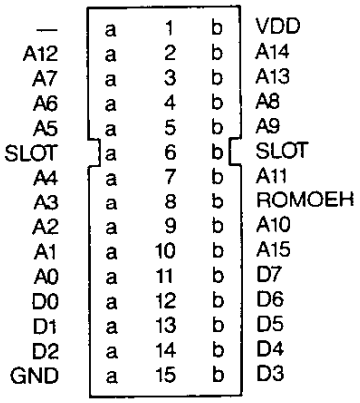
assemblers
compilers
debuggers
application software
etc.

ii. Software to expand the SuperBASIC system. For example:

special procedures
etc.

It is not possible to use ZX ROM cartridges on the QL.

pin out



Side b is the upper side of the connector; side a is the lower.

Signal	Function
AO..A15	Address lines
DO..D8	Data lines
ROMOEH	ROM Output Enable
VDD	5V
GND	Ground

warning

Never plug or unplug a ROM cartridge while the QL power is on.

screen

512 mode

The screen is 512 pixels across and 256 pixels deep. Only the solid colours:

black
red
green
white

can be displayed in this mode.

Low resolution mode also has a hardware flash.

256 mode

The screen is 256 pixels across and 256 pixels deep. The full set of solid colours is available in this mode:

blue
red
magenta
green
cyan
yellow
white

warning

A domestic television is not capable of displaying the complete QL screen. Portions of the screen at the top and the sides will not be reproduced. The default initial window will take account of this and will reduce the effective picture size. The full size can be restored with the **WINDOW** command.

Command	Function
MODE	set screen mode

slicing

Under certain circumstances it is possible to refer to more than one element in an array i.e. slice the array. The array slice can be thought of as defining a **subarray** or a series of subarrays to SuperBASIC. Each slice can define a continuous sequence of elements belonging to a particular dimension of the original array. The term array in this context can include a numeric array, a string array or a simple string.

It is not necessary to specify an index for the full number of dimensions of an array. If a dimension is omitted then slices are added which will select the full range of elements for that particular dimension, i.e. the slice (0 TO). SuperBASIC can only add slices to the end of a list of array indices.

syntax:

index:=	| *numeric_exp*	{single element}
	| *numeric_exp* TO *numeric_exp*	{range of elements}
	| *numeric_exp* TO	{range to end}
	| TO *numeric_expression*	{range from beginning}

array_reference:= | *variable*
| *variable* ([*index* *[,*index*] *])

An array slice can be used to specify a source or a destination subarray for an assignment statement.

example:

```
i.    PRINT data_array
ii.   PRINT letters$(1 TO 15)
iii.  PRINT two_d_array (3)(2 TO 4)
```

String slicing is performed in the same way as slicing numeric or string arrays.

Thus

a$(n)	will select the nth character.
a$(n TO m)	will select all characters from the nth to the mth, inclusively.
a$(n TO)	will select from a character n to the end, inclusively.
a$(1 TO m)	will select from the beginning to the nth character, inclusively.
a$	will select the entire contents of a a$

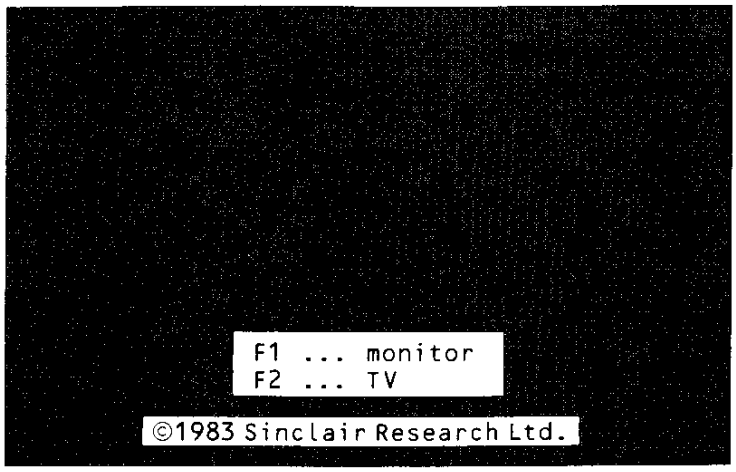
Some forms of BASIC have functions called LEFT$, MID$, RIGHT$. These are not necessary in SuperBASIC. Their equivalents are specified below:

SuperBASIC	Other BASIC
a$(n)	MID$ (a$,n,1)
a$(n TO m)	MID$ (a$,n,m+1-n)
a$(1 TO n)	LEFT$ (a$,n)
a$(n TO)	RIGHT$ (a$,LEN(a$)+1-n)

warning Assigning data to a sliced string array or string variable may not have the desired effect. Assignments made in this way will not update the length of the string. The length of a string array or string variable is only updated when an assignment is made to the whole string.

start up

Immediately after switch on (or reset) the QL will perform a RAM test which will give a spurious pattern on the display. If the RAM test is passed then the screen will be cleared and the copyright screen displayed.



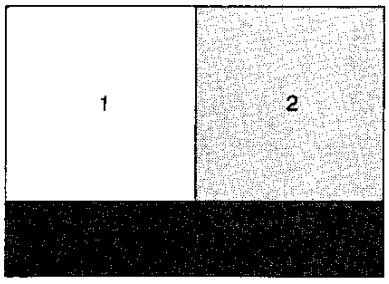
After start-up, the QL displays the copyright message and asks whether it is being used on a television or a monitor. The QL will set different initial screen modes and window sizes depending on the answer.

Press **F1** if you are using a monitor and **F2** if you are using a television set.

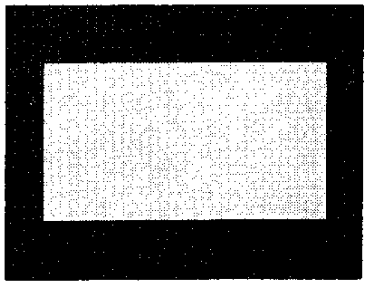
The QL has the ability to 'boot' itself up from programs contained in either the ROM cartridge slot or in Microdrive 2. If the ROM cartridge slot contains a self starting program then start up will continue under the control of the program in the ROM cartridge. If nothing suitable is found then the QL will check Microdrive 1 for a cartridge. If a cartridge is found and if it contains a file called **BOOT** it is loaded and run.

default screen

The QL has three default channels which are linked to three default windows.



Monitor


Television

warning

Channel 0 is used for listing commands and error messages, channel 1 for program and graphics output and channel 2 for program listings. The default channel can be modified using the optional channel specifier in the relevant command.

It is important NOT to switch on the QL with a Microdrive cartridge in position. If booting from a Microdrive cartridge is required then the cartridge must be inserted between switching on and pressing either **F1** or **F2**.

sound

Sound on the QL is generated by the QL's second processor (an 8049) and is controlled by specifying:

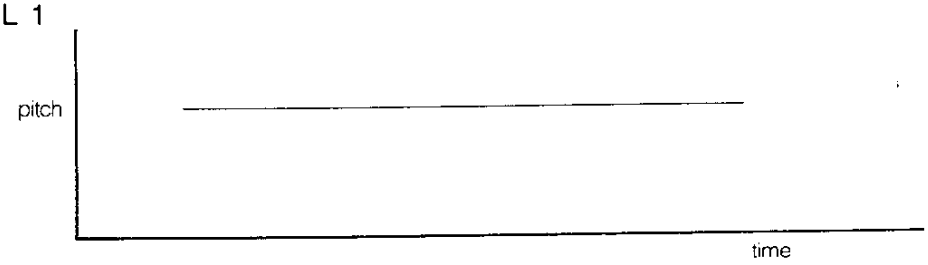
up to two pitches
the rate at which the sound must move between the pitches, the ramp
how the sound is to behave after it has reached one of the specified pitches, the wrap
if any randomness should be built into the sound, i.e. deviations from the ramp
if any fuzziness should be built into the sound. i.e. deviations on every cycle of the sound

Fuzziness tends to result in buzzy sounds while randomness, depending on the other parameters, will result in 'melodic' sounds or noise.

The complexity of the sound can be built up stage by stage gradually building more complex sounds. This is, in fact, the best way to master sound on the QL.

Specify a duration and a single pitch. The specified pitch will be beeped for the specified time.

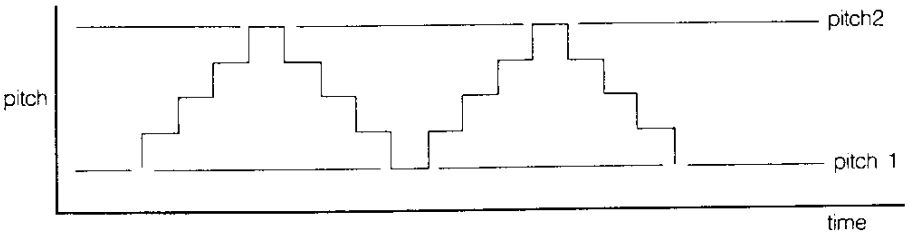
LEVEL 1



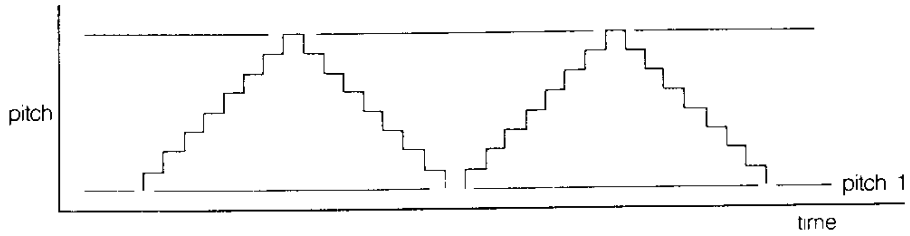
This is the simplest sound command, other than the command to stop the sound, on the QL.

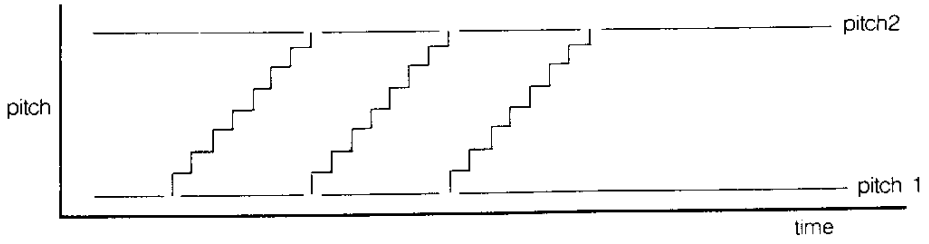
LEVEL 2 A second pitch and a gradient can be added to the command. The sound will then 'bounce' between the two pitches at the rate specified by the gradient.

The sounds produced at this level can vary between: semi musical beeps, growls, zaps and moans. It is best to experiment.



LEVEL 3 A parameter can be added which controls how the sound behaves when it becomes equal to one of the specified pitches. The sound can be made to 'bounce' or 'wrap'. The number of wraps can be specified, including wrap forever. It is even more important to experiment.

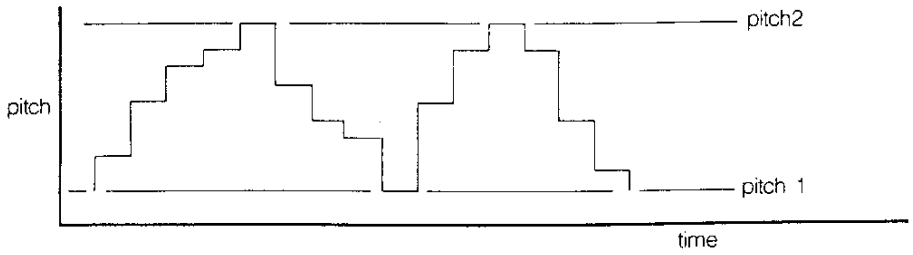




LEVEL 4

Randomness can be added to the sound. This is a deviation from the specified step or gradient.

Depending on the amount of randomness added in relation to the pitches and the gradient, it will generate a very wide and unexpected range of sounds.



LEVEL 5

More variation can be added by specifying 'fuzziness'. Fuzziness adds a random factor to the pitch continuously. Fuzziness tends to make the sound buzz.

Combining all of the above effects can make a very wide range of sounds, many of them unexpected. QL sound is best explored through experiment. By specifying a time interval of zero the sound can be made to repeat forever and so a sequence of BEEP commands can be used until the sound generated is the sound which is required. A word of warning: slight changes in the value of a single parameter can have alarming results on the sound generated.

statement

A SuperBASIC statement is an instruction to the QL to perform a specific operation, for example:

```
LET a = 2
```

will assign the value 2 to the variable identified by a.

More than one statement can be written on a single line by separating the individual statements from each other by a colon (:), for example:

```
LET a = a + 2 : PRINT a
```

will add 2 to the value identified by the variable a and will store the result back in a. The answer will then be printed out.

If a line is not preceded by a *line number* then the line is a *direct command* and SuperBASIC processes the statement immediately. If the statement is preceded by a line number then the statement becomes part of a SuperBASIC *program* and is added into the SuperBASIC program area for later execution.

Certain SuperBASIC statements can have an effect on the other statements over the rest of the logical line in which they appear i.e. IF, FOR, REPeat, REM, etc. It is meaningless to use certain SuperBASIC statements as direct commands.

string arrays
string variables

String arrays and numeric arrays are essentially the same, however there are slight differences in treatment by SuperBASIC. The last dimension of a string array defines the maximum length of the strings within the array. String variables can be any length up to 32766. Both string arrays and string variables can be *sliced*.

String lengths on either side of a string assignment need not be equal. If the sizes are not the same then either the right hand string is truncated to fit or the length of the left hand string is reduced to match. If an assignment is made to a sliced string then if necessary the 'hole' defined by the slice will be padded with spaces.

It is not necessary to specify the final dimension of a string array. Not specifying the dimension selects the whole string while specifying a single element will pick out a single character and specifying a slice will define a sub string.

comment

Unlike many BASICs SuperBASIC does not treat string arrays as fixed length strings. If the data stored in a string array is less than the maximum size of the string array then the length of the string is reduced.

warning

Assigning data to a sliced string array or string variable may not have the desired effect. Assignments made in this way will not update the length of the string and so it is possible that the system will not recognise the assignment. The length of a string array or a string variable is only updated when an assignment is made to the whole string.

Command	Function
FILL$	generate a string
LEN$	find the length of a string

FOR SERVICE MANUALS
CONTACT:
MAURITRON TECHNICAL SERVICES
www.mauritron.co.uk
TEL: 01844 - 351694
FAX: 01844 - 352554

string comparison

order

. (decimal point/full stop)
digits or numbers in numerical order
AaBbCcDdEeFfGgHhIiJjKkLlMmNnOoPpQqRrSsTtUuVvWwXxYyZz
space ! " # $ % & ' ( )* + , - . / : ; < = > ? @ [|] ^ _ / { | }~©
other non printing characters

The relationship of one string to another may be:

equal: All characters or numbers are the same or equivalent

lesser: The first part of the string, which is different from the corresponding character in the second string, is before it in the defined order.

greater: The first part of the first string which is different from the corresponding character in the second string, is after it in the defined order.

Note that a '.' may be treated as a decimal point in the case of string comparison which sorts numbers (such as SuperBASIC comparisons). Note also that comparison of strings containing non-printable characters may give unexpected results.

types of comparison

type 0 case dependent - character by character comparison

type 1 case independent - character by character

type 2 case dependent - numbers are sorted in numerical order

type 3 case independent - numbers are sorted in numerical order

usage

type 0 not normally used by the SuperBASIC system.

type 1 File and variable comparison

type 2 SuperBASIC <,< =,=,> =,> INSTR and < >

type 3 SuperBASIC == (equivalence)

syntax definitions

SuperBASIC syntax is defined using a non-rigorous 'meta language' type notation. Four types of construction are used:

| | Select one of
[ ] Enclosed item(s) are optional
** Enclosed items are repeated

.. Range

{ } Comment

e.g. | A | B | A or B
[A] A is optional
* A * A is repeated
A..Z A, B, C, etc
{this is a comment}

Consider a SuperBASIC identifier:

A sequence of numbers, digits, underscores, starting with a letter and finishing with an optional % or $

letter:= | A..Z
| a..z

{a letter is one of: ABCDEFGHIJKLMNOPQRSTUVWXYZ}
or abcdefghijklmnopqrstuvwxyz

digit:= | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 |

{a digit is 0 or 1 or 2 or 3 or 4 or 5 or 6 or 7 or 8 or 9}

underscore:= _

{an underscore is _}

identifier = *letter* *[*letter* | *digit* | *underscore*]* | % | $ |

must start with a letter

a sequence of letters digits and underscores i.e. repeat something which is optional

windows

Windows are areas of the *screen* which behave, in most respects, as though each individual window was a screen in its own right, i.e. the window will scroll when it has become filled by text, it can be cleared with the **CLS** command, etc.

Windows can be specified and linked to a channel when the channel is *opened*. The current window shape can be changed with the **WINDOW** command and a border added to a window with the **BORDER** command. Output can be directed to a window by printing to the relevant channel. Input can be directed to have come from a particular window by inputting from the relevant *channel*. If more than one channel is ready for input then input can be switched between the ready channels by pressing

CTRL C

The cursor will flash in the selected window.

Windows can be used for graphics and non-graphic output at the same time. The non graphic output is relative to the current cursor position which can be positioned anywhere within the specified window with the **CURSOR** command and at any line-column boundary with the **AT** command. The graphics output is relative to a graphics cursor which can be positioned and manipulated with the *graphics* procedures.

parts

Certain commands (**CLS**, **PAN** etc.) will accept an optional parameter to define part of the current window for their operation. This parameter is as defined below:

part	description
0	whole screen
1	above and excluding cursor line
2	bottom of screen excluding cursor line
3	whole of cursor line
4	line right of and including cursor

Command	Function
WINDOW	re-define a window
BORDER	take a border from a window
PAPER	define the paper colour for a window
INK	define the ink colour for a window
STRIP	define a strip colour for a window
PAN	pan a window's contents
SCROLL	scroll a window's contents
AT	position the print position
CLS	clear a window
CSIZE	set character size
FLASH	character flash
RECOL	recolour a window

turtle graphics

uperBASIC has a set of turtle graphics commands:

ommand	Function
ENUP	stop drawing
ENDOWN	start drawing
OVE	move the turtle
URN	turn the turtle
URNTO	turn to a specific heading

e set of commands is the minimum and normally would be used within another cedure to expand on the commands. For example:

```
100 DEFine PROCedure forward(distance)
110   MOVE distance
120 END DEFine
130 DEFine PROCedure backwards(distance)
140   MOVE -distance
150 END DEFine
160 DEFine PROCedure left(angle)
170   TURN angle
180 END DEFine
190 DEFine PROCedure right(angle)
200   TURN -angle
210 END DEFine
```

ese will define some of the more famous turtle graphic commands.

ally the turtle's pen is up and the turtle is pointing at 0°, which is to the right-hand e of the window.

e FILL command will also work with figures drawn with turtle graphics. Also ordinary phics and turtle graphics can be mixed, although the direction of the turtle is not dified by the ordinary graphics commands.

A

Arrays 1
  slicing 46
  strings 52
  storage 1

B

BASIC 2
Baud rates 13
BEEP 48
Monochrome monitor 33
Booting 50
Break 3

C

Cartridges
  ROM 44
  Microdrive 31
Channels 4
Character set 5
Circles 24
Clock 10
Coercion 11
Close channels 4
Commands
  keywords 28
  direct 18
  turtle graphics 55
  windows 56
  screen 45
Codes
  characters 5
  colour 12
Communications 13
  channels 4
  devices 16
  networking 34
Comparisons 53
Console device 16
Control characters 5
Conversion 11
Coordinates
  graphics 24
  pixel 36
CTS 13
Cursor 24

D

Data
  structures 1
  types 15
Data storage
  Microdrives 31
  arrays 1
date 10
DCE 13
DEFine FuNction 23
Defaults
DEFine PROCedure 23
Devices
  console (con) 16
I/O
  Microdrives (mdv) 16,31
  network (net) 16,34
  peripherals 36
  screen (scr) 16
  serial (ser) 13,16
  channels 4
  file types 22
Dimension 1
Direct command 18
DTE 13

E

Elements 1
Error handling 19
EXIT 43
Expansion
  ROM cartridge 44
  peripherals 36
Expressions 21

F

Files 22
Filename 16,26
Filling shapes 24
Floating point 15
FOR 43
Functions 23

G

Graphics 24
  turtle 55

H

Handshaking 13
Hex codes 5
High resolution monitor 33

I

Identifiers 26
Initialisation 50
Input
  channels 4
  devices 16
  windows 56
Integers 15
I/O
  devices 16
  monitor 33
  peripherals 36
  Qdos 40
  windows 56

J

Joystick 27

K

Keyboard conventions 5
Keywords 28

L

Lines ... 24
Line numbering ... 39,51
  direct commands ... 18
Local variables ... 23
Loops ... 43

M

Maths functions ... 29
Memory
  map ... 30
  expansion ... 36
Microdrives ... 31
Modes ... 45
Monitor ... 33
Multitasking ... 41

N

Name ... 26
NET ... 34
Network ... 34
NEXT ... 43

O

OPEN ... 4
Operators ... 35
Operating system ... 40
Output
  monitor ... 33
  channels ... 4
Ordering
  coercion ... 11
  precedence ... 35

P

Parameters ... 23
Peripheral expansion ... 36
Pictures ... 24
Pixel coordinates ... 38
Points ... 24
Power up ... 50
Precedence ... 35
Procedure ... 23
Programs ... 39

Q

Qdos ... 40

R

RAM ... 30
Repetition ... 43
RGB ... 33
ROM ... 30
RS-232-C ... 13
RTS ... 13
RXD ... 13

S

Scaling ... 24
Scheduler ... 39
Screen ... 45
  con ... 16
  scr ... 16
  windows ... 56
  colours ... 12
  modes ... 45
Serial communications ... 13
Signals ... 13
Slicing ... 46
Sound ... 48
Start up ... 50
Statements ... 51
Stipples ... 12
Strings
  variables ... 52
  slicing ... 46
  arrays ... 52
  comparisons ... 53
Switching on ... 50
Syntax definitions ... 54

T

Time ... 10
Trig functions ... 29
Turtle graphics ... 55
TXD ... 13
Type conversion ... 11

V

Variables ... 15
  local ... 23
  string ... 15

W

Windows ... 56

sinclair

QL

QL Quill

©1984 PSION LIMITED
by Dick de Grandis-Harrison *(Psion Limited)*

CHAPTER 1
ABOUT QL QUILL

sticated wordprocessor. It has been designed to give you the maximum bility, yet is still easy to learn and to use. As you will see later, you will formed as to what you can do next and how to do it.

rdprocessor in any circumstance where you would otherwise use a machines are very similar in function, although the wordprocessor ages that are not matched by a conventional typewriter. Perhaps the erence is the ease with which mistakes can be corrected. Since the mmediately as you type it in, you can make as many corrections as only print the text when you are sure that it is exactly what you want be assured of perfect results every time.

working through this manual, there are a number of other advantages. using a typewriter, it is necessary to press the carriage return key ine. In Quill, this function is performed automatically. Whenever the es the end of a line, a new line is started; you press **ENTER** when new paragraph. Whenever a new line is started you will notice that text in the last line will be adjusted so that the left and right margins ghout the text. This process, which is known as *justification*, gives a appearance to the final result, without any effort on your part. Like es of Quill, the form of justification that you use can be modified, requirements.

are not sure what to do, remember that you can ask for Help by remember that you can cancel any partially completed operation (e.g. by pressing **ESC**.

CHAPTER 2

GETTING STARTED

LOADING QL QUILL

Load QL Quill as described in the QL Program Introduction. When loaded the following message will be displayed:

```
LOADING QL QUILL

version x.xx

Copyright © 1984 PSION SYSTEMS

wordprocessor
```

where x.xx is the version number, e.g. 2.00.

Quill will only need to access the cartridge in Microdrive 1 whenever you print a document or ask for Help.

When a document is being entered Quill will only require a cartridge in Microdrive 2 when the text is longer than about three pages. Quill will ask for a cartridge when necessary. Once the cartridge is inserted it should not be removed until the document is saved or abandoned.

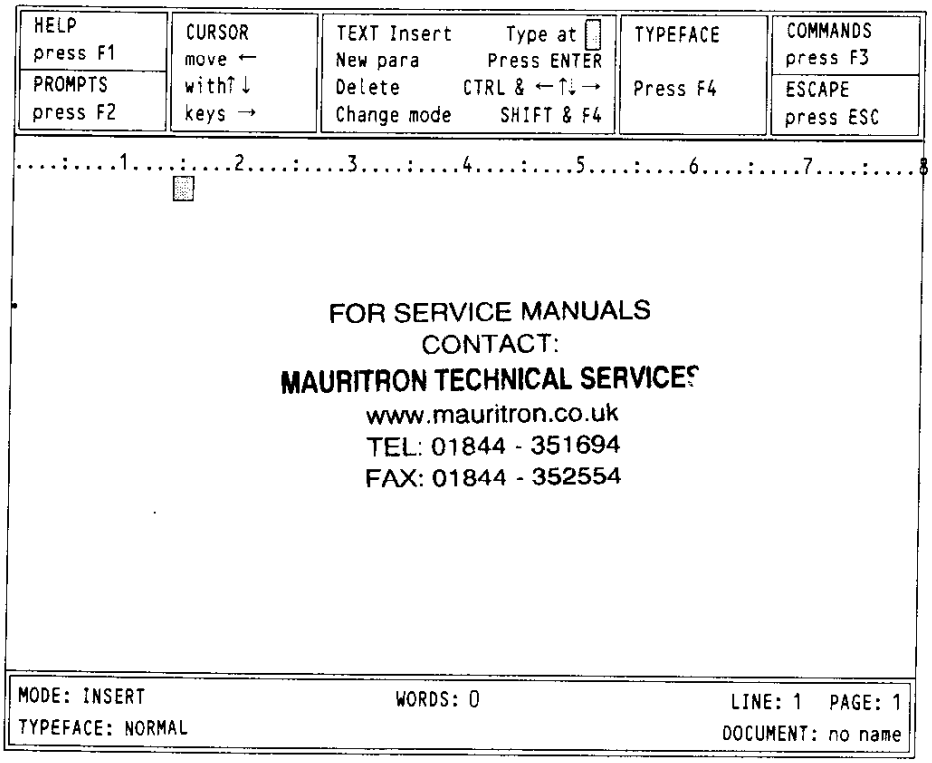


Figure 2.1 The main display with a monitor (80 characters)

GENERAL APPEARANCE

Initially the Quill display should look like that shown in Figure 2.1 or Figure 2.2. This is known as the *main display*.

Quill can show 80, 64 or 40 characters per line of the screen. If you are using a domestic television the display may not be clear enough for you to see 80 characters per line. If this is the case you will need to use 64 or 40 characters. The 64 character screen is very similar to that for 80 characters. The 40 character screen is arranged differently and the main display will look like Figure 2.2.

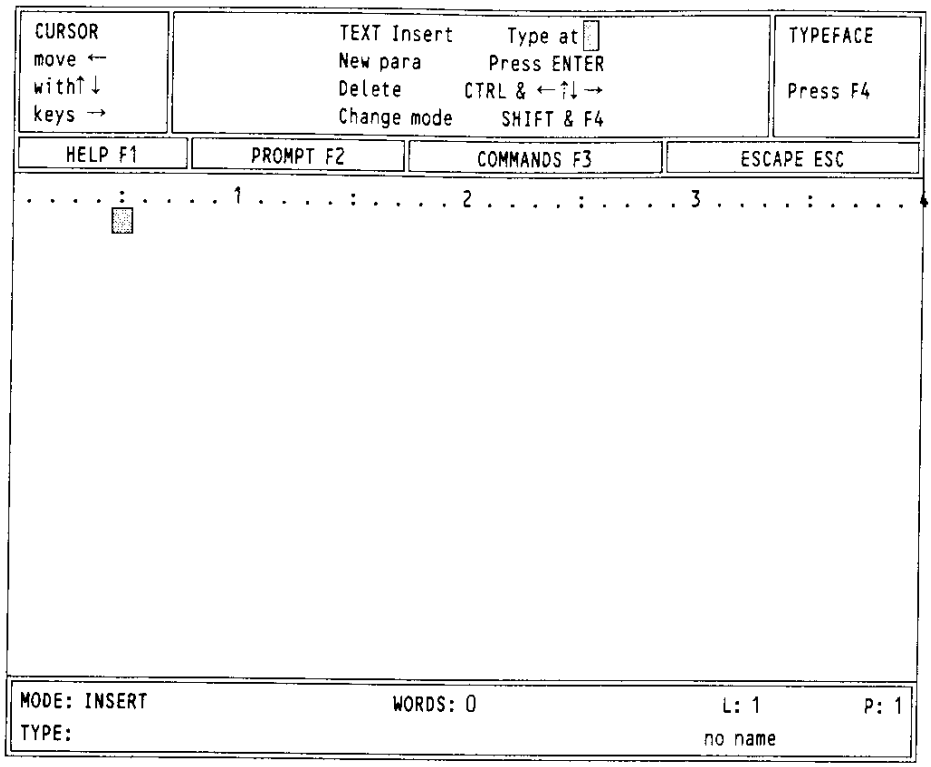


Figure 2.2 The main display with 40 characters

Quill initially selects either an 80 or a 64 character display – depending on whether you pressed **F1** or **F2** when you switched on the computer. You can change from one form of display to another at any time with the Design command which is described later.

Apart from the difference in appearance, Quill works in exactly the same way with all three forms of display. Most of the diagrams in this manual are shown for the 80 character display.

The screen is divided into three main sections: the *display area*, the *status area* and the *control area*.

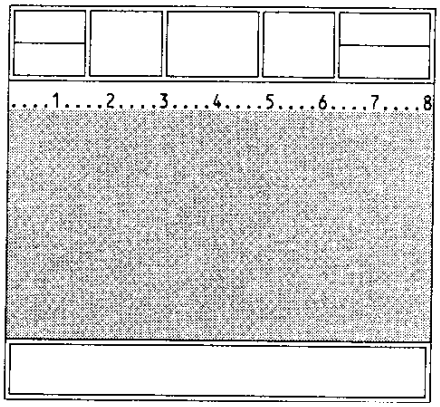


Figure 2.3 The display area

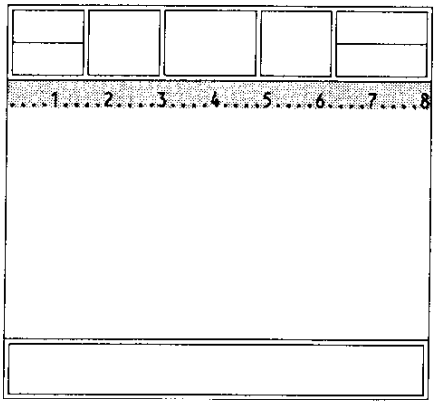


Figure 2.4 The ruler

The Display Area

The largest area, in the centre of the screen, is reserved for the text of your document. Almost everything that you type at the keyboard will appear in this area.

Across the top of the display area is the *ruler*. This is a row of dots, marking each character space across the screen. Every fifth space on it is marked with a colon (:) and every tenth space is numbered.

The Status Area

The *status area*, which uses the bottom three lines of the screen, shows information about your current document. For example, it normally shows its name. Initially you will not have given a name to a document and Quill shows "no name". Quill will show this for any text that you type, until you give the document a name.

The status area also shows that Quill is currently in *insert mode*, which means that anything you type into your document will be inserted (as opposed to writing over any following text). It also shows that there is no special *typeface*, i.e. that you are using a normal typeface. Bold (emphasised), underlined, subscript and superscript typefaces are also available and we shall see how to use them later on.

In addition, the status area shows the number of words in the current document, and the line and page number of the position of the cursor. It initially shows that you are at line 1 of page 1 of a document which contains no words.

The status area is also used for showing any special text that is typed in during the use of *commands* (the set of instructions available when you press **F3**). For example, the section of text being searched for when you use the Search command (see Chapter 5) will appear in the status area.

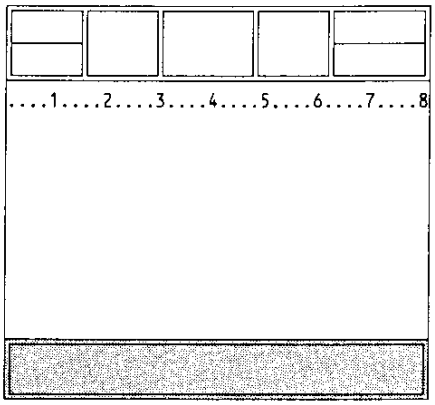


Figure 2.5 The status area

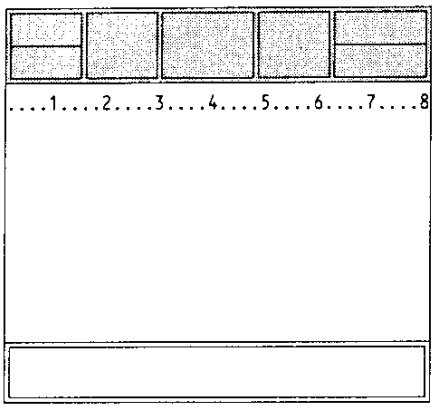


Figure 2.6 The control area

The Control Area

The control area occupies the top few lines of the screen. It shows the normal options to obtain Help (**F1**), to turn the prompts on and off (**F2**), to select a command (**F3**) and to cancel any incomplete operation (**ESC**). In addition there are three options that are specific to Quill. These are displayed in the three central boxes of the control area and are:

Cursor	– move the cursor
Text	– add or remove text
Typeface	– change the typeface

THE CURSOR

On the top line of the central display area you will see a small rectangle. This is known as a *cursor* and marks the position where the text you type will be placed.

The control area shows that you can move the cursor around the text area by use of the four cursor keys on the keyboard. When you have some text in your document, each time that you press one of these keys, the cursor will move by one space in the direction indicated by the arrow. The cursor will not pass the end of the text. If there is no text in your document you will not be able to move the cursor from its original position.

You can also move the cursor around the text in larger steps. If you hold down **SHIFT** and, press the left or right cursor keys the cursor will move left or right by units of one word. When you press **SHIFT** together with the up or down cursor keys the cursor will move backwards or forwards by one paragraph.

TEXT

The option shown at the centre of the control area indicates the various ways in which you can change the text of the document. Simply typing at the keyboard will insert the text at the cursor position.

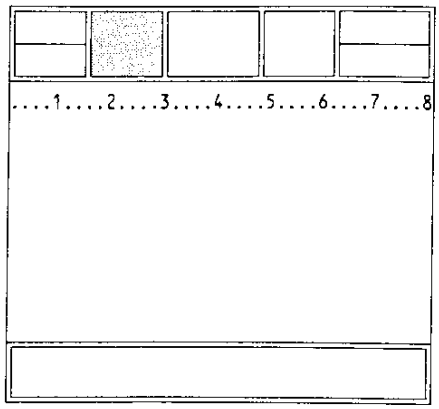


Figure 2.7 Moving the cursor

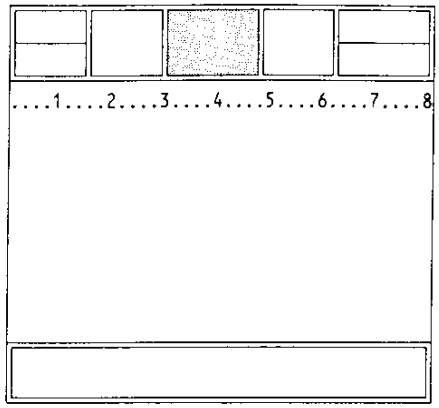


Figure 2.8 The text options

The second line of the Text option shows that pressing **ENTER** is used to mark the start of a new paragraph. You do not need to press **ENTER** when you reach the end of a line of text. If you continue typing in words until you reach the end of the first line, the new words will automatically appear on the second line and the spacing of the words on the first line will be adjusted. This is *justification*, which controls the way the text is aligned with respect to the left and right margins.

Try pressing **ENTER** and then typing in some text to see the effect of starting a new paragraph. Do not worry if the indentation of the new paragraph is not as you wish – you will find how to change it in Chapter 4.

You can include characters in your document which are not shown on the keyboard. They are selected by pressing **CTRL** or **CTRL** and **SHIFT** and another key. The *Concept* reference guide contains a full list of the usable characters together with the relevant keying.

You can delete text, one character at a time, to the left or right of the cursor position. Hold down **CTRL** and press either the left or the right cursor keys.

While you were typing in text you may have noticed some changes taking place in the status area at the bottom of the screen. The word and line counts will always agree with the contents of the document. The remainder of the status area will not have changed. In particular, the document will still be unnamed. You give a name to a document when you save it on a Microdrive cartridge, (as described in Chapter 7).

Now that you have some text in your document, you can try moving the cursor around the text area by use of all four cursor keys. When you have finished, move the cursor to the end of the text.

TYPEFACE

A further option in the control area is headed typeface and is used to modify the appearance of the text in your document.

Press **F4** and you are given five choices:

to use bold (or heavy) type
to display high script (superscript)
to display low script (subscript)
to produce underlined text
to 'paint' existing text.

Any one of these is brought into effect by pressing **F4** and then a single key from the list shown in the control area. As an example let us use this option to produce underlined text. Press **F4**, then the U key.

The display returns to normal and nothing seems to have changed, except that the typeface is marked as 'UNDERLINE' in the status area. If you now type in some more text you will see that it is underlined as it is displayed.

In Quill you see exactly what will appear in the final printed version. The only things that are not always shown on the screen are the upper and lower page margins, and the spacing between lines (when you select double or triple spacing). Quill does not show these since they would reduce the amount of text visible on the screen at any one time.

To turn off underlining, press F4 and then the U key again. If you now type in a few more words you will see that they are not underlined – the underlining option works like a simple on-off switch, or *toggle*.

You will find a fuller description of underlining, and the other three typeface options, in Chapter 4.

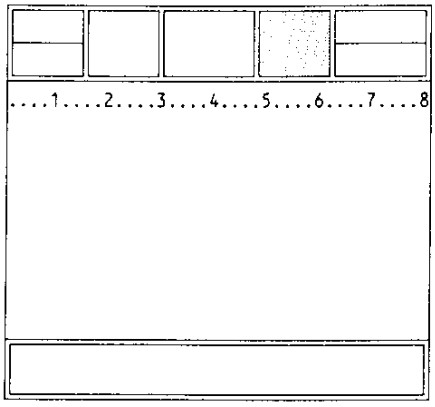


Figure 2.9 Typeface

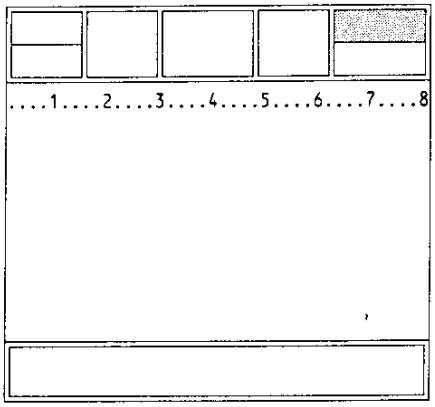


Figure 2.10 The commands

COMMANDS

You select a command by pressing F3. The list of commands in the control area is known as the *command menu*.

You can select any of the commands shown in the menu (list) at the centre of the control area by pressing the key corresponding to its first letter.

Quill has more commands than can be displayed in the command menu. You are therefore given two alternative lists and can switch between them with the Other command.

Since some commands start with the same letter it is important to check that the command you want is displayed in the control area before you select it.

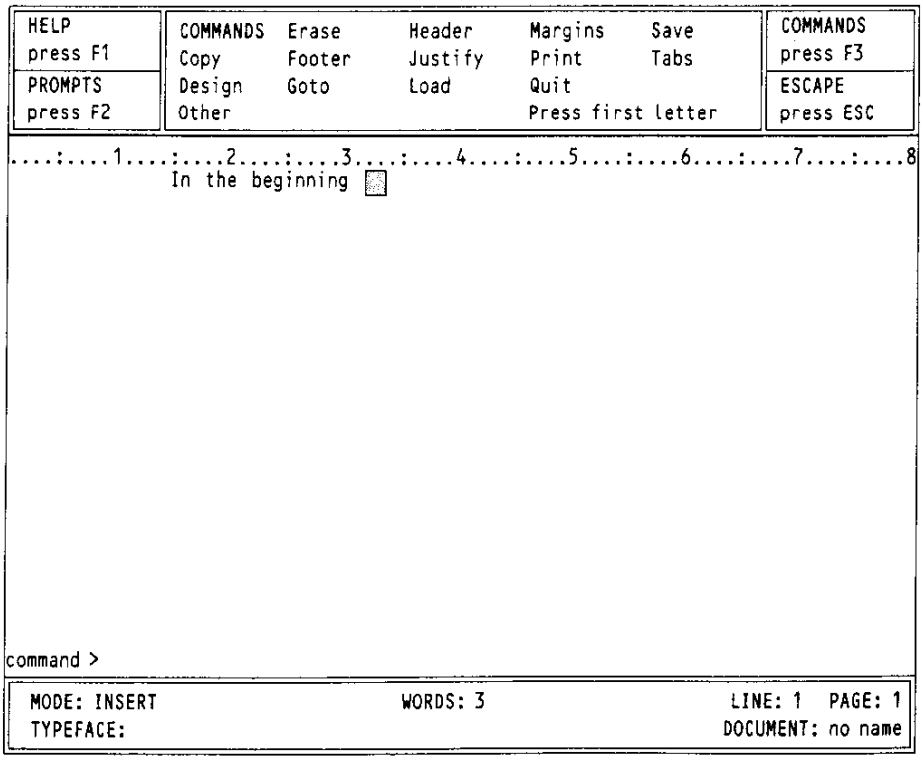


Figure 2.11 the first command menu

The descriptions of the various commands will take up much of the rest of this manual. For the moment we will describe the use of just two commands: Quit and Zap.

Quit is used when you have finished with Quill. Press **F3** and then the Q key to use **quit** and return to SuperBASIC. **Quit** will ask whether you want to save your current document on a Microdrive cartridge before quitting. Press **ENTER** to save it or press A to abandon it.

You can press **ESC** to cancel the command and return to your document.

The **Zap** command is in the commands II menu and so you will have to use the **Other** command before selecting **Zap**. You must press **F3**, O and then Z. **Zap** clears from memory the text of the current document, but does not return to SuperBASIC.

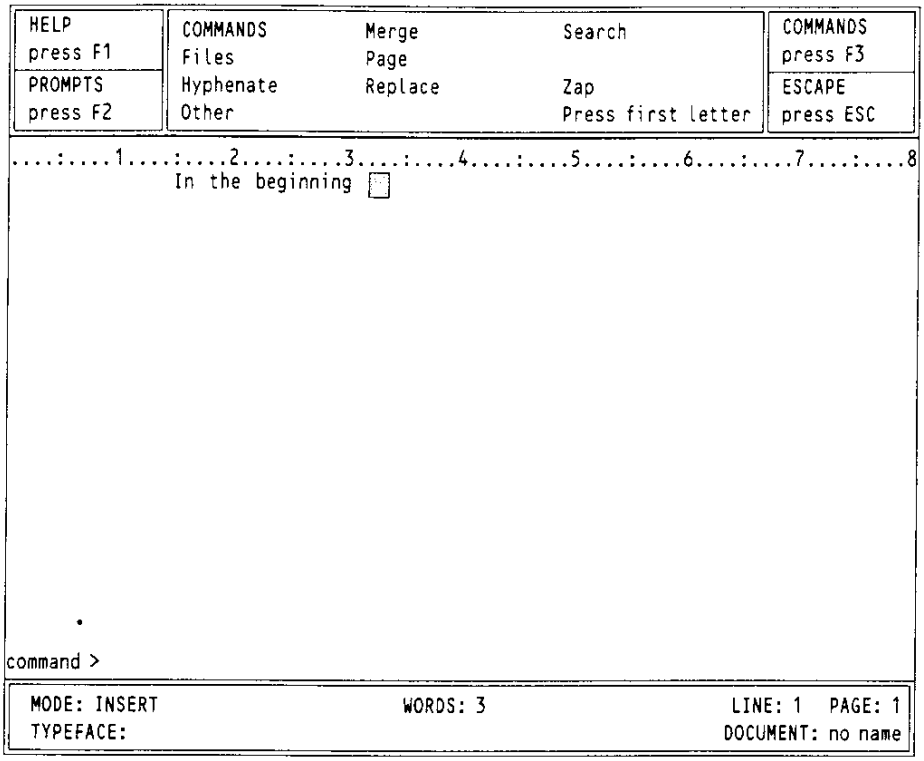


Figure 2.12 The second command menu

If you clear the text before you have saved it on a Microdrive cartridge you will not be able to recover it without typing it in again. Quill will therefore ask you to confirm your choice by pressing **ENTER**. You have the alternative choice of pressing **ESC** to cancel the command and return to your document.

CHAPTER 3
CURSOR EDITING

In this chapter you will learn how to use Quill's simple editing facilities. The changes to the text will always occur at the position of the cursor. You must therefore use the cursor keys to move the cursor to the place you want to alter before making any changes.

This form of editing is known, for fairly obvious reasons, as *cursor editing*. You may practise using these techniques on a piece of text that you type in yourself, or you may use the text provided with Quill. If you type in your own text, do not worry about any mistakes you make. In fact it may be a good idea to add deliberate mistakes – each mistake will give you extra practice in using the editing facilities.

INSERTING TEXT

Quill is initially in *insert mode*, so that any text you type is automatically inserted at the cursor position. To insert letters or words into the middle of the text, do the following:

Move the cursor, by using the four cursor keys, to the point where you want to make the insertion.
Type the letters or words that you want to insert. The characters are inserted immediately under the cursor position, and any existing text moves to the right to make room for them.

The text is rejustified automatically as you make the insertion.

If you wished to insert several words, it would be annoying to have to wait until the text was adjusted each time you pressed a key. Quill detects this situation and reacts by splitting the line at the point where you are inserting text. This is known as an *automatic text split*. You can then type in as much text as you like.

Quill will restore the text when you finish inserting text at that point (i.e. when you press a cursor key, a function key or **ESC**).

DELETING TEXT

The deletion of text at the cursor position is also very simple. You use the **CTRL** key together with the cursor keys.

To see the action of the left cursor key, position the cursor immediately after the character or characters that you want to delete. Now hold down **CTRL** and press the left cursor key briefly. The letter immediately to the left of the cursor position will be deleted and the cursor will move one space to the left. Each time you press the left cursor key, with the **CTRL** key held down, one more letter will be deleted. If you wish to delete several letters you can hold both the **CTRL** and the left cursor key down, using the auto-repeat facility. Always press the **CTRL** key before the cursor key.

If you use **CTRL** together with the right cursor key, text will be deleted, character by character, from beneath the cursor position, and the text to the right will close up to fill the gap.

You can delete whole words at a time, to the left or to the right of the cursor, by using **SHIFT** and **CTRL** together and pressing either the left or the right cursor key.

You can delete the whole line to the left or right of the cursor. Hold down the **CTRL** key and press the up cursor key. The line to the left of the cursor will disappear. Similarly, pressing the down cursor key will delete the whole of the line to the right of the cursor.

In all cases the text will be rejustified automatically.

OVERWRITING

In overwrite mode you can write over existing text and replace it with the new text.

You can change to overwrite mode by holding down **SHIFT** and pressing **F4**. The mode indicator at the left hand side of the status area will change from 'INSERT' to 'OVERWRITE' indicating that text typed at the keyboard will replace existing text. Pressing **SHIFT** and **F4** again will change back to insert mode.

With Quill set to overwrite mode, position the cursor at the start of the text to be replaced and type in the replacement. When you have finished making replacements, return to insert mode by pressing **SHIFT** and **F4** again, otherwise you will write over text that you want to keep.

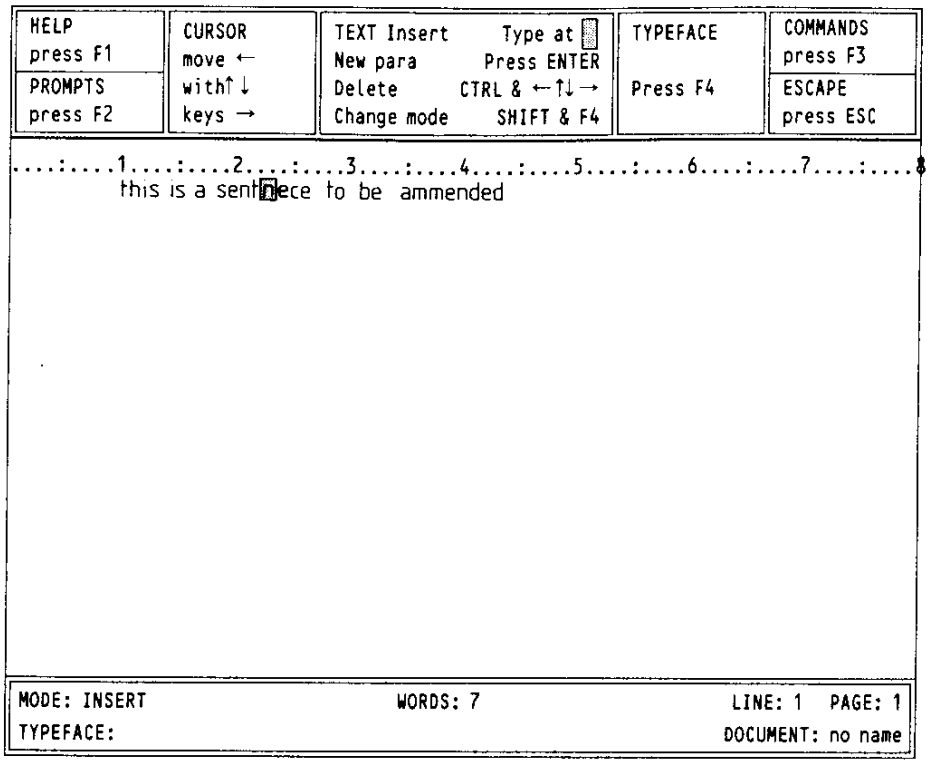


Figure 3.1 Overwriting

Figure 3.1 shows a typical situation where you would want to use the overwrite mode. With the display as shown, with the cursor on the 'n' of 'sentnece' you can overwrite with 'e' and 'n' to correct the word. (You can also practice deleting a character by removing one of the 'm's from 'ammended'!)

CHAPTER 4
TEXT FORMATTING

This chapter is concerned with the *format* of the text; that is, the layout and appearance, as opposed to the actual content. You will find out how to use the different typefaces, Bold, Underline, High and Low script. You will also learn how to move the position of the left, right and indent margins, and how to change the justification which affects the way the text is aligned with respect to the margins.

TYPEFACE

The underlining facility has already been used as an example of the use of the typeface option. In this section we shall examine its use more fully, together with the options to use bold characters, high script (superscript) and low script (subscript).

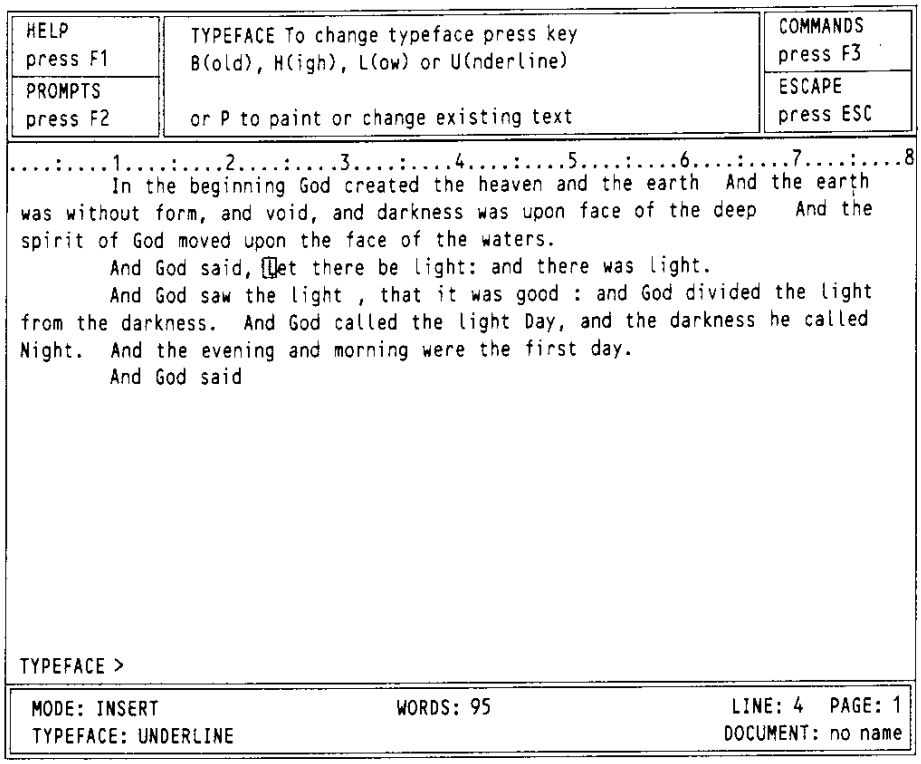


Figure 4.1 Selecting a typeface.

In general you can select any of these options by pressing **F4** and then the appropriate letter – Bold, Underline, High script or Low script. If one of these options is currently switched on, you can turn it off again by exactly the same method as you used to turn it on – by pressing **F4** and then the appropriate letter.

Note that any text that you type will always appear in the typeface shown in the status area. If you move the cursor into a region which is in bold type, for example, the status area will show Bold typeface, and any further text that you type within this region will also be in bold type. The typeface changes automatically as soon as you move to a region containing a different typeface.

Of course, you can only use one of High script or Low script at any one time. If you select one of these, the other is automatically switched off.

There are three ways in which you may want to use the typeface option:

Insert new text in a particular typeface,
Alter existing text to a new typeface,
Change or remove an existing typeface.

If you want to type in some text in a particular typeface you should press **F4** and select the typeface you want. Any text that you then type in will appear in the typeface you have selected. When you want to return to normal text you should switch off the typeface by pressing **F4** followed by the appropriate typeface letter(s).

It is easy to change the typeface used in existing text. The method is known as *painting* since you use the cursor like a paint brush, changing the typeface of any text over which it moves.

First you must move the cursor to the start of the text to be changed, press F4 and then press the P key. Next, select the combination of typefaces you want. Use the right and down cursor keys to move the cursor across the text to be changed. When you reach the end of the text you want to alter, leave the option by pressing **ENTER**. You do not need to switch off the typeface selection; it will revert to the correct typeface as soon as you move away from the area painted in the new typeface. Figure 4.2 shows the appearance of the screen while painting text with underlining.

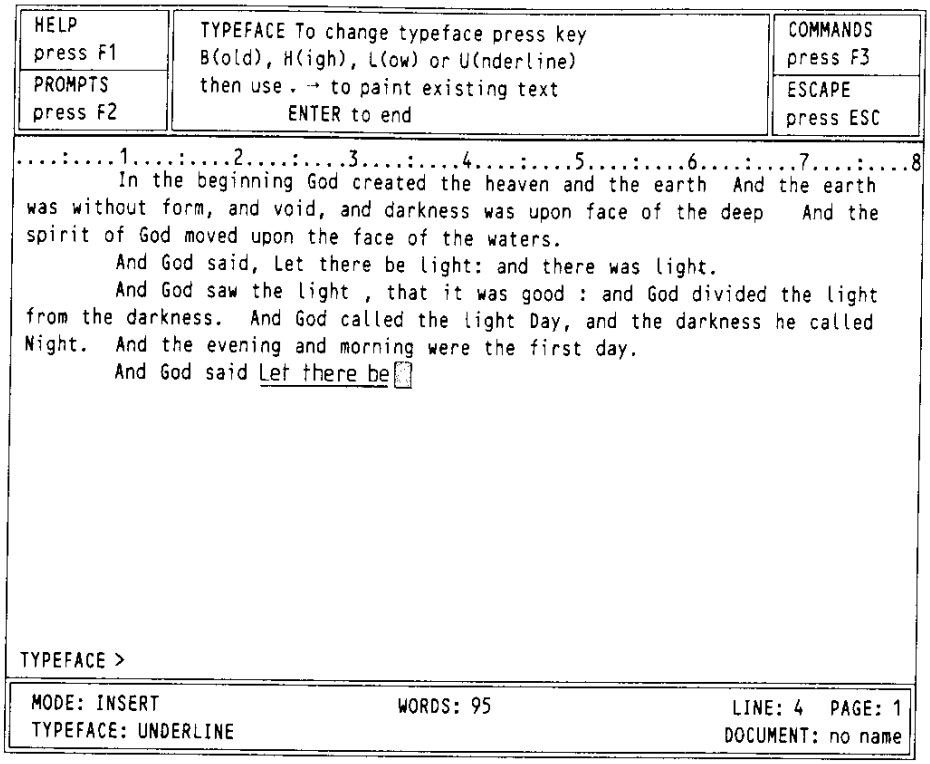


Figure 4.2 Painting underline typeface

You can change, or remove, an existing typeface in the same way in which you add a new typeface to existing text. Again you should move the cursor to the start of the text before pressing **F4**. Press the P key and then select (or switch off) the typeface combination you require. Move the cursor through the text you wish to change and then press **ESC**.

When you change text from an existing typeface to a new one, Quill does not remember the original typeface. Suppose, for example, you change text which was originally underlined to being in bold characters. If you later remove the bold typeface, the final text will be in plain characters, and will not revert to being underlined.

MARGINS

You change the widths of the margins with the **margin** command. Each new margin position takes effect from the current paragraph and remains in force for all following paragraphs, until you make another change to the position of the margin.

Press the command key (**F3**) and then the M key to start this command. In addition to other changes in the control area you will see that three choices – LEFT, INDENT and RIGHT will appear and that the LEFT option is highlighted. These options represent the three margins, and the one that is highlighted is the one that you can move. You can step the highlighting from option to option by pressing the space bar, or you can select a particular option by pressing the key corresponding to its first letter. When the name of a margin is highlighted in the control area you can move that margin with the left or right cursor keys.

Suppose you wish to move the left margin to the right by three characters, starting with the second paragraph of your document.

First move the cursor to any point in the second paragraph and then type:

F3 M

As indicated by the highlighting, the left margin is the one you can move, so you just have to press the right cursor key three times. The change in the margin takes place immediately, so that you can see the effect before you leave the command.

You can leave the command straight away by pressing ENTER, or you can continue to make further margin changes. Press the space bar until the correct margin is selected and move it with the left and right cursor keys. You can use the up and down cursor keys to move the cursor to another paragraph and make further changes to the margins. After you have made all the changes you want you can leave the command by pressing ENTER.

The indent margin marks the character position which is used for the start of a new paragraph. For an 80 character display it is initially set at the fifteenth character position.

There is no restriction on the relative positions of the indent and left margins. If you do not want to use indented paragraphs you may move them so that they are both in the same place. You may even place the indent margin to the left of the left margin. This is useful for producing numbered paragraphs as shown in the following example.

```
Indent Margin
|
|  Left Martin
|  |
1) This is the first of two
   paragraphs to show how you can
   use indent margins
2) The indent margin is three
   characters to the left
   of the left margin.
```

In this case, starting a new paragraph (by pressing ENTER) will allow text to be typed at the "indent" position. All following text will be displayed between the left and right margin positions until you press ENTER again.

JUSTIFICATION

The **justify** command allows you to alter the type of justification used in your document. Like the **margins** command, all changes take effect from the current paragraph (that containing the cursor) and remain in force until the end of the document, or until the next change of justification. When you select this command you will see that you are offered the choice of left, right or centred justification.

Initially, it assumes *right* justification, the text is aligned on both the left and right margins, producing text with an appearance like that of this manual. If there are not sufficient characters on a line to make the margins match, extra spaces will be added between the words until they do. The final effect is very professional. However, if you use an unusually large quantity of extra-long or hyphenated words in a document, unpleasant-looking spaces may result.

To choose *left* justification, press the L key after calling the Justify command. This will produce text which looks like the text in this paragraph. The left margin is aligned, but the spacing of the text within a line is not adjusted, so that the right hand margin is left uneven.

Centre justification, selected by using the C option of the Justify command, causes the text of each line to be centred between the left and right margins. The text could then appear as shown in this paragraph. Centre justification is useful, for example, in centering headings and titles, or for adding labels to diagrams.

As with the **margins** command, you may press the up or down cursor keys to move to another paragraph and make further changes of justification. Press ENTER to leave the command.

CHAPTER 5
COMMAND EDITING

This chapter will extend your knowledge of the editing facilities to include block copies, moves and erasures. In addition, the extremely powerful technique of search and replace editing will be introduced. These facilities are available through the Quill editing commands – copy, erase, search and replace.

COPY

In addition to copying a block of text from one place in the document to another, the **copy** command also allows you to move blocks of text.

The only difference between copying and moving text is that, in the case of a copy, the original text is left in position so that you end up with two 'copies'. You would use this, for example, if you wanted to create a table, with a piece of text repeated a number of times, or if you wanted to see the best place to include a particular paragraph.

If you move some text, the new copy is inserted and the old copy is deleted, so that you are left with only one version.

The **copy** command gives you the option of either keeping or deleting the old copy and therefore gives you both facilities in a single command.

When you select the **copy** command (by pressing **F3** and then the C key) you must first move the cursor to the beginning of the text you want to copy, and then press **ENTER**. Move the cursor to the end of the text to be copied. When you move the cursor the text that will be affected by the command is highlighted so that it is easy to see how much text will be copied. If you accidentally mark too much text you may use the left or up cursor keys to move backwards, but you may not pass the start of the marked text. After you have marked the text you should again press **ENTER**.

In response to the next prompt you should move the cursor to the point where you want the selected text to be inserted and press the C key. The copy will be made and inserted immediately. You are then asked if you want to delete the old copy. You should press the K key to keep the old version (to produce the effect of a copy) or press **ENTER** to accept Quill's suggestion to delete it.

You can then end the command by pressing **ENTER**, which will take you back to the main display.

However, you also have an option of making further copies of the same text at other places in your document. All you have to do is to move the cursor to the point where you want another copy and press the C key. You can repeat this as many times as you want. While you are making these extra copies you are not asked whether to keep or delete the old copy. When you have finished making copies you should press **ENTER** to leave the command.

As is normal in Quill, pressing **ESC** will cancel any partially completed action, but will not undo anything that has been completed. All copies that you have made will be left in the text if you press **ESC**.

ERASE

You should use this command (press **F3** and then E) if you want to remove any large blocks of text from your document. Remember that it is simpler to delete small bits of text with the cursor editing facilities described in Chapter 3.

As with the **copy** command, you are asked to move the cursor to the start of the text to be erased and then to press **ENTER**. You then have to move the cursor to the end of the text – again the text which will be affected is highlighted. When you are satisfied that you have marked the correct amount of text you should press **ENTER** and the marked text will be erased immediately.

SEARCH

The **search** command allows you to look for a particular word or phrase, through all or part of your document. You can use it, for example, to check whether you have used a particular word or phrase too often. The first search will start at the beginning of the text, but can then be continued from the current cursor position.

The **search** command is in the second command menu so you select it by pressing **F3**, O and then S.

When you use the command you are asked to type in the text which you want to find, finishing with **ENTER**. Quill will immediately start searching your document from the top until it finds the first occurrence of the text. The cursor is left positioned at the start

of the found text. If this is the occurrence you want, you can leave the command by pressing **ENTER**.

However, once you have given the **search** command some text to look for, you can use it again to find the next occurrence of that text. Instead of pressing **ENTER**, just press the C key. If you do this Quill continues to search from the current cursor position until it finds the next occurrence of the given text. You can repeat this as many times as you like, finding successive occurrences. Press **ENTER** to leave the command when you have found the occurrence you want.

If, at any stage, Quill does not find another occurrence of the text in your document it tells you so and waits for you to press the space bar and will then return to the main display.

REPLACE

The **replace** command is similar to the **search** command, but also gives you the ability to replace some or all of the occurrences that are found. The command is in the second command menu, so you select it by pressing **F3**, O and then R.

You are asked to type in the text to be found. When you press **ENTER** at the end of the text, Quill immediately finds the first occurrence and asks you to type in the replacement text (don't forget to press **ENTER** at the end)

Quill then asks if you want to replace the found text. Press the R key to replace the text – if you press the N key the text is not replaced. In either case Quill then continues the search for the next occurrence and offers you the same choice of keeping or replacing the found text. This continues until no further occurrences are found, or until you press **ENTER**.

If, at any stage, Quill does not find another occurrence of the text in your document it tells you so and waits for you to press the space bar and will then return to the main display.

You can use the command to make multiple replacements, insertions or deletions as illustrated in the following examples:

To *replace* occurrences of 'river' by 'stream', give 'river' as the text to be found and 'stream' as the replacement text.

To *insert* 'or stream', give 'river' as the text to be found and 'river or stream' as the replacement text.

To *delete* 'river', give 'river' as the text to be found and give no replacement text (just press R).

CHAPTER 6
MORE FORMATTING

In this chapter we shall cover the remaining options for modifying the appearance of the text. It includes setting tab stops and page breaks, and using bold characters, underlining, subscripts or superscripts. In addition there is a section on the **design** command, which you can use to change the settings of various options (such as the page size) that control the overall appearance of your documents.

TABS

Using Tab Stops

A very common way of controlling the layout of a document is by tab stops. These are marked positions, at particular columns of the text of your document. When you press the **TABULATE** key, the cursor will move to the right, from its present position, to the next tab stop in the line. If you have passed the last tab stop, then pressing the **TABULATE** key will move you to the start of the following line.

So that you know where the tab stops are, Quill draws the tab positions and their types (as described below) in the line immediately below the ruler.

Quill allows you to use several different types of tab stop, and to position them in any column. You can have up to sixteen tab stops in a line.

Tab Stop Types

There are four different types of tab stop.

The most common type is known as a *left* tab stop and this works in exactly the same way as the tab positions on a normal typewriter. When you press the **TABULATE** key the cursor will move to the next tab position and any text you type in will start at the tab column. It is called a left tab since lines of text at such a tab stop are aligned at their left hand edges.

A second type is a *right* tab stop. When you move to such a tab stop and start typing, the cursor will remain at the tab position and the text will appear to the left, so that it ends at the tab position. This will continue until the text to the left of the tab position has filled the space available or until you press the **TABULATE** key again to move to the next tab position. Lines of text at such a tab stop are aligned at their right hand edges.

There is also a *centred* tab stop. Text typed at such a tab position will be adjusted so that its central character is positioned on the tab stop. Again the aligning of the text will continue until the available space (to existing text or to the left margin) is filled, or you press the **TABULATE** key again.

The fourth type of tab stop is a *decimal* tab, and is used for typing in numerical values. When you type a number at such a tab stop it is positioned so that its decimal point is at the tab column. If you do not type a decimal point in the text, it will behave like a Right tab.

Figure 6.1 shows the appearance of text typed at each of the four different types of tab stops.

```
Left          Centre        Right         Decimal
|             |             |             |
a             a             a             a
piece of      piece of      piece of      piece of
text          text          text          text
|             |             |             |
12.345        12.345        12.345        12.345
123.4         123.4         123.4         123.4
1234.56       1234.56       1234.56       1234.56
|             |             |             |
```

Figure 6.1 The four types of tab stop

The Tabs Command

Initially tab stops are set at every tenth character position and are all Left tabs. You can change the number, position and type of tab stops with the **tabs** command.

You can place tab stops at any point in the line and mix the different types in any way you like. The only limit is that you may not have more than sixteen tab stops in the line. The new tab stops take effect from the current paragraph (that containing the cursor when you called the **tabs** command) to the end of the document, or to the next change of tab positions.

When you select the **tabs** command (**F3** and T) the positions are drawn in the display, immediately beneath the ruler.

Each tab stop is marked by a letter (L, C, R or D) to indicate its type. The cursor is positioned at the beginning of the line and you can move it to the left or right using the appropriate cursor keys.

You can make as many changes to the tab stops as you like. You may also press the up or down cursor keys to move to another paragraph and make more changes to the tab stops. When you have made all the changes you want, press **ENTER** to leave the command and return to the main display.

Inserting a Tab

To insert a tab stop, select the type you want, use the left and right cursor keys to move the cursor to the position the tab is required, and press T.

When you have selected the Tabs command the types are shown in the control area.

The control area contains the words (L)eft, (R)ight, (C)entre and (D)ecimal and the word (L)eft is highlighted. This shows that the next tab stop to be inserted will be a Left tab.

You can change the type of tab stop to be inserted either by pressing the space bar (each time you press it the highlight moves from one type to the next) or by pressing the key corresponding to its first letter. For example, if you want to change to a Right tab, you can either keep pressing the space bar until the word (R)ight is highlighted, or just press the R key.

Deleting a Tab

Remove a tab stop by moving the cursor until it is over the tab marker that you want to delete and pressing the X key.

DESIGN

You use the Design command to change features in the main display, such as:

characters per line
line spacing
lines per document page

The command is illustrated in figure 6.2 and a full description of each option appears in Chapter 8.

Press **F3** and then D to select the **design** command. Quill then shows the list of options. If, for example, you want to select a 40 character display, press the D key, for the 'Display Width' option. This option will be highlighted and Quill waits for you to press 4, 6 or 8 to select a display width of 40, 64 or 80 characters. It will not allow you to select any other option until you have chosen one of these three.

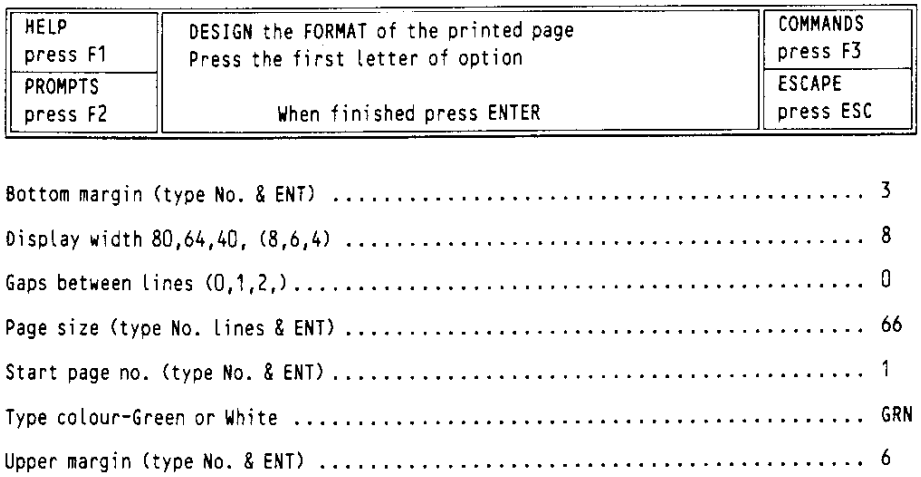


Figure 6.2 The design command

You then have the option of changing any or all of the items listed in the display. When you have made all the changes you want you should leave the command by pressing **ENTER**.

WIDE DOCUMENTS

If you move the right margin so that the number of characters in a line is greater than the screen width, Quill cannot show the full width of your document on the screen. In this situation the display area acts like a window, through which you see only part of the full document. As you move the cursor along a line, the window will slide across the width of your document, so that it always shows the region containing the cursor.

Paging

One of the options in the **design** command is to set page size, in terms of the maximum number of lines of text that can appear on a page of your document. In addition to the text, this number of lines includes the upper and lower margins, any header or footer and the lines of space between them and your text.

Suppose, for example, that you have an upper margin of 3 lines, a header separated by 2 blank lines from your text, a footer separated from the text by 4 blank lines and a bottom margin of 5 lines. This takes up a total of 3+1+2+4+5=16 lines. If you have a page size of 66 lines then there will be 66−16=50 lines of text on each page. If you were then to use the **design** command to set the gaps between lines to be 1 (double spacing) you would have only 25 lines of text on each page.

Page Breaks

A page break marks the point in your document where a new page will start, depending on the length of the page that is set in the **design** command. It is shown as a horizontal line across the screen and includes the page number. Quill allows for the upper and lower margins, headers and footers when calculating the length of a page. In the above example, with the gaps between lines set to zero, Quill will insert a page break after each block of 50 lines of text.

If you set a page size that does not leave space for five or more lines of text per page, Quill will turn off the paging. No page breaks are shown and Quill treats the whole document as a single page. You can make sure that automatic paging is turned off by setting the page size to zero.

Forced Page Breaks

You can use the **page** command to force a page break to occur at a particular line. This is very useful for making sure that a section of text, such as a list or a table, is started at the top of a new page and is not shown in two parts on different pages.

You can set a page break in your text at any time by using the **page** command which is in the second command menu – press **F3**, the O key and then the P key. You should then position the cursor anywhere in the line at which you want the page to end and press the P key. Quill will insert a page break after the end of this line.

You may set several forced page breaks, at different positions, but you may not set more than one forced page break in any one line of your document. When you have finished, press **ENTER** to leave the command.

You can remove a forced page break from your document at any time – you do not use the **page** command for this purpose. Remove a forced page break by moving the cursor with the up cursor key until it lies on the page break. Then press **CTRL** and, while holding it down, press the left cursor key.

CHAPTER 7
FILE OPERATIONS

When you have produced a document you will probably want to save a copy of it on a Microdrive cartridge. At some later date you may want to make some changes and keep a copy of the new version. If you have a printer, you will certainly want to produce printed copies.

Each document is saved on a Microdrive cartridge in the form of a *file* – a named chunk of information. This chapter describes the commands provided to save, load and print files.

SAVE

You use this command to save a copy of the text of a Quill document on a Microdrive cartridge. If you do not save a document after you have written it, you will lose its contents when you leave Quill.

When you use the **save** command (**F3** and then S) you are asked to type in a name for the document. The simplest way to use the command is, therefore, to type in something like the following sequence.

F3 S myletter ENTER

This saves your document with the name 'myletter_doc' on the cartridge in Microdrive 2.

If this name is the same as that of a document which is already saved on Microdrive 2, Quill will remind you that the document already exists and ask if you want to overwrite it with the new one. Press **Y** (yes) to replace the document or **ESC** to save the document with a different name.

When the document has been saved Quill asks you if you want to continue editing the document press **ENTER** to continue and the space bar if you want to change to another document.

When you name a document to save it, or if you load a previously saved document, Quill displays the document name in the status area. If, at some time later, you want to save the document again, Quill suggests the current document name as the name to be used. If you type in a name of your own choice, it will replace the one suggested. Alternatively you may accept Quill's suggestion by just pressing **ENTER**. In such a case you can just type in:

F3 S ENTER

The new version will then be saved on the Microdrive cartridge, replacing the old one.

LOAD

You should use the **load** command when you want to copy a document from a Microdrive cartridge into the computer's memory so that it may, for example, be edited.

You are first asked to type in the name of the document you want to load. If you have forgotten it you can type in a question mark, plus **ENTER**. Quill will then display a list of all the documents on Microdrive 2 and again asks you to type in the name.

If the name you type in does not correspond to the name of an existing document, Quill will tell you that the document does not exist and give you another chance to type the name.

FILES AND MERGE

The files command includes four options:

Backup – to copy a Microdrive document or other Microdrive file
Delete – to erase a Microdrive document or other Microdrive file
Format – to format a Microdrive cartridge
Import – to insert a Microdrive file, exported from Abacus, Archive or Easel, into the current document at the position of the cursor.

The **merge** command allows you to insert a document from a Microdrive cartridge into the current document at the position of the cursor.

With these commands you will often want to use a second data cartridge. For example, you will usually want to make a backup copy of a document on a different cartridge, and an import file will not usually be on the same cartridge as your Quill document.

You can remove the Quill cartridge in Microdrive 1 and replace it with another cartridge but remember to replace the Quill cartridge before printing a document or asking for Help. If you are using additional Microdrives then normally it will be necessary to remove the Quill cartridge in Microdrive 1.

PRINT

This command is used to produce a printed copy of all or part of a Quill document. It is, of course, necessary that you have a printer and that it is correctly connected to the computer, otherwise nothing much will happen!

Quill suggests that you print the document you are currently working on and waits for you to press a key. Press **ENTER** to accept this suggestion, or type in the name of the document to be printed (which must be a document on the cartridge in Microdrive 2).

Quill will ask you if you want the whole document to be printed. Press **ENTER** to accept the suggestion. Otherwise you type the number of the page of the document at which you want printing to start and also the page number of the last page you want printed, terminating each number by **ENTER**. You can only print complete pages of your document.

The **print** command has an option to print to a Microdrive file instead of to the printer. Press **ENTER** to use the printer, or type a new file name if you want to send the text to a file. The file produced will contain all the characters and control codes that would otherwise have been sent to the printer.

The simplest use is to print all of the current document. The keys you press in this case are:

F3 P ENTER ENTER ENTER

To print pages 2 to 4 inclusive of a document called "myletter_doc" to a new file "myletter_lis" (both on drive 2) you should type:

F3 P myletter ENTER 2 ENTER 4 ENTER myletter ENTER

Before starting to print Quill will read the current printer driver from the Quill cartridge in Microdrive 1. This will tell Quill what facilities are available on a particular printer and how they can be used. Quill will work with most makes of printer and you will find details of how to make changes for a particular type of printer in the *Information* section, where the printer driver program is described.

You may also wish to change things such as the line spacing and the number of lines per page of the printed document. These are all included in the **design** command, which is described in Chapter 6.

CHAPTER 8
QL QUILL REFERENCE

THE FUNCTION KEYS

In addition to the standard use of F1, F2 and F3, function key 4 is used as follows:

F4	change typeface
SHIFT & F4	switch between insert and overwrite

Quill does not use function key 5.

Select a command by pressing **F3**. This switches Quill to display a command menu. You can still move the cursor but you are not allowed to insert or delete text.

The control area display changes to show a list of the commands available. You select a command by typing its first letter. A second set of commands (COMMANDS II) is available and you can switch between them using the Other command.

Since there are commands in the two sets that start with the same letter you must always make sure that the command you want is shown in the control area before you select it.

In general, you can leave any partially completed command by pressing **ESC**.

At the end of most commands Quill returns to the main display. The exceptions are those commands that have their own internal menu (e.g. **Files**). In these cases you are left in the internal menu and must press **ESC** to go back to the main display.

In any command that requires text input (e.g. **save, load, files, replace**) you may edit the text with the line editor, described in the QL Program Introduction.

THE COMMANDS

The following commands are available:-

They are listed in alphabetical order. If they are part of the second command menu then this is shown by a II symbol after the command name.

COPY

Use this command for either moving or copying text from one place in the document to another.

You are first asked to move the cursor to the start of the text to be copied and then to press **ENTER**. Next move the cursor to the end of the text you want to copy. You can move the cursor backwards with the up and left cursor keys but you cannot move it back past the starting point. The affected text is highlighted. Press **ENTER** when you have finished. Press **ENTER** again to delete the original marked text or press K to keep it. Move the cursor to the position where you want the marked text to appear and press the C key to insert the text at the new position.

You can make further copies of the text at any other point in your document. Position the cursor where you want another copy to appear and press the C key. You may make as many copies as you like. When you have finished press **ENTER** to end the command.

DESIGN

This command allows you to set or change a number of features which control the overall appearance of your document. Within the command you are asked to choose, by pressing the appropriate key, from the following options:

Bottom margin	type in the number of lines to be left blank at the bottom of each printed page of your document. Press **ENTER** when you have typed in the number. The initial setting is for a bottom margin of 3 lines.
Display width	type in 4, 6 or 8 to select a display of 40, 64 or 80 characters per line. Quill will not accept any other characters. The initial setting is for either 80 or 64 characters depending on whether you are using a monitor or a television.
Gaps between lines	type in 0, 1 or 2 to select how many blank lines will be printed between each line of text in your document. Quill will not accept any other characters. The initial setting is 0.
Page size	type in the total number of lines to be used for each page of your document and press **ENTER**. This number includes the blank lines in both the upper and bottom margins. If you type

in a zero the document will not be split into pages. The initial setting is 66. (You can normally print 66 lines on a standard A4 page.)

Start page number — type in a number, followed by **ENTER**. This number is used to number the first page of your document. Successive pages are numbered consecutively from this value. You may want to change it if your document is a continuation of another document. The initial value is 1.

Type colour — to change colours used for normal and bold text. Each time you select this option the normal and bold text colours switch between green and white. The initial setting is for ordinary text to be green and bold text to be white.

Upper margin — type in the number of lines space to be left blank at the top of each page of your document and press **ENTER**. The initial setting is for 6 lines.

At the end of each option you may select another option, or press **ENTER** to leave the command.

ERASE

This command allows you to erase text from your document. You are first asked to move the cursor (with the cursor keys) to the first character that you want to erase, press **ENTER**, and then move the cursor through the text you want to erase. The marked text is highlighted. When you have marked the text you should press **ENTER** again and the text is erased immediately.

FILES II

There are four options provided in this command.

Delete — to delete a named document or file from a Microdrive cartridge. You are asked to type in the name of the file you want to delete. The file is deleted when you press **ENTER**.

Format — to format a cartridge in Microdrive 2. Since this erases all the information on the cartridge, you must confirm your selection.

Warning: all information on the cartridge is erased when you format it.

Backup — to make a second, security copy of a document on a Microdrive cartridge. You are asked to type in the name of the document and the name you want to give to the new copy. You would normally make the copy on a different cartridge and could therefore use the same name again.

Import — to insert another file from a Microdrive cartridge into your document, at the position of the cursor. The file must be a file exported from either QL Abacus or QL Archive, or a text file produced, say, from SuperBASIC. See the *Information* section.

FOOTER

This command allows you to specify a line of text to be used as the bottom line on each page. It does not appear on the display screen—only on the printed page

You are first asked to select the position of the footer from the four options:

None – no footer text
Left – at the left margin
Centre– centred in the page (the initial setting)
Right – at the right margin

Press the space bar until the required option is highlighted and then press **ENTER**. You are then asked to type the text for the footer, ending by pressing **ENTER**.

If you have previously specified a footer then this list is shown in the status area. You have the option of altering it with the line editor, rather than typing in the whole of the revised text.

You can include a page number anywhere in the text. The position and type of number is marked by a three character code:

Characters	Page Number Style
nnn or NNN	Arabic Numerals e.g. 1, 2, 3, 4
rrr or RRR	Roman Numerals e.g. I, II, III, IV
aaa or AAA	Alphabetic e.g. A, B, C, D

You are finally asked to type in a number, from 0 to 9 to indicate the number of lines to be left between the bottom of the text and the footer.

GOTO

You may use this command to move the cursor to the top, bottom or to a specified page in your document. You are offered three options:

Top	to move the cursor to the beginning of your document.
Bottom	to move the cursor to the end of your document.
a page number	typing in a number, followed by ENTER moves the cursor to the start of that page of your document. If there are no page breaks in your document this option will move the cursor to the end.

HEADER

This command allows you to specify a line of text to be used as the first line on each page. Note that the header does not appear in the display of your document on screen. Quill does not automatically provide a header for your document.

You are first asked to select the position of the header from the four options:

None	no header text (the initial setting)
Left	at the left margin
Centre	centred in the page
Right	at the right margin

You press the space bar until the required option is highlighted and then press ENTER. You are then asked to type the text for the header, ending by pressing ENTER.

If you have added a header at an earlier stage the existing text is shown in the status area. You then have the option of changing with the line editor, rather than typing in the whole of the text.

You can include a page number anywhere in the text. The position and type of number is marked by a three character code:

Characters	Page Number Style
nnn or NNN	Arabic Numerals e.g. 1, 2, 3, 4
rrr or RRR	Roman Numerals e.g. I, II, III, IV
aaa or AAA	Alphabetic e.g. A, B, C, D

HYPHENATE (II)

This command allows you to specify a point within a word where it can be split, with an automatically inserted hyphen, if it extends beyond the end of a line. Words not marked in this way will, if necessary, be moved to the next line in their entirety.

Hyphenation is particularly useful when you are using right justification, to avoid large gaps being left between the words.

Move the cursor to the first character following the position where you want to allow a split to be made and press the H key. You may repeat this process as many times as you want. Press ENTER to leave the command.

The command will have no apparent effect on the word if it is not at the end of a line.

JUSTIFY

Use this command to select the type of justification you want. It takes effect from the start of the paragraph containing the cursor, and remains in effect to the end of the document, or to the next change of justification.

You are offered the following options, selected by pressing the key corresponding to its first letter:

Left	the text is aligned at the left margin, but the right margin is uneven.
Centre	the text of each line is centred between the margins.
Right	additional spaces are inserted between words in each line so that the text is aligned at both the left and right margins.

You may make changes of justification to more than one paragraph. Press the up or down cursor keys to move up or down by a paragraph and change the justification as described above. Press ENTER to end the command.

LOAD

This command allows you to load a document into memory from a Microdrive cartridge, ready for printing or editing.

Type in the name of the document (the name you gave it when you saved it). If you just press "?" plus **ENTER**, Quill will show you a list of the names of all the documents saved on Microdrive 2. Edit the suggested text, Microdrive 2 – if you want a list of the files from a different Microdrive. When Quill has shown the list, you are again asked to type in a document name.

MARGINS

Use this command to set or change the positions of the left, indent and right margins of your document. All changes in the margins are shown in the text as you make them.

The control area shows the words left, indent and right, and on first entering this command the word left is highlighted. This means that you can use the left and right cursor keys to move the left margin.

You can select any of the three margins by pressing the space bar until the correct margin name is highlighted. You can move the selected margin by pressing either the right or left cursor key.

The change in each margin takes effect from the paragraph containing the cursor. It remains in effect to the end of your document, or to the next change of position of that margin.

You may make changes of margin positions to more then one paragraph. Press the up or down cursor keys to move up or down by a paragraph and change the margins as described above. Press ENTER to leave the command.

MERGE (II)

The **merge** command takes a copy of a named Quill document from a Microdrive cartridge and inserts it, at the position of the cursor, in the document currently in memory.

This command allows you the option to replace the Quill cartridge with a data cartridge. You must replace the Quill cartridge in Microdrive 1 at the end of the command.

Position the cursor at the point where you want the document to be inserted before selecting the command. Quill asks you to type in the name of the file you want to insert. If you insert the document in the middle of a paragraph, Quill will split it into two paragraphs at the position of the cursor and insert the document between them.

OTHER

This command allows you to switch to the display of a second set of commands in the control area. The list of commands in the control area alternates between the two lists each time you use Other.

Since several commands start with the same letter, you must make sure that the command you want is one of those displayed, before you choose it.

PAGE (II)

You can use this command to mark a point in your document where you want a new page to start.

Move the cursor to the point where you want the new page to start and press P.

You may add such page breaks at several points in your document. Move the cursor to the point where you want another page to start and press the P key. Press ENTER to leave the command.

Do not use the **page** command for deleting a forced page break. You can cancel a page break by moving the cursor to any point on the page break line and then pressing **CTRL** and the left cursor key together.

PRINT

This command prints all or part of the document currently in the computer's memory, or any other document on the cartridge in Microdrive 2.

Press **ENTER** to print the current document, or type in the filename of the document to be printed, followed by **ENTER**.

Quill then suggests printing the whole document. If you reply by pressing **ENTER** the whole document will be printed. If you only want to print some of the pages, type in the number of the first page you want printed, followed by **ENTER**. Then type the number of the last page you want printed, again ending by pressing **ENTER**.

Finally, press **ENTER** to send the text to a printer, or type the file name of a new file, followed by **ENTER** to send the output to a Microdrive file.

Before printing, Quill will read a "printerdat" file entering the printer driver information.

QUIT

This command allows you to leave Quill and return to SuperBASIC. You have three options:

ENTER	to save your current document before returning to SuperBASIC. You are given the further option of typing in a name for the saved document. If you just press **ENTER** the document will be saved with its old name, replacing the original version of the document on the Microdrive cartridge.
A	to abandon your current document and return to SuperBASIC without saving it.
ESC	to cancel the command and return to your document.

REPLACE (II)

You can use this command to replace some or all occurrences of one piece of text by another.

First type in the word(s) to be replaced, followed by **ENTER**. Then type in the replacement word(s), again followed by **ENTER**.

Quill searches from the start of the document until the first occurrence of the old text is found. It then offers you the option of replacing the old text with the new. Press the R key to replace the text, or N if you do not want to replace it.

Quill will then search for the next occurrence and again offer you the option to make the replacement. This process will continue until you reach the end of the document or until you end the command by pressing **ENTER**.

SAVE

You use this command to save a copy of your document on a Microdrive cartridge.

Type in a name for your document, so that it can be identified. The document is then saved under that name. If, instead of typing in a name, you just press **ENTER**, the document will be saved with its old name, replacing the original version.

Quill then asks you if you want to continue editing the document you have just saved. If you press **ENTER**, the text of the document remains in the computer's memory and you can continue working on it.

Alternatively, press the space bar if you want to work with another document.

SEARCH (II)

This command searches your document for a particular word or phrase.

First type in the text which you want to find. When you press **ENTER** Quill starts at the top of your document and searches for the first occurrence of the text.

You may press the C key to continue the search to find the next occurrence of the text. Press **ENTER** to end the command when you have found the occurrence you want.

TABS

The **tabs** command allows you to specify the positions and types of tab stops on a line of text. The tabulate key will then take you straight to the next tab stop along the rule which you have set. Each change of the tab stops will take effect from the start of the current paragraph (the one containing the cursor). It will remain in effect to the end of your document, or until the next change of tab stops.

There are four types of tab stop provided:

Left	the tab stop behaves like a left margin; the text is positioned to the right of the tab stop.
Centred	the text will be centred around the tab stop.
Right	the tab stop behaves like a right margin; the text is positioned to the left of the tab stop.
Decimal	this is used for aligning decimal numbers. Each number will be positioned so that its decimal point is at the tab stop. Until a decimal point is encountered it behaves like a right tab.

The tab positions are drawn on the screen, below the ruler, using the following symbols:

L - left
C - centred
R - right
D - decimal

The cursor is positioned at the start of that line. You can move the cursor along the line by using the left and right cursor keys.

You can remove a tab marker by moving the cursor with the left and right cursor keys until it is over the tab marker in the line under the ruler and then pressing the X key.

To insert a tab marker you should first select the type you want by either pressing the space bar until the correct type is highlighted in the control area, or pressing the L, C, R, or D key. Then move the cursor to the appropriate point and press the T key.

You can mix inserting and deleting tab markers in any combination. You may also press the up or down cursor keys to move to another paragraph and make further changes to the tab stops. When you have made all the changes you want you should press **ENTER** to leave the command and return to the main display.

ZAP

This command deletes the whole of your current document, without saving it on a Microdrive cartridge. It allows you to discard your current document and start again.

TYPEFACE

You can change the typeface of the text in your document by pressing **F4** and then the first letter of one of the four options listed below. The selected typeface affects all text subsequently typed in.

Alternatively you may press **F4** and then the P key to paint new text in a new style.

You are offered the following options:

Bold	text is converted to a bold, or heavy, typeface.
High script	text is printed in the upper half of the line.
Low script	text is printed in the lower half of the line.
Underline	text is underlined.

You may select any combination of these options except, of course, that you can not have both high and low scripts selected together. If you select either of these, the other will be switched off automatically.

If you want to select a combination of typefaces, you should select them one after another, by pressing **F4** and the appropriate letters.

If you press the P key to select the paint option, Quill allows you to select one or more typeface styles. Move the cursor to 'paint' the text to the new style and press **ENTER** when you have finished. Note that the original typestyle is restored after painting.

You can switch off any of the typeface options in the same way that you use to turn it on – that is by pressing **F4** and then the appropriate key (B, H, L or U).

INSERT AND OVERWRITE MODES

Initially Quill in insert mode and any text that you type in will be inserted into your document at the position of the cursor. Any surrounding text will be spread out to make room.

If you hold down **SHIFT** and press **F4**, Quill will switch to overwrite mode. In this mode any text that you type in will replace, character by character, any text from the cursor position onwards.

You can switch back to insert mode by the same method, that is by holding **SHIFT** down and pressing **F4**.

THE START-UP PARAMETERS

When you first load QUILL it is in the state described by the following list. You can change each of the properties by the method indicated in the right hand column

Feature	Initially	Change By
Mode:	insert	SHIFT & F4
Display width:	80(mon) 64(TV)	Design
Left margin:	10 0	Margins
Indent margin:	15 5	Margins
Right margin:	70 64	Margins (max 160)
Upper margin:	6	Design
Bottom margin:	3	Design
Justification:	Right	Justify
Tab stops:	Left, cols 10,20,..,80	Tabs
Page size:	66	Design
Gaps between lines:	0	Design
Page header:	none	Header
Page footer:	centred, "page nnn"	Footer
Start page number:	1	Design
Text colour:		
Normal	green	Design
Bold	white	Design
Typeface:		
Bold	off	F4
Underline	off	F4
High script	off	F4
Low script	off	F4

sinclair

QL

QL Abacus

FOR SERVICE MANUALS
CONTACT:
MAURITRON TECHNICAL SERVICES
www.mauritron.co.uk
TEL: 01844 - 351694
FAX: 01844 - 352554

©1984 PSION LIMITED
by Dick de Grandis-Harrison *(Psion Limited)*

CHAPTER 1
ABOUT QL ABACUS

QL Abacus is a spreadsheet which can be used for planning, budgeting, tabulating data, calculation, information storage or for presenting information. This information is represented on a tabulated *grid* divided into 255 rows and 64 columns. The data area you see on the computer screen is a *window* through which you can see part of the grid. You can move this window across the grid. The intersections of the rows and columns represent more than 16,000 *cells* or boxes in the grid. You can enter text into any cell or cells, or the cells may be used for the storage of numbers or data.

The real power of Abacus, however, comes from the use of rules, or *formulae*, which can connect different blocks, rows or columns of cells, or even individual cells of the grid. This means that information inserted in one area can immediately be evaluated and represented in another form elsewhere.

For example, you can use twelve of the columns to represent months of the year and you can then enter sales data along a 'sales' row. The next two rows can contain formulae to calculate the cost of sales (as a percentage of sales plus a fixed cost, say) and the profit. The monthly profits will then be evaluated automatically each time you type in a sales figure. The yearly totals can also be summed by another formula, so that a change in the sales of, say, March will immediately lead to a completely different profit profile and total for the year. All the figures are evaluated by Abacus automatically.

You can also represent the data from Abacus as graphics or in a table in the word processor, through the *export* commands of the Psion QL package.

In many respects Abacus is like a visual programming language, but one which is easy to use. You may manipulate text, data, or formulae, use input and output statements and text variables.

If, at any time, you are not sure what to do, remember that you can ask for Help by pressing **F1**. Also remember that you can cancel any partially-completed operation (e.g. typing in a number, or using a command) by pressing **ESC**.

CHAPTER 2

GETTING STARTED

LOADING QL ABACUS

Load QL Abacus as described in the Introduction to the QL Programs, don't forget that Abacus requires a formatted cartridge in Microdrive 2. When loaded the following message will de displayed:

```
LOADING QL ABACUS
version x.xx
Copyright © 1984 PSION SYSTEMS
spreadsheet
```

where x.xx represents the version number (e.g. 1.02).

The program will then wait for a few seconds before starting.

The Help information is not loaded into the computer's memory together with the program. It is only read from the Abacus cartridge when it is needed. **You should therefore not remove the Abacus cartridge from Microdrive 1 if you intend to use the Help facility.**

When Abacus is first loaded the appearance of the screen is as shown in Figure 2.1. This is the *main display*.

GENERAL APPEARANCE

Abacus can display 80, 64 or 40 characters per line of the display. If you are using a domestic television the display may not be clear enough for you to see 80 characters per line and you should use 64 or 40 characters. The 64 character display is very similar to that for 80 characters but the 40 character display is arranged slightly differently. This is shown in figure 2.2.

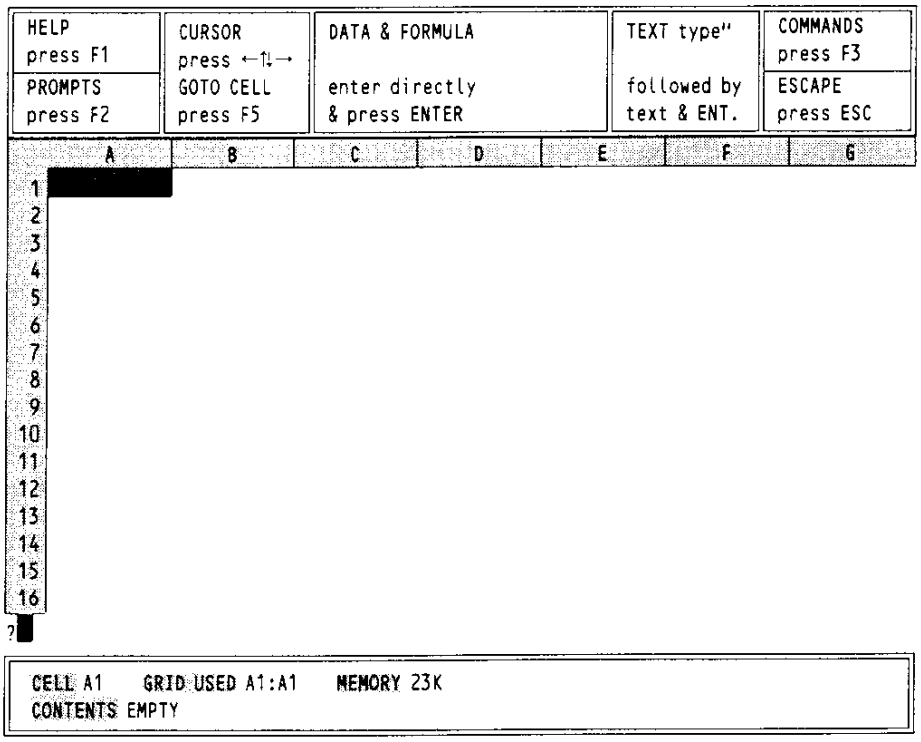


Figure 2.1 The main display with a monitor. (80 characters)

Abacus initially selects either an 80 or a 64 character display depending on whether you started from SuperBASIC in the Monitor or the TV.

Apart from the difference in appearance, Abacus works in exactly the same way with all three display formats. Most of the diagrams in this manual are shown for the 80 character display.

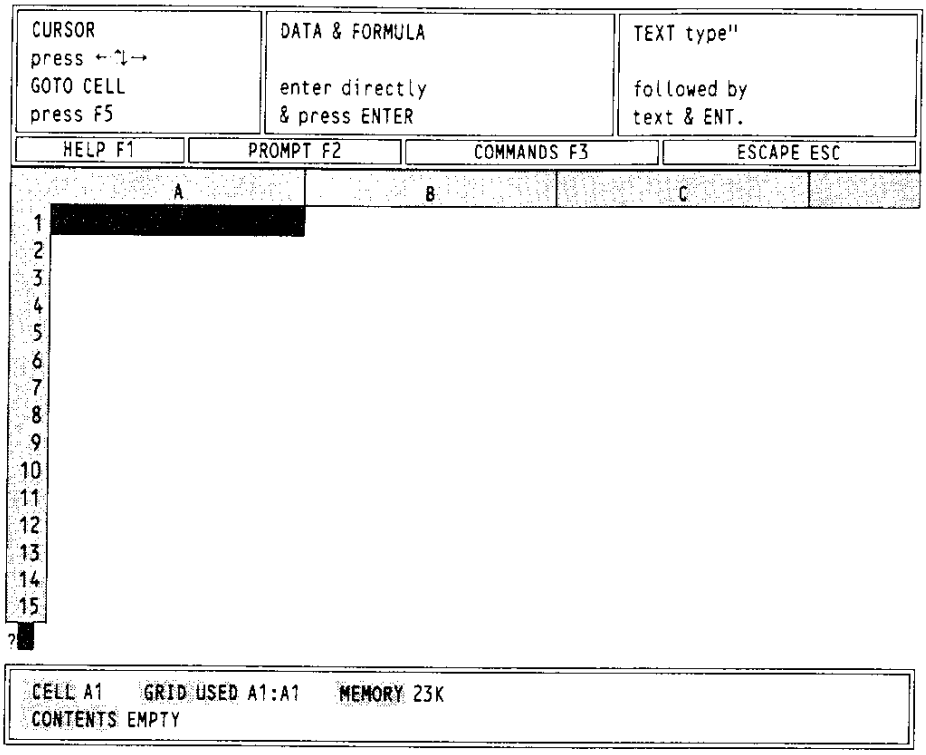


Figure 2.2 The main display for 40 characters

The Window

The central area of the screen contains the *window* showing part of the grid.

Across the top of the window you will see a line in which a number of letters appear. These letters label vertical columns of cells making up the grid. As you can see, columns A,B,C and so on are visible. Down the side of the window there is a series of numbers, from 1 to 15. These numbers label the rows of cells in the grid.

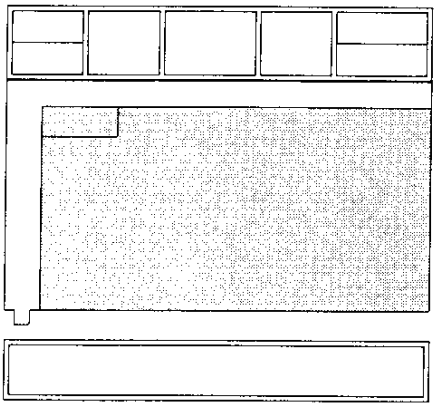


Figure 2.3 The window

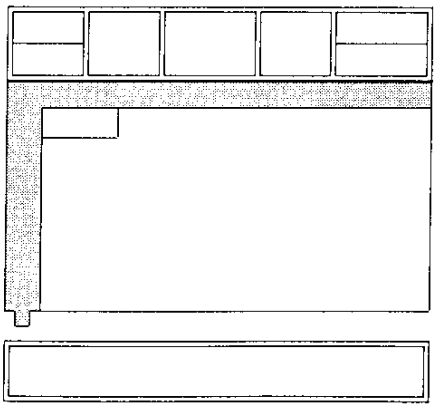


Figure 2.4 The grid labels.

A combination of a letter and a number will therefore identify one particular cell, and is known as a *cell reference*. For example, A1. This refers to the cell which is in column A and row 1, (the top left hand cell in the window).

You will see that this cell is different from all the others in that it is filled by a large red rectangle. This is known as the cursor and it marks the *current cell*, that is the cell which will receive any data you type in.

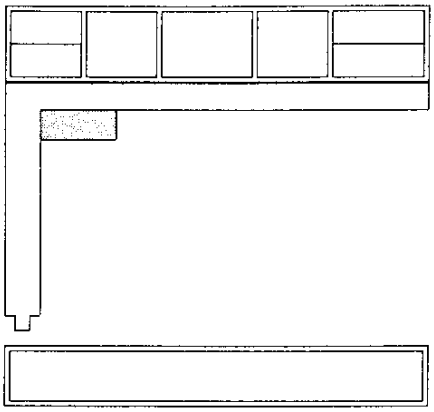


Figure 2.5 The cursor

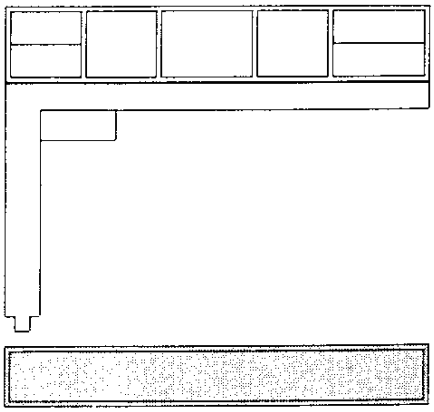


Figure 2.6 The status area

The Status Area

The bottom section of the display contains the *status area*, which gives information about the current state of the grid.

It contains the cell reference of the current cell and its contents. This cell is empty when you have just loaded Abacus. In addition, the status area shows the extent of the used portion of the grid (as the cell reference of the bottom right cell of the used portion) and the amount of memory left.

The Control Area

The control area shows the normal options to obtain Help (F1), to turn the prompts on and off (F2), to select a command (F3) and to cancel an incomplete selection (ESC). In addition there are three options that are specific to Abacus. These are:

move the cursor,
type in data or a formula,
type in text.

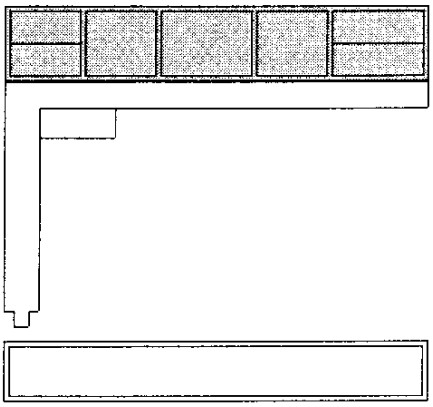


Figure 2.7 The control area

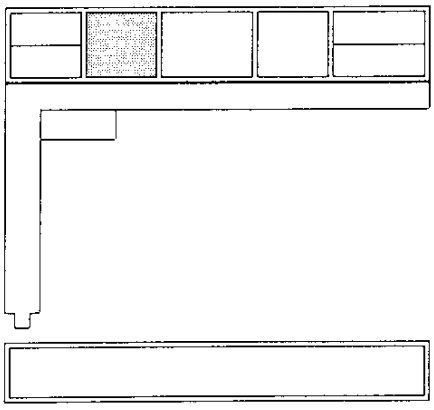


Figure 2.8 Moving the cursor

MOVING THE CURSOR

The four cursor keys move the cursor around the grid. Press the right cursor key once. The cursor moves one column to the right and the current cell indicator now shows B1. If you then press the left cursor key once the cursor returns to cell A1. Pressing the left cursor key again will have no effect because you are at the extreme left hand edge of the grid.

Move the cursor to the extreme right hand edge of the grid. Pressing the right cursor key again will not move the cursor but the letters across the top of the window will change. When you attempt to make the cursor leave the visible area of the grid the window will move across the grid so that the cursor remains in view.

The cursor keys are a useful way of moving the cursor, provided you only wish to move it one or two cells. They are very inefficient for making large movements across the grid. For such large movements it is more convenient to go directly to the required cell. You can do this by pressing F5, to select the **goto** option, and then typing the required cell reference, followed by ENTER.

As an example of using the **goto** option, ask Abacus to move the cursor to cell D11. First press **F5** to select the **goto** option. The words '**goto > A1**' will then appear in the line immediately below the window. Abacus is suggesting that the cursor be moved to the top left hand corner of the grid. If you accept this suggestion (by just pressing **ENTER**) the cursor will move to that point. To move the cursor to another cell, type in the cell reference – in this case type:

```
d11
```

and press **ENTER**. Note that the 'd' may be in upper or lower case – Abacus will accept either. The cell reference you type in replaces that suggested by Abacus and the cursor moves directly to the cell you have specified.

You should now move the cursor back to the top left hand corner of the grid by using this option again. This time you can accept the suggested cell reference (A1) so all you have to type is:

F5 ENTER

You will find that you go back to the original state of the display, with the cursor at the top left hand corner of the window, in cell A1.

Now move the cursor to cell Y1, by typing in

F5 y 1 ENTER

Look at the letters labelling the columns across the top of the window and you will find the column to the right of column Z is labelled AA, the next one is labelled AB, and so on. This enables you to refer to more than 26 columns.

There are 64 columns in total and, after AZ, the columns are labelled BA, BB and so on. The last column in the grid is labelled BL.

You can also move down the grid to find the last row but you will have to go a long way; there are 255 rows in the grid.

ENTERING NUMBERS

Return the cursor to cell A1 and then type

```
100
```

but don't press **ENTER** just yet. The 'Data or Formula' option box in the control area is now highlighted, to confirm your action. The prompt **value >**, followed by the number 100 will also have appeared in the line immediately below the window.

All typed input, and the text that Abacus shows while you are using a command, appears in this line. It is the *input line*.

The small rectangle in the input line marks where the next input character will appear, and is known as the input cursor, to distinguish it from the main cursor in the window. If you make a mistake at any time during typing to the input line, you can correct it by using the line editor, described in the Introduction to the QL Programs.

ENTERING TEXT

When you press **ENTER** the value 100 will be transferred to the current cell (A1) and the input line will clear, ready for more input. You will see that the value 100 also appears in the status area, at the bottom of the display.

Putting text into a cell is the same as entering a number except that text is preceded by double quotation marks. As soon as you type the quotation marks, Abacus responds by emphasising the TEXT option box in the control area and showing **text >** " in the input line. You then type in exactly what you want to appear in the cell, followed by ENTER. There is no need for a closing quotation mark. Try entering text into a few cells and, in particular, notice the difference between entering, say:

```
1000 ENTER (a number)
```

and

```
"1000 ENTER (text)
```

A number is shown at the right of the cell, whereas text is placed at the left. The status area also shows the type of information; text, numeric and so on, in the current cell.

THE COMMANDS

You select a command by first pressing F3.

The central part of the control area shows a list, or menu, of the available commands and is known as the *command menu*, illustrated Figure 2.9.

Most of the commands are described in later chapters but we can take a quick look at two of them. These are **Zap**, which you use to clear the whole grid, and **Quit**, which allows you to stop using Abacus and return to SuperBASIC.

Try the **Zap** command first. Press **F3** and locate the **Zap** command in the displayed menu. If you press the Z key, the word *Zap* will appear in the input line – you need never type more than the first letter of any command. Also, the command box in the control area changes to show the menu for **Zap**. Try pressing **ESC** first, to cancel the command.

Now return to the command menu by pressing **F3** and then press Z to call the **Zap** command again, but this time press **ENTER** next, to clear the grid. You will be left with a blank grid and with the cursor in cell A1, ready to start afresh.

Whenever you want to leave Abacus and return to SuperBASIC, you must use the **Quit** command. This works in a similar way to **Zap**, (press **F3** and then the first letter of the command (Q)). Quitting causes you to lose the contents of your grid, so you are again given the option of going back to the main level by pressing **ESC**.

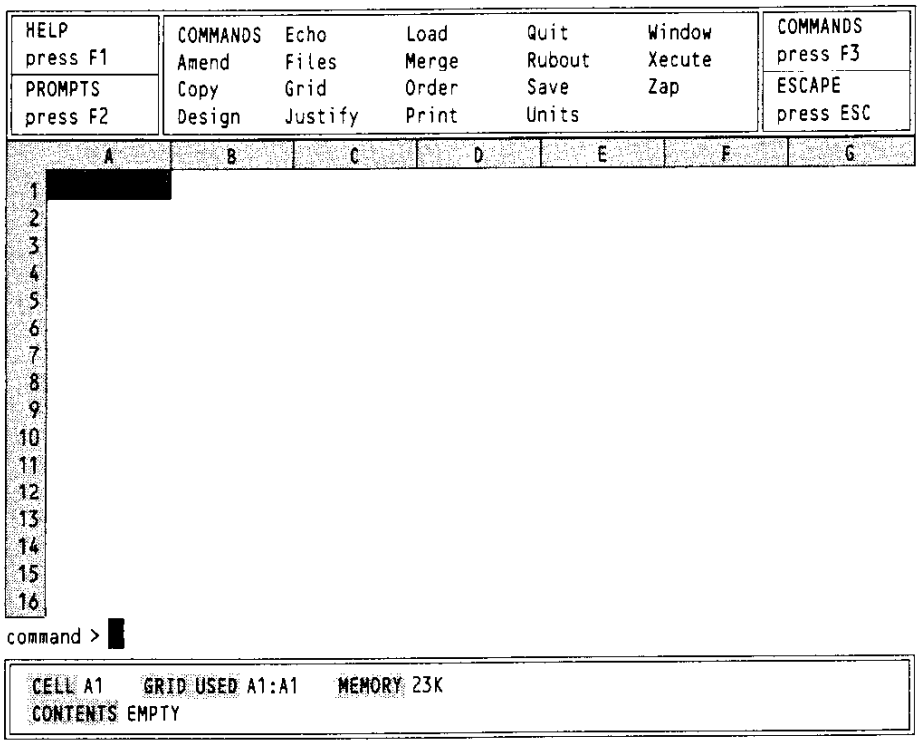


Figure 2.9 The command menu

CHAPTER 3
CELLS, ROWS, COLUMNS AND RANGES

Much of the power of Abacus lies in its ability to handle whole rows, columns or ranges of cells in a single operation. You do this by using simple expressions which allow you, for example, to fill all or part of a row of cells. The values in the cells may all be made the same or they may vary in a regular way.

This chapter describes some of the properties of cells and the ways in which you can refer to them.

CELLS

The cell is the basic unit for holding information in Abacus. Each cell can contain one item of information which may be text, a number or a formula.

For each cell that contains information, Abacus also keeps a record of how that information is to be displayed. You can, for example, display numbers or text at the left, centre or right of the cell, and you can display numbers in several different formats.

Justification

You use the **Justify** command to change the position of the display within a cell. It allows you to select the position of numbers or of text within a cell or group of cells.

Put a value of 100 in cell A1 and then use the **Justify** command by pressing **F3** and then the J key. Abacus first asks you to select between a Cells and a Defaults option; select the Cells option by pressing **ENTER**. Abacus then asks you to choose between either text or numbers. Select numbers, by pressing the N key. Next you must select Left, Centre or Right justificaton. Since Left is suggested by Abacus, select it by pressing **ENTER**.

Finally Abacus asks you to specify the range of cells that are to be affected. In this case just press **ENTER**. You will see that the value of 100 in cell A1 will move to the left hand side of the cell.

Note that you can change the numeric format or numeric justification of a cell which currently contains text. Nothing will appear to happen. If, however, you later change the contents of the cell to be numeric, it will be displayed with the format and justification that you specified. This also applies to a change of text justification for a cell which currently contains numeric information.

Empty Cells

Cells that contain no information do not exist as far as Abacus is concerned, and use no memory. They can therefore have no properties. If you attempt to use the Cells option of either the **Justify** or the **Units** command on an empty cell they will have no effect. Numbers subsequently placed into such a cell will be displayed in the general default format.

If you want to change these defaults you must use the Defaults option of either the **Justify** or the **Units** command (or both). For example, use the Defaults option of the **Units** command (press **F3**, U and then D) to select a default of percent format with one decimal place. The choices are similar to those in the Cells option, but you are not asked for a cell range.

The Defaults option of the **Justify** command works in the same way. Again you are not asked to type in a cell range because Abacus will use the new default each time you put informaton into any previously empty cell.

The new default settings will remain in effect until you change them again, or until you finish using Abacus and return to SuperBASIC.

To restore the defaults to their original state – numbers justified right, text justified left and numbers displayed in General format – use the following sequences:

F3 J D N R {number right justified}
F3 J D ENTER ENTER {text left justified}
F3 U D G {number displayed in general format}

ROWS

Very often you will want to fill several cells in a particular row with a particular value, or with values that vary in a regular way. Abacus provides simple ways of doing this. One method is to refer to the cells of a row with a *range identifier*. There are two range

identifiers, **row** and **col**. They refer to the cells of the current row or column – the row or the column that contains the cursor.

As an example, let us fill the first row, from column B to column D, with the value 100. We shall use the range identifier **row** as follows. Place the cursor in cell A1 and then type:

```
row = 100 [ENTER]
```

As soon as you press **ENTER** a prompt appears in the input line suggesting that the row be filled starting at column A (the column containing the cursor). The system will always make a reasonable suggestion for the starting point and this can be accepted simply by pressing **ENTER**. In this case, however, we want to start at column B so you should press:

```
B [ENTER]
```

The input line changes to show that the filling of the row is to start at column B and a further prompt appears with a suggestion of BL (the last column in the grid) for the end column. Again this will have to be changed, since we want to end at column D, so you should press:

```
D [ENTER]
```

The instruction is now complete and will be carried out – the value 100 will appear in each of the cells from B1 to D1 inclusive and the input line will clear, ready for your next input.

COLUMNS

Filling a column follows a very similar pattern except, of course, that you refer to a column by one or two letters rather than the number that identifies a row. Suppose we want to put the text 'hello' in each of the cells of column D, from row 5 to row 11. We can do this by using the second range identifier, **col**. Move the cursor to cell D5 and type:

```
col = "hello" [ENTER]
```

This time Abacus suggests the correct starting point (row 5) as this row contains the cursor, and you can accept this suggestion by pressing **ENTER**. Row 255 will then be offered as a suggested end point and you should change this by typing:

```
11 [ENTER]
```

The text will appear in cells D5 to D11 inclusive and the input line will clear, ready for the next input.

Each time you use **col** you will be asked to specify the first and last row to be affected. You may, as usual, accept or replace the values that Abacus suggests.

In addition to this way of using the range identifiers **row** and **col**, you can also use them to specify the range of cells for any function that needs such a range. For example:

For example put some numbers in all the cells of the rectangular area whose top left hand corner is the cell A1 and whose bottom right hand corner is the cell C3 (nine numbers in all). Now move the cursor to cell D1 and type:

```
col = sum(row) [ENTER]
```

This fills each cell of column D with the total of the values in the cells of the corresponding row. Abacus needs to know the ranges for both **row** and **col**. It will therefore ask for the range of columns for **row** (Abacus suggests column D to column D, which is correct – accept each by pressing **ENTER**) and then for the range of rows to be used by **col**. Abacus suggests from row 1, which is correct, to row 255 (or to row 11 if you type in this example immediately after the previous one). Accept the first by pressing **ENTER** and type the correct value, 3 (don't forget to press **ENTER**) for the second. Abacus will then calculate the total for each of the three rows and display the results in the cells of column D.

LABELS

The previous examples referred to rows and columns by an explicit use of their number and letter cell references. An important alternative for identifying rows or columns is to use *labels*, that is names which you may choose yourself. These labels are then used to refer to specific rows, columns or cells.

Any text that you put into a cell can be used as a label. You can use labels in any command or formula where you would otherwise use a letter and number reference. The advantage is it is much easier to remember names than numbers and letters when you want to refer to a particular cell.

This is an extremely powerful and flexible method which you can use to great advantage to simplify the setting out and operation of a grid. The following two sections explain how you can use these labels.

Row and Column Labels

A label may refer to either a row or a column, depending on the contents of the other cells in the grid. The basic rule when you use a label to identify a row or column of figures is that Abacus searches below and to the right from the cell containing the label.

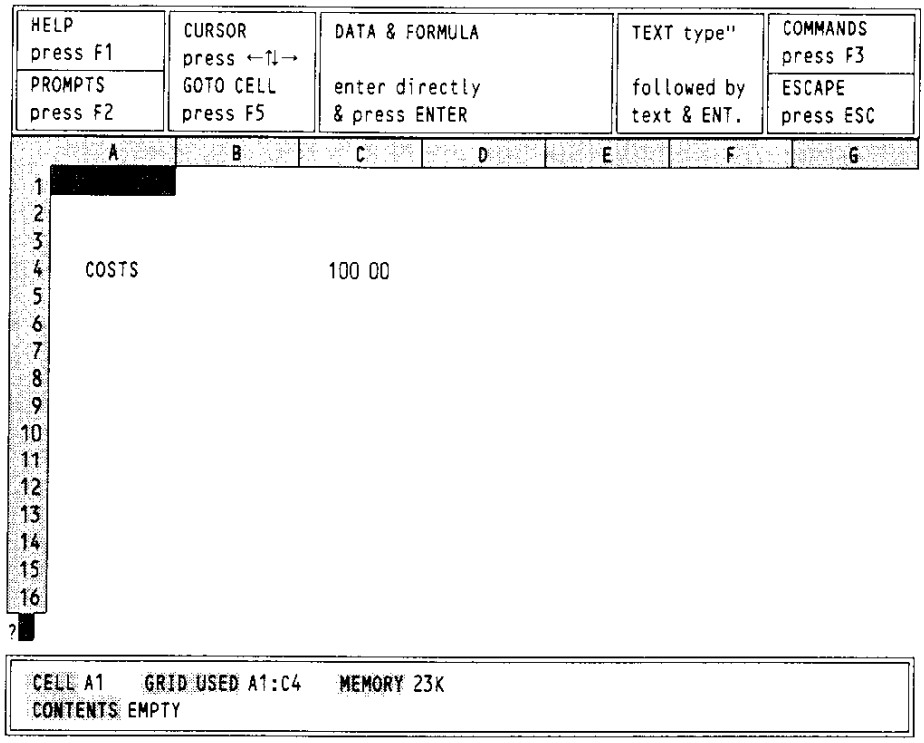


Figure 3.1 Labelling a row

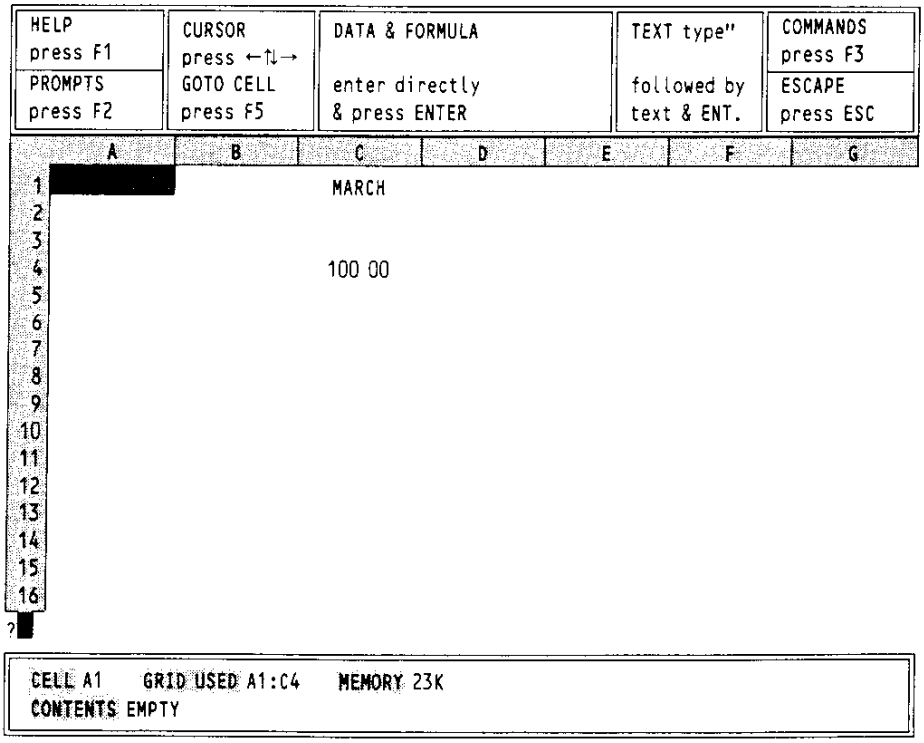


Figure 3.2 Labelling a column

The closest cell that contains a number, below or to the right of the position of the label, determines whether the label refers to a row or to a column. Figures 3.1 and 3.2 should help make this clear. In Figure 3.1 the label refers to a row and in Figure 3.2 it refers to a column.

In more complex cases, for example where there are numbers both to the right and below the label, the nearest number (measured by the number of cells separating the number from the label) determines whether it is a row or a column reference. If the two numbers are the same distance from the label, Abacus shows the message:

```
Cannot tell whether name is a row or col
```

and wait for you to press the space bar. Abacus will then put the text of your formula back into the input line so that you can correct it with the line editor.

You should replace the unresolved reference with either **row** or **col** and press **ENTER** again. You should consider rearranging the labels so that Abacus can resolve the reference in future.

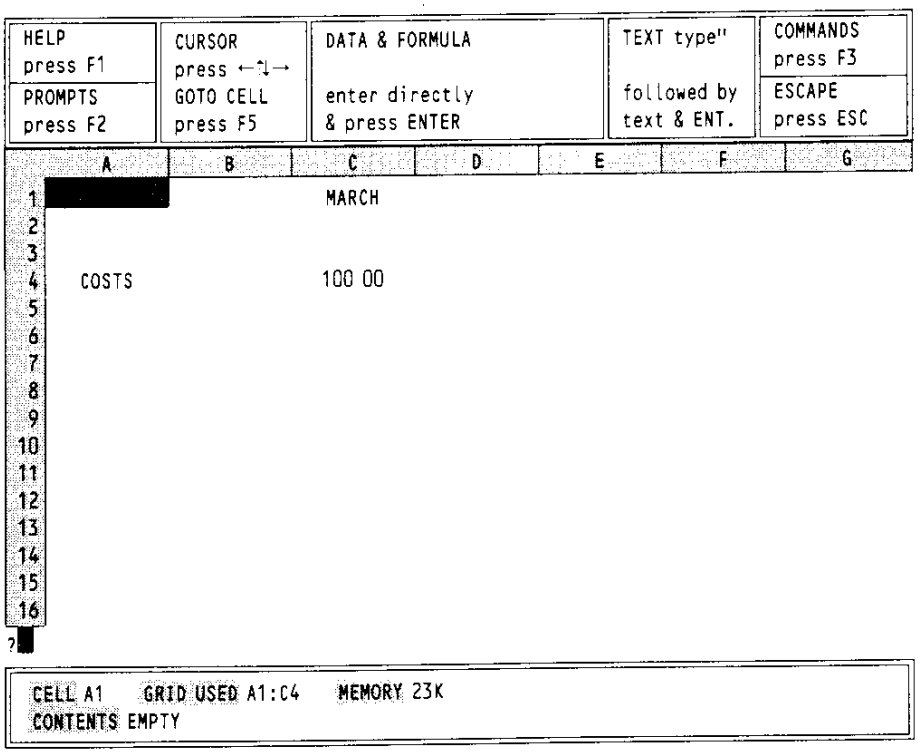


Figure 3.3 Labelling a cell

Labelling Cells

You can also use labels to refer to single cells, but in this case two labels are needed. In the following example the labels 'March' and 'Costs' can be used to refer to cell C4.

The reference is made up of the names of the two labels, separated by a full stop (e.g., March.Costs). It is not necessary to give the full names, and no distinction is made between upper and lower case letters. Also Abacus needs only enough letters of each name to make sure that the identification is unique. In the above example 'mar.cos' would be perfectly adequate. The order of the labels is also irrelevant, so you could also use 'cos.mar' to refer to the same cell.

RANGES

In addition to being able to refer to a whole row or a whole column, you can make an instruction work on a rectangular block, or *range*, of cells.

A range reference is made up of two parts. The first part is the row and column reference of the top left hand cell of the range. This is separated by a colon from the second part, which is the row and column reference of the bottom right hand corner of the range. An example of a range reference is:

```
A2:D27
```

An example of the use of a range reference would be the use of the **Copy** command to copy the contents of a range of cells to a similar range at a different place in the grid.

Many of the commands ask you to type in a range reference, to identify the cells on which they are to work. Since a range reference has a much wider set of possibilities than a row or column reference, Abacus can not suggest a possible range. You must type in the entire range reference yourself. You can specify the range in any one of four ways. These are:

1. With explicit row and column numbers and letters,
   e.g. A1:C7
2. With labels,
   e.g. january.sales:march.costs
3. With a combination of the above two methods,
   e.g. A1:march.costs
4. With a range identifier,
   e.g. row (or col)
   This refers to the cells of the row (or column) that contains the cursor. In this case, Abacus can suggest suitable start and end points.

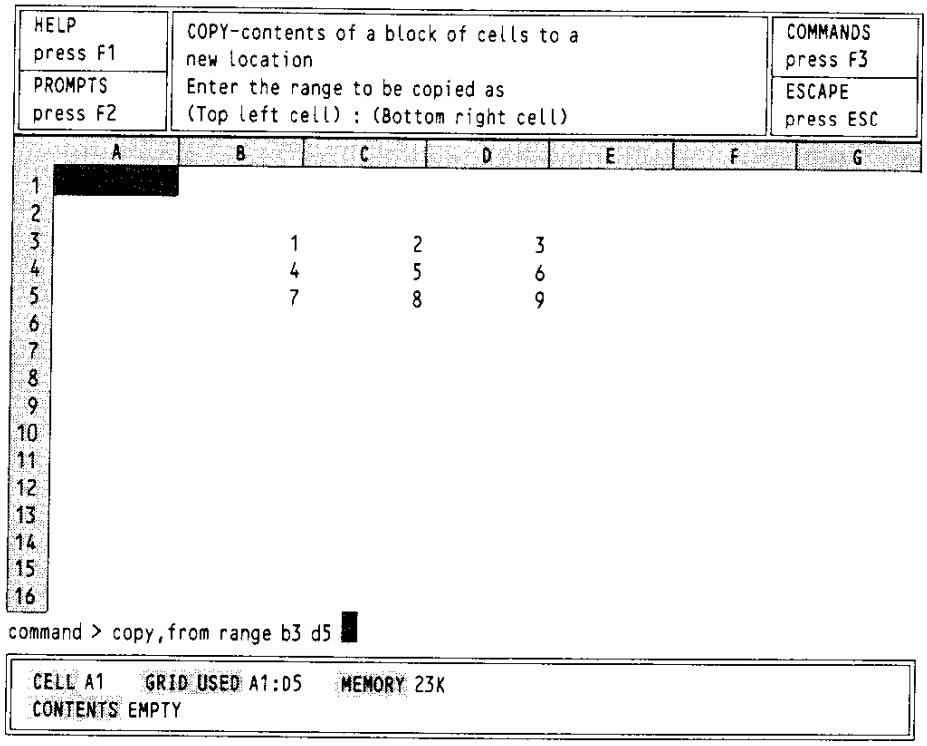


Figure 3.4 A range reference

MORE ABOUT NUMBERS AND TEXT

Now that we have seen how the position of a cell or range of cells can be specified to Abacus, we can go on to show how the appearance of the contents of these cells can be modified. First we must explain the way in which numbers are stored. Move the cursor to cell A1 and type in the number 123.456.

Abacus stores all numbers to an accuracy of 16 significant figures and it can display up to 14 significant figures – the extra two figures are used to make sure that the calculated value is displayed accurately. Although Abacus calculates and stores all numbers to this accuracy, you do not have to display all the significant figures.

Select the **Units** command (by pressing **F3** and then the U key). There are two options: Cells or Defaults. In this case press **ENTER** to select the suggested Cells option.

Abacus offers you several different forms of display.

Press the M key to select the Monetary form of display. Abacus asks you to choose how you want it to show negative values. Abacus suggests that they are displayed with a leading minus sign and you can accept this suggestion by pressing **ENTER**. Alternatively you can display negative values in brackets by pressing the B key instead. In this example it does not matter which we choose, but we shall assume the minus sign option.

Abacus then asks you to specify the range of cells which are to be affected. You could reply by typing in a range reference (e.g. A1:B3) or just the reference to a single cell. Abacus will always try to anticipate the range you require. However, in some circumstances Abacus is unable to do this and will simply suggest the range A1:A1. This range reference is identical to the single cell reference A1. You can either accept the suggestion by pressing **ENTER** or type in your own reference choice followed by **ENTER**.

We will assume that Abacus makes the default range suggestion A1:A1 and the complete sequence of keypresses is:

F3 U ENTER M ENTER ENTER

Just before you press **ENTER** for the third time the input line should contain:

```
Command>units,cells,monetary,minus sign,range A1:A1
```

When you press **ENTER** the display in cell A1 will change to £123.46, even though the actual value (123.456) is still kept, and shown in the status area. Abacus automatically takes you back to the main display.

The monetary form of display always shows the number rounded to two decimal places, with a leading currency sign. (You can change the sign to $, or anything else, by using one of the options in the **Design** command.)

Let us now change the display in cell A1 to Integer (whole number) format, by calling the Cells option of the **Units** command again, but this time pressing the I key. This format also allows you to choose whether to use a minus sign or brackets to show negative numbers and this time we can choose the bracket option by pressing the B key followed by **ENTER** (again we are only affecting cell A1).

The full sequence of keypresses in this case is:

F3 U ENTER I B ENTER

and the input line shows:

```
Command>units, cells, integer, brackets, range A1:A1
```

The cell display now shows 123 – the decimal point and all figures following it are not shown in integer format.

We can now try Decimal format. For this, and the remaining formats, you do not have the option of displaying negative values in brackets. Instead (except for the General format) you must specify the number of figures you want to be displayed after the decimal point; let's use five decimal places. Select the Cells option of the **Units** command. Decimal is the default format and can be selected simply by pressing **ENTER**, then specifying five decimal places. Finally, in response to the 'range' prompt, press **ENTER** to accept the default suggestion. The full sequence of keypresses and the corresponding input line contents are:

F3 U ENTER ENTER 5 ENTER ENTER

```
Command>units,cells,decimal,decimal places 5,range A1:A1
```

Cell A1 will now show 123.45600, as required.

Now use the command again, but this time press the P key, to specify the Percent format. Use one decimal place and select cell A1. The full sequence of keypresses is:

F3 U ENTER P 1 ENTER ENTER

The display will now show 12345.6%. The percent option shows a number multiplied by 100 with an added % sign. Note that the stored value, as shown in the status area, is still 123.456, regardless of the cell display.

We can now try the Exponential format, with three decimal places, by typing:

F3 U ENTER E 3 ENTER ENTER

Before you press **ENTER** for the third time the input line should contain:

```
Command>units,cells,exponent,decimal places 3,range A1:A1
```

and, after pressing it, the cell display will be 1.235E+02.

The exponential format is used to display numbers which are too large or too small to be written in decimal format. The number is written as a value between 1 and 10, multiplied by the appropriate power of ten. The number 2 300 000 000, for example, can be written as 2.3 multiplied by 1 000 000 000 and 1 000 000 000 is ten raised to the ninth power (nine tens multiplied together). So 2 300 000 000 could be written in exponential format as 2.3 E+09. Very small numbers are written using negative powers of ten. Thus, the number 0.000123, which is 1.23 divided by 10000 (ten raised to the fourth power), can be written in exponential format as 1.23 E−04.

The remaining option is the General format, which you can see in cell A1 by typing:

[F3] U [ENTER] G [ENTER]

The input line contains:

```
Command>units,cells,general,range A1:A1
```

This format again does not require you to specify the number of decimal places. Using the General format lets Abacus choose a sensible form for the display of each number. It does the best it can to display each number as accurately as possible in the space that is available.

Before we leave the **Units** command, try displaying the number in cell A1 in decimal format, with nine decimal places. Type:

[F3] U [ENTER] [ENTER] 9 [ENTER] [ENTER]

Cell A1 now shows # # # # # # # # # # indicating that the display will not fit in the space available. Whenever you see this, you should then either change the display format, or increase the width of that column.

Now clear the grid by using the **Zap** command. With the cursor at cell A1, type:

```
"This is a long bit of text
```

Although the text is too long to be contained in one cell, it is all shown. It overflows across the following cells. Now put the number 1 into cell B1. The text is cut off at the end of cell A1 as it is not allowed to overflow across another filled cell. Move the cursor back to cell A1 and verify that the whole of the text is still stored by looking in the status area.

Move the cursor back to cell B1 and use the **Rubout** command to erase. When you use this command you are asked to specify the range of cells whose contents you want to delete. In this case we only want to delete the contents of cell B1 and can do so by pressing **ENTER**. The full sequence of key presses is:

[F3] R [ENTER]

Now that cell B1 is empty, the full text in cell A1 appears again.

CHAPTER 4
FUNCTIONS AND FORMULAE

FUNCTIONS

Abacus contains a number of pre-defined *functions* which are used to perform specific calculations on the contents of one or more cells. A function takes a number of input values, known as *arguments*, and from them calculates a specific result. The result is said to be the value that the function *returns*.

In Abacus you must supply the arguments in brackets after the name of the function and, if there is more than one argument, you must separate each with commas. Most of the functions provided return a numeric value, for example, the function **sum( )**. This takes, as an argument, a range reference and returns a numeric value equal to the sum of the numeric values contained in all the cells within the range.

Some functions, such as **month( )**, return a text value. (**month(1)**, for example, returns the text 'January'). A few functions require no arguments, but you must still include the brackets. For example the function **pi( )** returns the numerical value of the mathematical constant π (approximately 3.14).

Two particularly useful functions are **col( )** and **row( )**. These return the number of the column (or row) which intersect at the cell that contains the function. They are used extensively in the examples in the next chapter.

For example, **col( )** will return a value of 1 from column A, 2 from column B, and so on. The function **row( )** simply returns the row number.

As an example we can use the two functions **month( )** and **col( )** to label columns of the grid. The object will be to place the headings January, February, and so on at the top of columns B to M. We use the function **col( )** to supply the number that **month( )** needs as its argument, so that it gets a different value in every column. Type in:

```
row = month(col())
```

and then press **ENTER**. Select the range from B to M when Abacus asks for the start and end columns. You will see that the result is not quite what we want in that, although the labels start at column B, the first label is February and not January. This is because, in column B, **col( )** returns the value 2 and **month(2)** is the text 'February'. All we have to do is to alter the instruction so that 1 is subtracted from the value returned by **col( )**, before calculating the month. Type in:

```
row = month(col()-1)
```

(Don't forget to press **ENTER** to mark the end of the input.) Select the column range from B to M, as before.

FORMULAE

A formula is usually used to relate the contents of one cell to the contents of one or more of the other cells in the grid. The idea of formulae is very important in the use of Abacus as it allows you to describe even the most complicated calculations in a simple way.

You enter a formula into a cell using the same method employed for entering numbers, that is, by moving the cursor to the cell, typing it and then pressing **ENTER**. Abacus assumes that anything it does not recognise as a number (starting with a numeric digit) or a text value (starting with quotation marks) is a formula.

Move the cursor to cell B3 and enter the number 100, move the cursor to cell C3 and enter 200. Now move the cursor to cell D3 and type in the following formula

```
B3 + C3
```

When you press **ENTER** you will see two things happen. First the value 300 will appear in cell D3; the formula's result has been calculated by adding together the contents of cell B3 and cell C3 and the total placed in cell D3. In addition you will see that the status area at the bottom of the screen shows the formula used to calculate the value in this cell. A cell which contains a formula will always show the result of the calculation. If you position the cursor on the cell then Abacus will show the formula itself in the status area at the bottom of the screen.

The rest of the examples using formulae make use of the labelling facility and the **row** and **col** range identifiers. They allow much more efficent methods of entering information into the grid than the direct use of letter and number cell references.

Note that any numeric formula that does not contain any cell references is not stored as a formula. In such a case Abacus calculates its value and stores the result as a pure number. For example, 37 + 100/20 is stored as the value 42, and not as the original formula.

A SIMPLE CASH FLOW EXAMPLE

	A	B	C	D	E
1		January	February	March	April
2	Sales	£1000.00	£1050.00	£1102.50	£1157.63
3	Costs	£722.00	£749.50	£778.38	£808.69
4	Profit	£278.00	£300.50	£324.13	£348.93
5					

Figure 4.1 Simple cash flow analysis.

Start this example with a grid containing month headings in cells B1 to M1. If you have anything else in the grid you should clear it with the **Zap** command.

Now move the cursor to cell A2, enter the text 'Sales' and then put the value 1000 in cell B2. Now move the cursor to cell C2 and type in the formula:

```
row=sales.january*1.05
```

Accept the range selection given by Abacus (column C to column M) by pressing **ENTER** twice. Note that Abacus knows the end of the row is at column M because that is where the previous row ended. When you press **ENTER** a second time you will see a whole series of values appearing in row 2, from column C onwards, and the formula B2 * 1.05 will appear in the status area at the bottom of the screen.

If you move the cursor along row two you will see that the formula for each cell is slightly different. In each case the formula takes the contents of the cell on the immediate left and multiplies it by 1.05 to obtain the value to place in the current cell. For example, the formula in cell E2 refers to cell D2, and the formula in cell H2 refers to cell G2, and so on.

In Abacus all formulae work in this way unless you specify otherwise. Each formula remembers the relative positions of all cells to which it refers. When such a formula is used in more than one cell the references are adjusted to maintain a *relative cell reference*.

It may prove helpful to point out that the initial value of 1000 placed in cell B2 was necessary for two purposes; to ensure that the label 'Sales' was recognised as a row reference and also to specify the first value to be used by the formula.

Now position the cursor at cell A3 and enter the text 'Costs'. Without moving the cursor, type in the formula:

```
costs = sales * 0.55 + 172
```

This formula calculates the cost from two components. They can be regarded as manufacturing costs (55% of sales) and fixed costs totalling £172.00.

Use the suggested start and end points of column B and column M. Since the contents of the row is defined in terms of the row reference 'Sales', the label 'Costs' will also be taken as a row reference, with the same range as 'Sales'.

Again you should move the cursor along the row, examining the different formulae shown at the bottom of the screen, in order to understand how the results have been calculated.

Finally, put the text 'Profit' in cell A4 and type in a further formula

```
profit = sales-costs
```

with the same range selection as before (i.e. columns B to M). Abacus will do all the rest of the work for you, producing a simple, but complete, example. If you now change the display to monetary format with the command:

F3 Units,Cells,Monetary,Minus sign,Range B2:M4

you should find that the first few columns appear as in Figure 4.1.

AUTO-CALCULATION

When you have typed in the simple cash flow application described in the previous section, try changing the number in cell B2 (Sales.January).

Move the cursor to this cell – the easiest method is to press **F5** and then type in the cell reference (either B2 or sal.jan) followed by **ENTER**. Now type in any number you like. When you press **ENTER** you will see that all the numbers in the grid will change!

All the formulae in the cells of the grid are recalculated automatically each time you make an entry to a cell. Since all the formulae in this example refer, directly or indirectly, to the value held in cell B2, all their values will change when you alter the contents of this cell. (Remember that we assumed that sales would increase by 5% per month, based on the January figure.)

You can switch off the auto-calculate facility by using the **Design** command. This is useful, for example, when you have many complicated formulae in the grid and do not want to wait for a recalculation each time you change a single value.

Select the Design comand by pressing **F3** and then the D key. The display changes to show a list of the options, as shown in Figure 4.2. You can select any one of these options by typing its first letter. Select the auto-calculate option by pressing A and the auto-calculate state changes automatically. You leave the command by pressing **ENTER**.

HELP press F1	DESIGN allows modification of options Press first letter of option	COMMANDS press F3
PROMPTS press F2	Press ENTER when finished	ESCAPE press ESC

```
AUTO-CALCULATE on input------------------------------------------- YES
BLANK if zero----------------------------------------------------- NO
CALCULATION order row or column----------------------------------- ROW
DISPLAY 80,64,40, columns (8,6,4)--------------------------------- 64
FORM feed between pages------------------------------------------- YES
GAPS between lines on printer------------------------------------- 0
LINES per page of printer paper----------------------------------- 66
MONETARY symbol (e.g. f,$)---------------------------------------- £
PRINTER paper width (characters)---------------------------------- 80
```

Figure 4.2 The design command.

If you now change the contents of cell B2 you will see that there is no change in the contents of any of the other cells.

You can also force a recalculation of all the formulae in the grid at any time by using the **Xecute** command. While you have the auto-calculate turned off, try using this command. Make sure that the command menu is displayed in the control area (press **F3**) and then press the X key. The values in the cells of the grid will be recalculated.

Before you go any further you should restore the auto-calculate facility by using the **Design** command again. Select the auto-calculate option by pressing the A key, as before, and leave the command by pressing **ENTER**.

CHAPTER 5
THE EXAMPLES

The following sections illustrate the use of Abacus by developing a number of examples. In addition to explaining the way a number of features work, the examples have been chosen to show some of Abacus's wide range of applications. The best way to learn about Abacus is to use it. The examples have been written with this in mind.

You are recommended to work through all the examples yourself, typing them in as you go along. Each contains some additional information, as well as giving more practice with the topics covered in earlier examples. You may well be able to think of modifications and improvements and they should give you ideas about how to construct applications of your own.

In all the examples in this chapter, text, numbers and formulae are shown exactly as you would type them in. If a cell range is required, it will be given in brackets at the end of the line. In many cases the range you need will be the one that Abacus suggests and you can select it simply by pressing **ENTER**. In other cases you will have to type in the range yourself. If the cursor needs to be positioned on a particular cell, its cell reference is shown in square brackets at the beginning of the line – do not type in any such cell reference. For example, the line:

```
[A4] row=month(col()-1)        {columns B to M}
```

should be read as:

> move the cursor to cell A4, and then type in
> `row=month(col()-1)`
> If necessary, modifying the range suggested by ABACUS to be from column B to column M.

Where you have to type in an explicit range reference, e.g. b3:e15, it will be given in that form.

When commands are given in full they are shown exactly as they will appear on the screen. Remember that you only need to type in the first letter of each option - the rest is filled in by Abacus. If you want to use the default option you should just press **ENTER**.

Each example assumes that you start with a completely blank grid. If necessary clear the grid with the **Zap** command before starting to type in the example.

CASH FLOW MODELLING

This is a more complete version of the simple cash flow example of Chapter 4. When you have finished the grid it should look like Figure 5.1.

The first two cell entries produce an underlined title for the grid.

```
[C1] "CASH FLOW
[C2] rept("=",len(c1))
```

The second entry uses the **rept( )** function which needs two arguments. The first is text, or a reference to a cell which shows a text value and the second is numeric. The function produces that number of repetitions of the first character of the text. In this case it underlines the title with '=' signs, to the exact length of the title. If you decide to change the title there is no need to alter the formula in cell C2 since it uses the **len( )** function to read the length of the text in cell C1.

```
[A4] row=month(col()-1)      {columns B to M}
[A5] row=rept("-",width()+1)      {columns A to M}
```

These row entries produce month headings, and rule a line across the whole of the used part of the grid. The function **width( )** gives the width, in character spaces, of each column. It can therefore be used to rule lines across a grid with columns of different widths. There is one extra character space separating each column of the grid, which is why the additional +1 is needed.

```
[A6] "SALES
[B6] 4000
[C6] row=sal.jan*1.02     {columns C to M}
```

Those entries fill in the sales figures for the year, assuming that the January sales were 4000 and that sales are increasing at 2% per month.

```
   |        A         |    B     |    C     |    D     |    E
 1 |                        CASH FLOW
 2 |                        =======
 3 |
 4 |                  January    February   March      April
 5 |-------------------------------------------------------------
 6 | SALES              4000.00    4080.00    4161.60    4244.83
 7 | COST OF SALES      2750.00    2790.00    2830.80    2872.42
 8 |-------------------------------------------------------------
 9 | GROSS PROFIT       1250.00    1290.00    1330.80    1372.42
10 |
11 | EXPENSES
12 |            wages    700.00     700.00     700.00     700.00
13 |      advertising    100.00     100.00     100.00     100.00
14 |             rent    200.00     200.00     200.00     200.00
15 |      electricity     50.00      50.00      50.00      50.00
16 |     depreciation     90.00      90.00      90.00      90.00
17 |-------------------------------------------------------------
18 | TOTAL EXPENSES     1140.00    1140.00    1140.00    1140.00
19 |-------------------------------------------------------------
20 | NET PROFIT          110.00     150.00     190.80     232.42
21 |=============================================================
```

Figure 5.1. The completed cash flow grid (first five columns)

```
[A7] "COST OF SALES
cos=sal*0.5+750   {columns B to M}
```

(The costs are assumed to be half of the selling price plus a fixed amount of £750.00.)

```
[A8] row=a5   {columns A to M}
[A9] "GROSS PROFIT
gro=sal-cos   {columns B to M}
```

This rules off the grid again and calculates the monthly gross profit figures.

```
[A11] "EXPENSES

[A12] "wages
row=700   {columns B to M}

[A13] "advertising
row=100   {columns B to M}

[A14] "rent
row=200   {columns B to M}

[A15] "electricity
row=50   {columns B to M}

[A16] "depreciation
row=90   {columns B to M}
```

These entries fill in the expense figures, assuming them to be constant throughout the year. You can, of course, change the expense headings and amounts to suit yourself. You can include more or fewer entries, as long as you make the necessary changes to the cell references in the rest of the example. You may want to have different values for each month, but it is faster to set up the table with fixed values and modify them later.

```
[A17] row = a5   {columns A to M}
[A18] "TOTAL EXPENSES
[B18] row=sum(col)   {rows 12 to 16, columns B to M}
[A19] row=a5   {columns A to M}
```

You now have the totals of the monthly expenses.

The **sum( )** function adds the contents of all the numeric cells in the range specified as its argument. All empty cells, together with those containing text, are ignored. The range could be given as an explicit range reference – B12:B16 for example. In this

case, however, each range is only a single column so we have used the range specifier 'col'. All you need to do is to answer the range questions asked by Abacus, and just press **ENTER** if the suggested range is what you want.

Note that this formula uses the range identifiers **row** and **col** in the two different ways. Firstly, **row** is used to indicate that the formula is to be placed in several cells of the current row. Secondly, **col** is used to specify the range of cells over which the addition should take place. Both of the range identifiers need you to confirm (or change) their beginning and end points. In this case Abacus deals with the range for the **sum( )** function first.

```
[A20] "NET PROFIT
net=gross-tot   {columns B to M}
[A21] row= rept("=",width()+1)   {columns A to M}
```

The table is now complete, with the net profit figures calculated as the difference between the gross profits and the total expenses. All you have to do now is to adjust the appearance of the table by using a few commands. Remember to press **F3** each time you want to use a command.

First we change the width of column a (note that the **Grid** command has its own menu of options).

```
Grid>Width, 15 FROM a TO a
```

Then we change the justification and numeric display format for a few cells:

```
Justify,Cells,Text,Right,Range a4:m4
Justify,Cells,Text,Right,Range a12:a16
Units,Cells,Decimal,Decimal places 2,Range Range a1:m21
```

We have chosen to display the figures in decimal format, with two decimal places. If you prefer the pound sign to appear you should replace the last command by

```
Units,Cells,Monetary,Minus sign,Range a1:m21
```

It is very simple to alter any of the figures. Suppose you want to increase the February advertising figure. All you have to do is press **F5** (go to a cell) and type the cell reference

```
feb.adv
```

The cursor will move to that cell and you can type a new value.

Remember that the sales figures were calculated by a formula which assumed a 2% increase each month. If you change one of these cells to a numeric value you will destroy the formula in that cell. The formulae in the other cells of the row will, however, be unchanged. The amounts in the following cells will still increase by 2% per month, starting from the new value.

MULTIPLICATION TABLES

This simple example may prove useful to a child who wants to learn the multiplication tables. It lets you request a particular table and then displays it.

When you have typed in the example you use it by forcing a recalculation of the grid with the **Xecute** command, i.e. you type:

F3 X

Abacus then asks you to type in a number and displays the corresponding multiplication table.

The table in Figure 5.2 shows an example of the display it produces.

First title the application as normal:

```
[B1] "MULTIPLICATION TABLES
[B2] rept("=",len(b1))
```

The next three lines give a heading to the table.

```
[B3] "The
[C3] askn("Which multiplication table do you want")
[D3] "times table
```

Here we have used the **askn( )** function to request input; it allows you to choose which table you want, by typing in a number.

```
   |        A        |   B    |   C    |   D    |   E    |   F
 1 |                  MULTIPLICATION TABLES
 2 |                  ====================
 3 |                       The    7     times table
 4 |                        1 *        7        =        7
 5 |                        2 *        7        =       14
 6 |                        3 *        7        =       21
 7 |                        4 *        7        =       28
 8 |                        5 *        7        =       35
 9 |                        6 *        7        =       42
10 |                        7 *        7        =       49
11 |                        8 *        7        =       56
12 |                        9 *        7        =       63
13 |                       10 *        7        =       70
14 |                       11 *        7        =       77
15 |                       12 *        7        =       84
```

Figure 5.2 A multiplication table.

This function takes a text string as its argument and displays the text in the input line, followed by a question mark. It then waits for you to type in a number, ending with **ENTER**. The number that you type in will be displayed in the cell which contains **askn( )**.

Note that **askn( )** will not wait for input during a normal auto-calculation of the grid. It will only display the message and request input when first put the fomula into the cell, or when you force a recalculation of the grid by using the **Xecute** command. Once you have input a value to the cell it will be retained until the next time you force a recalculation with the **Xecute** command.

The remaining grid entries use the column-filling facility to produce the body of the multiplication table.

```
[B4] col=str(row()-3,2,0)+" *"   {rows 4 to 15}
```

This is the most complicated formula of the example. It is used to display the multiplier in each row of the table. The number is converted to a text string so that we can combine it with the multiplication sign and display them both in a single cell.

The **str( )** function converts a number to the equivalent string of digits. It takes three values; the number to be converted, a code for the format (0 = decimal, 1 = exponential, 2 = integer, 3 = general) in which the number is to be displayed, and the number of decimal places to be shown. In this case the number is converted to integer format.

In this case the value is obtained from the expression '**row( )**−3', whose value is 1 in row four, 2 in row five, and so on, up to 12 in row 15. The next value (2) selects display as an integer (whole number). The third number normally specifies how many decimal places are to be used. Its value must always be given but is ignored for integers. It has been given a value of zero (any other value could have been used).

Finally the result is *concatenated* (the correct term for adding one text string to another) with the string *, so that both the multiplier and the multiplication sign are displayed in a single cell.

```
[C4] col=$c3   {rows 4 to 15}
```

Column C contains copies of the value typed in in answer to the **askn( )** function. The cell reference is preceded by a $ sign to make it an *absolute cell reference*. When you have entered the formula, move the cursor up and down the cells of column C and look at their contents. You will see that they all contain the cell reference $C3. The reference has not been adjusted in each row. An absolute cell reference always refers to one particular cell, from any position in the grid. You can make any cell reference absolute by adding a leading $ sign.

```
[D4] col="="   {rows 4 to 15}
[E4] col=$c3*(row()-3)   {rows 4 to 15}
```

These last two column entries are almost self-explanatory. They are used to produce the equals sign and the answer for each row of the table. The last formula multiplies

the value from the **askn( )** function in cell C3 (another absolute cell reference) by the **row( )−3** expression which, as we saw earlier, gives a value of 1 in row four, 2 in row five, up to 12 in row fifteen.

We now need to use a few commands to change the display of the table to a more convenient form. Use the following commands:

```
Justify,Cells,Text,Right,Range b3:b15
Justify,Cells,Text,Right,Range d4:d15
Justify,Cells,Numbers,Centre,Range c3
Grid>Width, 5 FROM b TO b
Grid>Width, 3 FROM c TO c
Grid>Width, 2 FROM d TO d
Grid>Width, 4 FROM e TO e
```

You use the table by forcing a recalculation of the grid with the **Xecute** command. The text of the **askn( )** function will appear in the input line – and you should type in a number between one and twelve.

CHEQUE BOOK RECONCILIATION

This example allows you to keep a check on your bank account. You enter details of your cheques in the spaces provided. At the end of the month you add the details of your salary, standing orders etc. You are then provided with a balance which you can compare with your bank statements.

The result, with a few figures added, is shown in Figure 5.3.

```
  |       A                        |      B       |      C
1 |                CHEQUE BOOK RECONCILIATION
2 |                ========================
3 |
4 |                                    Month          January
5 |
6 | Opening balance                                200.00
7 | Salary                                         527.35
8 | Miscellaneous income                             0.00
9 |
10|            CREDIT                              727.35
11|                                                =====
12|
13| Standing orders                                              130.00
14| Charges                                                        0.00
15|
16| Cheques                    Date       Cheque no.             Amount
17|                          3/01/84          123456              50.00
18|                         10/01/84          123457              50.00
19|                         14/01/84          123458              32.21
20|                         17/01/84          123459              50.00
21|                         24/01/84          123460              50.00
22|                         31/01/84          123461              50.00
23|                            ---               ---                ---
24|                            ---               ---                ---
25|                            ---               ---                ---
26|                            ---               ---                ---
27|
28|            DEBIT                                             412.21
29|                                                              =====
30| Closing balance                                315.14
31|                                                =====
```

Figure 5.3 Cheque book reconciliation

```
[B1] "CHEQUE BOOK RECONCILIATION
[B2] rept("=",len(b1))

[C4] "Month
[D4] askt("Enter month")
```

The **askt( )** function works in the same way as **askn( )**, but the expected input is text instead of a number. When you use **Xecute** Abacus will display the message in the input line and then wait for you to type in some text. You should type in the name of the month for your balance.

```
[A6] "Opening balance
[A7] "Salary
[A8] "Miscellaneous income
[C6] askn(a6+" for "+$d4)
```

The prompt string for **askn( )** is constructed from the text of other cell entries, using both relative and absolute cell references.

We now use the **Echo** command to copy the formula from cell C6 into cells C7 and C8. Instead of typing the range reference C7:C8, we can use the range identifier **col**.

```
Echo,cell c6,over range col (rows 7 to 8)
```

```
[B10] "CREDIT
[C10] sum(col)    {rows 6 to 8}
```

Cell C10 is used to contain the total of all credits for the month. This cell is labelled; its reference is "credit.month".

The cell's contents are calculated using the **sum( )** function which we met in the first example in this chapter. This function adds the numeric contents of all cells in the range specified by its argument. Remember that it ignores any cell in the range that is empty or that contains text.

In this case we have again used it as sum(col), which specifies that the cells to be summed lie in the current column. As normal, Abacus asks you to specify the exact range, suggesting reasonable values based on your previous work.

```
[C11] rept("=",len(str(credit.month,0,2)))
```

Cell C11 underlines the total, using the usual **rept( )** and **len( )** functions. In this case, however, we do not know in advance the number of characters to underline. We therefore have to convert the number to a string of characters with the **str( )** function, assuming that it is to be shown in decimal format with two decimal places. The length of this string gives the correct number of characters to underline.

```
[A13] "Standing orders
[A14] "Charges
[D13] askn(a13+" for "+$d4)
[D14] askn(a14+" for "+$d4)
```

These allow you to enter the monthly debits in response to prompts, using **askn( )** in the same way as described earlier.

```
[A16] "Cheques
[B16] "Date
[C16] "Cheque no
[D16] "Amount
[B17] row="---"    {columns B to D}
```

These cells set up an area of the grid which you will later use to enter the details of your cheques.

```
[B28] "DEBIT
[D28] sum(col)    {rows 13 to 26}
```

This calculates the total of the debits. Remember that **sum( )** only adds numeric values in the cells of the specified range. Cells containing text, and empty cells, are not included. The sum will therefore ignore all unused entries in the list of cheques, as well as the table heading in column D.

```
[A30] "Closing balance
[C30] credit.month-debit.amount
```

The calculation of the closing balance completes the grid entries. You should now use the commands to tidy up the appearance of the application.

First we can use the **Echo** command to fill the rest of the cheque table and complete the underlining of the totals. This command makes copies of the contents of a single

cell into all the cells in a range. The first of the following three uses, for example, copies the contents of cell B17 into all the cells in a rectangle whose top left and bottom right corners are B18 and D26 respectively.

```
Echo,Cell b17,over range b18:d26
Echo,Cell c11,over range d29:d29
Echo,Cell c11,over range c31:c31
```

Next we need to set the numeric display to decimal, with two decimal places, for the whole of the application, with integer format for the cheque numbers:

```
Units,Cells,Decimal,Decimal places 2,Range a1:d30
Units,Cells,Integer,Minus sign,Range c17:c26
```

We have already explained that empty cells do not exist as far as Abacus is concerned and so a change of format will therefore only affect non-empty cells. We can fill the cheque table with '---' before making the change to ensure that these cells are changed to decimal format. An alternative method is to change the default format.

Finally we can modify the justification of the text, including the underlining, to improve the final appearance.

```
Justify,Cells,Text,Right,Range b16:d26
Justify,Cells,Text,Right,Range c11
Justify,Cells,Text,Right,Range d29
Justify,Cells,Text,Right,Range c31
```

The part of the grid that is used is too large for it all to be visible in the window at once. In order to see the final results, together with the values entered via the **askt( )** and **askn( )** functions, you might like to use the split window facility. The **Window** command splits the window, either vertically or horizontally, into two windows, using the position of the cursor to determine the position of the split.

A vertical split is most suitable for this grid and you can set it up by moving the cursor to the centre of the window and then using the command:

```
Window,Vertical,Separately
```

You can move the cursor from one window to the other by pressing **F4**. For this example you should use the cursor to adjust the left hand window to show cell A1 at its top left corner, and cell B15 at the top left corner of the right hand window.

STANDARD DEVIATION

This example calculates the mean and standard deviation of a set of numbers. It makes use of the labelling facilities of Abacus so that the formulae used in the calculations are mostly self-explanatory.

```
  |      A     |    B     |    C     |    D     |    E
1 |            STANDARD DEVIATION
2 |            ==================
3 |
4 |            Value      Deviation  Square of dev.
5 |             5.00       -4.50       20.25
6 |             6.00       -3.50       12.25
7 |             7.00       -2.50        6.25
8 |             8.00       -1.50        2.25
9 |             9.00       -0.50        0.25
10|            10.00        0.50        0.25
11|            11.00        1.50        2.25
12|            12.00        2.50        6.25
13|            13.00        3.50       12.25
14|            14.00        4.50       20.25
15|
16|  Mean      9.50        Variance       8.25
17|                        Std. Dev.      2.87
18|                                    ---------
```

Figure 5.4 Standard deviation calculation.

In addition it uses a grid layout which requires calculation in column order, rather than the normal row order.

In general, a formula should only refer to cells that are in the region above and to the left of the cell containing the formula, including the row and column containing the formula.

If you do not follow this rule, as in this example, it is likely that the results may be incorrect. In most cases you can obtain a correct result by forcing a recalculation of the grid with the **Xecute** command or, as in this case, calculating the grid in column order.

```
[B1] "STANDARD DEVIATION
[B2] rept("=",len(b1))
[B4] "Value
[C4] "Deviation
[D4] "Square of dev.
[B5] col=row()   {rows 5 to 14}
```

This last formula inserts a set of dummy values in the cells of column B for testing the application. When the grid entries are complete you can replace them with other values. The table described in this example will only hold ten values – you can change this to cope with more if you want.

```
[A16] "Mean
[B16] ave(value)   {rows 5 to 14}

deviation=value-$mean.value   {rows 5 to 14}
square=dev*dev   {rows 5 to 14}

[C16] "Variance
[D16] ave(square)   {rows 5 to 14}
```

These formulae show that the variance of a set of numbers is defined as the average of the squares of the deviations from the mean,

```
[C17] "Std. Dev.
[D17] sqr(variance)   {columns D to D}
```

and that the standard deviation can be calculated as the square root of the variance.

```
[D18] rept("-",len(str(std.sq,3,0)))
```

The numbers in this example are left in general format so that it can handle any range of values. The underlining uses the length of the text string corresponding to the number in the cell above (with cell reference "std.sq") expressed in general format.

You can improve the appearance of the display by changing to centre justification for the text in the range B4:D4, and using left justified numbers in the range B16:D17.

If you try using this example by putting different values in the cells of column B, you will find that it does not give the correct answers. The reason is that the recalculation of the grid is performed row by row, from the top downwards. Any alteration you make will therefore be worked out on the basis of an incorrect mean value (since the new mean will not be calculated until after the deviations from the mean). The solution is to make the recalculation of the grid be in column order, from left to right. You do this with the **Design** command.

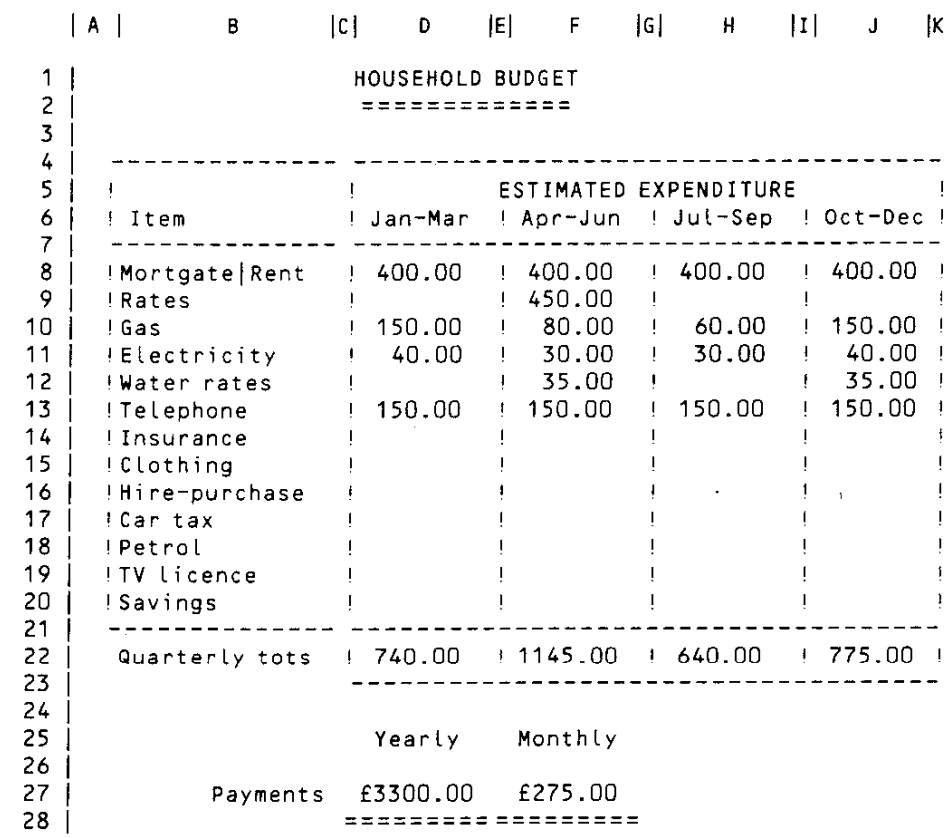
Use the 'C' option to change the column order and leave the command by pressing ENTER, as indicated in the control area. When you next change a value in column B, the calculation will be correct, since the new mean is now calculated before the deviations. Although this ability to change the order of calculation is very useful, you should not get into the habit of using it too often – calculating in column order is much slower than row order.

If you save a grid to a Microdrive file, the current settings of all the **Design** options are saved with it and they are used whenever you reload the file.

A HOUSEHOLD BUDGET

This example will allow you to plan your household expenditure over the year. You can enter your estimated expenditure under a number of headings for each quarter. You are then provided with quarterly totals, your expenditure for the whole year and the averaged monthly cost.

Do not type any numbers into the table until you have completed it. This allows you to change the form of numeric display, with the Defaults option of the **Units** command, as described later.



```
    | A |        B        |C|     D     |E|     F     |G|     H     |I|     J     |K
  1 |                        HOUSEHOLD BUDGET
  2 |                         =============
  3 |
  4 |  --------------- ------------------------------------
  5 |  !               !           ESTIMATED EXPENDITURE          !
  6 |  ! Item          ! Jan-Mar   ! Apr-Jun   ! Jul-Sep   ! Oct-Dec !
  7 |  --------------- ------------------------------------
  8 |  !Mortgate|Rent  !  400.00   !  400.00   !  400.00   !  400.00  !
  9 |  !Rates          !           !  450.00   !           !          !
 10 |  !Gas            !  150.00   !   80.00   !   60.00   !  150.00  !
 11 |  !Electricity    !   40.00   !   30.00   !   30.00   !   40.00  !
 12 |  !Water rates    !           !   35.00   !           !   35.00  !
 13 |  !Telephone      !  150.00   !  150.00   !  150.00   !  150.00  !
 14 |  !Insurance      !           !           !           !          !
 15 |  !Clothing       !           !           !           !          !
 16 |  !Hire-purchase  !           !           !           !          !
 17 |  !Car tax        !           !           !           !          !
 18 |  !Petrol         !           !           !           !          !
 19 |  !TV licence     !           !           !           !          !
 20 |  !Savings        !           !           !           !          !
 21 |  --------------- ------------------------------------
 22 |   Quarterly tots !  740.00   ! 1145.00   !  640.00   !  775.00  !
 23 |                  ------------------------------------
 24 |
 25 |                    Yearly     Monthly
 26 |
 27 |          Payments  £3300.00   £275.00
 28 |                   ========= =========
```

Figure 5.5 Home budget example.

```
[D1] "HOUSEHOLD BUDGET
[D2] rept("=",len(d1))
```

Now we can set up the structure of the table with its ruled divisions.

```
[A4] row=rept("-",width()+1)    {columns A to K}
[A5] col="!"    {rows 5 to 20}
```

The following commands complete the table structure.

```
Grid>Width, 16 FROM b TO b
Grid>Width, 8 FROM d TO j
Grid>Width, 1 FROM a TO a
Grid>Width, 1 FROM c TO c
Grid>Width, 1 FROM e TO e
Grid>Width, 1 FROM g TO g
Grid>Width, 1 FROM i TO i
Grid>Width, 1 FROM k TO k
```

```
Echo,Cell a5,over range c5:c22
Echo,Cell a5,over range e6:e22
Echo,Cell a5,over range g6:g22
Echo,Cell a5,over range i6:i22
Echo,Cell a5,over range k5:k22
Echo,Cell a4,over range b7:j7
Echo,Cell a4,over range b21:k21
Echo,Cell a4,over range c23:k23
```

```
[A7] "!-
[F5] "ESTIMATED EXPENDITURE
[B6] "Item
[D6] "Jan-Mar
[F6] "Apr-Jun
[H6] "Jul-Sep
[J6] "Oct-Dec
```

FOR SERVICE MANUALS
CONTACT:
MAURITRON TECHNICAL SERVICES
www.mauritron.co.uk
TEL: 01844 - 351694
FAX: 01844 - 352554

```
[B8] "Mortgage/Rent
[B9] "Rates
[B10] "Gas
[B11] "Electricity
[B12] "Water rates
[B13] "Telephone
[B14] "Insurance
[B15] "Clothing
[B16] "Hire-purchase
[B17] "Car tax
[B18] "Petrol
[B19] "TV licence
[B20] "Savings

[B22] "Quarterly tots

[D22] sum(col)   {rows 8 to 20}
[F22] sum(col)   {rows 8 to 20}
[H22] sum(col)   {rows 8 to 20}
[J22] sum(col)   {rows 8 to 20}

[D25] "Yearly
[F25] "Monthly
[B27] "Payments

[D27] sum(d22:j22)
[F27] year.pay/12
[D28] rept("=",len(str(year.pay,0,2))+1)
[F28] d28
```

Note that the underlining of the two final figures assumes a monetary format. The length of the underlining is for a number in decimal format, with two decimal places, plus one (for the currency symbol).

You also should use a few more commands, to justify text right in the range B22:B27 (quarterly tots and payments) and to justify numbers left over the cells containing the yearly and monthly payments.

You must also modify the numeric display format. Since many of the cells are still empty, simply changing the format will have no effect. You must change the default format of the cells to make the effect permanent.

The following command will change the display default to monetary units over the whole of the budget application.

```
Units,Defaults,Monetary,Minus sign
```

The display of Figure 5.5 uses decimal format, with two decimal places, except for the yearly and monthly payments, which are in monetary format. The appropriate commands are:

```
Units,Defaults,Decimal,Decimal places 2
Units,Cells,Monetary,Minus sign,Range d27:f27
```

This last command can use the Cells option since the cells concerned already exist.

You can enter values in this table by moving the cursor to the appropriate cell and typing in the number. The easiest way of moving the cursor is to press **F5** (Go to cell) and then use a cell label, such as:

```
apr.gas
```

AN AUTO-SCALING BAR CHART

The chart displays twelve values, labelled by month. The values are read from twelve cells above the chart. The vertical scale is adjusted automatically to make sure that all values will fit the display. It is only suited to displaying positive values.

First you should set the column widths to five in column A, one in column B and three in columns C to N, using the Width option of the **Grid** command.

```
[C2] row=0   {columns C to N}
```

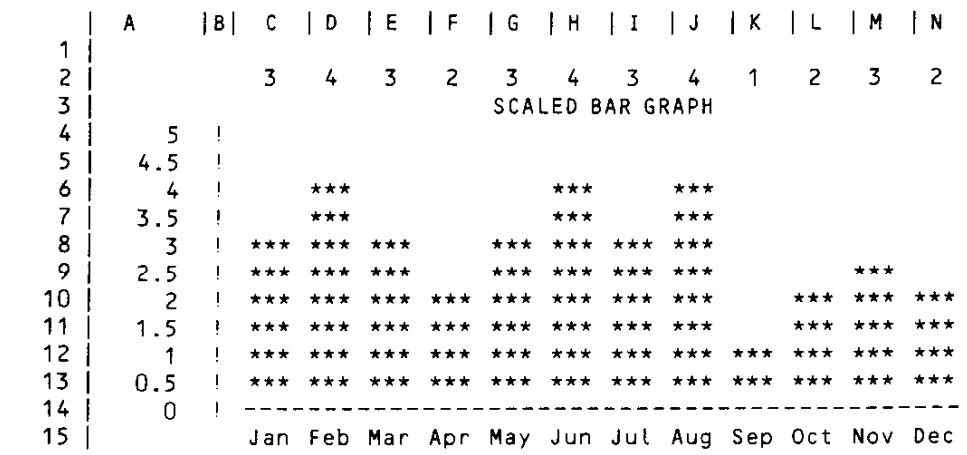


Figure 5.6 A scaled bar chart

Row two will contain the values to be displayed -- initially filled with dummy dots.

```
[F3] "SCALED BAR GRAPH

[P2] int(max(c2:n2)/5+1)*5
[Q2] int(min(c2:n2)/5)*5
```

Cells P2 and Q2 contain the maximum and minimum values for the vertical scale of the graph. These cells are chosen so that they do not appear in the final display of the chart. Their initial values are five and zero respectively.

The **max( )** function finds the maximum, or largest, numerical value in the range of cells specified by its argument. Similarly, the **min( )** function finds the minimum, or smallest, value in the range.

Let us first examine the formula in cell Q2. The **min( )** function finds the minimum, or smallest, value in the specified range which is then divided by five. The **int( )** function then removes the fractional part of the result of the division. If, for example, the minimum value is 13, dividing by 5 gives a value 2.6, and **int(2.6)** is 2. When this is multiplied by 5 we end up with a value of 10, which is the largest multiple of 5 that is less than the minimum.

The formula in cell P2 is similar, except that it finds the largest value in the range and adds 1 to the number before the final multiplication by 5. If, as an example, we assume that the maximum value is 21, you can verify that the formula will give a value of 25 -- the smallest multiple of 5 that is greater than the maximum.

The two values in these cells will therefore always bracket the values in the cells from C2 to N2. Their difference is always a multiple of five.

The next formula displays the vertical scale of the graph in column A.

```
[A4] col=$q2+(14-row())*($p2-$q2)/10    {rows 4 to 14}
```

The interval between successive numbers in the scale is **(P2 -Q2) / 10**. Note that we made the difference between the contents of P2 and Q2 a multiple of five so that this interval always has a simple value.

This interval is multiplied by a number **(14-row( ))** which starts at zero in row fourteen and increases by steps of one to a value of ten in row four. The result is added to the smallest value, from cell Q2, to produce the number for each cell.

The net result is that the value in cell Q2 is displayed in A14, the value from P2 is displayed in A4 and the intervening cells contain a set of equally spaced values between these two limits.

```
[B4] col="!"    {rows 4 to 14}
[B14] row=rept("-",width()+1)    {columns B to N}
[C15] row=month(col()-2) ( to 3)    {columns C to N}
```

These draw the axes for the chart and add the horizontal axis labels, using the months of the year. Note that we have used the string slicing operator, similar to that of SuperBASIC, to display only the first three characters of each month.

```
[C4] if(index(1,row()) > index(col(),2) ,"","***")
```

This is the formula that does all the work of producing the bars themselves. It must be copied into every cell in the display area:

```
Echo,Cell c4,over range c4:n13
```

The formula itself needs some explanation. It uses the **if( )** function to decide whether to display part of a bar. The **if( )** function takes three arguments. The first is an expression which must give a numeric result. If this result is non-zero the cell displays the second argument, which may be text or numeric. If, however, the result is zero the third argument is displayed in the cell. Again this may be text or numeric.

In each cell the formula compares the number in column one of that row (the value labelling the vertical axis) with the number in row two of that column (the value to be displayed in the graph). If the axis label is greater than the display value, the condition is true (it evaluates to 1) and nothing is displayed. If the axis label is less than or equal to the display value, the condition results in a value of zero, and three asterisks are shown in the cell. The net result is that a bar is drawn to the correct height in each column.

Since a single formula is used for all the cells in the display, the cell reference can be neither absolute nor relative. The reference to the display values must change as we move from column to column (i.e. it must be relative along a column) but must always refer to row two as we move down, from row to row. We need a form of cell reference which is relative with respect to columns, but absolute with respect to rows.

Fortunately the **index( )** function can be used to produce this effect. It takes two parameters, a column number and then a row number, returning the contents of the specified cell. With this we can construct any combination of absolute and relative references. For example:

Function	Column Ref.	Row Ref.
index(5,5)	absolute	absolute
index(col( ),5)	relative	absolute
index(5,row( ))	absolute	relative
index(col( ),row( ))	relative	relative

The function **index(col( ),2)** therefore returns the contents of the cell in row two of the current column, and **index(1,row( ))** returns the contents of the cell in column one (A) of the current row.

Try putting different values in cells C2 to N2 and see what effect they have on the display.

MORTGAGE CALCULATOR

This example enables you to calculate the monthly payments due on a repayment mortgage. You are asked to type in the amount of the loan, the interest rate, the length of the loan in years and the month of the first payment. The required repayments are calculated and displayed, together with a complete repayment table for the whole period of the loan. This table shows you the outstanding sum at the beginning of each month until the loan is repaid.

Several of the calculations in the grid make use of values that are input by use of the **askn( )** function.

Mortgage Repayment Calculations

In this section we shall produce the part of the grid that accepts your input and calculates the monthly repayments. When you have typed in the formulae and added a few figures in response to the **askn( )** functions, it should look like Figure 5.7.

```
[C1] "MORTGAGE REPAYMENT CALCULATOR
[C2] rept("=",len(c1))

[B4] "Loan
[C4] askn("Amount of loan")
```

The next three entries request the input of the interest rate. The original input is to a cell (H4) well away from the displayed portion of the grid so that you do not normally see it. You type in a percentage value, e.g. you type 12 to mean 12%. The value needed by the rest of the formulae is a fractional value (e.g. 12% must be converted to 0.12) and this is calculated from the input value by the formula in cell C5.

```
  |      A      |     B      |     C      |     D      |     E
1 |                MORTGAGE REPAYMENT CALCULATOR
2 |                =========================
3 |
4 |              Loan         £25,000.00
5 |              Int rate     14.00%                  Mnth
6 |              Term         25           Start       4
7 |                                                    (April)
8 |                                        REPAYMENTS
9 |                                        ---------
10|                           Annual         £3637.46
11|                           Monthly         £303.12
12|                                        --------
```

Figure 5.7 Calculating the repayments.

```
[H4] askn("Percentage interest rate")
[B5] "Int rate
[C5] h4/100

[B6] "Term
[C6] askn("Period of loan in years [maximum 35]")

[E5] "Mnth
[D6] "Start
[E6] askn("Month of first payment [Jan=1, Feb=2, etc]")
[E7] '(' + month(e6) + ")"
```

In this last formula we enclose the literal text with single quotation marks. If the first character had been a double quote, Abacus would have interpreted the following characters as text input, rather than a formula.

```
[D8] "REPAYMENTS
[D9] rept("-",len(d8))
[C10] "Annual
[D10] mor.loan*mor.int/(1-(1+mor.int)^(-mor.term))
```

This formula, which calculates the annual repayment, assumes that the interest is calculated annually and added to the loan before the twelve monthly repayments are made.

```
[C11] "Monthly
[D11] ann.rep/12
[D12] d9
```

The grid is now sufficiently complete to calculate mortgage repayments. Try using the **Xecute** command and enter the figures requested, so that you can see it working.

To make the example look better, we can change the format of some of the numbers with the **Units** command. In this example there is no need to alter the default numeric format since you do not need to make new entries in any grid cell once the application is completed.

```
Units,Cells,Percentage,Decimal places 2,Range c5
Units,Cells,Monetary,Minus sign,Range c4
Units,Cells,Monetary,Minus sign,Range d10:d11
```

In addition it improves the appearance if we move the numbers in rows 4, 5 and 6 to the left hand side of the cells:

```
Justify,Cells,Numbers,Left,Range c4:e6
```

Mortgage Repayment Table

This section describes how you can add a repayment table to the mortgage calculator. The first part of a repayment table for the values appearing in Figure 5.7 is illustrated in Figure 5.8.

```
    |       A       |     B      |     C      |     D      |     E
 15 |                 REPAYMENT TABLE
 16 |                 =============
 17 |
 18 |                 Year                    1            2            3
 19 |                 ----------------------------------------------------
 20 |                 April          28500.00     28343.30     28164.65
 21 |                 May            28196.88     28040.17     27861.53
 22 |                 June           27893.76     27737.05     27558.41
 23 |                 July           27590.63     27433.93     27255.29
 24 |                 August         27287.51     27130.81     26952.17
 25 |                 September      26984.39     26827.69     26649.04
 26 |                 October        26681.27     26524.57     26345.92
 27 |                 November       26378.15     26221.44     26042.80
 28 |                 December       26075.03     25918.32     25739.68
 29 |                 January        25771.90     25615.20     25436.56
 30 |                 February       25468.78     25312.08     25133.44
 31 |                 March          25165.66     25008.96     24830.31
 32 |
 33 |                 Year                    1            2            3
 34 |
 35 |  End-of-year balance           24862.54     24705.84     24527.19
```

Figure 5.8 The repayment table.

If you have a mortgage, type in your own figures. Don't spend too much time over the results for the first few years – they make rather depressing reading!

```
[C15] "REPAYMENT TABLE
[C16] rept("=",len(c15))
[B18] "Year
[C18] row=col()-2    {columns C to AK}
[B19] row=rept("-",width()+1)    {columns B to AK}
[B20] col=month(row()-20+$mnth.start)    {rows 20 to 31}
```

These entries set up the headers for the table: now we must add the formulae that will calculate the values. We start with the first item which is the initial amount due. It is calculated by adding the first year's interest to the amount of the loan.

```
[C20] mor.loan*(1+mor.int)
```

Then the rest of the first row is calculated by subtracting the yearly payment and adding the interest for the current year. These values should not be calculated beyond the year in which the loan is repaid and we allow for this by using the **if( )** function. If the year number (given by **col( )−2**) is greater than the term of the mortgage, zero is placed in the cell.

```
[D20]
row=if((col()-2)>$mor.term,0,(c20-$ann.rep)*(1+$mor.int))
                                                {columns D to AK}
```

The remainder of the table can be filled with a single formula. We fill the first cell with a formula which just subtracts the monthly repayment from the amount in the cell above. Again we use the **if( )** function to prevent the calculations extending beyond the year in which the loan is repaid.

```
[C21] if((col()-2)>$mor.term,0,c20-$mon.rep)
```

You can then use the **Echo** command to copy the formula from cell C21 to the range C21:AK31.

```
Echo,Cell c21,over range c21:ak31
```

We can now complete the table by adding a final row to give the outstanding balance at the end of each year. It is probably a good idea to add a copy of the year, from row 18, for easy reference.

```
[B33] row=year.term    {columns B to AK}
[A35] "End-of-year balance
[C35] row=if((col()-2)>$mor.term,0,c31-$mon.rep)
                                                {columns C to AK}
```

The entire table, and the end-of-year balances should be set to either monetary format or to decimal format with two places of decimals. The ranges for these changes are C20:AK31 and C35:AK35.

FOURIER ANALYSIS

The French scientist Fourier showed that a repetitive wave of any shape can be built up from a set of sine or cosine waves of the correct amplitudes and frequencies. The building up of complex waves from pure sine and cosine waves is known as Fourier synthesis and is employed, for example, in many of the music synthesisers in use today.

The opposite process, decomposing a complex wave shape into a number of pure sine and cosine waves, is known as Fourier analysis. This example allows you to perform a Fourier analysis of any shape of wave. All you have to do is type in the height of the wave at sixteen equally spaced intervals and let the formulae in the grid do the rest. The formulae assume that the wave repeats its shape after the sixteenth value, i.e. that the seventeenth value is the same as the first, the eighteenth is the same as the second, and so on.

Calculating the Fourier Transform

Since the calculation takes an appreciable time it is worth turning off the auto-calculate, by use of the **Design** command, before typing in the example.

```
[C1] "FOURIER ANALYSIS
[C2] rept("=",len(c1))

[B3] "Function:
[A7] "Input
[A8] "Values
```

The Cosine Components

The input values are placed in the sixteen cells from B9 to B24 inclusive.

```
col=row()-9    {rows 9 to 24}
```

We shall now set up the headings for the table which will calculate the cosine components of the wave. The result contains the amounts of all cosine-like waves in the input.

```
[E3] "Transform:
[E4] "Cosine
[D6] "Cycles
row=col()-5    {columns E to T}
[D8] "Sample
```

Surprisingly, the entire cosine transformation can be performed by a single formula. In each row the input value is multiplied by the cosine of an angle (in radians) which is calculated as follows:

angle = 2 * pi( ) * rownumber * colnumber / 16

The row number and column number are the values given in the row labelled 'Cycle' and the column labelled 'Sample' respectively. They each count up from zero to fifteen. The final divisor is simply the number of points in the input (or output).

```
[E9] index(2,row())*cos(pi()*(row()-9)*(col()-5)/8)
```

Now use the **Echo** command to copy the contents of cell E9 to the cells in the range from E10 to T24.

The final result is calculated by summing the contents of each column to produce the sixteen output values.

```
[A26] "Components
[E26] row=sum(col)    {rows 9 to 24, columns E to T}
```

The Sine Components

The calculation of the sine components follows exactly the same pattern as for the cosine ones. The resulting values are the amounts of all sine-like waves in the input.

```
[X4] "Sine
[X6] row=col()-24    {columns X to AM}
[X9] index(2,row())*sin(pi()*(row()-9)*(col()-24)/8)
                                        {columns X to AM}
```

Now **Echo** the contents of cell X9 over the range from X10 to AM24, to fill in the rest of the table, and **Echo** the contents of cell C9 to column V, from V9 to V24 (this makes a copy of the 'Sample' values).

```
[X26] row=sum(col)    {rows 9 to 24, columns X to AM}
```

The Power Spectrum

Any input wave that is not a pure sine or pure cosine wave will generally produce components in both the sine and cosine transforms. Furthermore, when you calculate the transform of many types of wave, some of the components will turn out to be negative. In order to obtain results which combine both transforms, and are never negative, we shall make one more calculation. This will add the squares of the sine and cosine components. In the case of a real wave this result shows how much power (energy per second) is present in the wave at each frequency, irrespective of whether it is in the sine or the cosine components. It is usually called the power spectrum (a spectrum records how much of each frequency is present). In this case we shall calculate the square root of the power spectrum, to avoid having too large a range of values for the simple graphical display we are using.

```
[C28] "Power
[E28] row=sqr(cos.comp*cos.comp + sin.comp*sin.comp)
                                                {columns E to T}
```

Graphical Display of the Fourier Transform

The results of this calculation can be made clearer by presenting them in graphical form. If you would like high-quality graphs the best way is for you to use the **Export** command to create files that can be read by Easel, containing the input and output values of the calculation. The following additions to the grid will allow you to see very simple graphical results.

```
   |       A       |     B      |     C      |     D
31 | max =                10.00        80.00 = max
32 | min =               -10.00         0.00 = min
33 | ................*       *
34 | ...............*        ..................*
35 | ............*            *
36 | ...........*             *
37 | .........*               *
38 | .....*                   *
39 | ..*                      *
40 | *                        *
41 | *
42 | *
43 | ..*
44 | .....*
45 | .........*
46 | ...........*
47 | ............*
48 | ..............*
              INPUT                    POWER SPECTRUM
```

Figure 5.9 Simple graphical output

The output graphs are only half the size of the input graph, since the highest detectable frequency is numerically equal to half the number of input points. All the information is present in the first half of the results.

The first part produces a bar graph of the input values.

```
[A30] "Graph
[A31] "max=
[B31] max(col)    {rows 9 to 24}
[A32] "min=
[B32] min(col)    {rows 9 to 24}
[A33] col=rept(".",(func.max-$func.min)
       *18/($func.max-$func.min+1))+"*"    {rows 33 to 48}
```

The second set of entries graphs the power spectrum.

```
[D31] "= max
[C31] max(e28:t28)
[D32] "= min
[C32] 0
[C33] col=rept(".",(index(row()-28,28)-$pow.min)
        *18/($pow.max-$pow.min+1))+"*"   {rows 33 to 40}
```

The next set of entries graphs the cosine components.

```
[F31] "= max
[E31] max(e26:t26)
[F32] "= min
[E32] min(e26:t26)
[E33] col=rept(".",(index(row()-28,26)-$cos.min)
        *18/($cos.max-$cos.min+1))+"*"   {rows 33 to 40}
```

The final set of entries gives a graph of the sine components.

```
[Y31] "= max
[X31] max(x26:am26)
[Y32] "= min
[X32] min(x26:am26)
[X33] col=rept(".",(index(row()-9,26)-$sin.min)
        *18/($sin.max-$sin.min+1))+"*"   {rows 33 to 40}
```

Using the Fourier Transform

As was mentioned earlier, you should put the input values in cells B9 to B24 inclusive. You may try any set of values you like, but here are a few suggestions.

```
[B9] col=10*cos(pi()*(row()-9)/8)      {rows 9 to 24}
[B9] col=10*cos(pi()*(row()-9)/4)      {rows 9 to 24}
[B9] col=10*sin(pi()*(row()-9)/8)      {rows 9 to 24}
[B9] col=10*sgn(cos(pi()*(row()-9)/8))     {rows 9 to 24}
[B9] col=10     {rows 9 to 24}
```

Remember that, since the auto-calculate is turned off, you must use **Xecute** to calculate each result.

A further advantage of including lots of labels is that you can move the window to most of the interesting points in the grid by using the **goto** (**F5**) facility, followed by a cell reference in its label form.

CHAPTER 6
QL ABACUS REFERENCE

THE FUNCTION KEYS

In addition to the standard use of F1, F2 and F3, function keys 4 and 5 are used as follows:

F4 move cursor between the two halves of a split window

F5 Go to a cell

CELL REFERENCES

You can refer to single cells, rows, columns or ranges either by using explicit letter and number references or by using text labels.

Single Cells

A reference to a single cell consists of two parts, a column and a row reference.

There are 64 columns in the grid and they are labelled from A to BL. There are 255 rows, numbered from 1 to 255. Typical cell references are

```
A1 AC13 BD200
```

Range References

A range reference is made up of two cell references, separated by a colon. You must always type in the colon to separate the two parts of the reference. The first cell reference specifies the top left hand corner of the block and the second one identifies the bottom right hand corner. Examples of range references are:

```
B5:D9
AZ23:BA155
```

Row and Column References

A part of a row or column can be considered as a range that is only one column wide (or one row deep). You can therefore use a range reference to specify part of a row or column, such as:

```
A3:L3    {cells A to L of row 3}
D7:D11   {cells 7 to 11 of column D}
```

Range Identifiers

There are two range identifiers: **row** and **col**. They refer to the cells of the current row or the current column respectively (those that intersect at the cell containing the range identifier).

Each time you use one of them in a formula you will be asked to specify the exact range of cells within the row or column. Abacus will suggest reasonable starting and ending points for the range and you can either accept this choice or change it.

There are two ways in which you can use range identifiers. You can fill the current row or column by use of either

row = (*formula*) or **col** = (*formula*)

You can also use them as the argument for any function that requires a range, for example, count(row). You can, of course, only use them in this way when you just want to refer to the cells of a single row or column.

You can mix the two methods freely, for example,

col = ave(*row*)

Each occurrence in a formula will result in Abacus asking you for a particular range.

Relative and Absolute Cell References

Abacus normally assumes that all cell references are relative, i.e., that the important thing is the difference in position between the cell containing the reference and the cell to which you refer. When you copy such a reference into another cell, the references are modified to keep this relative difference. For example, imagine that a formula in cell B2 contains a reference to cell A1 (one column to the left and one row above). If the formula in cell B2 is copied into cell D4 it will, in this new location, refer to cell C3 (again one column to the left and one row above).

This is illustrated in Figure 6.1. A formula in cell X contains a reference to the lightly shaded cell. If this formula is copied to cell Y it then refers to the heavily shaded cell. The two cells in each pair have the same relative positions.

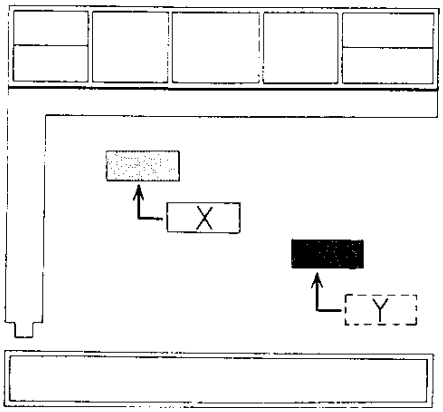


Figure 6.1 Relative cell references.

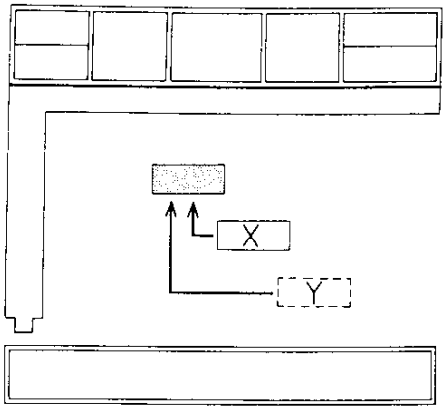


Figure 6.2 Absolute cell references

Suppose we put the formula A1*2 into cell A2, and then use the **Echo** command to copy the formula into cells in the range B2:G2. Examining the cells of row 2 will show that they have the following contents:

Cell:	A2	B2	C2	D2	E2	F2	G2
Contents:	A1*2	B1*2	C1*2	D1*2	E1*2	F1*2	G1*2

You can make any cell reference absolute by prefacing it with a $ sign. Such a reference will not be modified when the formula is copied to other cells. For example, if a reference in cell B2 was to $A1, any copy of the formula will also contain the reference $A1. You can also use labels to give an absolute cell reference (e.g. $march.costs).

Figure 6.2 shows the effect of an absolute cell reference. A formula in cell X contains an absolute reference to the shaded cell. A copy of the formula in cell Y refers to the same cell.

Let us try the previous example, but this time we shall use an absolute reference. Put the formula $A1*2 in cell A2 and **Echo** it to cells B2 to G2 inclusive. You will then find that the cells contain the following:

Cell:	A2	B2	C2	D2	E2	F2	G2
Contents:	$A1*2	$A1*2	$A1*2	$A1*2	$A1*2	$A1*2	$A1*2

See also the **index( )** function.

Cell ranges, in any form (including the range identifiers, row and col) are always relative.

LABELS

Row and Column Labels

A label is a cell containing text. The text must only include letters and digits. Any such cell can be used to identify a row or column in the grid. You can also use labels to refer to a single cell, but you may not use them to replace a range reference or to refer to a whole block of cells.

Whenever you refer to a label in an expression or formula, Abacus uses a set of rules to determine whether it refers to a row, a column or a cell. The rules for rows and columns are:

1 The row and column intersecting at the label are scanned (to the right and below) to find the numeric entry.
   a) If only a row entry is found, the label refers to the row, starting at the found entry.
   b) If only a column entry is found, the label refers to the column, starting at the found entry.
   c) If entries are found in both the row and the column, the entry closest to the labelled cell is used to make the choice.

2 If no decision can be made under 1), but the label is used on the left hand side of an expression, it will be given the type of any label(s) used on the right hand side. For example, if "Costs" is a row label:

```
Sales = Costs * 0.5
```

then "Sales" will also be a row label.

If both of these rules fail, you are told that Abacus cannot decide the meaning of the label.

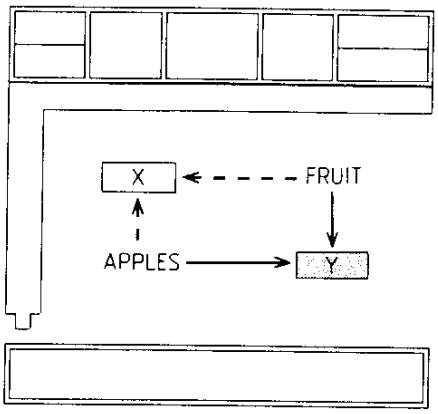


Figure 6.3 Labelling a cell

Cell Labels

You need to use two labels to identify a single cell and you make the cell reference by giving both labels, separated by a full stop. For example, if you have two labels "fruit" and "apples", you can refer to a cell as

```
fruit.apples
```

(or by any unique abbreviation, such as fr.ap). The order of the two labels is unimportant so you could also use apples.fruit, ap.fr and so on.

Such a reference refers to the cell at the intersection of the row s and columns containing the labels but, as Figure 6.3 shows, there are two such cells (labelled X and Y).

The cell that is selected is the one in the right-most column and the lower of the two rows. In the previous example, the cell labelled Y will be selected. You should, therefore, always place labels above or to the left of the cells to which they refer.

FORMULAE

A formula is any allowed combination of functions, cell references, labels and arithmetic operators. Examples are:

```
A1*B3
month(col()-1)
if(instr(B6,"is"),1,0)
rept("=",len(G23))+":"
```

Master Formulae

Each new formula, in addition to being used in one or more grid cells, is stored separately in a list of master formulae. Each master formula may therefore appear in one cell or in many. When you fill cells by use of the row and column fill operations, or by using the **Copy** or **Echo** commands, all the filled cells share a single master formula. If a master formula contains relative cell references they are adjusted, in each cell using the formula, to be valid for that particular location. The formulae may therefore appear superficially different but are all based on the one master formula.

You can modify all copies of the formula by editing only one of the copies. If you use the **Amend** command to change any copy of a master formula, the master is also modified and all copies are changed simultaneously.

THE COMMANDS

This section contains a full description of all the commands available in Abacus.

AMEND (A)

This command allows you to change the contents of a cell. The contents of the cell containing the cursor are copied to the input line, ready for editing with the line editor described in the Introduction to the QL Programs. When you press ENTER the edited version replaces the original cell contents.

COPY (C)

You use this command to copy a range of cells from one area of the grid to a similar range in another place. Abacus first asks you to give the range reference of the cells to be copied, e.g. A1:B3, and you should then press **ENTER**. Abacus next asks you to specify the cell reference for the top left hand corner of the area to which the range of cells is to be copied. When you then press **ENTER** the range will be copied to the new location.

DESIGN (D)

You use the **Design** command to modify a number of the features of Abacus that affect the appearance of the whole grid, such as whether the display should be set for a domestic television or a monitor. The choices remain in force until you modify them again, or until you leave Abacus. When you save an application these choices (except for the **Display** option) are saved with it so that they are used every time you load the application.

Changing the defaults, however, does not affect Abacus itself. You must set them to the values you want each time you load Abacus from SuperBASIC.

When you have finished you return to the main display by pressing **ENTER**. The options are:

Auto-calculate on input
used to specify auto-calculate or no auto-calculate. Each time you press the A key the auto-calculate option switches between YES and NO.

If you choose YES, the whole spreadsheet will be recalculated after each entry. Selecting NO, however, means that the spreadsheet will only be recalculated when you use the **Xecute** command. The initial value is YES.

Blank if zero
switches between two ways of treating zero values in the grid. The original option is to display the value zero in the appropriate format for that cell. You may select the alternative, which is to display a blank cell if its contents evaluate to zero.

Note that, in this option, a blank cell will only be shown if the value is truly zero. Suppose you have selected decimal display format, with two decimal places, and the value in such a cell is 0.003. The cell will show 0.00, rather than being blank, since the true value is non-zero.

Calculation order
selects between calculating the spreadsheet in ROW or COLumn order. The option changes each time you press the C key (as for auto-calculate). The specified order will be used for both auto-calculate and the **Xecute** command. The initial value is for row order.

Display 80,60,40 columns
selects the number of characters displayed across the screen. You are asked to type in 8, 6 or 4 (followed by **ENTER**) to select an 80,64 or 40 character display. The initial value is either 80 or 40, depending on whether you select the Monitor or Television option when you load Abacus from its Microdrive cartridge.

Form feed between pages
selects whether or not a form-feed is issued at the end of each page of printed output, in the same way as for auto-calculate. The initial value is YES.

Gaps between lines on printer
sets the line spacing on printed output by specifying the number of gaps between the lines of text. You are asked to type in 0,1 or 2 (no **ENTER** is necessary). You can set ordinary double-spaced printer output, for example, by specifying one gap between each line. The initial value is zero.

Lines
specifies how many lines on a page of printed output. You should type in a number, followed by **ENTER**. The initial value is 66 and the maximum is 255.

Monetary
specifies the currency sign to be used in the display of monetary values. You should type in the single character that you want (no **ENTER** is necessary). The initial value is the pound sign.

Printer
sets the number of characters per line of printed output. You should type in a number, followed by **ENTER**. The initial value is 80 and the maximum is 255.

ECHO (E)

The **Echo** command makes a copy of the data or formula in a particular cell to all the cells in a specified range.

You are given the option of specifying the cell reference of the cell to be copied, or pressing **ENTER** to copy the current cell. You then should type in the range over which the cell contents are to be copied, followed by **ENTER**.

FILES (F)

This command allows you to modify Abacus files, previously saved on a Microdrive cartridge. The options ask you to type in the names of files. Each time you are asked for a file name you can press ? for a list of all files on Microdrive 2.

You are offered the following options:

Backup
used to make a backup copy of an Abacus file. You are asked for the name of the file to be copied. You are strongly recommended to make copies of all your files, to protect yourself against accidental loss of, or damage to, the cartridge.

Delete
deletes a named file from a Microdrive cartridge. **Note that this command is NOT reversible and should therefore be used with GREAT CARE.**

Export
exports a named file. The file is saved in a form suitable for being imported by Archive, Easel or Quill.

Abacus first asks you whether you want to export to Quill, Archive or Easel. Accept the suggestion of export to Quill by pressing **ENTER**, or select export to Archive or Easel by pressing either the A key or the E key.

In all cases you are then asked to type in the range reference for the section of the grid that you want to export, ending your input by pressing **ENTER**.

If you have chosen to export to Archive or Easel you can export the file by rows or by columns. Abacus asks you to press **ENTER** to accept the suggestion of exporting by rows, or to press the C key to choose export by columns. You are not given this option if you choose to export to Quill. In this case the data is always exported by rows.

Abacus finally asks you to type in a name for the exported file. If you do not specify a file name extension Abacus will supply an extension of **_exp**.

Format
formats the cartridge in Microdrive 2. Abacus gives you the Microdrive specifier, mdv2_ and you must type in a volume name for the cartridge.

Make sure that the cartridge in Microdrive 2 contains no files that you want to keep – ALL the contents of the cartridge are erased.

Import
imports a named file. It allows Abacus to read files exported from Archive or Easel. There is a full description of Import in the *Information* section in the User Guide.

You may import a file in either row or column order, and are asked to select which. You are also asked for the cell reference of the top left hand corner of the area of the grid into which the data is to be imported.

If you do not specify a file name extension Abacus will assume an extension of **_exp**.

GRID (G)

The **Grid** command is used to make changes which affect the entire spreadsheet. It allows you to insert or delete an entire row or column, or to change the number of characters displayed in one or more columns.

The options are:

Insert
allows you to insert empty rows or columns into the grid. You are first asked if you want to insert rows (press **ENTER**) or columns (press C). You are then asked to give the number of rows or columns to insert, and a row (or column) reference. When you then press **ENTER** empty rows or columns are inserted before the one specified. The last rows (or columns) will be lost from the grid. If, for example, you insert three rows, the last three rows of the old grid will be lost.

You will not be able to recover any of the data that these lost rows contain, unless you type it in again.

Delete
allows you to delete one or more rows or columns from the grid. You are first asked if you want to delete rows (press ENTER) or columns (press C). You are then asked to give the reference of the starting row (or column) of the region you want to delete, followed by ENTER. You are then asked for the row (or column) reference of the end of the region.

When you then press ENTER the selected region is deleted and the following rows (or columns) close up to fill the gap. Empty rows (or columns) will be inserted at the bottom (or at the extreme right) of the grid.

In both of these options all formulae in the rows or columns that are moved will be adjusted to correct them for their new positions

Width
allows you to change the width (number of characters) of one or more columns. You are first asked to specify the number of characters in a column, and then to specify the starting and ending columns over which you wish the change to take effect.

JUSTIFY (J)

The **Justify** command is used to modify the positioning of text and numbers in a range of cells. It has two main options; to modify existing cells, or to set the default justification that Abacus will use when you put data in a cell which is currently empty. You should press ENTER to select the Cells option, or the D key to select the Defaults option.

You are then asked to specify whether you want to modify the justification of text (by pressing ENTER) or of numbers (by pressing the N key). In either case you can then select left (ENTER), right (R) or centre (C) justification.

In the case of the Cells option you are finally asked to give the range over which the change is to act.

You do not have to give a range in the Defaults option. The new default will apply to all newly created cells, at any point in the grid, until you make a further change in the default justification.

Some of the different types of justification, together with the original settings (text justified left and numbers justified right) are shown in Figure 6.4.

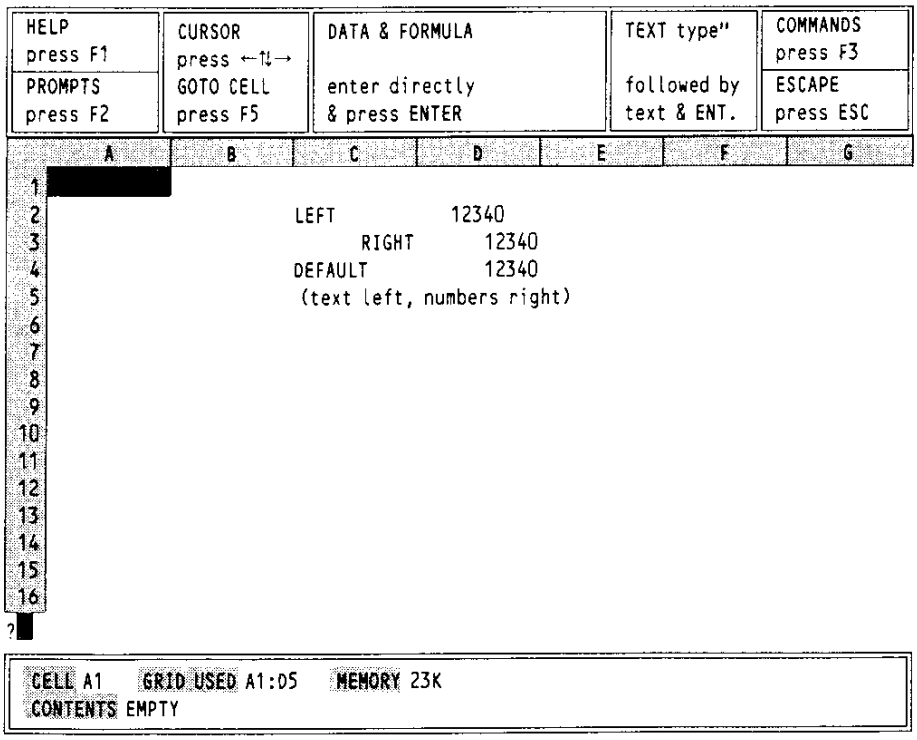


Figure 6.4 Justification

LOAD (L)

This is used to load a file from the Microdrive. You are first asked to specify the file name; pressing the ? key at this point gives you a list the files on Microdrive 2.

If you do not include an extension in the file name you type in, Abacus will assume an extension of __aba.

MERGE (M)

This command is used to combine, or consolidate, data from a previously saved file with the data in the current grid. You are first asked for the name of the file to be merged from the Microdrive cartridge and will then have to indicate whether the data in this file is to be added to (press **ENTER**) or subtracted from (press S) the data in the current grid.

Whenever a cell (in the file) containing a number or a formula matches a corresponding data cell in the grid, the value from the file will be added to or subtracted from the grid data. The contents of any other cells are not affected. The command will not have any effect on grid cells containing text, which are therefore protected against alteration.

The resulting grid contains purely numeric values in each cell that have been affected by the merge. The formulae that produced these values in the original grid cells will be destroyed. These formulae would not have any meaning in the consolidated grid.

This command offers a fast and easy method of combining the data in two similar models. It is, of course, essential that you have laid out the two grids in exactly the same way, using the same cell locations, for the results of the command to make sense.

ORDER (O)

You use this command to sort the rows of the grid into ascending order, based on the contents of one particular column.

You are first asked to specify the column on which the sorting is to be based. You are then asked for the first and last rows to be sorted. The exact ordering sequence that is used is:

Empty cells
Numeric cells, in ascending numeric order
Text cells in alphabetic order

Only use the **Order** command on rows or columns which contain data. It is likely to invalidate any formulae present in the affected portion of the grid, as they are not adjusted for their new locations.

PRINT (P)

This command is used to send a selected portion of the grid to a printer or to a Microdrive file. You are first asked whether you want the printed grid to show the values or the formulae in each cell. Press **ENTER** to show the values, or press the F key to show the formulae. Abacus next asks you to specify the range of cells which you want printed. Then you are asked if you want the grid border to be included (press **ENTER**) or not (press the N key). Following this you should specify whether the output should be sent to the printer (press **ENTER**) or to a Microdrive file (press the F key). If you choose to send the output to a file, you are also asked to type in a file name (ending with **ENTER**).

The selected portion of the grid will be sent to the chosen destination. You can stop the printing at any time by pressing **ESC**.

If you have asked for a display of the formulae, Abacus will first print a numbered list of all the formulae used in the grid. It then prints the grid itself. The formula number is shown in any cell that contains a formula.

If you do not, in the case of the option to print to a file, specify an extension when you type in the file name, Abacus will assume an extension of __lis.

QUIT (Q)

You use this command to leave Abacus when you have finished using it.

When you leave Abacus the current grid contents are lost. You are asked to confirm your request so that you have the chance to change your mind. You can cancel the command and return to your spreadsheet by pressing **ESC**. If you press **ENTER** you will confirm your wish to leave Abacus and return to SuperBASIC.

RUBOUT (R)

This command is used to rub out, or delete, the contents of one or more cells in the grid. When you use this command you will be asked to specify a range of cells. All the cells in that range will be cleared.

SAVE (S)

This is used to save a file on a Microdrive. You are first asked to specify the file name; pressing the **?** key at this point gives you a list of the files on Microdrive 2.

If you do not include an extension when you type in the file name, Abacus will assume an extension of __aba.

UNITS (U)

The **Units** command is used to change the way that numbers are displayed within a cell, or group of cells. It does not affect the values of the numbers in any way.

You are first asked to select whether you want the command to affect existing cells (just press ENTER) or to set the default format that Abacus will use for all subsequently created cells (press the D key).

In either case you are then asked to choose the display format from the following list:

Decimal
Numbers are displayed in a fixed point decimal notation, that is, all numbers are shown in the same way, with a fixed number of decimal places. Numbers which actually contain more decimal places than are displayed will be rounded up or down as necessary. You are asked to type in the number of decimal places you require. It will not accept a value greater than 14.

If you want the cell values themselves to be rounded, rather than just being displayed in rounded form, you must do the rounding yourself. For example, to round a value to two decimal places:

1. multiply by 100 (1000 for rounding to 3 decimal places, and so on)
2. add 0.5
3. discard the decimal fraction with the int( ) function
4. divide by 100 (or 1000)

The following formula will round the value in cell C3 to 2 decimal places:

```
int(c3100+0.5)/100
```

Integer
Numbers are shown as integers, or whole numbers, as for the int( ) function. You are given the option for negative values to be enclosed in brackets, rather than with a leading minus sign. Press the B key for bracketed negative values, or enter for a leading minus sign.

Use the int( ) function if you want the cell values to be converted to integers, rather than just being displayed in integer format.

Exponent
numbers are displayed in exponential, or scientific notation. The option asks you to type in the number of decimal places you want to be shown. It will not accept a value greater than 14. Again the displayed number is rounded as necessary, to the number of decimal places that you select.

Percent
this displays numbers as percentages so that, for example, the value 0.55 is displayed as 55%. The option asks you to type in the number of decimal places you want to be shown. It will not accept a value greater than 14.

General
this is a general numeric format in which any of the previous formats is chosen, depending on the value of the number, to make best use of the space available in the cell.

Monetary
numbers are displayed in fixed-point decimal format, with two decimal places and a leading currency symbol. You are given the option for negative values to be enclosed in brackets, rather than with a leading minus sign. Press the B key for bracketed negative values, or ENTER for a leading minus sign.

In the case of the Cells option Abacus asks you, at the end of any of the above choices, to specify the range over which the change is to act. You can type in any form of cell or range reference (including labels or range identifiers). Press ENTER to mark the end of the reference.

Abacus does not ask you to specify a range if you selected the Defaults option. In this case the selected format will be used for all new cells, as they are created.

WINDOW (W)

You use this command to control whether the display is a single window or is split into two windows which can be used to show two separate portions of the grid.

You are first asked to choose between a vertical (V) split, a horizontal (H) split, or to join (J) a split display back into a single window. If the window is initially split and you want to change from, say, a horizontal to a vertical split you must first join the two windows before making the new split.

If you choose to split the window then the split will occur at the column or row containing the cursor. You should therefore position the cursor at the point where you want the split

to occur before making the split. Whole columns will always be displayed. Each window in a vertical split will never be less than ten characters wide.

You then are given a further choice as to whether the two windows should move together (T) or separately (S). If you specify the T option, this means that any change in the position of one window – in the direction parallel to the split – will cause a corresponding change in the position of the other. Movements at right angles to the split are not related in this way. The S option allows the two windows to move around the surface of the grid independently.

XECUTE (X)

This command is used to force a recalculation of all formulae appearing in the grid. A recalculation is normally performed automatically when you make any new entry in the grid. You will only need to use this command if you have switched off the automatic recalculation option or if you want to activate any **askn( )** or **asks( )** functions stored in the cells of the grid.

ZAP (Z)

This command clears the entire contents of the grid and returns you to the beginning of Abacus for a fresh start. Since this command is drastic (and irreversible) in its action, you will be asked to confirm your request. If you press **ESC** you will return to the command menu without any deletion taking place. You should press **ENTER** to confirm your wish to clear the grid.

FUNCTIONS

Think of a function as a kind of recipe which converts a number of values, known as the function's *arguments* into a different value, which is said to be the value that is returned by the function. In Abacus this is the value which would be shown in a cell containing the function.

The functions provided by Abacus may take three, two, one or no arguments which are placed in brackets after its name. You must not leave a space between the name and the opening bracket, but spaces are allowed between items within the brackets. If a function takes more than one argument, then they are separated by commas. All functions must be followed by the brackets, even if they take no arguments. The presence of the brackets is a useful reminder that you are referring to a function.

In the descriptions of the functions:

n is either a numeric expression or a reference to a cell displaying a numeric value,

text is either a text expression or a reference to a cell displaying a text value,

range is a grid range reference.

A numeric expression is either a number or an expression which gives a numeric result.

A text expression is either a text string (enclosed in quotes) or an expression which gives a text result.

The following functions are provided.

ABS(*n*)
Returns the absolute value (that is, the value ignoring any minus sign) of the argument.

For example, **abs(3)** returns 3 and **abs(–7)** returns 7.

ASKN(*text*)
This function is used for the input of numeric data. It displays the given *text* (which may be up to 40 characters in length) as a prompt in the input line, followed by a '?', and waits for a reply to be typed in. The reply is shown in the cell containing the function. Input will only be requested when you first put the function into a cell, and when you recalculate the grid by use of the **Xecute** command. It is not asked for during an auto-calculate after each grid entry.

ASKT(*text*)
This function is used for the input of text strings. It works in exactly the same way as **askn( )**, except that it expects you to type in text instead of a number.

ATN(*n*)
Returns the angle,in radians, whose tangent is *n*.

AVE(*range*)
Returns the average of the numeric values contained in all the cells in the specified *range*. Empty cells and cells containing text are ignored in the calculation of the average. If there are no numeric cells in the range it will return a value of zero.

CHR(*n*)
This function returns the ASCII character whose code is *n*. A character with an ASCII code less than 32 has no effect on the screen, but is sent to the printer (when you print the portion of the grid containing it) if preceded by an ASCII null. For example, **chr(0)+chr(13)** passes the ASCII character for a carriage return to a printer, when the cell containing it is printed.

You can show an 'A' on the screen with **chr(65)**.

CODE(*text*)
This returns the ASCII value of the first character found in the specified *text*.

COL( )
Returns the number of the current column.

COS(*n*)
Returns the cosine of the given (radian) angle.

COUNT(*range*)
Returns the number of non-empty cells in the specified range. Both text and numeric cells are included in the count.

DATE(*n*)
Returns today's date as a text string in one of three forms:

n	date string
0	"YYYY/MM/DD"
1	"DD/MM/YYYY"
2	"MM/DD/YYYY"

You must first have set the system clock, as described in the SuperBASIC keyword guide.

DAYS(*text*)
Returns a number of days, from the first of January 1583, to a date given by a text expression of the form 'YYYY/MM/DD'. The conversion assumes the Gregorian (modern) calendar is being used. The formula is therefore only valid for dates after 1582.

DEG(*n*)
Takes an angle, measured in radians, and converts it to the same angle in degrees.

EXP(*n*)
Returns the value of e (approximately 2.718) raised to the power *n*. The returned value will be in error if *n* lies outside the range from −87 to +88, since the result will then exceed the numeric range of Abacus.

IF(*expression,true,false*)
The value of the expression is calculated and used to determine which of the following two arguments should be returned.

expression:= *n*
true:= *n* | *text*
false:= *n* | *text*

If the expression evaluates to 0 it is considered to be false and the 'false' argument is returned. Any non-zero value for the expression is interpreted as being true and causes the 'true' argument to be returned. The 'true' and 'false' arguments may be either text or numeric in nature. Thus all the following examples are valid uses of the function;

```
if(A1=B1,"equal","not equal")
if(A1,1,0)
```

You can also mix a text and a numeric argument as in the following example. Try this one out if you are not sure how if( ) works.

```
[A1] 1
[B1] 0
[C1] if(A1 or B1,"either",0)
```

You should see the word 'either' appearing in cell C1 since the first parameter of if( ) returns a non-zero (true) value if either cell A1 or cell B1 contains a non-zero value. If you change the contents of cell A1 to be zero then you will see a zero displayed in cell C1.

INDEX(*column,row*)

column:= *n*
row:= *n*

Returns the contents of the cell at the intersection of the specified column and row.

INSTR(*main,sub*)

main:= *text*
sub:= *text*

This finds the first occurrence of *sub* within *main* and returns the position of the first character of *sub* in *main*. It will return a value of zero if no match is found. The match is case-dependent.

```
instr("January","Jan")    {returns 1}
instr("January","an")     {returns 2}
instr("January","AN")     {returns 0}
```

INT(*n*)

Returns the integer value of the number, by truncating at the decimal point. The truncation always makes the number less positive. Thus;

```
int(3.7)  returns 3
int(-4.8) returns -4
```

IRR(*range,period*)

period:= *n*

Calculates the Internal Rate of Return for the numeric data in the specified range, which may be either a row or a column.

The data in the range represents a cash flow for each of a series of periods, separated by *n* months. Negative values represent cash outlays and positive values represent cash returns.

IRR(*range,period*)

period:= *n*

Calculates the Internal Rate of Return for the numeric data in the specified range, which may be either a row or a column.

The data in the range represents a cash flow for each of a series of periods, separated by *n* months. Negative values represent cash outlays and positive values represent cash returns.

The function returns the rate of interest necessary so that investment of your outlay would match the proposed returns.

For example, suppose you are offered a return of twenty thousand pounds at the end of each of the next seven years, in return for an initial outlay of one hundred thousand pounds. Is this a good deal?

```
[A1] "flow
[A2] -100000
[A3] col=20000    {rows 3 to 9}
```

We can refer to the range of the data by the label 'flow' and the interval between successive periods is twelve months:

```
[C2] irr(flow,12)    {rows 2 to 9}
```

The completed grid should look like Figure 6.5, showing that the internal rate of return is 9.1%. If you can invest your hundred thousand pounds at a higher rate of interest you should do so, and forget the deal.

Note that the first item in the range is counted as period zero, the next is period one, and so on. The function assumes that each amount is payable in full at the end of the relevant period.

	A	B	C
1	flow		
2	-100000.00		9.10
3	20000.00		
4	20000.00		
5	20000.00		
6	20000.00		
7	20000.00		
8	20000.00		
9	20000.00		

Figure 6.5 Internal rate of return

LEN(*text*)
Returns the number of characters in the specified text.

LN(*n*)
Returns the natural, or base e, logarithm of *n*. An error results if *n* is negative or zero, since logarithms are not defined in this range.

LOOKUP(*range,offset,value*)

offset:= *n*
value:= *n*

This function implements a look-up table in the grid. Two tables of values are assumed to be present. The first table occupies the specified range (which can be in a row or a column). The second table runs parallel to the first, in the following row or column. For example, if the first table is in column G, from G10 to G25, the second will be assumed to be from H10 to H25. Every entry in the first table should have a corresponding entry in the second. The first table is searched for the largest value that is less than or equal to the specified value. The function returns the corresponding entry from the second table. Note that it is assumed, for the correct operation of this function, that both tables contain numeric values, and that those in the first table are arranged in ascending order.

The first value in the first table is a dummy. It must be less than the second value, which is the lower limit for the table lookup process. It is otherwise ignored. The first value in the second table is the value that is returned if **lookup( )** is called with any number less than the lower limit.

MAX(*range*)
Returns the largest numeric value found in the cells in the specified range. If there are no numeric cells in the range the function will return the smallest possible number (1.7−E+38).

MIN(*range*)
Returns the smallest numeric value found in the cells within the specified range. If there are no numeric cells in the range the function will return the largest possible number (−1.7 E+38).

MONTH(*n*)
Returns, as text, the name of month *n*.

For example **month(3)** returns the text "March".

If an argument larger than 12 is used, it is replaced by the remainder after division by 12 so that, for example, **month(13)** and **month(1)** will both give the result **'January'**.

NPV(*range,percent,period*)

percent:= *n*
period:= *n*

Calculates the Net Present Value for the cash flow data in the specified range. Percent is the annual interest rate (14 represents a 14% rate). The data is assumed to refer to a series of periods, separated by equal intervals of period months.

The net present value is the amount of money required now to produce a given future cash flow, assuming an interest rate. For example suppose you are given the opportunity to buy, for a single payment of seventy thousand pounds, a ten-year lease on a shop which is currently producing a yearly net income of ten thousand pounds. You expect the income to increase by 10% per year. If you did not buy the shop your seventy thousand pounds would earn 14% interest. What should you do?

You should calculate the net present value of the income and compare it with the sum you are asked to pay

```
[A1] "flow
[A2] 0
[A3] 10000
[A4] col=a3*1.1    {rows 4 to 12}
[A14] npv(flow,14,12)    {rows 2 to 12}
```

The result is shown in Figure 6.6

```
   |          A          |        B
 1 |  flow
 2 |                0.00
 3 |            10000.00
 4 |            11000.00
 5 |            12100.00
 6 |            13310.00
 7 |            14641.00
 8 |            16105.10
 9 |            17715.61
10 |            19487.17
11 |            21435.89
12 |            23579.48
13 |
14 |            75088.51
```

Figure 6.6 Net present value

The net present value (in cell A14) of the cash flow from the shop is more than the asking price, so you should go ahead.

The first item in the list is for period zero, the second is for period one, and so on. This is consistent with the assumption, made by the function, that the returns are received at the end of each period. You therefore have to wait for one period before you obtain any return on your investment. In a real situation of this type you would probably work on a monthly basis, rather than on twelve month periods.

PI( )
Returns the value of the mathematical constant π.

RAD(*n*)
Takes an angle, measured in degrees, and converts it to the same angle in radians.

REPT(*text,n*)
This function will fill the current cell with *n* copies of the first character of the given text. For example,

```
rept("*",5)    {will put five asterisks in the current cell}
rept("abc",3)    {makes three repetitions of "a"}.
```

ROW( )
Returns the number of the current row.

SGN(*n*)
Returns +1, −1, or 0, depending on whether the argument is positive, negative or zero.

SIN(*n*)
Returns the value of the sine of the specified (radian) angle.

STR(*n,type,dp*)

num:= n
type:= n
dp:= n

Converts a number, *num*, to the equivalent text string. Type indicates the form of the converted string as follows;

0 decimal (floating point)
1 exponential, or scientific, notation
2 integer.
3 general format

The third parameter, *dp*, specifies the number of figures after the decimal point in the converted string. It should always be included, although its value is ignored for integer, general and monetary formats.

SQR(*n*)
Returns the square root of the number *n*, which must not be negative.

SUM(*range*)
Note that the value returned by the function is the sum of the exact values in the relevant cells. It does not take into account any rounding that may result from use of the **Units** command. For example, if two cells contain the values 3.44 and 9.73, the **sum( )** function will add them to give 13.17. If you then select a display in decimal format with one decimal place, the two numbers will be rounded to 3.4 and 9.7. The sum, whose value will still be 13.17, will be rounded to an apparently inaccurate 13.2. See the **units** command.

TAN(*n*)
Returns the tangent of the specified (radian) angle.

TIME( )
Returns, as text, the time of day in the format "HH:MM:SS". You must first have set the system clock, as desribed in the SuperBASIC keyword guide.

VAL(*text*)
Val converts the *text* to its equivalent numeric value. It will only convert text composed of valid numeric characters and the conversion will stop at the first character that cannot be interpreted as a digit. For example, **val("2.2ABC")** will return the value 2.2, and **val("ABC")** will return 0.0

WIDTH( )
Returns the width, in character spaces, of the current column. Note that there is one space separating adjacent columns.

ERRORS

Grid Errors

Any syntax error in a formula – such as supplying the wrong number of arguments for a function, or mis-matched brackets – will be reported at the time you type in the formula. You are told the nature of the error and the formula is left in the input line. You can then examine it, and then correct it with the line editor.

The possible error messages are listed below.

Message	Example
Missing closing quotes in a string	`"abc" + "def`
Badly formed numeric constant	`1.5e` (missing number after 'e')
Number too large	`1.5e99`
Illegal character	`12_5` (underscore instead of minus)
All names must refer to columns All names must refer to rows Name references may only be relative (see the section on Cell References, earlier in this chapter)	
Badly formed range reference	`a1:`
Badly formed name reference	`c3.`
Name is not a row or column	(see Chapter 3)
First name reference undefined Second name reference undefined (the text does not appear in the grid, above and to the left of this cell)	
Function requires a range reference	`irr(1,2,3)` (see description of **irr( )**)
Illegal range Syntax error Mismatched brackets	
Type mismatch	`1 + "abc"`

Wrong number of function arguments	`sqr(1,2)`
String bigger than 255 characters	`rept("*",256)`
Division by zero	
Illegal function arguments	`sqr(-1)`
String subscript out of range (either subscript less than zero or greater than 255, or first subscript greater than length of text)	
Reference out of range (to a cell outside the grid)	
Reference to an error cell (the formula refers to a cell containing a formula which produces one of the errors described below)	
Out of memory, use RUBOUT to make more room	

Other errors, such as a formula which adds the contents of two other cells, one containing a numeric value and the other containing a text value, will not be detected until the result of the formula is calculated – after the formula has been placed in the grid.

If a formula contains a reference to an empty cell, Abacus will assume that the cell has a numeric value of zero. This may well cause an error when the formula is calculated.

If Abacus detects an error when a formula is calculated it gives a brief error message in the relevant cell. You can then move the cursor to the cell to examine the formula and find out what is wrong.

The possible errors are:

##TYPE - the formula contains a reference to a cell containing information of the wrong type, i.e. numeric instead of text, or vice versa.

##LONG - the formula contains a reference to a text string that is more than 255 characters long.

##ZERO - The formula is attempting to divide something by zero.

##ARG - The formula contains a function called with a non-valid value for one or more of its arguments e.g. ln(-5).

##SUB - The formula uses a string slicing operator with an error in one or more of its subscripts.

##REF - The formula contains a reference to a cell which is outside the grid. The formula in such a cell will show the word 'ERROR' for each cell reference that is not valid.

ERR - The formula contains a reference to a cell which contains an error. You can ignore these messages since they will disappear when the original error, in the cell to which the formula refers, is corrected.

File Errors

The following error messages will only appear if an error occurs while you are using a file-related command e.g. Load or Files.

File does not exist
the file name you gave was not found on the cartridge in Microdrive 2.

File I/O incomplete
the loading or saving of a file has started successfully but has failed at a later stage – this may mean that the data in the file has been corrupted, or that the cartridge has been damaged.

Unable to open file
the file can not be opened – for one of the reasons given for the previous error.

Wrong file type
the file name extension is not the one that Abacus was expecting – e.g. attempting to load an export file instead of importing it.

Illegal file name
e.g. "3test" file names must start with an alphabetic character and may not be more than 8 characters.

Illegal import file format
this is only likely to occur if you attempt to import a file not created with an **export** command.

FOR SERVICE MANUALS
CONTACT:
MAURITRON TECHNICAL SERVICES
www.mauritron.co.uk
TEL: 01844 - 351694
FAX: 01844 - 352554

sinclair

QL

QL Archive

©1984 PSION LIMITED
by Dick de Grandis-Harrison *(Psion Limited)*

CHAPTER 1
ABOUT QL ARCHIVE

QL Archive is a database program which enables you to create filing systems for any type of information you choose. You are free to decide how this information will be stored and retrieved.

You will quickly discover how Archive can be used for creating simple card index systems such as address lists or customer records. Once you have mastered the creation of straightforward systems such as these, you may wish to develop more complex multi-file relational systems where information is shared between, for example, purchase and stock control records.

Information may be presented using the screen layout that Archive provides, or you may design your own. Printed forms and reports can be produced from the information in the file in any format you choose.

One of the most powerful features of Archive is its command structure. Once you have created a file and stored some records in it, these commands can be used to find particular records, make searches and selections or display the information in the file in a particular order.

The commands combine to form a powerful programming language, similar to SuperBASIC, which can be used to construct a multitude of specialist applications.

At all times you will be guided by an informative set of prompt messages which never leave you in doubt about what your options are or what you are expected to do. If you require further information you can use the Help files. You may ask for Help at any stage, no matter what you are doing, and will automatically be given the information that is most relevant to your current needs.

The real power of Archive becomes apparent when you write your own procedures in the command language. You can create a named procedure to do exactly what you want and then use it as an additional command, in the same way as you use the commands provided with Archive.

The mechanics of writing and modifying a program are aided by a full *procedure editor* which, together with the *input line editor* (which is available at all times), make editing a simple and painless task.

The data files themselves use variable length fields and records. Not only does this lead to the most efficient use of available memory and cartridge space, but also to simplified file creation. You never need to decide in advance how large a record needs to be.

This manual contains a number of working examples. Try these out to see some of the range of things that can be done. They contain many general purpose procedures which you might include in your own programs.

If, at any time, you are not sure what to do, remember that you can ask for Help by pressing **F1**. Also remember that you can cancel any partially completed operation (e.g. typing in a number, or using a command) by pressing **ESC**.

Archive has been designed to give you the greatest possible flexibility. As a consequence it cannot give as much assistance with the selection of options as the other QL programs. If you are not familiar with computers and computer programming you may find it helpful to read the Beginner's Guide to SuperBASIC before attempting to write Archive programs.

CHAPTER 2
GETTING STARTED

LOADING QL ARCHIVE

Load QL Archive as described in the Introduction to the QL Programs. When loaded Archive will display the following message:

```
LOADING QL ARCHIVE
version x.xx
Copyright © 1984 PSION SYSTEMS
database
```

where x.xx represents the version number, e.g. 2.00.

The program will then wait for a few seconds before starting.

The Help information is not loaded into the computer's memory together with the program. It is only read from the Archive cartridge when it is needed. **You should therefore not remove the Archive cartridge from Microdrive 1 if you intend to use the Help facility.**

GENERAL APPEARANCE

When you have loaded Archive the screen should look like Figure 2.1. This is the *main display.*

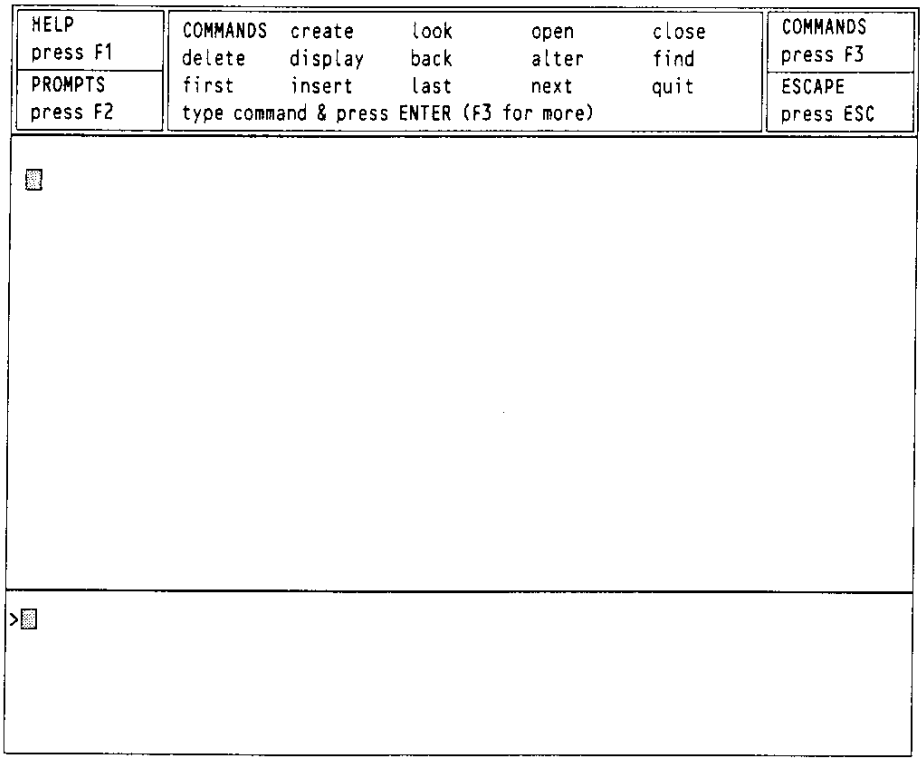


Figure 2.1 The main display with a monitor. (80 characters)

If you are using a domestic television, the screen is arranged slightly differently. This is because a television is not normally able to show clearly 80 characters per line. Archive therefore only shows 64 characters.

The screen is divided into three sections: the display area, the work area and the control area.

The Display and Work Areas

As its name suggests, this is where all information produced by Archive is shown.

The work area uses the bottom four lines of the screen. All commands that you type in, together with any error messages, are shown here.

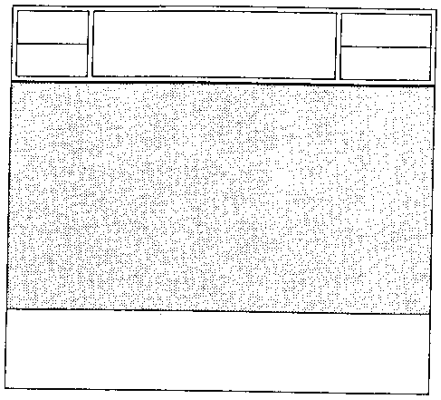


Figure 2.2 The display area

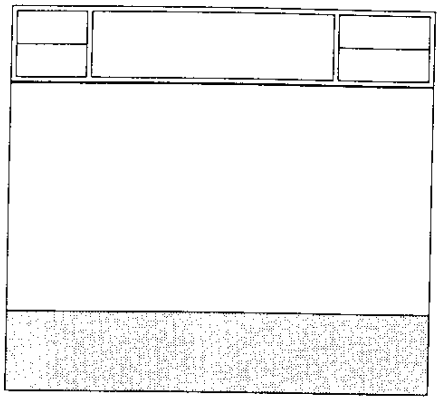


Figure 2.3 The work area

These two areas almost invariably work together, since commands typed into the work area produce their results in the display area.

As an example, type in the following short program, exactly as it is shown below.

```
let x=13:while x>0:print x:let x=x-1:endwhile ENTER
```

The text of this program will appear in the first line of the work area. When you press **ENTER**, the numbers from thirteen down to one will be printed on successive lines of the display area. The bottom line of the display area will be left blank except for a red cursor indicating the next position at which text will be displayed. The numbers from fifteen to one are displayed which, together with the bottom blank line, occupy all sixteen lines of the display area.

The command:

```
cls ENTER
```

will clear the display area completely.

The Control Area

The control area occupies the top few lines of the screen. It shows the normal options: Help (**F1**), to turn the prompts on and off (**F2**), cancel any incomplete operation (**ESC**), and use a command (**F4**).

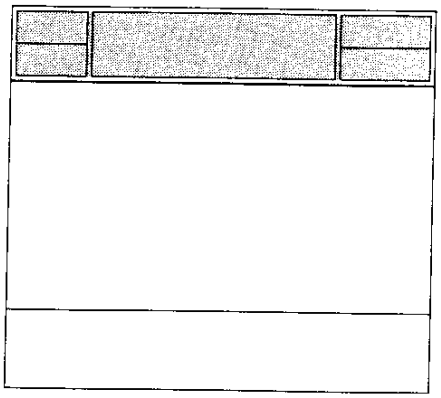


Figure 2.4 The control area

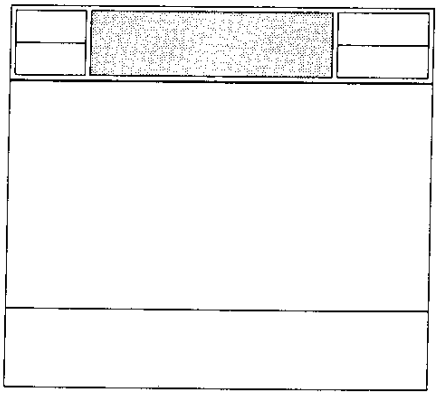


Figure 2.5 The commands

USING THE COMMANDS

Archive's commands form a programming language and you must type their names in full. This may seem long-winded at first, but later you will be shown how to create procedures which allow you to enter commands with a single keystroke.

There are four different lists of commands which can be displayed by pressing **F3**. If a command list is already being shown, pressing **F3** will display the next list in sequence. These commands are used simply by typing in the name and pressing **ENTER**. However, some commands need further information and will ask for it.

You can use any of the commands, even if its name does not appear in the current display in the control area.

THE MODE COMMAND

You can combine the control, display and work areas into a single area with the **mode** command. Used by itself **mode** will combine the three areas into a single area. Typing **mode 0** will also have the same effect. Try

```
mode ENTER
```

and the input from the keyboard and anything displayed by a command or program will share the whole of the screen. A value of 1 divides the screen back into three areas.

You can also use the **mode** command to change the number of characters displayed across the screen. To do this you must supply a second number separated by a comma from the first. The second number must be a 4, 6 or 8 to select a 40, 64 or 80 column display. Try typing

```
mode 0,4 ENTER
```

to change the display to 40 characters and to combine all three areas on the screen. Note that the 0, which originally was optional must be typed to change the size of the display.

Try some different combinations to see the effect on the display. Finish with a command that leaves the screen divided into its three areas, but choose the number of characters that gives a clear display on your television or monitor.

CHAPTER 3
QL ARCHIVE FILES
FILES RECORDS AND FIELDS

An Archive file behaves rather like a card index. A real card index consists of a box containing a set of record cards, each card containing various items of information. For such a card index to be useful, there have to be rules to determine where each piece of information is written.

Suppose, for example, that we have a name and address index. You would normally write the person's name across the top, followed by the address and telephone number (if any). It would be very difficult to use if some cards had the name written at the top and others had it written near the bottom. You would normally expect to be able to use the index by flipping through the cards, reading only the top line, until you found the name you were looking for.

If you had two sets of record cards, such as a set of name and address records and a set of stock records, you would not normally store them both in the same box. You would use two boxes and label them, for example, "Customer Records" and "Stock Records".

The card index system contains most of the ideas necessary to understand how an Archive *file* works. A file is like the card index box and is given a name to identify it. The file is made up of a collection of *records*, each of which serves the same purpose as a record card. A file, then, is simply a collection of related records.

Like a card index, the information in each record is organised in a regular way. Individual items of data, such as telephone numbers might be kept on a specified area of the card. A record in an Archive file is organised in the same way. Each item is stored in a separate region of the record, known as a *field*. A record in a customer file, such as that described above, would contain a name field, an address field, a discount field and so on.

If this were the whole story there would be little point in using an Archive data file in preference to a physical card index. There are, however, many advantages when you use computerised records. A customer record card index would normally be arranged in alphabetical order of customer names which makes it an efficient way to find the information about a particular customer. Suppose, however, you want to send a letter to all your customers who have not placed an order with you during the last six months. It would be a very tedious task to go through the entire contents of a card index to compile such a list. In Archive you can make such a search by using a few simple commands. Furthermore, it is easy to arrange for a set of address labels to be printed at the same time.

You can save a great deal of time and effort by using Archive to store and manipulate your records.

CHAPTER 4
EXAMINING A FILE

The best way to start learning about Archive is to look through the demonstration file **gazet**, provided on the Archive cartridge. This is a file which contains information about various countries – the continent, the capital, the currency, the language, the population, the land area and the gross domestic product per capita.

Most of the examples in chapters 4 and 5 refer to the "gazet" file. Before using it, you should make a copy of it using the following procedure:

When you have loaded Archive, put a formatted cartridge into Microdrive 2 and type:

```
backup ENTER
mdv1_gazet_dbf ENTER
mdv2_gazet_dbf ENTER
```

Wait until the two Microdrives have stopped; be patient as the file is quite long and can take a while to copy. Use the copy, now on the cartridge in Microdrive 2, for experimenting.

From now on we will not always write **ENTER** at the end of every command but please remember that it must still be used.

The **look** command opens a file so that you may read its contents, but you are not able to make alterations or additions to the file. It is a safer command than **open** if you are merely looking through a file because the file is protected against accidental modification. You can examine the copy of the "gazet" file on Microdrive 2 by typing:

```
look "gazet"
```

DISPLAYING A RECORD

To look at the first record type:

```
first
display
```

Don't forget to type **ENTER** after each command and then the display will show the first record of the file.

Note the first line shows the logical name of the file; Archive automatically supplies the name "main" for a single file. Logical file names are usually used when you are using more than one file at a time and are described later.

EXAMINING OTHER RECORDS

Having looked at the first record of the file, you may want to move on to the following record. Type:

```
next
```

and the display shows the next record in the file. When you are typing single commands after a **display** command the display area is continuously updated to show the contents of the current record. You can use the **next** command to step through the file, record by record until you reach the end (it will not pass the last record).

There are three other related commands which you can use to control which record of the file is displayed.

```
back    - which displays the previous record,
first   - which displays the first record,
last    - which shows the last record of the file.
```

Try using these commands to move around the file, displaying any record you like. Note that the four commands **first, last, next** and **back** do not themselves display the record. They merely move from record to record regardless of whether or not you have used **display** command.

SEARCHING A FILE

Find

The first and simplest search command is **find**. This will search from the beginning of a file, looking for the first occurrence of a specified piece of text in any of the text fields.

For example:

```
find "africa"
```

When you press **ENTER** there will be a slight pause and then the first record containing the word 'africa' in any of its text fields will be displayed. Note that this search is independent of whether the text is in upper or lower case and will therefore find 'Africa', 'AFRICA' or 'africa'.

Continue

If the first record that is found containing the text is not the one that you want, you can find the next occurrence by typing:

```
continue
```

The **continue** command will repeat the previous search, looking for the next occurrence of the text in any text field of the following records.

It is possible that you may have to repeat a search several times before finding the record you require. Press **F5** and Archive will put the previous command back in the command line. Press **ENTER** and the command will be executed.

Search

Another method of locating a particular record is to use the **search** command. This allows you to find a record by specifying the contents of one or more specific fields, for example:

```
search continent$="EUROPE" and language$="FRENCH"
```

will find the first record in the file which matches both conditions. You must type in the full field name.

Unlike the **find** command, **search** will only test the fields you specify and will differentiate between upper and lower case letters. Use the **upper( )**or **lower( )** case functions to make the search case independent, for example:

```
search lower(continent$)="europe"
```

Again the **continue** command can be used to find the next occurrence of the text.

Select

In many cases, you may want to look at a sub group of the records within a file. Suppose, for example, you only want to look at the details of countries in Europe. You can use the **select** command to pick out from the file all those records which satisfy a certain condition. The file will then behave as though only those selected records are present. Try this command on the "gazet" file to see how it works. First type:

```
print count()
```

which will tell you how many records there are in the file. Then type:

```
select continent$="EUROPE"
print count()
```

and you will see how many records have been selected. The records that are removed from the file are still held in the computer's memory and you can restore them to the file at any time by using the **reset** command. Type:

```
reset
```

and print the value of **count( )** again, to check that the file has been restored to its original state.

When you use the **print** command from the keyboard, any file shown on the screen will be erased. This is because, in general, **display** and **print** use areas of the screen which overlap. After using **print** you must type **display** again to restore the display.

SORTING A FILE

The file records may not always be in the order you need. You can sort the file by the contents of numeric or text fields. **Only the first eight characters of text are taken into account by order.**

Suppose, for example, you want to sort the records of the "gazet" alphabetically by capital city. You can do this by using the **order** command as follows:

```
order capital$;a
```

The "a" following the semicolon specifies that you want to sort the file in ascending order. Replace it by "d" if you want the file sorted in descending order. The capital$ field becomes the *sort key* for the file. You can specify a sort key composed of up to four fields by

giving a list of fields after the **order** command. For each of the keys you must specify whether the sort is to be in ascending or descending order. The following command, for example, will sort the file into descending order by population and ascending order by capital.

```
order pop;d,capital;a
```

Note that a semicolon separates each field name from the "a" or "d" that specifies ascending or descending order, but that each pair (field name and letter) is separated from the next by a comma.

When more than one field is specified for sorting purposes the records are initially sorted according to the contents of the first field in the list. If two or more records have the same contents for this field, they are ordered according to the next field in the list. If records exist which are equal in respect of the contents of both of these two fields, they are ordered according to the contents of the third field, and so on.

LOCATE

When a file has been sorted, you can use the **locate** command to make any particular record the current record in the file. Its action is to find the first record whose first sort field is greater than or equal to the given expression. This record becomes the current record in the file.

For example, if the "gazet" file has been sorted as described in the last example, the command:

```
locate "100"
```

locates the first country in the sorted file which has a population of 100 million. If there is no such country Archive will locate the first country with a population less than 100 million (remember that the file was sorted in descending order).

Locate is followed by an *expression* which may be either text or numeric, but must be of the same type as the field used to sort the file. (See the *Reference* chapter.)

You can locate a record with respect to the contents of more than one sort field by using **locate** with multiple expressions, separated by commas. For example,

```
let a="100"
let b$="D"
locate a,b$
```

will find the first country with a population of 100 million or less, and with a capital whose name either starts with "D" or is after "D" in the alphabet. In this example Archive will locate Bangladesh, which has a population of 76.1 million and whose capital is Dacca.

The only restriction on the number of expressions that you can use with **locate** is the number of fields used to sort the file.

You cannot use **continue** after **locate**. Repeating a **locate** with the same condition will always locate the same record.

Locate is the fastest way of locating a record in a large, sorted, file. Because of the uncertainty in the record that is located, you may have to make a further check on the record to make sure it is what you want.

CLOSING A FILE

When you have finished looking at a file you must tell Archive. You can do this by typing

```
close
```

This will only act on files and will leave any program or screen layout intact. You can close all your files and clear out your data and display area by typing

```
new
```

This will clear Archive to its initial state after loading.

This only acts on the data files, leaving any program, or screen layout, intact.

Alternatively, if you have finished using Archive, you can go back to SuperBASIC by using **quit**. This command closes all open files automatically before leaving Archive.

Remember that you should never remove a cartridge from a Microdrive while it contains open files.

CHAPTER 5
MODIFYING A FILE

Before typing in examples in this chapter, type **new** first to ensure that Archive is cleared and ready for a fresh start.

The **open** command prepares a file for both reading and writing.

If you open a file with the **open** command, instead of **look** you will be able to write to the file to change its contents as well as read it. This means that any additions, deletions or modifications will make a permanent change to the copy of the file when it is closed. Type:

```
open "gazet"
```

If you have opened a file for reading with **look** then you must not use any commands which will attempt to modify the data. If you do, Archive will report an error. The commands described in this chapter modify data files and so should only be used with a file opened with **open**.

Display the first record of the file with:

```
first
display
```

CLOSING THE FILE

When you have finished modifications to the file you must close the file (using **close** or **new**) to ensure that all the changes are recorded.

If you do not close a file properly (for example, if you just turn off the computer when you have finished) the file may be changed and your most recent changes will not be recorded. **Always make sure that there are no open files on a cartridge before you remove it from the Microdrive. Do not switch off the computer without first closing all open files and removing the cartridges from the Microdrives.**

INSERT

The **insert** command is used to add one or more records to the current file. When you use **insert** you will be asked to type in the contents of each field of the new record. Type:

```
insert
```

The display area will now show:

```
Logical name : main
country$     :
continent$   :
capital$     :
currency$    :
languages$   :
pop          :
area         :
gdp          :
```

You can now type in the contents of each field. You can step from one field to the next by pressing **ENTER** or **TABULATE** or you can step back to the previous field by holding down **SHIFT** and pressing **TABULATE**. You can make as many changes as you like to the fields until you are satisfied. The new record can be inserted into the file by pressing **F5**. Press **F4** to leave **insert**. Try typing:

SCOTLAND	TABULATE
EUROPE	TABULATE
EDINBURGH	TABULATE
POUND STERLING	TABULATE
ENGLISH	TABULATE
10	TABULATE
30	TABULATE
50	TABULATE

The display area should now show:

```
Logical name : main
country$     : SCOTLAND
continent$   : EUROPE
capital$     : EDINBURGH
currency$    : POUND STERLING
languages$   : ENGLISH
pop          : 10
area         : 30
gdp          : 50
```

When you are satisfied that you have typed in the new information correctly, press F5 to insert the new record into the file. The fields you have just typed in will then be blanked out ready for you to insert a new record. Press F4 when you have finished inserting.

You can also end the entry for each field and move to the next one by pressing ENTER. The new record is added to the file automatically when you press ENTER after the last value.

If the file has been sorted the new record is inserted at the correct position to maintain the order.

DELETE

You can use the **delete** command to remove a record from the file. **delete** removes the current record (the one shown by **display**) from the file. All you have to do to remove a particular record is to display it, and, having made certain that it is the correct one, type:

```
delete
```

CHANGING A RECORD

It is also simple to modify the contents of any or all of the fields within an existing record. There are two methods.

Alter

Select the record you want to change (use **display** and **find**) then type **alter**. **Alter** works in the same way as **insert** except each field shows its old contents. You can step over those fields you do not want to change (use TABULATE or ENTER). Type in a new value or use the cursor keys to modify an old one. Press F5 to replace the record.

As with **insert**, the record is replaced automatically if you press ENTER after the last field in the record.

Update

Select the record you want to change then change the contents of the field variables until the displayed record is as required. Type **update** to change the record.

For example, suppose that you decide that Iceland should be in Europe instead of the Arctic. Find the record by typing

```
find "Iceland"
display
```

Use the **let** command to change the contents of the continent$ field:

```
let continent$ = "Europe"
```

Finally put this change into the record by typing **update**.

In both of the above methods the new record will be inserted in the correct position if the file has been sorted. Otherwise the replacement record is inserted in an unspecified position in the file.

The **alter** command is simpler to use, but always affects the current record. The **update** command can be useful when you are using multiple files.

Remember that you must close the file with the **close**, the **new** or the **quit** command, before switching off the computer.

CHAPTER 6
CREATING A FILE

If you have been following the examples up to this point, you will have been using Archive only to look at the file provided for you. This chapter will show you how to create your own file with your own choice of file names.

If necessary, type **new** to clear anything in the computer's memory and to close any open files. Make sure that the formatted cartridge on which you are going to create the file is in Microdrive 2.

Suppose you want to use ARCHIVE to make a catalogue of your books. To do this, you will have to create a new file called "books". The first thing to do when creating a file is to decide what it is going to contain, that is, what fields you will use in each record. In this case you will obviously need to record the author, title and subject; you may also like to include other details, such as the type (fiction or non fiction), ISBN (International Standard Book Number), shelf location, a brief description and so on. In this example we shall simply use three text fields to contain the author, title and subject and one numeric field which will be used to hold the ISBN.

CREATE

You create a file with the **create** command. You must specify the name of the file to be created and the names of the fields to be used in each record. The $ sign indicates that the field contains text. When you have finished defining the fields of a record you end the **create** command with **endcreate**. You can create a simple book catalogue file, as described above, by typing in the following sequence.

```
create "books"
 author$
 title$
 subject$
 isbn
 endcreate
```

Note that you do not have to type in the final **endcreate** command. You can do so if you want, but you can end the creation of the file simply by pressing **ENTER** on a blank input line. You must, however, include **endcreate** if you use **create** in an Archive program.

ADDING RECORDS

When you have created a file, it is open for both reading and writing, but it contains no records. Records can be added using **insert**. Type:

```
insert
```

and the display area will show:

```
logical name : main
author$ :
title$ :
subject$ :
isbn :
```

All you have to do is to type in the contents of each field. For example, type:

```
Bloggs, J          TABULATE
A Boring Manual    TABULATE
Cannon Making      TABULATE
1234567            TABULATE
```

the display area should show

```
Logical name : main
author$      : Bloggs, J
title$       : A Boring Manual
subject$     : Cannon Making
isbn         : 1234567
```

Insert the record into the "look" file by pressing **F5**. The field value will be cleared ready for inserting another record.

Remember that you can also end the entry for each field and move to the next one by pressing **ENTER** and that pressing **ENTER** after the last value will add the record to the file.

When you have finished press F1, and remember to use **close** or **quit** to save the file first.

CHAPTER 7
SCREEN LAYOUTS

When you use the **display** command on a file that you have created, the records are shown using the standard Archive screen layout.

DEFINING A SCREEN LAYOUT

You can design your own screen layout, better suited to the information in your data file. **Open** an existing file and type in:

```
display
```

You select screen editing with the **sedit** command – type in:

```
sedit
```

The display area shows the current screen layout, which will be the one that Archive creates automatically. If there is no screen layout in the computer's memory, the display area may be blank.

You will see that the values of the fields of any file are not included. The spaces where these values are normally shown are marked by rows of dots. You should think of a screen layout as a background against which the values of a number of variables are shown in specific positions. Archive shows a screen layout into two stages – first it draws the background text and then it shows the values of the variables at the marked positions on the screen.

You are initially at the *main level* of the command and you have three options:

type background text into the screen
press **ESC** to leave **sedit**
press **F3** to use a screen editing command

To design a screen layout, press **F3** and then C to clear the screen and make a fresh start. Press **ENTER** to confirm your choice; any other key will return you to the main level of **sedit.**

Choose paper and ink colours by pressing either P or I and pressing any key to switch between the four available colours. Press **ESC** to return to the main level to enter background text.

Background text might be explanatory, such as:

Andrew Young's World Gazeteer

Or it might consist of a new name for one of the fields in your file:

Population (millions):

You can move the cursor to any point in the display area by using the four cursor keys. Anything that you type will immediately appear in the display area at the position of the cursor and will become part of the background of the layout. The only exception is if the cursor is positioned within an area of the screen reserved for the display of a variable. Archive shows the name of the variable in the work area at the bottom of the screen. You cannot type background text into this area unless you first free the area, as described later.

The four screen edit commands enable you to produce attractive and colourful formats for displaying your data. Clearing the screen has already been explained. You may need to experiment to completely master the remaining three so make sure you are using a copy of your data file which is expendable.

SCREEN EDIT COMMANDS

Mark Variable (V)

Suppose you want to show the value of the variable country$ at a particular position in the screen. Move the cursor to that point and press **F3** and then the V key. Archive asks you to type in the name of the variable. You type:

```
country$
```

Note that this name does not appear on the screen – you are just marking the point where the *value* is to be shown. When you press **ENTER** Archive asks you to show

how much space is to be reserved for showing the value. You press any key except **ENTER** to mark the space with a row of dots. **CTRL** and the left cursor key can be used to delete reserved space. When you have reserved enough space you press **ENTER** and Archive takes you back to the main level of **sedit**.

If you move the cursor into one of the reserved areas, (marked by dots), Archive shows the name of the variable for which space is reserved in the work area.

If you reserve space for a variable in a region which overlaps any area that is already reserved, you are given the option of cancelling the old area. You can then use the option again to allocate space for a new variable.

Ink (I) Suppose you want to change the ink colour. Move the cursor to the point where you want the new colour text to start and press **F3** and then the I key. Archive shows the four available colours in the control area. The one that is selected will be the one that is highlighted. Press any key to change the selected colour and then press **ENTER** to record your choice. Any subsequent text that you type will appear in the new colour until the **ink** command is used again.

Paper (P) Changing the paper colour works in the same way – except that you press **F3** and then the P key.

If you want a colour change to affect only part of a line, you should move the cursor to the start of the region and select the paper and ink colours that you require. You should then move the cursor to the end of the region and make a second selection of paper and ink colours, returning them to their original values.

ACTIVATING A SCREEN LAYOUT

Once you have designed a screen layout and have left **sedit**, the screen layout will be *active*. This means that the values of all the variables in the screen layout will be displayed automatically every time Archive completes a command or a program. If, for example, you type the command **next** Archive moves to the next record of the current file and shows those fields that are included in the screen layout. Any active screen is deactivated each time you use the **cls** command.

If a screen layout is not active, you can activate it with the **screen** command. This displays the background text of the screen layout, but does not show the current values of the variables.

SAVING AND LOADING SCREENS

You can save your screen design on a Microdrive cartridge using the **ssave** command:

```
ssave "filename"
```

where "filename" is a name of your choice. The screen layout is saved exactly as it appears.

You can reload the screen layout by typing in the command:

```
sload "filename"
```

When you load a screen layout, it is automatically displayed on the screen and made active.

Archive will not automatically update an active screen layout from within a program. Suppose you want to show all the records of the current file, one after another, and tried to do so by typing the one-line program:

```
first: let x=0: while x<count():next:let x=x+1:endwhile
```

(The **while** and **endwhile** commands cause the section of program that they enclose to be performed repeatedly, while the condition following **while** is true. For correct operation every **while** command must have a matching **endwhile**.)

This program would fail to do what you want, since Archive only updates the contents of the screen layout at the end of the program.

THE SPRINT COMMAND

You can, however, force a display of the values of the variables in an active screen from within a program using the **sprint** command. The following one-line program will show all the records, as required.

```
first:let x=0:while x<count():sprint:next:let x=x+1:endwhile
```

If there is no active screen **sprint** has no effect.

THE DISPLAY COMMAND

Remember that the **display** command uses the standard layout. It will always replace any screen layout with its own simple list of the fields of the current record of the current file. You must therefore **ssave** your screen layout before you next use **display**. If you do not, your screen layout will be replaced and you will not be able to get it back again except by redesigning it with **sedit**.

CHAPTER 8

PROCEDURES

To use the examples in this chapter, first type **new** to clear the computer, then type **look "gazet"** to open the example file on your data cartridge, which is assumed to be in Microdrive 2.

The commands and functions of Archive together form a programming language which you can use to write programs that will manipulate your files. You will find that Archive programs are simple to write.

An Archive program is made up of one or more separate sections. Each section is known as a *procedure* which is simply a named section of program. You can refer to a procedure by its name, like the procedures which you write and use in SuperBASIC. In Archive you can run a procedure by typing its name at the keyboard. When you write a procedure you are effectively adding a new command to Archive.

No procedure may contain more than 255 lines, and each line must not contain more than 160 characters.

CREATING A PROCEDURE

You use the *program editor* whenever you want to write or change a procedure. This editor allows you to change, delete or add to the text of procedures.

The program editor is described in detail in Chapter 9, but in this chapter we will look briefly at some of its features so that we can write a few short procedures. We shall assume that initially there are no procedures in the computer's memory.

Type:

```
edit
```

to enter the program editor. The control area changes, showing that you should type in the name of the procedure. Entering the editor will always allow you to create a new procedure if none are defined or loaded.

The first thing to do, therefore, is to decide what the new procedure should do. Let us start with a very simple task; to make life easier by renaming the **display** command. We will save typing by giving it the name "d".

Just type

```
d
```

The left hand side of the display area now shows the name, and the right hand side a listing of the procedure. The procedure, as yet, contains no commands; the **proc** and **endproc** which mark the beginning and end of the procedure were automatically added by Archive.

The *body* of the procedure must be added; that is sequence of actions it is to perform.

The control area shows that you can add lines of text to the new procedure. In terms of the current example this text is the **display** command. Type:

```
display
```

and Archive will insert the new text into the procedure below the highlighted line. If you have followed this example the display will contain:

```
d   proc d
    display
    endproc
```

You could add more lines of text – each line would be inserted below the highlighted line.

In this case, however, the procedure is complete so you can leave the **edit** command by pressing **ESC** twice.

All you have to do to use the procedure is type its name, followed by **ENTER**. This new procedure will perform the same function as typing the command **display** in full.

LISTING AND PRINTING PROCEDURES

Whenever you call the **edit** command you are shown a list of the names of all the defined procedures present in the computer's memory.

You can list any one of these procedures from within **edit** by pressing the **TABULATE** key to move down the list or the **SHIFT** and **TABULATE** keys together to move up the list until the particular procedure name is highlighted. The procedure is automatically listed at the right hand side of the screen. If the procedure is too long to fit in the display area, you will be shown the first part and you can then scroll up and down through the procedure using the up and down cursor keys. When you have finished you can leave the **edit** command by pressing **ESC**.

If you want a printed listing of your procedures you can use the **llist** command. Type:

```
llist
```

and all the procedures currently in the computer's memory will be listed on a printer.

WARNING: Do not use this command unless a printer is attached since this will cause the program to "hang".

SAVING AND LOADING PROCEDURES

If you want to keep the procedures that you have defined, you can use the **save** command. This stores all defined procedures in a single named file on Microdrive cartridge. If you want to save the new display procedures that you have just defined in a file called "myprocs", you should type in

```
save "myprocs"
```

At any later time you can bring these procedures back into the computer's memory by typing:

```
load "myprocs"
```

The **load** command deletes any existing procedures in memory before loading the new ones from the Microdrive cartridge. If you want to add the new procedures to those already in memory, you can use the **merge** command. For example:

```
merge "myprocs"
```

This works like **load**, except that the existing procedures are not deleted. If a new procedure has the same name as an existing one, the new one will replace the old version.

EXAMINING FILE RECORDS

Renaming commonly used commands with single-character names is one way of making life easier for yourself. An alternative would be to write a longer procedure to replace several commands by single key presses. Try using the **edit** command to define the following procedure. It allows you to open and examine any of your data files, providing, of course, that the file you wish to use is not already loaded.

If you have already defined a procedure, typing:

```
edit
```

will not automatically give you the option to create a new procedure. From within **edit** you must press **F3** and then the N key to start a new procedure.

Don't worry if you make a few mistakes while typing in the example – you will learn how to correct them in the next chapter.

```
proc vufile
  cls
  input "which file? ";file$
  look file$
  display
  let key$="z"
  while key$<>"q"
    sprint
    let key$=lower(getkey())
    if key$="f":first:endif
    if key$="l":last:endif
    if key$="n":next:endif
    if key$="b":back:endif
    endwhile
  close
  endproc
```

Remember that you leave **edit** by pressing ESC twice.

You can use the procedure by typing:

```
vufile
```

It will first clear the display area and then prompt you to type in a file name such as "gazet". If "gazet" is already loaded, however, you will receive an error message. To recover, type **new** and load and run the procedure again. When you have entered the name of one of your data files the procedure will open that file in read-only mode and display its first record. It will then wait for you to press a key and will respond to the keys f, l, n, b or q. The first four of these will cause the appropriate display action (first, last, next or back) and pressing the q (quit) key will close the file and end the procedure.

Since this is the first program of any great length that we have written, a few comments might prove helpful. First note how the example is indented to clarify the structure of the procedure. There is no need for you to type it like this, the indents are added automatically as you write, list or print the procedure.

The main part of the procedure (waiting for a key to be pressed and performing the appropriate action) is enclosed between **while** and **endwhile** commands. This repetitive loop will only be left when the condition following **while** is false, in this case, when you press the q key.

The **if** command, used several times within this loop, also requires that each **if** has a matching **endif** to mark the end of the sequence of instructions to be executed if the condition is true. **If** and **endif** are separate commands and can be used on different lines. We could, for example, have written the first of the **if** statements in this procedure as:

```
if key$="f"
  first
  endif
```

You may include several lines of statements between **if** and **endif**; they will all be executed, provided the condition following if is true. In the **vufile** procedure these statements are sufficiently short that each can be written on a single line, using the colon to separate the individual statements.

As you can see, a **sprint** command is used within the main loop of this procedure to make sure that each new record is shown on the screen. Remember that, although the display commands (**first**, **last** etc.) always move to the correct record, the data in the display area is not automatically changed until the end of the procedure. If we had not included the **sprint** command, no information would have been shown in the display area until you pressed the q key to leave the procedure. In that case all you would see would be the result of the last of any sequence of keypresses that you have made.

CHAPTER 9
EDITING

This chapter describes the program editor. We shall include a few simple examples, but the best way to learn is by using them yourself. Start by typing **new** to clear the computer's memory.

When you have read this chapter you could try writing a few simple programs of your own, or you could try modifying the procedures you typed in while working on the last chapter. If you want to use longer examples you could use the editor to type in all or part of the programs in the following chapters.

THE PROGRAM EDITOR

You enter the *main level* of the program editor with the **edit**

As an example we can create a procedure and add a couple of statements to it. From the main level of **edit**, press **F3** and N to create a new procedure. Type in **test** when prompted for the name of the procedure.

Press **ESC** twice to leave the editor without adding any statements. Then use the **edit** command again. If you have no other procedures loaded, the screen will show:

```
test   proc test
          endproc
```

If the procedures you created in the last chapter are still loaded, then **test** is highlighted on the left as the current procedure among these other procedures. Press **F4** to insert lines of text. The line containing **proc** will be highlighted.

Now type:

```
print "this is a test" ENTER
print "there are two statements" ENTER ENTER
```

Pressing **ENTER** twice in succession takes you out of **insert**. When you have finished the screen will look like:

```
test   proc test
          print "this is a test"
          print "there are two statements"
          endproc
```

The line containing the second print statement is highlighted.

Remember that until you press **ENTER** you can use the line editor to correct any text that you type. However, once you have pressed **ENTER** the line is inserted into the procedure. To get it out again to **edit** it you must press **F5**. Pressing **ENTER** will then replace the old line with the new line.

You are not allowed to edit the **endproc** statement at the end of the procedure. You are also not allowed to edit the word **proc** but you may edit the rest of the contents of this line. You can, therefore, rename a procedure by using the line editor to delete the old name and replace it with a new one. The list of procedures at the left of the screen is rearranged automatically to keep the procedures in alphabetical order.

Editing Commands

There are four separate editing commands which you will have noticed in the command section when creating a new procedure. You can select one by pressing **F3** and then typing the first letter of its name.

New Procedure (N)

You type in the name of the procedure you want to create. If you type in the name of an existing procedure, you will not be allowed to create a second procedure but will be offered the option of editing the existing procedure.

When you press **ENTER** at the end of the name the new procedure becomes the current one, listed at the right of the screen. You are presented with an empty procedure – that is, one containing only the **proc** and **endproc** statements.

Delete Procedure (D)

This command deletes the current procedure from your program. You must first select the procedure you want to delete by using the **SHIFT** and **TABULATE** keys, as described earlier, to make it the current procedure. You then select the command by pressing **F3** and then the D key.

You must press **ENTER** to confirm that you really do want to delete the procedure. If you change your mind at this stage you can press any other key to go back to **edit** without deleting the procedure.

Be careful when you use this command since there is no way to restore a deleted procedure, except by typing it in again.

Cut (C) This command removes one or more lines of text from the current procedure. The text that is removed can be inserted in another position, or even in another procedure, by means of the **paste** command.

Before you select the command you should use the up and down cursor keys to make the current line either the first or the last line of the section you want to remove. You can then select the command by pressing F3 and then the C key.

If you then press ENTER the current line will be removed from the procedure. Alternatively you can use the up or the down cursor key to move the cursor to the other end of a section of text that you want to remove. The region of text that will be removed is marked by highlighting. When you have marked the text you want to remove you should press ENTER. Archive will immediately remove the marked text.

Paste (P) This command inserts the text removed by the last use of the **cut** command into the current procedure, below the current line. The text can be inserted in another position, or even in another procedure.

Before you select the command you should, if necessary, use the SHIFT and TABULATE keys to select the procedure in which you want to insert the text. You should also use the up and down cursor keys to highlight the line immediately above the position where you want to insert the text.

Archive immediately inserts the text, underneath the current line. When you have used **paste** to insert the text, the paste buffer is empty. You can not, therefore, insert the same text in more than one position.

FOR SERVICE MANUALS
CONTACT:
MAURITRON TECHNICAL SERVICES
www.mauritron.co.uk
TEL: 01844 - 351694
FAX: 01844 - 352554

CHAPTER 10
PROGRAMMING IN ARCHIVE

This chapter will describe the development of an actual working example and each new technique will be described as it is needed.

Suppose you are involved in running a club or society which charges a subscription and produces a newsletter. You will need to send a copy of each issue to every paid-up member. You will also need to send a reminder to each member when his or her subscription falls due.

This example allows you to construct a mailing list and then print a set of address labels on request. The address label includes a reminder when a subscription is due. The example assumes that you send out six issues of the newsletter per year and that a person's subscription falls due when he or she has received six issues. It could easily be adapted to any situation where you regularly send out some form of circular letter to a number of people on a mailing list.

A MAILING LIST

In this example we shall make as much use as possible of the existing facilities and introduce some new ones. If you need help with a feature or command you have not yet encountered, or one that seems to do things you don't understand, you may now find it quicker to look for help in the reference section or use the **help** function by pressing F1. We use the **insert** and **alter** commands for all additions and changes to the file records. We shall, however, need to write special routines to print out the address labels.

We shall have to cater for the following set of requirements:

Add a new record to the file.
Delete a record.
Modify a record.
Record subscription payments.
Produce the address labels.
Leave the program.

We shall write a procedure to handle each of these tasks and link them together by another procedure which will allow you to select any of these options.

In this application it is quite clear what fields each record must contain. The name and address are essential plus one field to record the number of issues the person has received. We can create the necessary file immediately, as shown below.

```
create "mail"
   title$
   fname$
   surname$
   street$
   town$
   county$
   postcode$
   issues
   endcreate
```

We have used three string fields for the person's name; to hold the title (Dr, Mr, Mrs etc.), the first name and the surname respectively. We could probably have managed with just a single field.

There are four string fields for the address, nominally reserved for the street address, the town, county and postcode. You do not always have to use them in this way, but can treat them as four general fields to hold the address. Four fields should normally be quite sufficient.

There is only one numeric field, to hold the information about how many issues remain to be sent.

Now that we have the file, we can use it to test the various procedures as we write them. It is a good idea to test each procedure as far as possible as you go along. You can then spot each mistake as it occurs and correct it immediately. If you leave all the testing to the end it will be much more complicated as several things may be going wrong at the same time. Keep things as simple as possible while you are still testing your procedures. Try to make sure that each procedure works correctly before you move on to the next one. That way you will find that your final program will usually work as soon as you have written the last procedure.

Insertion

We do not need to write a procedure to add a record. We can use **insert**. Remember that you must use **sprint** to force the display of the contents of the record from within a procedure. You can use **insert** immediately to add a few records to the file so that you can test the other procedures on a real file.

Deletions

At some time you will want to remove the records of people who have not renewed their subscriptions. We shall write a procedure, **wipe**, which allows you to scan through the file, examining the records of all people who have not renewed, and to decide which should be deleted.

We shall use the field variable **issues** to hold the number of issues that a person is entitled to receive. All records for which the value of **issues** is zero are therefore candidates for deletion.

```
proc wipe
  rem ***** delete non-paying subscribers *****
  cls
  display
  select issues =0
  all
    sprint
    print at 10,0; "DELETE (y/n)? ";
    let ok$ =lower(getkey())
    print ok$
    if ok$ ="y"
      delete
      print "DELETED"; tab 15
      else
      print tab 15
      endif
    endall
  reset
  endproc
```

Since a deleted record cannot be recovered, the full contents of the record are displayed and you are asked to confirm that you really want to delete it. We use the **getkey()** function which waits for a key to be pressed and then returns the ASCII code of that key. Note that **lower()** converts the code to the lower case character so that you can type the letter in either upper or lower case.

Once you are satisfied you have correctly entered this procedure, you may try it out on your file, (provided, of course, that you have entered some test records). First, leave **edit** by pressing **ESC** (twice if necessary) and save your procedure in a file called "Maillist".

Type:

```
save "Maillist"
```

The procedure called **wipe** is now stored and can be called whenever "Maillist" is loaded.

After entering each of the following procedures, repeat these steps, each time storing the new procedure in "Maillist".

Payments

You will normally want to record a batch of subscription payments from a list of names and addresses. You will therefore need to get the record of a particular person. The quickest way is to write a separate procedure, **getrec**, to locate a particular record and then incorporate it in a **pay** procedure.

The **getrec** procedure asks for a text string (n$) and then locates the first record in the file which contains that text. If you reply by just pressing **ENTER**, n$ is set to the empty string and no search is made. This will, however, indicate that you have finished recording payments.

. From the **edit** level, press F3 and N to start entering **getrec**.

```
proc getrec
  rem ***** locate a particular record *****
  cls
  let ok$ ="n"
  input "who? "; n$
  if n$ <>""
    find n$
    while ok$ <>"y" and found()
      print title$ ; " "; fname$(1); " "; surname$
      print street$
      print "OK (y/n)? ";
      let ok$ =lower(getkey())
      cls
      if ok$ <>"y"
        continue
        endif
      endwhile
    if not found()
      print n$ ; " not found"
      endif
    endif
  endproc
```

The search uses the **find** command, so that the text is found in any string field. You can therefore identify a record by name or by address. Of course, the first record which matches may not be the one you want, so we have to be able to continue the search. This is the purpose of the **while endwhile** loop. This prints out the name and first line of the address, to identify the record, and asks you if that is the right record. If you do not respond by pressing the Y key, it continues the search. The loop ends either when you answer by pressing the Y key or when the text is not found in any of the remaining records. Note that the function **found()** returns a true (non-zero) value if the search is successful.

Since ok$ could initially be "y" (from a previous successful search) we must give it some other value at the beginning of the procedure, before entering the loop. This makes sure that the loop will be used at least once.

We can now write the **pay** procedure:

```
proc pay
  rem ***** record subscription payment *****
  cls
  let n$ ="x"
  while n$ <>""
    getrec
    if ok$ ="y"
      let issues =issues +6
      update
      endif
    endwhile
  endproc
```

The loop in this procedure continues until n$ is an empty string. This allows you to record several payments without having to select the **pay** option for each one. When you have finished, just press **ENTER** in response to the "who?" prompt. If the value of ok$ is "y" after the call to **getrec** then the payment is recorded by marking it as valid for a further six issues.

Again we have to set the initial value of n$ to some appropriate value (anything except the empty string) to make sure that the procedure is not affected by a previous operation.

Changes

The procedure to allow you to change the contents of a record is now very easy. Again you must be able to select a particular record to change, so the general structure can be identical to **pay**.

```
proc change
  rem ***** alter record *****
  let n$ ="x"
  cls
  while n$ <>""
    getrec
    if ok$ ="y"
      alter
      cls
      endif
    endwhile
  endproc
```

FOR SERVICE MANUALS
CONTACT:
MAURITRON TECHNICAL SERVICES
www.mauritron.co.uk
TEL: 01844 - 351694
FAX: 01844 - 352554

PARAMETERS

We shall now take a short break from the development of the program to describe the use of *parameters* with procedures. You can use a *parameter* to pass a value to a procedure, rather than using the value of a variable. We shall show you a few examples of how they can be used. You do not need to save these procedures in "maillist" and you may delete them before moving on to the section of the program which deals with labels.

Try the following simple example. Using the line editor, you add the parameter to the line containing the procedure name.

```
proc test; a
  print 5*a
endproc
```

This defines a procedure called **test** which requires one parameter, "a". Notice that the parameter is separated from the name of the procedure by a semicolon. Whenever you use the procedure you must always supply a value for the parameter. For example, you could type:

```
test; 3
```

which will print the value 15 – the number (3) has been passed to the procedure as the value of the variable a.

You may specify any number of parameters for a procedure, provided you separate them by commas. For example:

```
proc trial; a,b,c
  print a * b * c
endproc
```

which you can call by:

```
trial; 3,4,5
```

The values you supply do not have to be literal values, but could be variables, as shown below:

```
let x = 2
let y = 5
let z = 7
trial; x,y,z
```

Note that the names of the variables do not have to be the same as the names used within the procedure. We can distinguish between the *formal parameters* (e.g. a,b,c) in the definition of the procedure, and the *actual parameters* which are the actual values that are passed to the procedure.

You can also pass the results of expressions:

```
trial; x*2,z/y,(z-y)*x
```

You are not restricted to using numeric variables but can also pass strings (or string expressions) as parameters, provided you specify string variables in the definition of the procedure. For example:

```
proc try; a$
  print a$
endproc

let t$ = "message"
try; t$
```

The only requirement is that the number and types of parameters supplied must match the list of formal parameters in the definition of the procedure.

Address Labels

The reason for the brief interlude about parameters is that they give a neat way of writing the procedure to print an address label. For the purposes of testing we shall first write the procedure to show the addresses on the display and later convert it to send the output to the printer. We shall assume that the labels are eight lines of print-out in length. If this is not right for your printer and label combination you will have to change the number of lines of space in the procedure so that it matches your requirement. Remember to start saving your procedures in "Maillist" again.

First we shall write a procedure that displays a single line, the contents of which are passed via a parameter.

```
proc doline; x$
  print x$
  endproc
```

We can now use this procedure to display eight lines of text for the address label.

```
proc dolabel
  rem ***** print labels *****
  if issues
    if issues =1
      doline; "REMINDER - Subscription Now Due"
      else
      doline; ""
      endif
    doline; ""
    doline; title$ +" "+fname$ (1)+". "+surname$
    doline; street$
    doline; town$
    doline; county$
    doline; postcode$
    doline; ""
    let issues =issues - 1
    update
    endif
  endproc
```

The procedure includes a reminder in the address label if the person is about to receive his or her last issue. Each time a label is printed, that person's issue count is reduced by one. If this number has reached zero then the label is not printed.

You can begin to see how useful parameters can be – without them this procedure would be much longer. Look how easy it is to combine the title, initial and surname for the first line of the address.

Perhaps you are wondering why we went to the trouble of defining **doline** when we could have just used **print** statements throughout **dolabel**. The reason is that the routine in its present form shows the addresses on the display screen. We can convert it to send its output to the printer merely by changing one line in **doline**, instead of having to change every print statement in **dolabel**. All we need to do is change **doline** to read:

```
proc doline; x$
  lprint x$
  endproc
```

Finally we can write the procedure to print all the address labels:

```
proc despatch
  cls
  all
    dolabel
    endall
  endproc
```

Leaving the Program

The final option is to leave the program when you have finished. This procedure can be very simple – all it has to do is to make sure that the file is closed properly before returning control to the keyboard. We have also added a short sign-off message to make it clear that the program has ended.

```
proc bye
  close
  print "bye"
  stop endproc
```

ERRORS

It is quite likely that sooner or later you will make an error while using this program. You may, for example, accidentally press the **ESC** key or you may type in some text when a number is expected. This type of mistake is detected by Archive and normally results in the display of an error message and a return from your program to the keyboard.

You can use the **error** command to mark a procedure to be treated specially if any error is detected. Any error occurring in the marked procedure, or any procedure that it calls, results in an immediate, premature, return.

The normal method of handling errors is *switched off* for the marked procedure and it is left to you to decide how to deal with it. You can find out the number of the last error that occurred by using the **errnum( )** function. You can use it to read the error number more than once as the value is only cleared to zero by the next use of the **error** command. If no errors have occurred since the start of the program, or since the last time **error** was executed, then **errnum( )** will return a value of zero.

This method, although not easy to understand at first, gives you a very powerful and flexible control of how to deal with errors. The following example shows a typical way of using **error**. It gives you an error-resistant method of inputting a number.

```
proc dotest
  input x
  endproc

proc test
  let n =1
  while n
    error dotest
    let n =errnum()
    if n
      print "You made error number " ;n ;", try again"
      endif
    endwhile
  endproc
```

The first procedure simply waits for your input to the variable x. The second procedure handles any error during the execution of the input procedure. If any error occurs within **dotest** it will be terminated prematurely and the error number will be set. This number is then read by **errnum( )** and, if it is non-zero, the error message is printed (this error message could, of course, be anything you like). Since these statements are enclosed in a **while endwhile** loop, any error will cause them to be executed again. The error number is cleared by **error**, ready for the next try. You can not leave **test** until you have typed in a valid number.

This example reports the number of the error that was detected. On most occasions you will not be concerned about which error occurred. The main use of **errnum( )** is to differentiate between there being no error and there being a detected error of any type. A list of error numbers and possible explanations is included in the *Reference* chapter.

We can now write a procedure which will allow you to select any one of the six options with a single keypress. It is sufficiently simple that no explanation is necessary.

```
proc choose
   rem ***** choose an option *****
   cls
   print
   print " Add Despatch Pay Change Wipe Quit";
   print "? ";
   let choice$ =lower(getkey())
   print choice$
   if choice$ ="a": insert : endif
   if choice$ ="d": despatch : endif
   if choice$ ="p": pay : endif
   if choice$ ="c": change : endif
   if choice$ ="w": wipe : endif
   if choice$ ="q": bye : endif
   endproc
```

All that remains to be done to complete our program is to write a start-up procedure which opens the file and calls **choose**. We must include **choose** in a loop so that you are offered the options again, each time you complete your previous selection.

You will see that the **while endwhile** loop in the following procedure will never end. Such a loop will only come to an end when the expression following **while** has a zero value. In the above procedure the expression always has the value 1, so the loop will continue indefinitely. The only way of leaving this loop is to choose the Quit option. The **stop** command in **bye** immediately returns control to the keyboard.

```
proc start
   ***** rem start procedure *****
   cls
   open "newmail.dbf"
   while 1
      error choose
      let n =errnum()
      if n
         print "Mistake - Press any key to continue"
         let m$ =getkey()
      endif
   endwhile
endproc
```

Within this loop is a sequence of statements which handles any errors, using a similar method to that described in the previous section. If you make a mistake the program will not continue until you press a key. This allows you to look at what you have just done so that you can find out how you made the error.

THE RUN COMMAND

The main procedure in the mailing list program is called "start". This is so that you can use the **run** command when using the program. We have already used this command when we used the "loader" program to load the "gazet" data file

Save this final procedure in "maillist". When you want to run the program you will need to load the procedures into the computer's memory and then execute the main procedure, which will call all the others. One way is to use the **load** command and then type in the name of the main procedure, for example:

```
load "maillist"
start
```

The **run** command will load a named program and then automatically execute the procedure called "start" (if it exists). You can run the program exactly as in the previous example just by typing:

```
run "maillist"
```

The remaining two sections of this chapter include some general purpose procedures which you may find useful.

LOCAL VARIABLES

Most variables that appear in procedures are *global*. This means that they are recognised throughout the program. They may be used or changed in any procedure, and not just the procedure in which they are first assigned a value.

The variables used as formal parameters in a procedure are *local variables* and they are not recognised outside the procedure in which they appear.

The following example may help to make the distinction clear. Before going on, type **new** to clear the computer's memory. First we create a procedure which uses two local variables a and b$, as well as assigning values to two normal (global) variables u and v$.

```
proc demo; a,b$
  print a,b$
  let u=3
  let v$="text"
  print u;v$
  endproc
```

Then we use **demo**:

```
demo 5;"words"
```

All four values are printed showing that all four variables are recognised inside **demo**. Typing

```
print u;v$
```

shows that both of these variables are also recognised outside the procedure. However, typing

```
print a,b$
```

results in an error because a and b$ are not recognised outside **demo**. All formal parameters are local variables, but you can also declare other variables to be local, as in the following example:

```
proc dumbo
  print "inside dumbo"
  print p; q; r
  endproc
proc dummy
  local q,r
  let p = 2
  let q = 3
  let r = 4
  print "inside dummy"
  print p; q; r
  dumbo
  endproc
```

If you attempt to use **dummy** by typing:

```
dummy
```

you will find that the values of p, q and r are all recognised (and therefore printed) in **dummy**, but **dumbo** does not know the values of q and r, which are local to **dummy**.

The values of local variables are not defined anywhere except in the procedure in which they are declared – not even in procedures called from the declaring procedure. The variable p is global and is recognised everywhere.

You may be wondering why local variables are necessary. To illustrate their usefulness, suppose you write a program containing several procedures that you, or someone else, originally write for use in other programs. It is quite possible that two or more of these procedures might use variables with the same name for quite different purposes. If these variables were global then one procedure could alter a value so that it would be wrong for another. In such a situation you would have to check all the procedures that you use and, if necessary, change the names of the variables. If, however, the variables were local it would not matter if they had the same name. Provided they were in different procedures, changing one would have no effect on the other.

Furthermore, it does not matter if a procedure calls another which uses the same name for a variable – provided at least one of them is local. For example, the procedure **choose** in the section on errors, earlier in this chapter, declared the variable choice$ to be local. This means that there is no need to check whether any of the many procedures called by **choose** also use choice$ – the called procedures cannot change the value of choice$ in **choose**.

PROMPTS

Displaying a prompt and waiting for a key to be pressed is one of the most commonly needed actions, so it is worth writing a general-purpose procedure. The procedure must be able to display a wide range of messages. A simple way of allowing the procedure to print any message is to pass the message to the procedure in the form of a parameter.

```
proc prompt; m$
  print m$ + ": ";
  let x$ =lower(getkey())
  print x$
  endproc
```

The message to be displayed is passed to the procedure as a parameter in the local variable m$. The function **getkey( )** waits for a key to be pressed and returns the ASCII code for the key. In this procedure the ASCII code is converted to lower case by the function **lower( )**, so that the result is independent of upper or lower case. Finally the resulting value is assigned to the variable x$. This is a global variable, so that the key that was actually pressed is available to any other procedure in the program.

PAUSE

A useful procedure is **pause**. It uses **prompt** to print a message and then simply waits until a key is pressed. Since you are not usually interested in knowing which key was actually pressed, it uses a local variable, y$, to preserve the original contents of x$.

```
proc pause
  rem ***** wait for any key *****
  local y$
  let y$ =x$
  print
  prompt; "press any key to continue"
  let x$ =y$
  endproc
```

DATA ENTRY

Text

Accepting text as typed input is quite simple. Any collection of characters is a valid text string (even if it does not make sense) and will not cause an system error. You will not normally need to take any special precautions when accepting text input. It will usually be sufficient to use a line such as the following, which asks you to type in your name:

```
input "Please type your name: ";name$
```

Note that a space is included as the last character of the prompt text; this small point makes a lot of difference to the appearance of your program when you use it.

You can input several items with one input statement. All you have to do is to include all the prompts and variable names, separated by semicolons.

```
input "Your first name? ";fname$;"Your surname? ";sname$;
```

This last input statement also ends with a semicolon – this stops the cursor moving to the following line after you have typed your input.

Numbers

When you use the **input** command to enter text to a string variable the computer will accept anything that you type, without complaint. If, however, you try the same thing with input to a numeric variable you will get an error message if you type anything except a valid number. Assuming that you do not want to leave your program every time your finger slips while you are typing in a number, you must make sure that your program can cope with such errors.

The most useful way is to make use of the **error** command, which was described earlier. The following procedure, for example, will accept any valid number within a specified range. It even provides the display of any prompt message you want to appear.

```
proc getnum; m$,min,max
  rem ***** get number in range *****
  local wrong
  let wrong=1
  while wrong
    print m$; "? ";
    error readnum
    let wrong=errnum()
    if not wrong
      if num<min or num>max
      let wrong=1
      print "Allowed range is ";min;" to ";max
      endif
    endif
    if wrong
      print "Try again"
      endif
    endwhile
  endproc
```

Since **error** must be followed by the name of a procedure, we define **readnum** to input a value for the variable **num**.

```
proc readnum
  input num
  endproc
```

Suppose you want a procedure that checks that a number is within the range 1 to 10. You can do this using **getnum** in the following way:

```
proc check
  getnum; "Numeric value?",1,10
endproc
```

CHAPTER 11
USING MULTIPLE FILES

LOGICAL FILE NAMES

This chapter extends the explanation of how to use the Archive programming language by describing how to work with two or more open files. When you have more than one file open at the same time you must be able to identify which file you want to use for any particular operation. You must give each file a unique *logical file name* when you open or create it and then refer to it by that name in all commands that refer to the file.

Archive automatically supplies the *logical file name*, "main", when you open a single file. It is called a logical file name to distinguish it from the *physical file name* – the name you give to the file when you save it.

Since a program refers to a file by its logical file name, you can write a program that will work with several different files.

Logical file names are essential for multiple file operations since you can only open a second file by using both its physical file name and its logical file name. Note that the logical file name is not saved with the file when it is closed and must be specified each time the file is opened.

Two or more data files could contain fields with the same name. When this happens you can identify the file to which the field belongs by adding the logical file name to the field name. For example, if the field country$ appears in two files whose logical file names are "main" and "b" you could refer to each of them respectively as "main.country$" and "b.country$".

CHANGING THE RECORDS OF A FILE

The first example demonstrates how to add, delete or rename fields within an existing file.

Suppose that you want to make some changes to the "gazet" file, to create a new file containing only European countries. The "continent$" field becomes irrelevant and we need not include it. We shall also rename the "pop" field as "population".

The most convenient way of changing the file is to create a second file containing the fields you want and then to copy the required records from the old file to the new one. Let us call the new file "europe". The following procedure will do the rest of the work.

```
proc start
  rem ***** create europe file *****
  create "europe" logical "e"
    country$
    capital$
    languages$
    currency$
    population
    gdp
    area
    endcreate
  look "gazet" logical "g"
  select continent$="EUROPE"
  all "g"
    print at 0,0;g.country$;tab 30
    let e.country$=g.country$
    let e.capital$=g.capital$
    let e.language$=g.language$
    let e.currency$=g.currency$
    let e.population=g.pop
    let e.gdp=g.gdp
    let e.area=g.area
    append "e"
    endall
  close "e"
  close "g"
  print
  print "DONE"
  endproc
```

THE CURRENT FILE

You can see, from this example, that you can use the same name for a field in both files – they can be distinguished by including the logical file name. If you do not include the logical file name then it will be assumed that the *current file* is to be used. The last file to be opened automatically becomes the current file. In this example the current file will be "gazet" (with logical file name "g") so we could make use of this by simply writing the g before the field name in the previous program.

If you do not include the logigal file name in any case where it is optional, Archive will assume that the command refers to the current file. It is usually safer to include the logical file name explicitly, to avoid any possibility of confusion.

You can, at any time, specify the current file by means of the **use** command. If you included the command:

```
use "e"
```

in the above example, then "europe" would be the current file until you changed it again, either by opening another file or by means of the **use** command.

STOCK CONTROL

Now for a more complex example. In a stock control system you will need to:

Find information on a particular stock item.
Obtain a report on the current stock levels of all items.
Record sales and modify the stock records accordingly.
Order new supplies, to maintain adequate stock levels.
Record deliveries of stock.

You will obviously need a file to hold the details of all items held in stock and it is convenient to have a second file to hold details of all your suppliers. You will need to be able to access either file from the other – for example you may want to know all the possible suppliers of a particular item, or to find out what items are supplied by a particular company.

In order to keep the application as simple as possible we shall not use the menu-driven approach of the examples in the previous two chapters. We shall write it as a series of separate commands which can be used – like the standard commands – by typing their names.

Since the procedures will be strongly dependent on the file structure we use, we must first give some thought to their appearance.

The Stock File

The stock file must contain full details of the stock situation for each item. The following list explains all the fields we shall use.

Field Name	Use	Example
`stockno$`	The internal stock code	`A101`
`description$`	Item description	`Widget, large`
`qty`	Number in stock	500
`sellpr`	Selling price	1.25
`reorderlev`	Reorder when stock level falls below this value.	200
`buyqty`	How many to order	400

We can create the file by:

```
create "stock" logical "sto"
  stockno$
  description$
  qty
  reorderlev
  sellpr
  buyqty
  endcreate
```

The Supplier File

This file holds the names, addresses and telephone numbers of the companies that supply the goods you sell. It will be useful also to include the name of a contact person in the company. In order to be able to access this information efficiently we shall include a code for each company. We shall use the following fields:

Field Name	Use	Example
coname$	The company's name	Wonder Widgets plc
street$	First line of address	27 Belmont House
town$	Second line of address	LIVERPOOL
county$	Third line of address	Merseyside
postcode$	Last line of address	L31 2HK
contact$	Name of a contact	Andrew Cummins
tel$	Telephone number	051-532 7133
code$	Your code for the company	a

We can create the file by:

```
create "supplier" logical "sup"
  coname$
  street$
  town$
  county$
  postcode$
  contact$
  tel$
  code$
  endcreate
```

The Orders File

This file forms the link between the previous two files. It uses the following fields:

Field Name	Use	Example
stockno$	Your stock code	A101
code$	Your code for the supplier	a
scode$	The supplier's code for the item	123-456
price	The supplier's selling price	0.87
delivery	The supplier's delivery time, in days	28

Each record in this file links one record in the stock file with one record in the supplier file. The above example shows that Wonder Widgets (supplier code "a") can supply you with large widgets (stock code "A101"). In addition, we include details of the price, delivery time and the supplier's own stock code. These items are useful when you order more stock.

Using this file allows you to cater for the cases where one supplier supplies more than one stock item (equal values for code$, but different values for stockno$) and where one stock item is obtainable from several suppliers (equal stockno$ but different code$).

Create the file with:

```
create "orders" logical "ord"
  stockno$
  code$
  scode$
  price
  delivery
  endcreate
```

Enquiries

Having created these files, we now need some procedures to handle the information they will contain. You will find that the most frequently-needed facility is to find information about a particular stock item in response to customer enquiries. You will need to find the information as quickly as possible, but may need to find a particular record from either the part number or the description. We shall therefore use the **find** command so that you can give any valid text to start the search.

The procedure must be able to ask for you to confirm that the record is the one you require. We shall delegate this task to a separate procedure, so we can use it in different situations if necessary.

```
proc confirm
  print : print "Confirm (y/n)";
  let yes=lower(getkey())="y"
  cls
  endproc
```

It leaves the variable **yes** containing 1 if you press the Y key – otherwise the value is zero. Note the use of the = sign for assignment and also in a logical condition.

```
proc inquire
  rem ***** inquire stock item *****
  print
  input "Stock item? "; name$
  use "sto"
  find name$
  let yes=0
  while found() and not yes
    display
    sprint
    confirm
    if not yes
     continue
     endif
    endwhile
  if not found()
    print
    print name$; " does not exist"
    endif
  endproc
```

This procedure merely locates the correct record. A more usable procedure for interrogating the stock file is **query**:

```
proc query
  inquire
  clear
  endproc
```

This uses another procedure, **clear**, which waits until you press a key, clears the screen and then prints a list of the commands you can use. We shall leave this procedure until we have written the procedures it must list. Remember to leave **edit** from time to time to save these procedures as you enter them.

Stock Report

We can also write a simple procedure to produce a general stock report.

```
proc report
  rem ***** stock report *****
  cls
  print tab 2; "ITEM"; tab 11; "CODE";
  print tab 20; "QUANTITY"; tab 31; "PRICE";
  print tab 40; "STOCK VALUE";
  print
  let total=0
  use "sto"
  all
    print description$( to 10);tab 11;sto.stockno$;
      tab 20;qty;
    print tab 31;"£";sellpr; tab 40;"£";sellpr*qty
    let total=total+sellpr*qty
    endall
  print
  print "Total stock value =£"; total
  clear
  endproc
```

Recording Sales

All we need to do to record a sale is to subtract the number of items sold from the relevant stock record. It is advisable to include some form of confirmation that we are dealing with the right stock item and that the stock is sufficient to meet the order.

```
proc quantity
   rem ***** print items in stock *****
   inquire
   print
   input "How many? "; num
   print
   cls
   print num;" * ";sto.stockno$;" (";sto.description$;")"
   endproc

proc sale
   rem ***** process sale *****
   quantity
   if num<=sto.qty
   print "Order value:- £"; num*sto.sellpr
   confirm
   if yes
      let sto.qty=sto.qty-num
      update
      sprint: rem *** show the modified record ***
      endif
   else
   print "Not enough stock"
   endif
clear
endproc
```

Recording Incoming Stock

The following procedure allows you to record the delivery of stock. Again it requests confirmation of the details you type in before accepting them and updating the relevant stock record.

```
proc delivery
   rem ***** in case stock on delivery *****
   quantity
   confirm
   print
   if yes
      print "Accepted"
      let sto.qty=sto.qty+num
      update
      sprint
      else
      print "Delivery not recorded"
      endif
   clear
   endproc
```

Ordering New Stock

So far our procedures have only referred to the stock file. When we want to order more stock we shall have to refer to the supplier and orders files for the name and address of the company, the price, and so on.

Assuming that we have identified the item in the stock file (with **inquire**) we select, from the orders file, those records that have the correct stock code. These records contain the codes for all the companies that can supply the item. Since the records also contain the price and delivery time for each supplier, we can decide whether we want the cheapest item or the shortest delivery time.

We use **locate** as a fast way of finding the required supplier record. This means that the supplier file must be ordered (with respect to the supplier code, code$) before we use **doorder**.

```
proc doorder
  rem *****order new stock *****
  inquire
  use "ord"
  select sto.stockno$=ord.stockno$
  print
  print "fast or cheap (f/c)";
  if lower(getkey())="f"
    fast
    else : cheap
    endif
  let ycode$=scode$
  reset
  use "sup"
  locate comp$
  doform
  print
  print "Expected delivery is ";del;" days"
  clear
  endproc
```

The procedure **cheap** finds the supplier with the lowest price, and **fast** works in the same way to find the supplier with the shortest delivery time.

```
proc cheap
  rem ***** find cheapest *****
  use "ord"
  let pri=price
  let comp$=code$
  let del=delivery
  all
    if price<pri
      let pri=price
      let comp$=code$
      let del=delivery
      endif
    endall
  endproc

proc fast
  rem ***** fastest delivery *****
  use "ord"
  let del=delivery
  let comp$=code$
  let pri=price
  all
    if delivery<del
      let del=delivery
      let comp$=code$
      let pri=price
      endif
    endall
  endproc
```

The procedure **doform** produces the actual order form. You should modify it to your own requirements. We shall use a simple version which shows the order details on the screen.

```
proc doform
  rem ***** produce order form *****
  cls
  print
  print sup.coname$
  print sup.street$
  print sup.county$
  print sup.postcode$
  print
  print "Please supply "; sto.buyqty;
```

```
print " * part number ";
print ycode$
print "("; sto.description$; ") ";
print "at £"; pri; " each."
print
print "Total value: £"; sto.buyqty*pri
endproc
```

The final command that we need is one to close all the files when we have finished using them.

```
proc bye
  confirm
  if yes
    cls
    print : print "bye"
    close "sto"
    close "sup"
    close "ord"
    cls
    endif
  endproc
```

We can now write a short procedure to run the application. It must open all three files with the correct logical file names, clear the display and show you the additional commands that you have. Note that, in normal use, the stock file is the only one whose records will need to be changed. The other two files are opened as read only files. It also orders the supplier file so that we can **locate** a company by its reference code.

```
proc start
  cls
  print at 5,5; "STOCK CONTROL DEMONSTRATION"
  print
  open "stock" logical "sto"
  look "supplier" logical "sup"
  look "orders" logical "ord"
  use "sup"
  order code$; a
  clear
  endproc
```

Finally we can write **clear**, which simply clears the screen and shows a list of the extra commands available:

```
proc clear
  rem ***** clear screen and get command *****'
  local x$
  print
  print "Press any key to continue ";
  let x$=getkey()
  cls
  print
  print "Query Report Delivery Doorder Sale Bye": print
  print "Type in your choice"
  endproc
```

CHAPTER 12
QL ARCHIVE REFERENCE

VARIABLES

Variable names may be up to thirteen characters in length, and must not start with a digit (0 to 9). They may contain any combination of upper or lower case alphabetic characters, or digits. Other characters are not allowed, except for $ and . which have special meanings.

If a variable name ends with a $ it is a string variable. Strings may be up to 255 characters in length. If the name does not end with a $ the variable is numeric. A variable name may refer to the contents of a record in a file and is then known as a field variable. Field variables are normally assumed to refer to the current file but may be made to refer to another open file by including a logical file name, separated by a . from the variable's name. Such a field variable is written as:

logical_file_name . field_name

For example **main.continent$**. If a variable name includes a dot then it must refer to a field in an open file. If there is no dot an attempt is made to match the name to an existing variable in the following sequence:

1 a field of the current file
2 a local variable (a parameter in the current procedure, if any)
3 a global variable

An error message is given if no match is found.

SYNTAX

The term *syntax* refers to the exact structure of a command or function. The syntax of a command specifies the parameters that the command needs, in what order they must appear, and the symbols (if any) used to separate them.

This section describes the notation used to express the syntax of Archive's programming language.

EXPRESSIONS

An *expression* is a combination of literal values, variables, functions and operators which results in a single value. A *numeric expression* results in a numeric value and a *string expression* results in a text value. Examples are:

```
3 * y * sin (x) + len (a$)      {numeric}
"abc" + a$ + rept (" - ", 5)    {string}
```

An expression may, as in the above examples, be composed of several sub-expressions. In such a case you may not mix sub-expressions of different types. They must all be string expressions or all numeric.

Syntax Conventions

The syntax definitions are similar to those used to define the syntax of SuperBASIC, i.e.:

Symbol	Meaning
italics	denotes a syntactic entity
[ ]	encloses an optional item
**	encloses items that may be repeated
|	or
{ }	comment

Syntactic Entities

s.lit	literal string
s.exp	string expression
n.exp	numeric expression
exp	expression, either string or numeric
ptm	print item
var	variable name, either string or numeric
lfn	logical file name
fnm	physical file name (up to 8 characters)
pnm	procedure name

A literal string is text enclosed in quotes, for example 'text' or "text".

A string expression is a literal string, or a combination of literal strings, string variables and string functions that results in a text value for example:

```
"fred"+a$+chr(72)
```

A numeric expression is either a number, or a combination of numbers, numeric variables and operators (+, −, *, /, etc) that results in a numeric value for example:

```
(3+x) / sin(y)
```

A print item is one of four possibilities: **at, tab, ink, paper.** A full description of a print item in our syntax notation is:

```
print_item:= | at n exp , n exp
             | tab n exp
             | ink n exp
             | paper n exp
```

Logical file names and procedure names have the same restrictions as variable names. Physical file names must also not exceed eight characters.

As an example of a *syntax definition,* consider the syntax of the order command. In our notation it appears as:

```
order spec:= var; a | d
order order spec *[ , order spec ]*
```

Order therefore needs to be followed by at least one *order specification* which itself consists of a *variable* separated by a colon from a letter which must be either **a** or **d.** In addition you can also include up to three further order specifications provided each pair is separated by commas. Clearly the syntax notation provides a much more compact description.

Note that the syntax notation does not tell you the meaning or purpose of the symbols so you will have to read the rest of the description for each command. The syntax only gives you a formal description of the number and kind of items that go to make up a valid command. In addition the syntax notation does not tell you the maximum number of repetitions allowed for the repeated items. **Order** will accept up to four pairs of a variable and a letter.

ARCHIVE DATA FILES

A Field

A *field* is the space reserved to hold either a string or a number.

In Archive, each field is identified by a field variable name. Whether a particular field can hold a string or a number is dependent on the name given to the field at the time it was created – string fields have a name ending with a **$**. An Archive string field may hold up to 255 characters. A numeric field has a name that does not end with a **$** sign. All numbers are stored in the same amount of space, regardless of their value. The possible range for a number is the same as the valid numeric range for the arithmetic operators.

A Record

A *record* is a collection of fields, whose contents are related in some way. The fields of a record might, for example, be used to hold the name, the address and the telephone number of a particular person. In Archive the records are of *variable length* so that each record only takes up as much room as is necessary to hold the information contained in its fields. There may be up to 255 fields in an Archive record.

A File

A *data file* is made up from a number of related records. To continue the above example, a data file could consist of a collection of name, address and telephone number records for many different people. The number of records in an Archive data file is limited to roughly 15 000. In practice, you are limited to the capacity of one Microdrive cartridge, which will hold about 1000 records of 100 characters. A file is the basic unit that you can save on, or load from, a Microdrive cartridge. Each file has a name to identify it. In Archive you give a physical name to the file when it is created, but you can change the logical name at any time.

Opening and Closing Files

When you want to read from or write to a data file you must first *open* it. Generally speaking, opening a data file transfers a copy of the file from the Microdrive cartridge into memory although, in the case of a long file, it is possible that only part of the file will be present in memory at any one time.

You can open a data file in *read only* mode with **look** which, as its name suggests, means that you can not change its contents. You also have the option of opening a data file in *update* mode with **open** so that you are allowed both to read and to change its contents.

Every time you open a data file, Archive reserves space for the field variables needed by a record within the file. The field variables always contain the values of the current record.

When you close a data file with **close** or **quit** any changes that you have made are copied into the file stored on the Microdrive cartridge. The copy held in memory is discarded. Closing a file is the only way of ensuring that the copy on the Microdrive cartridge contains your latest version. Since an open file uses part of the computer's memory, you should not leave files open if you are not using them.

When you leave Archive with the **quit** command, all open files are closed automatically.

Do not turn off the computer, or remove a cartridge from a Microdrive, while the cartridge contains open files.

Logical File Names

Each open data file has an associated *logical file name*, given to it when the file is opened. If you do not specify a logical file name when you open the file, it is automatically given the logical file name "main".

The logical file name is used to identify a particular file when you are using several files at once.

PROCEDURES

A procedure is a named section of program, starting with a procedure declaration of the form:

proc *pnm*[; *var* *[, *var*]*]

and ending with:

endproc

It may be referred to by name from any other program or procedure, including itself. It acts as though its code had been inserted at the point from which it is called.

In Archive, the **proc** and **endproc** commands cannot be entered directly at the keyboard, but are added automatically when you use the program editor to create a procedure.

THE PROGRAM EDITOR

The program editor is entered using the **edit** command.

If there are no procedures present in memory, you will be immediately offered the option of creating a new procedure. Otherwise you are given a list of all the procedures in memory on the left hand side of the display area. The first procedure is highlighted and is listed in full on the right hand side of the display. The first line of the procedure is highlighted to mark the current procedure and the current line.

Once in **edit** you have five options:

Select a procedure
Press **TABULATE** to move down the list of procedures, press **SHIFT** and **TABULATE** to move up the list. The listing on the screen always shows the current procedure.

Select a line
Use the up and down cursor keys to select a line within the current procedure. The current line is highlighted.

Press F3 for the menu of editing commands.
There are four commands, which are selected by pressing the key corresponding to the first letter.

Delete Press **ENTER** to delete the procedure highlighted on the left of the display. Press any other key to leave the command without deleting the procedure.

New Type in the name of the new procedure and press **ENTER**. If a procedure of that name already exists you will be offered the opportunity to edit it.

Cut Removes text from the current procedure and transfers it to the paste buffer. Before calling this command use the up or down cursor keys to make the first (or last) line of the region to be removed the current line. Then use the up and down cursor keys to mark the region of text to be removed. Press **ENTER** to remove the text into the paste buffer.

Paste Copy the contents of the paste buffer into the current procedure below the current line. **Paste** will clear the paste buffer.

Insert text
Press F4 to insert one or more lines of text below the current line in the current procedure. Type the text and press ENTER. Pressing ENTER without any preceeding text will leave the insert option.

Edit text
Press F5 to edit the current line of the current procedure. The line of text is copied into the input line and can be edited with the line editor. Press ENTER to replace the old line with the new line.

THE SCREEN EDITOR

The screen editor is entered with the **sedit** command. It allows you to design a new screen layout or modify an existing one. Once you have designed a layout you can save it on a Microdrive cartridge with the **ssave** command and load it with the **sload** command.

A screen layout is composed of two parts, the fixed background text and the variable values that are displayed in it. The **screen** command shows the background text and the **sprint** command adds the current values of the variables it contains.

Sedit has two options:

type text into the screen background
press F3 to use a screen editing command.

There are four screen editing commands available after pressing F3:

C – clear the screen
V – mark a region to show a variable
I – set the ink colour
P – set the paper colour.

A screen layout is made active by:

```
sload
screen
```

When a particular screen is active it will show the current values of its variables after **sprint**, or when control returns to the keyboard after executing a program (or a command). A screen layout is made inactive by clearing the screen with **cls**. If there is no active screen, **sprint** has no effect. You may only have one screen layout in the computer's memory at any one time.

The **display** command creates and uses its own screen layout. It will therefore replace any other screen layout with its own design.

THE COMMANDS

The following commands are available.

ALL

Scans through the logically present records of the file in the fastest possible time.

Syntax: **all** [*lfn*] : ... : **endall**

This scan will not, in general, be in any particular sequence. The optional logical file name will force it to refer to a specified open file. If the logical file name is not given then it will scan the current file.

The **all** loop is primarily designed for examining the file records rather than for changing them. Do not use **update** within an **all** loop, unless you are sure that the length of the record will remain unchanged. You may, for example, change the value of a number, or convert a text field to upper case. If in doubt, use a **while** loop – using the value of **eof( )** to detect the end of the file. For example

```
first
while not eof( )
  ...
  update
  ...
  next
  endwhile
```

ALTER

Alters the current screen layout to display the current values of the variables.

Syntax: **alter**

You can change the contents of any one or more fields of the current file whose values are shown in the screen layout. Note that it is not necessary for all the field variables to be shown. You can not change a field that is not shown. If none of the field variables appear in the screen, Archive forces a **display** of the file.

First select the field to change by pressing **TABULATE** or **ENTER** until the cursor is at the correct field (variables that are not fields of the file are skipped). You can then type a new value or use the line editor to modify the existing value. Press **TABULATE** or **ENTER** to move to the next field. (Pressing **SHIFT** and **TABULATE** together moves back to the previous field.)

When you have made all the changes you want, press **F5** to replace the old record with the new one. The record is replaced automatically if you press **ENTER**. If the file is ordered the new version of the record is inserted in sequence.

APPEND

Adds a record to the specified file, or to the current file if the logical file name is not given.

Syntax: **append** [*lfn*]

The fields of the record take the current values of the field variables. If the file is ordered, the insertion is in sequence.

BACK

Moves backwards one record in the specified file, or in the current file if the logical file name is not given.

Syntax: **back** [*lfn*]

BACKUP

Makes a copy of the specified file. You should make copies of all your files, to protect against accidental damage or erasure.

Syntax: **backup** *oldfnm* **as** *newfnm*

CLOSE

Closes the specified file, or the current file if no logical file name is specified.

Syntax: **close** [*lfn*]

CLS

Clears the display area and switches off any display screen. See **screen**, **sload**, **sprint**.

Syntax: **cls**

CONTINUE

Continues the previous **search** or **find**, from the record following the current record in the current file.

Syntax: **continue**

CREATE

Creates a named open file whose records contain the fields given by the list of variables specified in the command. You have the option of specifying a logical file name – if you do not the file is created with the logical file name "main".

Syntax: **create** *fnm* [**logical:** *lfn*] : *var* *[: *var*]* : **endcreate**

DELETE

Deletes the current record from the specified file, or from the current file if no logical file name is given.

Syntax: **delete** [*lfn*]

Warning: Use this command with care since you can not recover the deleted record.

DIR

Displays a list of files on a Microdrive cartridge.

Syntax: **dir** [*drive*]

You may specify the Microdrive to be either **mdv1** or **mdv2**. If you do not include the Microdrive name Archive will automatically list the files on the cartridge in Microdrive 2.

Before showing the list of files, Archive displays the volume name of the cartridge (the name you gave when you formatted it).

DISPLAY

Shows the logical file name of the current file and a list of the field names and the values of the field variables for the current record. If the file is sorted, it also shows the sort fields and their sort priority.

Syntax: **display**

The command replaces any existing user-defined screen layout with this list, which becomes the active screen layout.

DUMP

Syntax: **dump** [; *var*] *[, *var*]*

Prints the specified fields of the selected records of the current file in tabular form **ser1** output. If you do not give a list of field variable names, *all* the fields are printed.

You can divert the output to a Microdrive file with **spoolon**.

EDIT

Calls the procedure editor to create a new procedure or to edit an existing procedure.

Syntax: **edit**

ENDALL

See **all**.

ENDCREATE

See **create**.

ERROR

Syntax: **error** *pnm*[; *exp* *[, *exp*]*]

Marks a procedure for the purposes of error-handling. Any error which occurs during the execution of this procedure, or any other procedure which it calls, causes a premature return from the marked procedure. The procedure can determine the nature of the error by using the **errnum( )** function to read the error number. This error number is cleared each time that **error** is executed.

EXPORT

Saves the named fields of the selected records of the current Archive file on a Microdrive cartridge in a form suitable for import to QL Abacus or QL Easel.

Syntax: **export** *fnm* [; *var*] *[, *var*]* [**quill**]

If you do not specify a list of field variable names, *all* the fields are exported. If you include the optional parameter **quill**, (separated by at least one space from the last variable name) the file is exported in a form suitable for import by QL Quill.

The export file is named *fnm* and, unless you specify your own file name extension, Archive uses the extension **__EXP**.

See the *Information section* for a full discussion of import and export.

FIND

Rewinds the file to the beginning and searches for the first record containing a match to the specified string in any string field. The match is independent of upper or lower case text.

Syntax: **find** *s.exp*

You can continue the search with the **continue** command, and determine whether the search was successful by examining the value returned by the **found( )** function.

FIRST

Finds the first record of the specified file, or the current file if no logical file name is specified.

Syntax: **first** [*lfn*]

FORMAT

Formats the cartridge in Microdrive 2 (the right hand drive). It gives the cartridge the name you specified. This name is reported when you subsequently use **dir** to show a directory of the files on that cartridge.

Syntax: **format** "*you specified*"

IF

Allows a specified condition to control subsequent processing.

Syntax: **if** *n.exp* : ... [: **else** : ...] : **endif**

Without **else**.
If the expression is non-zero, the following statements are executed. If the expression is zero execution transfers to the statement following **endif**.

With **else**.
If the numeric expression is non-zero, the statements between **if** and **else** are executed. Otherwise the statements between **else** and **endif** are executed. In either case execution continues with the statements following **endif**.

IMPORT

Reads a file, *name1*, exported from QL Abacus or QL Easel and produces an Archive data file *name2*. As with **open** and **look** you have the option of specifying a logical file name for the data file.

Syntax: **import** *name1* **as** *name2* [**logical** *lfn*]

where: *name1*:= *fnm*
name2:= *fnm*

See the *Information section* for a full description of import and export.

INK

Sets the foreground colour for all following text to the colour specified by the value of the expression.

Syntax: **ink** *n.exp*

The colours are: 0 and 1 black
2 and 3 red
4 and 5 green
6 and 7 white

If the expression evaluates to more than 7, the value taken is the remainder after division by 8, for example **ink** 9 is equivalent to **ink** 1, both setting the print colour to black. If ink is used within a **print** command it will only change the print colour for the duration of that command.

INPUT

Requests input from the keyboard to the variables listed in the command. Each variable in an input list may be preceded by a initial string which will be displayed as a prompt for the input. All input items must be separated from each other by semicolons. If the list has a final semicolon, the cursor will not move to a new line after the input.

Syntax: **input** [*var* | *s.lit* | *ptm* *[; *var* | *s.lit* | *ptm*]*][;]

The list of input items may include the cursor-positioning items

at *line,column*
tab *column*

where: *line*:= *n.exp*,
column:= *n.exp*

The first of these positions the cursor at the specified line and column position, and **tab** moves the cursor to the specified column within the current line. If the cursor is already to the right of the specified column, tab will have no effect.

These two items may not be used outside an **input** or a **print** command.

You may also use ink and paper as input items. If used within an input command they will only affect the ink and paper colours to the end of the input, when the colours will return to their original settings.

INSERT

Adds a new record to a file.

Syntax: **insert**

Uses the current screen layout to display the current values of the variables. You can type a new value for any one or more fields of the current file whose values are shown in the screen layout. Note that it is not necessary for all the field variables to be shown. You cannot type a value for a field that is not shown. If none of the field variables appear in the screen, Archive forces a **display** of the file.

First select a field by pressing **TABULATE** or **ENTER** until the cursor is at the correct field (values that are not fields of the file are skipped). You can then type a new value. Press **TABULATE**or **ENTER** to move to the next field. (Pressing **SHIFT** and **TABULATE** together moves back to the previous field.)

When you have typed all the values you want you should press **F5** to add the new record to the file. The record will also be added to the file if you press **ENTER** when the cursor is in the last field. Any field that you have not given a value will be zero (if it is a numeric field) or an empty string (if it is a text field). If the file is ordered, the new record is inserted in sequence, otherwise the insertion takes place at an unspecified position.

Erases the specified file from the Microdrive cartridge. **KILL**

Syntax: **kill** *fnm*

Warning: Use this command with care since you cannot recover the erased file.

Finds the last record of the specified file, or the current file if you do not specify a logical file name. **LAST**

Syntax: **last** [*lfn*]

Used to assign a value to a variable (as in SuperBASIC). **LET**

Syntax: **let** *var* = *exp*

Lists all the procedures currently in memory on a printer. **LLIST**

Syntax: **llist**

Loads the specified procedure file from a Microdrive cartridge into memory. **LOAD**

Syntax: **load** [**object**] *fnm*

If you include the optional **object** Archive will expect the file to be in binary rather than ASCII form, see **save**. **LOCAL**

Within a procedure, forces the following list of variables to be local variables. These variables exist only within the procedure in which they are declared and are undefined in any other procedure. Their values are destroyed on exit from the procedure.

Syntax: **local** *var* *[, *var*]*

Finds, in an ordered file, the first record whose field contents match the expression(s). **LOCATE**

Syntax: **locate** *exp* * [,*exp*]*

The record is located much more quickly than if you used **find**, but **the file must first have been sorted**. Each expression must explicitly refer to the contents of a particular sort field. In the case of a string field the match is case-dependent.

If you have ordered the file with respect to more than one field, you can specify several expressions (one for each sort field). The expressions are separated by commas and must refer to the fields used to order the file. They must be in the same sequence as in the preceding **order** command. For example:

```
order animal$ ; a , weight ; a
locate "Elephant" , 2000
```

will find the first record in which the field animal$ contains the text "Elephant" and a weight that equals (or exceeds) 2000.

If there is not an exact match **locate** will still find a record. This record will be the first one whose field contents "exceed" – in the sense of the ordering (i.e. "d" comes after "e" if the file is sorted in descending order) – the specified values.

Opens the named file for read access only. If the logical file name is not specified, it is given the default value "main". **LOOK**

Syntax: **look** *fnm* [**logical** *lfn*]

LPRINT

Displays the values of the following list of items on a printer attached to SER1, in the same way as for **llist**.

Syntax: **lprint** [*exp* | *ptm* *[; *exp* | *ptm*]*] [;]

MERGE

Adds the procedures of the specified program file to the procedures already in the computer's memory. If the file contains a procedure with the same name as one already in memory, the new procedure replaces the old one.

Syntax: **merge** [**object**] *fnm*

If you include the optional **object** Archive will export the file to be a binary rather than ASCII format. See **Save**.

MODE

Changes the form of the display.

Syntax: **mode** *var,var*

The first variable may have a value of 0 or 1. A value of 0 joins the control, display and work areas into a single region. A value of 1 separates them back into three distinct areas.

The second variable may have a value of 4, 6 or 8 and switches the display between showing 40, 64 or 80 characters per line.

The initial setting, when you load Archive for use with a monitor, is equivalent to:

```
mode 1,8
```

NEW

Deletes all the data from the computer's memory, ready for a fresh start. Any open files are closed. (The command does not delete files stored on a Microdrive cartridge.)

Syntax: **new**

NEXT

Moves to the next record in the specified file, or in the current file if you do not specify a logical file name.

Syntax: **next** [*lfn*]

OPEN

Opens the specified file for both reading and writing. The file is given a logical file name "main" if you do not specify one.

Syntax: **open** *fnm* [**logical** *lfn*]

ORDER

Orders the records of the file according to the contents of the specified fields.

Syntax: **order** *order_spec* *[, *order_spec*]*

where: *order_spec*:= *var*; **a** | **d**

The first field specified in the list is the primary sort field. Records which have equal contents of their primary sort field are further sorted according to the contents of the next field in the list (if it is specified) and so on. For each specified field an ordering direction must be given. This must be either **a** or **d** to specify ascending or descending order respectively.

Order only takes account of the first 8 characters of a text field and you may not specify more than four fields to **order** the file.

PAPER

Sets the background colour for all following text to the colour specified by the value of the expression.

Syntax: **paper** *n.exp*

The colours are:

0 and 1 black
2 and 3 red
4 and 5 green
6 and 7 white

If the expression evaluates to more than 7, the value taken is the remainder after division by 8, i.e. **paper** 11 is equivalent to **paper** 3, both setting the colour to red.

If **paper** is used within a **print** command, it will only change the background colour for the duration of that command.

POSITION

Makes the record whose record number is given by the expression the current record.

Syntax: **position** *n.exp*

PRINT

Displays the values of the following list of items – which must be separated by semicolons – on the screen. If the list has a final semicolon, the cursor will not move to a new line after the display. See also **lprint**.

Syntax: **print** [*exp* | *ptm*]*[; *exp* | *ptm*]*] [;]

QUIT

Closes all files and returns to SuperBASIC.

Syntax: **quit**

REM

When used within a procedure, it marks the rest of the line as containing a comment. Any following text on that line is ignored when the procedure is executed.

Syntax: **rem**

RESET

This command restores all the records in the current file which were removed by an earlier use of **select**. It destroys any ordering of the file.

Syntax: **reset**

RETURN

Used within a procedure to cause an immediate termination of the procedure by returning to the calling procedure.

Syntax: **return**

RUN

Loads the specified procedure file into memory and starts execution of the procedure called **start**.

Syntax: **run** [**object**] *fnm*

SAVE

If you include the optional **object** Archive will expect the file to be in binary rather than ASCII form, see **save**.

Saves all procedures currently in memory as a single named file on a Microdrive cartridge.

Syntax: **save** [**object**] *fnm*

If you include the optional **object**, Archive will save the file in binary, rather than ASCII, format. This means that Archive does not have to convert the program into ASCII characters before saving it and is therefore much faster. You can **load, run** or **merge** such a program by adding the optional **object** to the appropriate command. These operations will also work more rapidly since no conversion is necessary. Such files have an extension of _pro, rather than the normal prg.

You may also save such an **object** program in a form that is protected against examination or modification. Include, instead of **object**, the optional **protect**. A program saved in this way can only be loaded, run or merged – using the optional **object** with the appropriate command.

A protected program cannot be listed, edited or saved. If you merge a protected program with any other program then the combination will be similarly protected. The only way to clear the protected status is with the **new** command.

Saving a protected version does not affect the copy of the program in the computer's memory. You can still list, edit or save the program in the normal way.

SCREEN

Displays the formatted screen layout previously sloaded. It does nothing if there is no screen layout present. It does not display any of the variables in the screen.

Syntax: **screen**

SEARCH Searches the current file from the beginning until a record is found in which the specified expression is true. This record becomes the current record.

Syntax: **search** *n.exp*

SEDIT Calls the screen editor, to enable you to define a new screen layout. See Chapter 7.

SELECT Scans the whole file selecting only those records for which the specified expression is true. The file then behaves as if only the selected records are present.

Syntax: **select** *n.exp*

You can restore all the discarded records with the **reset** command.

SINPUT Waits for input to the variables in the following list, using the order specified in the list. All the variables in the list must be currently displayed in an active screen layout.

Syntax: **sinput** *var* *[, *var*]*

SLOAD Loads a previously defined and saved display screen layout. It also displays this screen layout and activates the display of any variables within the screen.

Syntax: **sload** *fnm*

The displayed values are then updated automatically whenever control returns from a procedure to the keyboard interpreter.

SPOOLOFF Direct all following **lprint** and **llist** output to the printer. This cancels the effect of **spoolon**.

Syntax: **spooloff**

SPOOLON Directs all following **lprint, llist** and **dump** output to the specified file – or to the screen – instead of to the printer.

Syntax: **spoolon** <*fnm*> [**export** | *dump*]
or:
spoolon screen

If you are directing output to a file, it is directed via the currently installed printer driver so that it contains all the special codes that your printer needs.

If you include the optional **export**, Archive ensures that the file contains only printable ASCII codes, carriage returns and line feeds. The resulting file is suitable for importing into Quill.

The optional **dump** allows the text to be transmitted to the file without being processed by the printer driver. In this case all ASCII codes (including control codes) are passed straight into the file.

Unless you specify a file name extension, Archive assumes an extension of **__lis** (**__exp** or **__dmp** if you include the optional **export** or **dump**).

The alternative form of the command – **spoolon screen** – directs the output to the monitor screen instead of the printer.

SPRINT Used within a procedure to force a display of the fields of the current record.

Syntax: **sprint**

There must be an active screen layout (the screen layout is made active by a previous use of **screen**, **sload** or **display**). If there is no active screen layout, the command will have no effect.

SSAVE Saves, as a named file on a Microdrive cartridge, the current display area as a defined screen layout.

Syntax: **ssave** *fnm*

It saves the text of the screen and a list of the variables in the display, together with their positions.

STOP Terminates the execution of all procedures and returns control to the keyboard.

Syntax: **stop**

TRACE

Switches the trace mode on and off.

Syntax: **trace**

Type:

```
trace
```

to turn on the trace. In trace mode each line of the program is displayed in the work area of the screen, as it is executed. Press the space bar and keep it held down to pause. The trace will continue when you release the space bar. To turn the trace off again, type:

```
trace
```

UPDATE

Replaces the current record in the specified file (or the current file if no logical file name is given) with a record containing the current values of the field variables.

Syntax: **update** [*lfn*]

USE

Makes the specified file the current file.

Syntax: **use** *lfn*

WHILE

Repeatedly executes the statements between **while** and **endwhile** for as long as the value of the expression is non-zero (true).

Syntax: **while** *n.exp* : ... : **endwhile**

FUNCTIONS

Think of a function as a kind of recipe which converts one or more initial values, known as the function's *arguments*, into a different value, which is said to be the value that is *returned* by the function.

The functions provided by Archive may take three, two, one or no arguments. The arguments for a function are placed in brackets after its name. You must not leave a space between the name and the opening bracket, but spaces are allowed between items within the brackets. If a function takes more than one argument, the arguments are separated by commas. All functions must be followed by the brackets, even if they take no arguments. The presence of the brackets is a useful reminder that you are referring to a function. They allow you to distinguish between a variable and a function, even if they have the same name.

The following functions are provided.

ABS(*n.exp*) Returns the absolute value of the argument, i.e. ignores any minus sign.

ATN(*n.exp*) Returns the angle, in radians, whose tangent is *n.exp*.

CHR(*n.exp*) This function returns the ASCII character whose code is n.exp. A character with an ASCII code less than 32 is only sent to the printer if preceded by an ASCII null. For example:

```
lprint chr(0)+chr(13)
```

passes the ASCII character for a carriage return to a printer. This is useful if your printer needs control code sequences to produce special effects – refer to your printer manual for any special codes that it needs.

You can, for example, send an "A" to the screen with:

```
print chr(65).
```

CODE(*s.exp*) This returns the ASCII value of the first character found in the specified text.

COS(*n.exp*) Returns the cosine of the given (radian) angle.

COUNT([*lfn*]) Returns the count of the number of records in the current file.

DATE(*n.exp*) Returns today's date as a text string in one of three forms:

n.exp	date string
0	"YYYY/MM/DD"
1	"DD/MM/YYYY"
2	"MM/DD/YYYY"

You must first have set the system clock, as described in the *SuperBASIC Keyword Guide*.

DAYS(*s.exp*) Returns a number of days, from the first of January 1583, to a date given as a text expression of the form "YYYY/MM/DD". The conversion assumes the Gregorian (modern) calendar is being used. The formula is therefore only valid for dates after 1582.

DEC(*value,dp,width*)
value:=(n.exp)
dp:= (n.exp)
width:= (n.exp)

Converts the given numeric *value* to the equivalent text string, in decimal format with *dp* decimal places. The text is justified right in a field of *width* characters. For example:

dec(1.23e1,3,10) returns the text " 12.300" (with 4 leading spaces).

DEG(*n.exp*) Takes an angle, measured in radians, and converts it to the same angle in degrees.

EOF([*lfn*]) Returns a value indicating whether you have attempted to read past the end of the current file, or the specified file if a file identifier is given. The value returned is 1 if you have attempted to read past the end of the file, otherwise it is zero.

ERRNUM() Returns the number of the last error which occurred (an error number of zero indicates no errors). The error number is the same as that displayed together with the error message when Archive reports a detected error.

EXP(*n.exp*) Returns the value of e (approximately 2.718) raised to the power of (*n.exp*). The returned value will be in error if *n.exp* is greater than +88 since the result will then exceed the numeric range of Archive.

FIELDN(n.exp[, *lfn*])
Returns the name of the specified field in the current record of the specified file (or the current file if no logical file name is given). Note that **fieldn(0)** returns the name of the first field.

FIELDT(*n.exp* [, *lfn*])
Returns the type of the specified field in the current record of the specified file (or the current file if no logical file name is given). Note that **fieldt(0)** returns the type of the first field.

It returns the value 0 if the field is numeric, otherwise it returns 1.

FIELDV(*n.exp*[, *lfn*])
Returns the value of the specified field in the current record of the specified file (or the current file if no logical file name is given). Note that **fieldv(0)** returns the value of the first field.

FOUND() Returns one if a record is found by use of **search** or **find**, otherwise returns zero.

GEN(*value,width*)
value:=n.exp
width:=n.exp

Converts the given numeric *value* to the equivalent text string, in general format. The text is justified right in a field of *width* characters. For example:

gen(1.23e1,10)

returns the text " 12.3" (with 6 leading spaces).

GETKEY() Waits for a key to be pressed and returns a single text character which corresponds to the key that was pressed.

INKEY() Returns the single text character corresponding to any key that was being pressed at the time the function is called. It does not wait for a keypress, but will return a null string ("") if no key is pressed.

INSTR(*main,sub*)

main:= *s.exp*
sub:= *s.exp*

This finds the first occurrence of *sub* within *main* and returns the position of the first character of sub in main. It will return a value of zero if no match is found. The match is case-dependent.

```
instr("January","Jan")    {returns 1}
instr("January","an")     {returns 2}
instr("January","AN")     {returns 0}
```

INT(*n.exp*) Returns the integer value of the number, by truncating at the decimal point. The truncation always operates towards zero. Thus;

```
int(3.7)     {returns 3}
int(-4.8)    {returns -4}
```

LEN(*s.exp*) Returns the number of characters in the specified text.

LN(*n.exp*) Returns the natural, or base e, logarithm of n.exp. An error results if n.exp is negative or zero, since logarithms are not defined in this range.

LOWER(*s.exp*) Converts the specified text to lower case.

MEMORY( ) Returns the number of unused bytes of memory remaining.

MONTH(*n.exp*) Returns, as text, the name of a month.

For example **month(3)** returns the text "March".

If an argument larger than 12 is used, it is replaced by the remainder after division by 12 so that, for example, **month(13)** and **month(1)** will both give the result "January".

NUM(*value, width*)

value:= *n.exp*
width:= *n.exp*

Converts the given numeric *value* to the equivalent text string, in integer format. The text is justified right in a field of *width* characters. For example:

num(1.23e1,10) returns the text " 12" (with 8 leading spaces).

NUMFLD([*lfn*]) Returns the number of fields in the records of the specified file (or the current file if you do not give a logical file name).

PI() Returns the value of the mathematical constant π.

RAD(*n.exp*) Takes an angle, measured in degrees, and converts it to the same angle in radians.

RECNUM([*lfn*]) Returns the number (counting from zero at the first record) of the current record of the specified file (or the current file if you do not give a logical file name).

REPT(*s.exp,n.exp*)

This function returns a string consisting of a number of copies of the first character of the given text. The resulting text may be up to 255 characters in length. For example,

```
print rept("*",5)      {will print five asterisks}
print rept("abc",3)    {prints "aaa"}
```

SGN(*n.exp*) Returns +1, −1 or 0, depending on whether the argument is positive, negative or zero.

SIN(*n.exp*) Returns the value of the sine of the specified (radian) angle.

SQR(*n.exp*) Returns the square root of the argument, which must not be negative.

STR(*n,type,dp*) *n*:= *n.exp*
type:= *n.exp*
dp:= *n.exp*

Converts a number, n, to the equivalent text string.

The second parameter, type, indicates the form of the converted string as follows;

0 decimal (floating point)
1 exponential, or scientific, notation
2 integer
3 general format

The third parameter, dp, indicates the number of figures after the decimal point in the converted string. It should always be specified, although its value is ignored for integer and general formats.

For example:

```
let a$=str(12.3456,0,2)    {gives a$ the value "12.35"}
let a$ str(12.3456,1,4)    {gives a$ the value "1.2346e1"}
```

TAN(*n.exp*) Returns the tangent of the specified (radian) angle.

TIME( ) Returns, as text, the time of day in the format "HH:MM:SS". You must first have set the system clock, as described in the SuperBASIC Keyword Guide.

UPPER(*s.exp*) Converts the specified string to upper case.

VAL(*s.exp*) Converts the text to its equivalent numeric value. It will only convert text composed of valid numeric characters and the conversion will stop at the first character that can not be interpreted as a digit. For example, **val("1.1ABC")** will return the numeric value 1.1, and **val("ABC")** will return 0.0

VALUE(*s.exp*) Returns the value of the variable whose name is given by *s.exp* – for example:

```
let a$=''len''
let length=15
print value(a$+''gth'')
```

will print the value 15.

Note that value(fieldn(y)) is exactly equivalent to fieldv(y).

ERRORS

When ARCHIVE detects an error in a command typed at the keyboard or in a procedure, it displays an error number and a short error message. Examples of errors that would be detected are:

attempting to divide by zero
if not matched with an **endif**
supplying a procedure with the wrong number of parameters.

If the error comes from keyboard input, the text of the statement remains visible in the work area. You can press **F5** to recall the text so that you can use the line editor to correct the error. You can then press **ENTER** to execute the corrected statement.

If the error comes from a program statement, ARCHIVE shows the name of the procedure and the line in which the error occurred. You can then use the program editor to correct the error.

When you use the **error** command in your programs, ARCHIVE will not report any error that it detects in a procedure marked with **error**. You are free to deal with any such error in any way that you want (including ignoring it). You can find which error has occurred by examining the value returned by **errnum( )**. This number is the same as the one ARCHIVE gives when it prints an error message.

The following list shows ARCHIVE's error numbers, together with the corresponding messages. Where possible, the list includes a short example of a statement that would give the error. The error messages are not designed to pinpoint the precise error, but are intended to give you an idea of what type of error to look for.

Those error messages for which there is no short example are marked with an asterisk. They are dealt with in the notes which follow the list.

No.	Message	Example
0	no error	
1	command not recognized	apend
2	end of statement expected	let x=3 let y=4
3	variable name expected	let 31=x
4	unrecognized print item	print create
5	wrong data type	* (1)
6	numeric expression expected	let x="fred"
7	string expression expected	let x$=4
8	variable not found	let x=qq (qq undefined)
9	variable undefined	print qq
10	missing separator	print at 5
11	name too long	let thisverylongname=4
12	duplicate name	create:n$:n$:endcreate
13	string literal expected	* (2)
14	missing endproc	* (3)
15	bad proc statement	* (3)
16	premature end of statement	create"test":endcreate
17	program structure fault	* (4)
18	too many numbers	* (5)
50	missing closing quote	let x$="fred
51	missing exponent after "E"	let x=1.2E
52	number too big	let x=1.2E100
53	unknown symbol	let x=%
70	evaluator syntax error	let x=3+
71	mismatched parenthesis	let x=(3+5)/7)
73	type mismatch	let x$="fred"+3
74	wrong number of arguments	let x$=str(1,2)
75	string too long	let x$=rept("*",256)
76	divide by zero	let a=0: let x=5/a
77	bad function arguments	let x$=sqr(-4)
78	string subscript error	let x$="fred" (to 97)
80	out of memory	* (6)
90	no room to open a file	* (7)
91	incomplete file transfer	* (8)
93	out of range	print at 100,100;37
94	file not open	append (without first opening a file)
100	cannot open file	look"xxx" (non-existent)
101	write to read only file	look "names":insert
103	wrong file type	sload"names" (data file)
104	bad file name	save"3test"
105	error reading file	* (9)

Notes

1) The most likely cause of error 5 – "wrong data type" – is that you have inputted text when a number is expected, e.g. in response to an **input** statement such as:

```
input x
```

2) Error 13 – "string literal expected" – can occur, for example, during the import of a file that you have constructed yourself (without using any of the **export** commands in the QL programs). It means that Archive has found a number, or a numeric or text expression, where it was expecting to find a literal text value. In most situations where Archive finds numeric data when expecting text, or vice versa, it will give error 7 or error 8.

3) Errors 14 – "missing endproc" – and 15 – "bad proc statement" – should never occur in normal use. They indicate that Archive has detected a missing **endproc** or an error in the structure of a **proc** statement in a procedure. They are only likely to occur if you construct a program file with an editor other than the one included in Archive.

4) Error 17 – "program structure fault" – usually indicates that an **all**, **if** or **while** is not paired with a corresponding **endall**, **endif** or **endwhile** in a procedure. You

can also generate this error by including an **endproc** inside another program structure, or by using **return** directly from the keyboard.

5) Error 18 – "too many numbers" – indicates that you are trying to input more numbers than will fit into the memory reserved for input. The error may occur either in a line of input from the keyboard, or while loading a program that includes a procedure with many numbers in one of its lines. The exact limit depends on circumstances – a typical limit would be 15 to 20 numbers, so you are unlikely to get this error.

6) Error 80 – "out of memory" – should only be given if you use a very large program The size of an ordinary data file is not limited by the amount of memory in the computer since only part of a large file is in memory at any one time. If Archive gives you this error you will have to reduce the size of your program before continuing. You can, if necessary, break your program into several sections, in different files, and use **merge** to load each section as it is needed. This technique will, however, normally need a considerable amount of programming skill.

7) Error 90 – "no room to open a file" – occurs when the area of memory Archive reserves to store internal information about the files currently in memory becomes full. This may happen even if there is still memory available (i.e. if the value returned by memory() is still not close to zero.)

8) Error 91 – "incomplete file transfer" – means that the loading or saving of a file has failed for some reason. This may mean that the data has been corrupted, or that the cartridge or the Microdrive has been damaged.

9) Error 105 – "error reading file" – means that some of the data in a file is in the wrong format, the wrong order, or has been corrupted. This is only likely to occur if you construct your own import file – or your own program file without using the Archive program editor (advanced uses).

sinclair

QL

QL Easel

©1984 PSION LIMITED
by Dick de Grandis-Harrison *(Psion Limited)*

CHAPTER 1
ABOUT QL EASEL

QL Easel is *fully interactive*, which means that you see the results of everything you do immediately. From the moment you start you can just type in a series of numbers and see them displayed as a graph, as you type them in. You never need to worry about building up tables of values; Easel takes care of that kind of thing for you, and keeps them where they should be – out of sight.

You can add *text* to the graph just as simply as you enter data and, once it is there, you can edit it or move it around (easily, of course!) until you are satisfied with the result.

Easel is organised in a series of levels and exhibits a *pyramidal* structure. The top level, which is immediately available when you start, allows you to do the most commonly needed operations, for example, entering data or text. The full power of Easel becomes apparent as you become more familiar with it and dig more deeply into the pyramid.

Despite this power, Easel still remains simple to use at all levels. You do not need to remember lots of numbers and commands, since you are guided through each process by a carefully designed sequence of *prompts* which explain what you can do at each stage. In particular, Easel has a *design by example* facility which allows you to select or design anything from a single line or bar to a whole graph, simply by choosing from a set of pictures. With this facility you need never be in any doubt as to what the final appearance of your graph will be.

If, at any time, you are not sure what to do, remember that you can ask for Help by pressing **F1**. Also remember that you can cancel any partially completed operation by pressing **ESC**.

GETTING STARTED

LOADING QL EASEL

Load QL Easel as described in the Introduction to the QL Programs. When loaded Easel will display the following message:

```
LOADING QL EASEL
version x.xx
Copyright © 1984 PSION SYSTEMS
business graphics
```

where x.xx is the version number, e.g. 1.04.

Easel will, from time to time, read more information from the Easel cartridge. **You must not take the cartridge out of Microdrive 1 until you have finished using Easel and returned to SuperBASIC.**

APPEARANCE

When you have loaded Easel the display should look like that shown in Figure 2.1. The display is divided into three main areas, known as the status area, the display area and the control area.

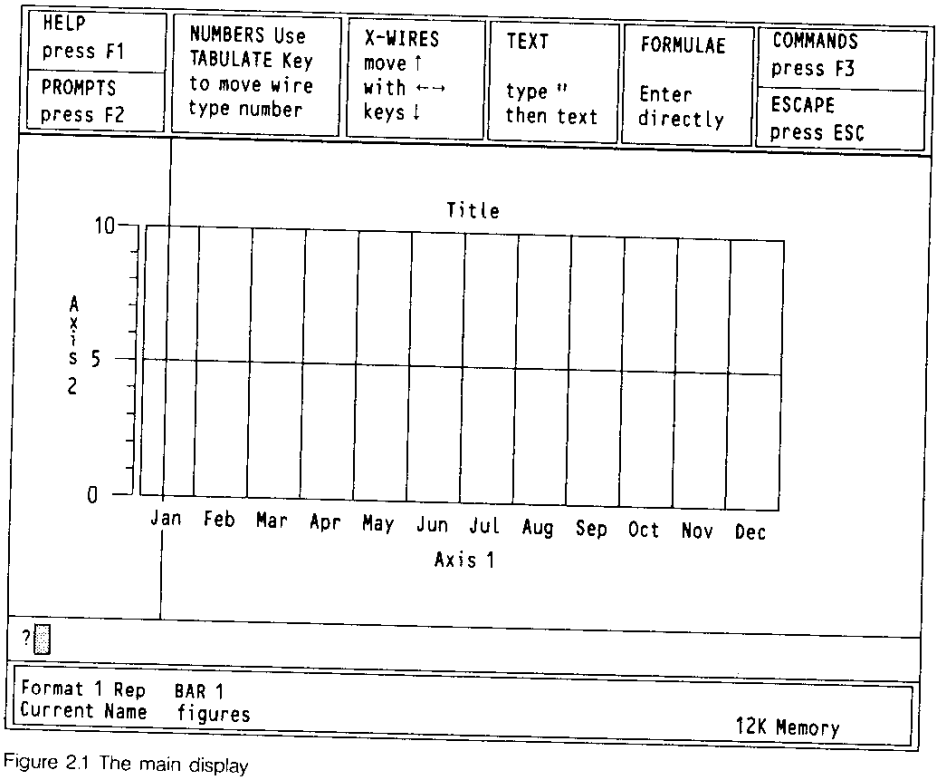


Figure 2.1 The main display

The Status Area

The *format* tells you how the values you type in will be shown. There are eight different display formats (numbered 0 to 7) to choose from, pre-defined to give an assortment of bar, line and pie chart displays. Initially, the format is set to give you a bar graph display (format 0).

You are also told the name of the set of data (or figures) for your graph. If you have more than one graph there will be a named set of figures for each graph. The current set of figures is the set that is changed when you type in numbers.

In addition you are told the *style* which will be used. Easel can show a set of figures in one of three different representations as—a bar graph, a line graph or a pie chart. Easel initially selects a bar graph representation and uses bar number 0 (there are 16 different bar designs ready for you to use).

The amount of memory available at any time is displayed in the status area together with *error messages* when necessary.

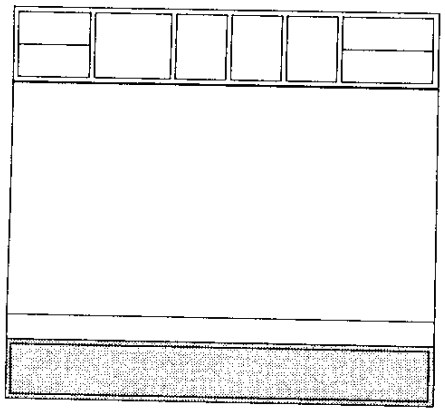


Figure 2.2 The status area

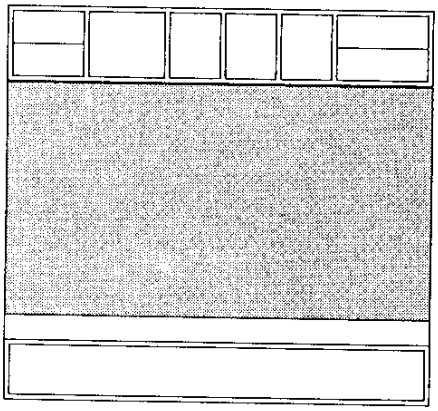


Figure 2.3 The display area

The Display Area

All graphs produced by Easel are shown in the display area.

Initially, there is an empty bar graph in the display area marked with a grid of horizontal and vertical lines. The horizontal lines correspond to the values shown on the vertical axis ('Axis 2') and the vertical lines divide the graph into *cells*. Each cell marks the position where one value of a set of figures will be plotted.

Each cell has a *cell label*, along the horizontal axis ('Axis 1'). Easel automatically supplies the text "Jan", "Feb" and so on, up to "Dec" for cell labels but you can change the text to anything you want.

Think of each set of figures as a row of cells, each containing one of the values to be plotted.

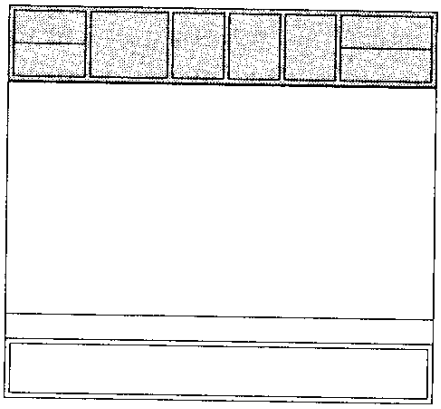


Figure 2.4 The control area

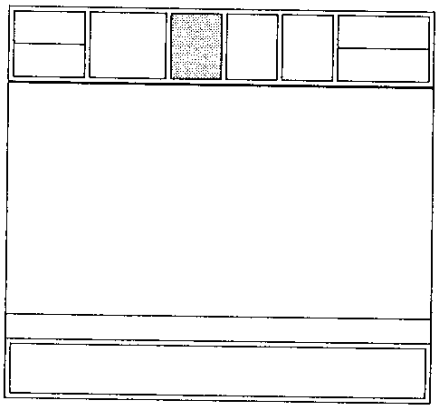


Figure 2.5 The crosswires

The Control Area

The control area shows the normal options: Help (**F1**), to turn the prompts on and off (**F2**), to select a command (**F3**) and to cancel an incomplete selection (**ESC**). In addition, there are four options that are specific to Easel. These are:

move the crosswires,
type in a number,
type in text,
type in a formula,

THE CROSSWIRES

Press the right cursor key and hold it down briefly. You will see the *vertical crosswire* moving across the screen, from left to right. The left and right arrow keys move it across the screen.

The up and down cursor keys move the *horizontal crosswire*.

You can indicate any point in the display area by moving the intersection of the crosswires to that point.

In addition, the vertical crosswire marks the position in the graph where a number that you type in will be plotted.

If the crosswires are not visible press any cursor key; press either the left or the right cursor key to display the vertical crosswire, and either the up or down cursor key to show the horizontal crosswire. Note that you can only do this from the main display, and not from the command menu.

If you press a cursor key and release it immediately the crosswire will move a short distance in the appropriate direction, but if you hold the key down the crosswire will move more rapidly across the display area.

NUMBERS

Type in a number (and then press **ENTER**). It will be displayed immediately on the graph, at the current position of the vertical crosswire. The crosswire will move one cell to the right, ready for the next number.

Each time you type in a number that exceeds the range of values shown along the vertical axis, Easel will redraw the graph with a scale that allows the new value to be shown.

If you press **TABULATE**, you will find that each press of the key makes the vertical crosswire move to the right by one cell. Hold down **SHIFT** and press **TABULATE**, and the vertical crosswire moves left by one cell. The position of the vertical crosswire marks the *current cell* – the cell that will show the next number you type in.

If you put an incorrect value into your graph you can correct it by moving the vertical crosswire to the cell where the mistake appears and typing in the correct value.

If you spot a mistake before you press **ENTER** you can correct it by using the line editor. Alternatively you can cancel the number by pressing **ESC** and then typing in the correct value.

Whether you move the crosswire with **TABULATE** or with the cursor keys, the next value you type in will always be shown in the cell containing the vertical crosswire.

TEXT

You can add text to your graph by typing a double or single quotation mark (' or ") as the first character of your input.

The crosswires will appear (if they were not already visible) and any following text that you type in will be written in the display area starting at the intersection of the crosswires and in the input line. Press **ENTER** when you have finished.

If the text is not in the exact position you want, move it using the cursor keys. The crosswires will move across the screen, carrying the text with them. When the text is in the position you want, press **ENTER** and the crosswires will disappear.

FORMULAE

A formula can be used to create a new set of figures, or to change an existing set.

Easel interprets any keyboard input that does not start with a numeric digit or quotation marks as a formula. For example, we can change the current set of figures (which, as you can see from the status area, has the name "figures").

```
figures = figures + 2 ENTER
```

The new graph is similar to the old one, except that each value has been increased by 2. If you want to return to the original graph you can type in another formula:

```
figures = figures - 2 ENTER
```

A formula always starts with the name of a set of figures. This name could be the name of an existing set or it could be a new name. In either case the contents of that data set are defined by the expression to the right of the equals sign in the formula. It is important to realise that the formula will affect all the values in the set, rather than just one value.

THE COMMANDS

The commands allow you to use some of the more sophisticated aspects of Easel. Press **F3** to select a command. The contents of the control area will change to show a list of the available commands – the command menu.

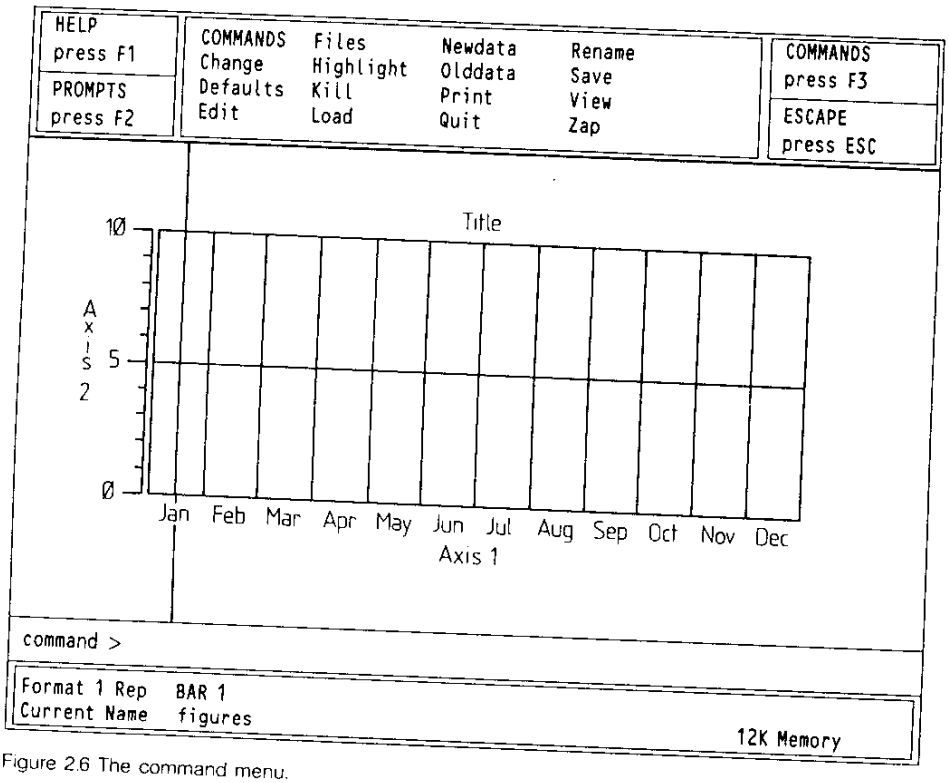


Figure 2.6 The command menu.

When the command menu is displayed you can select a command by typing its first letter.

For example, the **Quit** command leaves Easel and returns to SuperBASIC. Select it by pressing **F3** and then Q. Easel gives you the option to press **ESC** to stay in Easel (in case you selected the command by mistake). If you decide you really do want to leave Easel, you press **ENTER**.

You cannot type in a number to a cell or type in a formula when the command menu is visible. Also, you cannot move the crosswires, except when given this option as part of a command.

At the end of a command, Easel remains in the command menu and you must press **ESC** to go back to the main display.

DELETING A VALUE

You can delete a value from the graph. Use the **TABULATE** key (or the **SHIFT** and **TABULATE** keys) to position the vertical crosswire on the number you want to rub out and then press **F4**. If your graph is showing more than one set of figures, pressing **F4** deletes all values shown in that cell. It has no effect on sets of figures that are not shown. If you delete the values from a cell that has no label, then that cell will not be included in the graph when it is next redrawn.

Easel will only delete a cell that has no label and no value. If you want to delete a cell you should delete its contents and also delete any label that it has. The cell will not be included next time the graph is redrawn with the **View** command.

INSERTING A VALUE

You can insert a new value to the right of the one marked by the vertical crosswire. Press **F5** and a gap is opened up, ready for you to type in a new number. The new cell will not have a label, but you can add one.

Inserting and deleting values from pie charts is slightly different and is explained in Chapter 9.

CHAPTER 3
DESIGNING A BAR

This chapter shows how you can modify the appearance of your graph by using a different design of bar.

All the options to modify the various features of the graph work in the same way. Learning how to change your graph to use a new bar design explains the methods you will use to change any other aspect of the graph.

We assume that you have typed in a few numbers and have a bar graph shown on the screen.

SELECTING A BAR

You use the **Change** command to select a different bar, **F3** and then the C key. You are offered many options – to change an Axis, Text, and so on. Select the Bar option by pressing the B key.

There are two routes to using a new style of bar—selection by number or by example.

Selection by Number

When you have selected the Bar option, the input line shows the text:

```
COMMAND>Change to BAR ?
```

and Easel waits for you to type in a number. There are 16 different bars, numbered 0 to 15 and you can select any one of them by typing its number, followed by **ENTER**.

This is a very quick method of changing the bar you use, provided you know the number of the one you want.

Selection by Example

If you do not know the number of the bar, or you want to use your own design, press **ENTER**, instead of typing a number. Try this method by typing in

F3 C B ENTER

(You do not have to press **F3** if you are still in the command menu.) The display changes to show examples of all the available bar styles, together with their associated numbers.

The selected bar is surrounded by a box. You use the left and right cursor keys to move this box from bar to bar until it is positioned on the one you want. When you press **ENTER** the bar you have chosen will be used.

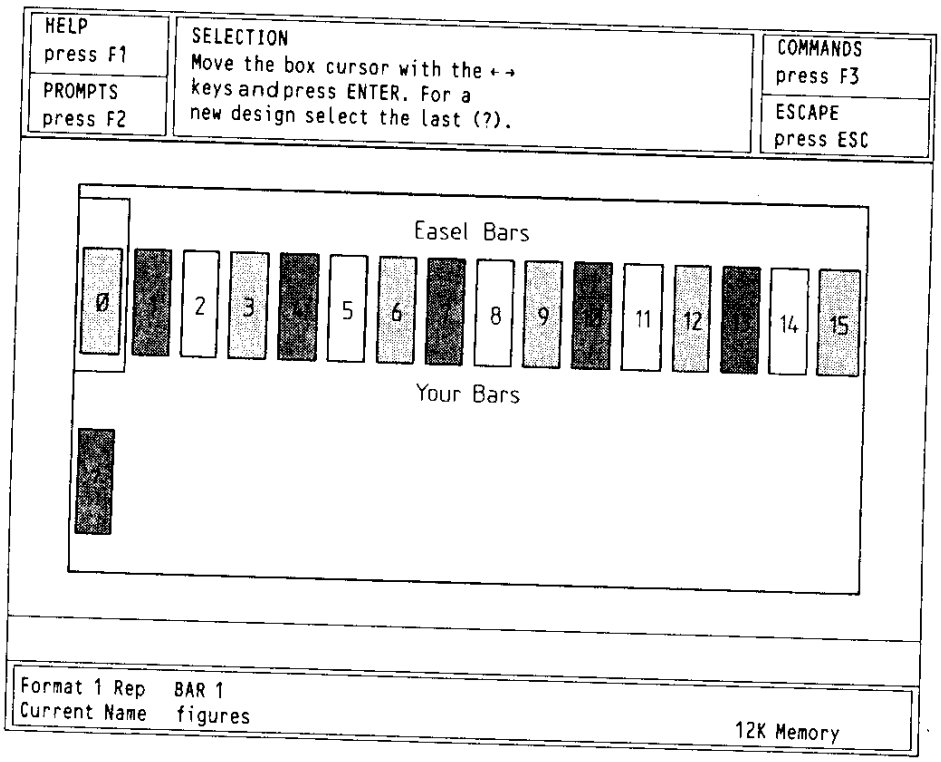


Figure 3.1 Selecting a bar.

When you use the option to select by example you will notice that there is one bar present, in the second row, which shows a question mark in place of its number. You select this bar if you want to make your own design.

Position the selection box on this bar and then press **ENTER**. The design by example continues by presenting you with a blank bar design and a list of options.

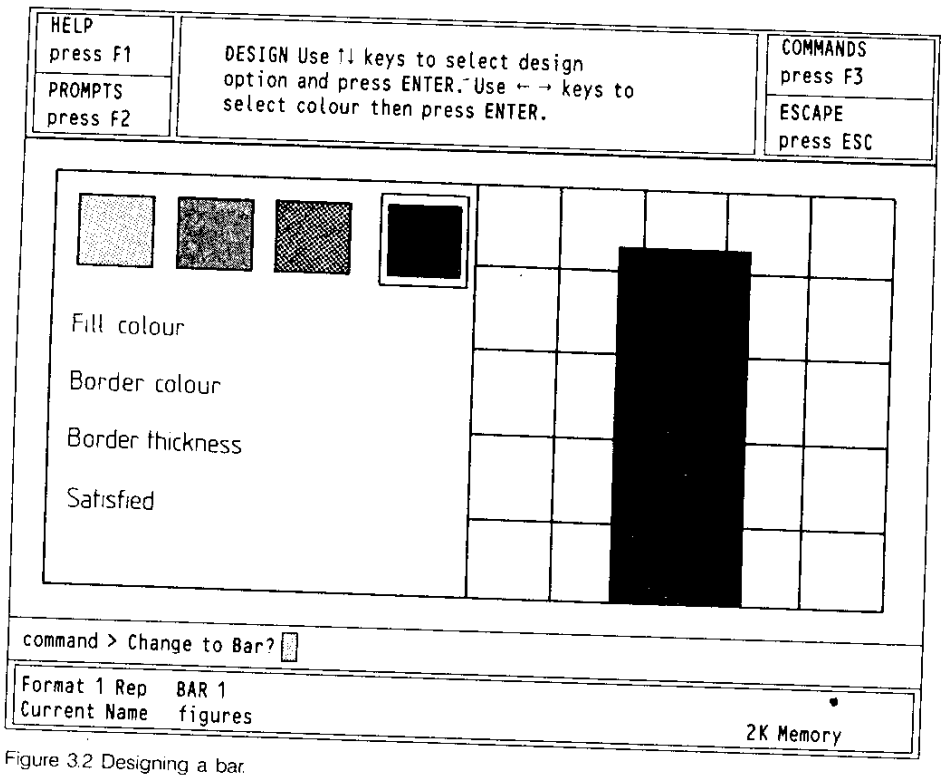


Figure 3.2 Designing a bar.

Bar Colour

The first option highlighted is bar colour and allows you to choose the bar colour from the palette shown across the top of the display. You can accept the option by pressing **ENTER** or select another option by using the up and down cursor keys.

If you accept the bar fill option a box is drawn around the first colour in the palette and the specimen bar is filled with that colour. You can move from colour to colour by pressing the left or right cursor keys. Make your selection by pressing **ENTER** when the bar is filled with the colour you want. Easel draws the bar against a background of the current graph paper.

Border Colour

The next option in the list, to select a border colour for the bar is then highlighted automatically. Again you can either select this option by pressing **ENTER**, or move on to one of the other options. If you select this option you can choose the border colour for the bar in the same way as you chose the main fill colour. (If the width of the border is currently set to zero you will not, of course, see the colour in your bar design.) You should press **ENTER** when you are satisfied with the result.

Border Width

The third option is to select the width of the border. In this case you are asked to type in a number to represent the width of the border as a percentage of half the width of the bar.

You are finally given the option of deciding whether you are satisfied with the design as shown. If you are you should press **ENTER**, when the new design will be added to the list of bar designs and it will automatically be used for display of the current set of figures. If you are not satisfied with the design you can go back to one of the other options, using the up and down cursor keys, and try a new combination. At any time before you accept the design you can terminate the command by pressing **ESC** and you will leave the command without creating a new bar design.

CHAPTER 4
USING TEXT

Each time you edit some text, or add new text, it is shown in the colour and direction—horizontal or vertical—that you last set with the Text option of the **Change** command.

Easel recognises three basic types of text:

Ordinary Text (including the Title).
Axis Names
Cell Labels

ORDINARY TEXT

Ordinary text – i.e. all text except for the axis names and cell labels – behaves as though it were pasted on the screen. It is always printed over the top of the graph or chart and remains on the screen until you delete it, regardless of any other changes you make.

The **Edit** command has options to edit the 3 types of text listed above, and a fourth option relating to the Key. The Key option is only relevant when you have more than one set of figures in your graph. It is described in the next chapter.

Press the T key to select the Text option. You then use the cursor keys to move the intersection of the crosswires close to the text which you want to change. It is not necessary to position the crosswires exactly; press any key and the crosswires will attach themselves to the nearest piece of text. A copy of the text also appears in the input line.

You can delete the text by pressing **F4**, or modify the text using the line editor. If you choose to delete the text this will also end the command.

When you are completely satisfied with the wording of the text you should press **ENTER**. Easel then gives you the opportunity to reposition the text. Press **ENTER** when you are satisfied with the position.

Easel treats a graph title in the same way as any other text. The only difference is that Easel supplies the text "Title", centred above the graph, when you load it from its Microdrive cartridge.

AXIS NAMES

Axis names only appear on bar and line graphs and are not shown when you select a pie chart representation.

Select the Axis option of the **Edit** command to edit either of the two axis names. Press V or H to select the vertical or horizontal axis. You can then edit, delete or move the text, as described for the Text option. Easel redraws the text in the current ink and paper colours.

CELL LABELS

The cells of the graph are provided with labels which are initially set to show the months from January to December. These labels are shown along the horizontal axis of a bar or line graph. In a pie chart they are used to label the segments of the chart.

You use the Labels option of the **Edit** command to change the cell labels. When you do so the crosswires will attach themselves to the nearest label which will then be displayed in full. Cell labels can be up to ten characters long but normally only the first few characters are shown. The text is also copied into the input line. You can then delete the label by pressing **F4**, or edit it with the line editor. Press **ENTER** to finish editing. Although the labels initially have their own text colour, when you edit any label, all the labels are shown in the current text colour. You cannot move a cell label.

TEXT COLOUR AND DIRECTION

You use the Text option of the **Change** command to alter the colour of the text and its background. You can also select whether the text is vertical or horizontal.

Easel uses the new text colour and direction for all new text that you add to the graph – and for any old text that you edit.

A convenient way of changing the colour of text is first to change the text colour and then to use the **Edit** command – described in the following section – on the existing text, without actually changing its wording or position.

Select the Text option of the **Change** command. Easel offers you a list of text design options in a similar way to designing a bar by example. You can step through the options with the up and down cursor keys, and select the highlighted option by pressing **ENTER**.

Ink Colour

The first option is to select the ink colour. You use the left and right cursor keys to select the colour and press **ENTER** when the text is shown in the colour you want. The following option is highlighted automatically, ready for selection by pressing **ENTER**.

Paper Colour

This second option is to select a background paper colour. You select the colour with the left and right cursor keys, press **ENTER** to confirm your selection and move to the next option.

The third option is to select a transparent background for the text. If you select this option Easel ignores your selection of paper colour and allows the background graph to be seen around the text. Each time you select this option the background switches between the chosen background colour and a transparent background.

Text Direction

The fourth option is to select the direction in which the text is printed. Each time you select this option Easel switches the text between horizontal and vertical.

Finally you are given the option of deciding if you are satisfied with the appearance of the text. At this stage you can press **ENTER** to keep your selection of text style and return to the command menu. Alternatively, you can use the up or down cursor keys to go back and make further changes.

CHAPTER 5
SEVERAL SETS OF FIGURES

So far we have only described how to create and display a single set of figures. On many occasions you may want to display two or more sets of data on the same graph, for example to compare the sales figures for two successive years. This chapter describes the techniques you can use to produce, modify and display graphs containing several sets of figures.

THE CURRENT FIGURES

No matter how many sets of figures you have in your graph, you can only modify one set at a time. The set that you can add to or change is known as the *current figures*, and its name is shown in the status area. Initially you have one set of figures called "figures". If a set of figures is current it will be displayed on the screen.

THE RENAME COMMAND

Suppose that you have typed in a set of numbers to "figures" and want to change the name to "sales". You do this with the **Rename** command. Press **F3** and then the R key. Easel asks you to type the old name of the set of figures and mark the end of the name by pressing **ENTER**. You then type in the new name. To change the name of "figures" to the new name "sales", you should type:

F3 R figures ENTER sales ENTER

The set of figures that you have renamed becomes the current figures.

THE NEWDATA COMMAND

There are two methods that you can use to produce new subsequent sets of figures. These are by using the **Newdata** command, or by using a formula. The two methods are described in this and the following section.

Suppose you have created a set of figures called "sales", as described above, containing monthly sales figures, and that you now want to include a display of the monthly costs. You can do this by pressing **F3** and then the N key, to select the **Newdata** command. You then type in a name for the new set of figures, ending by pressing **ENTER**.

To create a new set of figures called 'costs' you therefore type:

F3 N costs ENTER

Easel immediately gives you a new, blank graph (assuming you are in a bar or a line format) with the vertical crosswire set on the first column. The status area shows that the current figures are the new set, with name "costs". All you have to do is type in the new numbers which are immediately displayed on the graph as normal.

If you want to create a third set of figures, you can use the **Newdata** command again, exactly as has been described, giving the new set of figures a different name. You can create as many sets as you like, the only limit is the amount of available memory.

USING A FORMULA

On occasions you may wish to produce a new set of figures related in some way to one or more existing sets.

Suppose you have already entered sets of figures for "sales" and "costs" and want to generate a graph showing the resulting profits. All you have to do is type in a formula which describes the new set of figures, for example:

```
profits = sales - costs
```

This creates a new set of figures with the name "profits", each value being the difference between the corresponding values of the "sales" and "costs" figures. "Profits" will become the current figures and the graph will be displayed immediately.

You can also use a formula without having to refer to existing sets of figures. You could, for example, write a formula such as

```
wave = 10 * sin(cell/2)
```

This formula creates and displays a new set of figures with the name "wave", whose values are calculated using the **sin ()** function. In this formula we have also used "cell". This gives the cell number, counting from 1 at the left hand side of the graph. To see how this works, type in the formula:

```
a = cell
```

and look at the graph that is drawn. When you use 'cell' in a formula, the number of values in the set of figures is made equal to the number of columns currently being shown on the graph.

There is another reserved word in Easel – "cellmax". It has a value equal to the number of cells currently shown on the screen. You can use "cellmax" to adjust the scale of the horizontal axis in a formula. For example, the formula:

```
curve = sin(2*pi()*(cell - 1)/(cellmax - 1))
```

draws one complete cycle of a sine curve, regardless of how many cells are shown on the screen.

THE OLDDATA COMMAND

When you use the **Newdata** command the set of figures that you create becomes the current set. Remember that this is the set that can be added to or changed by typing in numbers. If you want to make some changes to an existing set of figures that is not the current set, you can do so by using the **Olddata** command. When you select this command you are asked to type in the name of an existing set of figures, and that set becomes the current figures.

Suppose that you have the three sets of figures called "sales", "costs" and "profits", and that "profits" is the current set of figures. If you want to change or add to the "costs" figures you should select it with the **Olddata** command. The costs figures will then be shown on the graph and you can modify the data by typing in replacement numbers.

Note that any change you make in the "costs" figures will not automatically change the "profits" graph. (This is a job for Abacus.)

VIEWING THE DATA

You can see the effect of displaying all of your figures on a single graph with the **View** command.

Try selecting this command. As you see, Easel suggests that all the sets of figures should be shown on the graph and you can accept this suggestion by pressing **ENTER**. Easel then suggests the display format to be used and again you can accept the suggestion by pressing **ENTER**. The graph is drawn immediately, containing all the data that you have defined – together with a *key box* which shows the name of each set of figures and the way that it is represented (the key is not shown if you only have one set of figures on the graph).

If you have defined a large number of sets of figures the graph will be very crowded and make very little sense. In general it is a good idea to display only a small number of sets of figures on any one graph to make the best visual impact. This does not mean that you should only define a small number of sets of figures, since the View command allows you to select which sets of figures you want to see.

You do this by not accepting the "all figures" suggestion that Easel gives in the **View** command. Instead of just pressing **ENTER** at this point, you can type in a list of the names of those sets of figures which you want to be displayed, separating the items in the list by commas. When you have typed in all the names of the sets of figures that you want to be displayed, press **ENTER**.

You can also select a different format for the display instead of accepting the suggestion made by Easel. Instead of just pressing **ENTER** to accept the suggested format you can type in a number between 0 and 7. Easel is provided with eight pre-defined formats, providing various styles of bar charts, lines or pie diagrams. You can type in a question mark to see a menu of all possible formats. Try using the **View** command to display three or four sets of figures in a number of the different formats available.

THE KEY

One of the options in the **Edit** command is to move the key. The key is replaced by its outline, and you are then offered the option of either deleting the key – by pressing **F4** – or moving the key by means of the cursor keys. If you choose the move option the cursor keys move a box equal in size to the key around the display area. When you press **ENTER** the graph will be redrawn with the key in its new position.

You may at some time want to restore the display of a key which you had deleted earlier. You can do this by using the **Edit** command and selecting the Key option. The outline of the key will appear. You can move the key to a new location. Pressing **ENTER** will redraw the graph, including a display of the key.

The only change that you can make to the contents of the key box is to change the colour of the text that it includes. This text is always drawn in the colour last set by using the **Change** command. The symbols shown in the key box will, of course, always match the symbols which you use to display the graphs.

FOR SERVICE MANUALS
CONTACT:
MAURITRON TECHNICAL SERVICES
www.mauritron.co.uk
TEL: 01844 - 351694
FAX: 01844 - 352554

CHAPTER 6
GRAPH FORMATS

CHANGING FORMAT

Easel is provided with 8 *display formats* (numbered 0 to 7) and you can use one of these numbers to specify which format should be used each time you use the **View** command. In addition to using different styles of background and bar colour, these formats give you a range of display styles.

You can also use the Format option of the **Change** command to select one of the 8 formats. Easel puts the text:

```
COMMAND Change to format ?
```

in the input line and you can select a particular format by typing in a number between 0 and 7 (followed by **ENTER**). If you press **ENTER**, then Easel shows you the appearance of all 8 formats and again asks you for the format number.

REDESIGNING A FORMAT

You can redesign the entire appearance of any or all of the eight different formats provided by EASEL.

You will normally have some idea of how you want your graph to appear. In this case you would select the format that is closest to the one you want and then modify it until it matches your requirements.

Use the options of the **Edit** command to change the text of the cell labels and the axis names. You use the **Change** command to modify the text and bar styles.

If you want a line graph you can use format 3, or you can use the Line option of the **Change** command. Pie charts are described in Chapter 8.

Graph Paper

The Graph paper option of the **Change** command allows you to select the paper colour and the colour of the grid markings. You can select a graph paper from an existing set of 7 styles, or you can design a new one. The method of design is exactly as described for the Bar option of **Change**.

Axis

The Axis option of the **Change** command allows you to select the axis style for your graph. You can select an axis from an existing set of 10 styles, or you can design a new one. The method of design is exactly as described for the Bar and Graph paper options of **Change**.

The Axis design option allows you to select a colour for the axis line, whether or not the axis line is drawn, the colour of the numbers labelling the vertical axis and the axis limits.

EASEL normally chooses the limits for the range of values shown on the vertical axis. It chooses a range that allows you to see all the values in your graphs. If you select the option to change the axis limits you are offered one of three possibilities.

Press the A key to select automatic limits. In this option Easel selects a suitable range, depending on the values in your graphs. The range might not include the zero point if, for example, all the values are large and positive.

Press the Z key to select automatic limits which always include zero. This is the type of axis limits that Easel uses if you do not make your own choice.

Press the M key if you want to make your own choice of limits. Easel asks you to type in the lower limit and then the upper limit (mark the end of each value by pressing **ENTER**). Note that Easel will override your selection if it does not cover the full range of values in your graphs.

In all cases the two limits are adjusted so that the intervals in the scale are sensible ones. Note that the specimen axis, shown at the right of the screen, does not necessarily show the exact range that will be used in the resulting graph. It is a representative axis and is only intended to illustrate the general type of axis that you have chosen.

CHAPTER 7
LINE GRAPHS

As you may have seen when you experimented with the different display formats, the sets of figures can also be represented by line graphs or by pie charts. This allows you to display a given set of the figures in many different ways so that you can choose the method most appropriate for your needs.

Format 3 uses lines to display the sets of figures. Each value may be marked by a symbol and the values are joined by lines of various thicknesses and colours. You can also use "filled" lines where the space between the line and the zero level is completely filled with colour.

You may find filled lines useful for showing "critical values", such as a break even level, as a background to your graph.

Since bars and lines are both displayed on the same type of grid, you can mix bars and lines in any combination. Titles, axis labels, general text and the key box all behave in exactly the same way for both bars and lines.

SELECTING A LINE STYLE

If you select the Line option of the **Change** command you can change the representation of a set of figures to use a line graph. First make the graph you want to change to be the current figures (e.g. with Olddata). Then select the Line option of the **Change** command.

There are 16 pre-defined line styles and Easel first asks you to type in the number of the line you want. Type in the number and press **ENTER**, or just press **ENTER** to see the selection available. Select a line by pressing the left or right cursor key. When the box encloses the line you want, press **ENTER**. Easel immediately draws the set of figures with the line style that you have selected.

Select the line shown with a Question mark instead of a number if you want to design your own style of line.

Easel gives you a list of options for your line and you use them exactly as was described for the Bar option, in Chapter 3. Press **ENTER** to select the highlighted option, or use the up and down cursor keys to step to the option you want.

Line Colour selects the line colour. Select the colour with the left and right cursor keys and press **ENTER** to move to the next option.

Symbols allows you to choose whether to mark each point on the line with a symbol. Each time you select this option the symbols are switched between on and off.

Symbol Colour selects the symbol colour in the same way as you select the line colour. You can step over this option if you have chosen not to use symbols.

Filled Lines switches between a normal line and a line which is filled with the line colour to the horizontal axis. Each time you select this option Easel switches between the two types of line.

Line Thickness allows you to choose the thickness of the line. Type in a number between 0 (thinnest) and 100 (thickest) and press **ENTER**. You can step over this option if you have selected a line filled to the axis.

Easel offers you a final option to check that you are satisfied with the result. Press **ENTER** to see the graph with your style of line, or use the up or down cursor keys to go back to modify your selections.

Figure 7.1 was created in format 2 (stacked bars) with one set of figures changed from a bar to a line representation.

NUMERIC ENTRY FOR LINES

If you select a format which uses a line graph for your current figures, you can enter your data in exactly the same way as described for bar charts – simply type them in.

The only real difference between line and bar representations appears when you are typing numbers into a set of figures represented by a line. In order to allow you to type new numbers – or change existing ones – without redrawing the whole graph for each number, Easel does not use the true line colour.

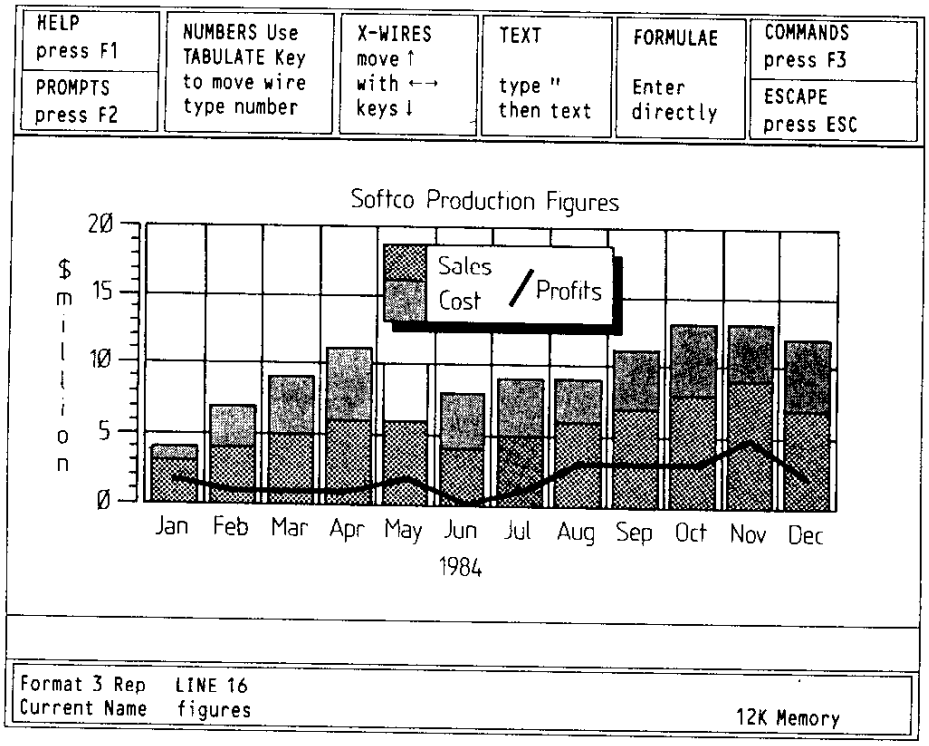


Figure 7.1 Stacked bars and lines

While you are typing in the numbers, the graph is drawn using a thin white line – or a filled line – depending on the line style you have chosen. The colour of the line changes as it passes over any bars, lines or text. Easel warns you in the status area that the line colour is not being shown correctly.

When you have finished typing in numbers, use the **View** command to see your graph with the correct colour and thickness of line.

Note that you can only show one set of figures at a time in pie chart format, and that any negative values are ignored. Easel will warn you if data has been omitted.

Since you can only have one set of figures in a pie chart, the **View** command offers you the option of viewing the current set of figures - instead of the usual "all Figures". You can type in a replacement name. If you type in a list of names (separated by commas) Easel will display the first set of figures in the list, ignoring the rest.

ENTERING NUMBERS

To illustrate entry into a pie chart, use the **Change** command to change to format 7, which is a pie chart format. Then use the **Newdata** command to create a new, empty set of figures called, for example, "costs". Easel draws a filled circle, labelled with the first cell label which, unless you have changed it, is "Jan".

Type a number and press **ENTER**. Easel redraws the circle, but this time the number you typed in is shown under the label. The diagram is a pie chart with only one value. Type in a few more numbers, exactly as if you were typing numbers into a bar chart.

During data entry into a pie chart the next cell to receive data will be indicated by having its label highlighted. If this is not possible it will be indicated by a special highlighted display box at the bottom left of the display area. In Figure 8.1 this is the label "PORTUGAL".

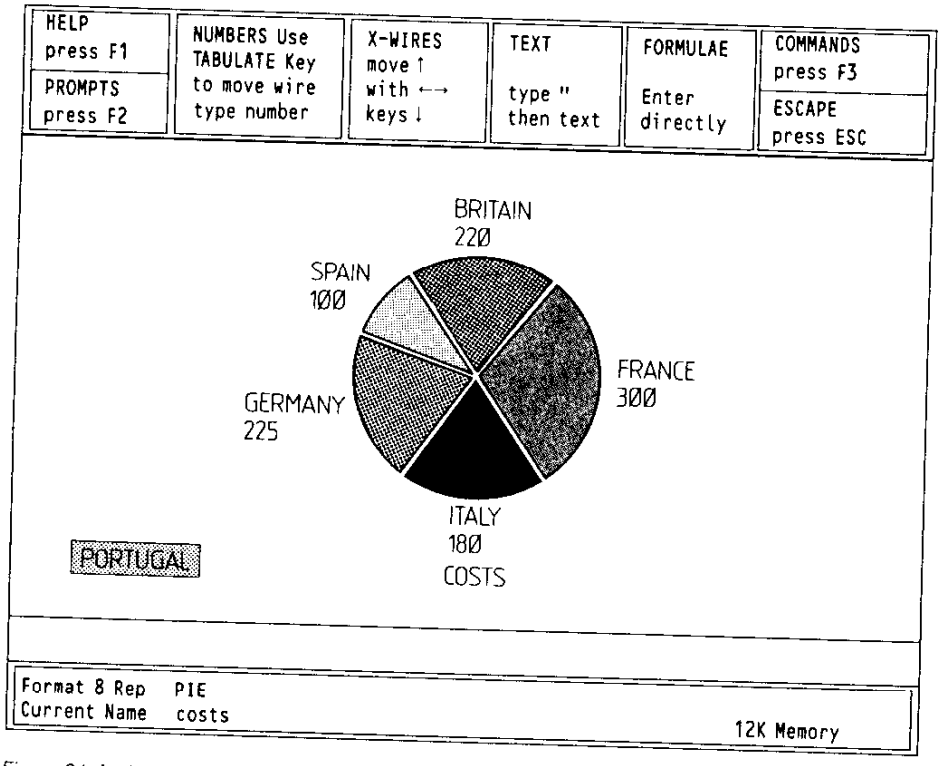


Figure 8.1 A pie chart

Since the chart must be redrawn each time you add or change a value, you may find it more convenient to enter the numbers in one of the other formats changing to pie chart format later.

CELL LABELS

You move from cell to cell with the **TABULATE** or **SHIFT** + **TABULATE** keys, just as in a bar chart. Remember that as always the option is not available from the command menu.

Press **F5** to add another cell after the one whose label is highlighted. Easel gives each new cell the label "unnamed". Type a number into the cell as normal. You can edit the

cell label with the Label option of the **Edit** command. The cell that you can edit is highlighted, and you can step from label to label with the **TABULATE** and **SHIFT** keys.

As in bar charts and line graphs you must delete both the cell label (use the label option of **Edit**, and press **F4**) and the number in the cell (step to the cell with **TABULATE** and press **F4**) before Easel deletes the whole cell. Easel deletes a blank cell when you next use the **View** command.

TEXT

You can add, delete, and move text and titles exactly as described for lines and bars in Chapters 2 and 4. You use the horizontal and vertical crosswires in the normal way for adding, editing or moving ordinary text.

THE CHANGE COMMAND

The Text and Format options of the **Change** command work in exactly the same way with a pie chart as in any other format.

Graph paper, bars, lines and graph axes have no meaning for a pie chart and Easel does not allow you to use these options.

The Segment option can only be used in the pie chart format. It allows you to change the colour of a segment of the chart. First select the segment whose colour you want to change.(Press **TABULATE** until its label is highlighted.) Then select the Segment option of the **Change** command.

Easel draws the palette of possible colours in the display area. Press the left or right cursor key to select the colour you want and then press **ENTER**. Easel redraws the pie chart with the segment in your chosen colour.

FOR SERVICE MANUALS
CONTACT
MAURITRON TECHNICAL SERVICES
www.maurtron.co.uk
TEL: 01844 - 351694
FAX: 01844 - 352554

CHAPTER 9
PERMANENT COPIES OF YOUR GRAPHS

PRINTING

If you have a dot matrix printer that is compatible with the Epson FX80 (for example the Brother HR-15 or the Canon PW1080A) you can make printed copies of your graphs immediately.

The **print** command makes a printed copy of the graph shown on the screen. Press the P key to select the **print** command. Before printing Easel reads the printer driver from the file 'gprint__prt' from the cartridge in Microdrive 1.

Press the S key to dump the screen into a Microdrive file; you must type in the name of the file to use followed by **ENTER**. This file can then be subsequently processed for example by a SuperBASIC program and sent to a printer not supported by Easel. Note that this file is very large and normally no more than three can be stored on a Microdrive cartridge.

Press the I key to install a different printer driver. Several other printer drivers are supplied on the Easel crtridge, they are in files with the extension **__prt**. Some of these are colour printers, for example the Integrex 132 and the Okimate 80. Type in the name of the printer driver you want and press **ENTER**.

The new printer driver is not installed permanently and Easel will revert to the Epson FX80 the next time Easel is loaded. The default printer driver is contained in the file 'gprint__prt' and so you can make the installation of a new printer permanent by simply renaming the files. First copy the original 'gprint__prt' to another file, for example 'FX80__prt' and delete the original file. Then copy the file containing the driver you require to the file 'print__prt'. Note that the original Easel cartridge is write protected so you must use the copy you made.

You can use a baud rate different from the initial 9600 baud assumed by the QL. For example, if you wanted to set 4800 baud start Easel by typing:

```
BAUD 4800
LRUN mdv1_boot
```

instead of having the Easel cartridge in Microdrive 1 when you press **F1** or **F2**.

Alternatively you could make the change of baud rate permanent by adding an extra line to the 'boot' program. First load and renumber the program by typing:

```
LOAD mdv1_boot
RENUM 10,10
```

Then add, for example, the line:

```
5 BAUD 4800
```

Delete the old copy of the program and save the new version on the Easel cartridge, type:

```
SAVE mdv1_boot
```

Again this change must be made to a copy of the Easel cartridge.

PHOTOGRAPHY

The simplest, and fastest, way of obtaining a permanent copy of one of your graphs is to take a photograph of the screen. You must, however, take a little care if you want to obtain good results.

One of the most common causes of a poor photograph of a television screen is using too short an exposure time. The picture is made up of 625 separate lines, displayed one after another. It takes a 25th of a second to display all the lines in the picture and if you use an exposure time of about this length, or shorter, the picture will be unevenly lit. It is best to use an exposure time of around a quarter of a second – this means that you must support the camera on a tripod. An average colour film (for prints or transparencies) with a speed of, say, 100 ASA will need an aperture of around F5.6. Use a long focal length lens (about 100mm) if you have one, as this will reduce the distortion caused by the curved surface of a TV screen.

Try to take the photograph in a darkened room, to avoid reflections of the surroundings on the surface of the screen. It is surprising how strongly such reflections show up on the photograph, even if you do not notice any when you look through the camera viewfinder.

Press **F2** to remove the control area and give you a larger graph. You can also press **SHIFT** and, while holding it down, press **F2** to erase the text in the status area.

Before taking the picture make sure that all text, cell labels, axis names and the key appear exactly as you want them.

FOR SERVICE MANUALS
CONTACT:
MAURITRON TECHNICAL SERVICES
www.mauritron.co.uk
TEL: 01844 - 351694
FAX: 01844 - 352554

CHAPTER 10
QL EASEL REFERENCE

THE FUNCTION KEYS

In addition to the standard use of F1, F2 and F3, function keys 4 and 5 are used as follows:

F4 delete
- text
- labels
- numbers
- the key
- user-defined objects

Note: user-defined objects are bars, lines, graph paper and axes.

F5 insert a cell

THE COMMANDS

The commands give access to the deeper levels of Easel and allow you to use many of the more advanced facilities. The following commands are provided.

CHANGE

The **Change** command allows you to modify the appearance of any feature of the graph.

You are offered the following options:

Axis to select the axis markings. You can alter the colour of the axes and of the numbers labelling the y-axis. You may also select whether or not the axis lines are to be drawn. Easel will not allow you to select this option in format 7 (a pie chart).

The option to change the axis limits allows you to choose between automatic or manual limits. Press the A key for automatic limits or the Z key for automatic limits which always include zero.

Alternatively, press the M key to select manual limits. In this case you must type values for both limits. Easel may modify your choices of limits to ensure that the whole of your graph is shown, with simple numeric values on the scale.

Bar to select or define the style of bar used to represent the current set of figures. You may choose one of 16 previously-defined bars by its number, or by example. The design by example option allows you to select a bar or to design a new one. Easel will not allow you to select this option in format 7 (a pie chart).

Format to redefine the appearance of the entire graph. You may choose one of 8 defined formats by its number, or by example.

Graph_paper to select one of 7 different graph papers, or to replace one with your own design. You can select both the background colour and the colour of the grid markings. Easel will not allow you to select this option in format 7 (a pie chart).

Line to select one of 16 defined line styles, or design your own line. You can choose the line colour and thickness, and the colour of the symbol used for each point on the line, or select a *filled line*, where the space between the line and the zero level on the graph is colour-filled. Easel will not allow you to select this option in format 7 (a pie chart).

Segment to select the colour of a particular pie chart segment. Easel will only allow you to select this option in format 7 (a pie chart).

Text to select the colour used for both the text and its background. You can select a transparent background so that the underlying graph will show through. You can also select whether the text is to be drawn horizontally or vertically.

Any existing text will retain its original colour and direction, but new text will appear in the selected style, until you change it again. (The text in a key box is always drawn in the current text colour.)

DEFAULTS

The **Defaults** command allows you to select a number of features, such as whether you use a 40, 64 or 80 character display. You can select an item by pressing the key corresponding to its first letter in the list of options shown.

EDIT

The **Edit** command allows you to modify or move text, labels and the key.

You are asked to choose between the following four options:

Text — the crosswires lock on to the nearest piece of text and you can use the line editor to change the wording. Press **ENTER** and you are offered the option of moving the text to a new position. Press **ENTER** when you are satisfied with the position.

Labels — the crosswires lock on to the nearest cell label and you can then edit the text of the label as in the Text option. Press **ENTER** when you have finished. Cell labels can not be moved.

Key — you are immediately offered the option of moving the key box with the cursor keys. Press **ENTER** when the outline of the key box is in the position you want. The key box is then redrawn in its new position.

Axis — you are asked to press either the V or the H key to select the vertical or the horizontal axis name. The crosswires lock on the chosen name and you can edit the text and then reposition it.

After editing, all text is shown in the colour and direction set by the last use of the Text option of the **Change** command. The only exceptions are the axis names, which are fixed in direction.

FILES

This command allows you to modify Easel files, previously saved on a Microdrive cartridge, or to transfer data files to another of the Psion QL programs.

You are offered the following options:

Backup — used to make a backup copy of an Easel file. You are asked for the name of the file to be copied and the name you want to give to the new copy. **Making copies of your files is strongly recommended, to protect yourself against accidental loss of, or damage to, the cartridge, and against making a mistake which causes your application to be corrupted or deleted.**

Delete — deletes a named file from a Microdrive cartridge.

Warning – this command is not reversible and should be used with great care.

Export — exports a named file. The file contains all the sets of figures currently in the computer's memory. It is saved in a form suitable for being read by QL Abacus or QL Archive. Import and export are described in the Appendix.

If you do not specify a file name extension for an exported file, Easel will supply an extension of **__exp.**

Import — imports a named file and allows Easel to read data files exported from QL Abacus or QL Archive and display them in graphical form.

If you do not specify a file name extension for an imported file, Easel will assume an extension of **__exp.**

Format — formats the cartridge in Microdrive 2, or another named Microdrive. Accept Easel's suggestion to format mdv2 or type in another Microdrive specifier, e.g. mdv3. Easel asks you to confirm your selection of this option.

Warning - all information on the cartridge is erased when you format it.

HIGHLIGHT

This command allows you to use a special symbol to represent a particular number in a set of figures, or all negative values in a bar chart. The value to be highlighted is the one at the current position of the intersection of the crosswires, or the one whose label is highlighted in a pie chart.

Easel first asks you to press either the V key to highlight a particular value, or to press N to highlight all negative values. You are not allowed to select this second option for a pie chart.

If you choose to highlight a value Easel asks you to select the value. Press **TABULATE** (or **SHIFT** and **TABULATE**) to select the cell you want to highlight and then press **ENTER**. In the case of a bar graph you are shown the selection of defined bars, and can choose one – or design a new one. In a pie diagram the selected segment is detached from the remainder of the pie.

If you select the option to highlight negative values, Easel immediately asks you to select or design a bar.

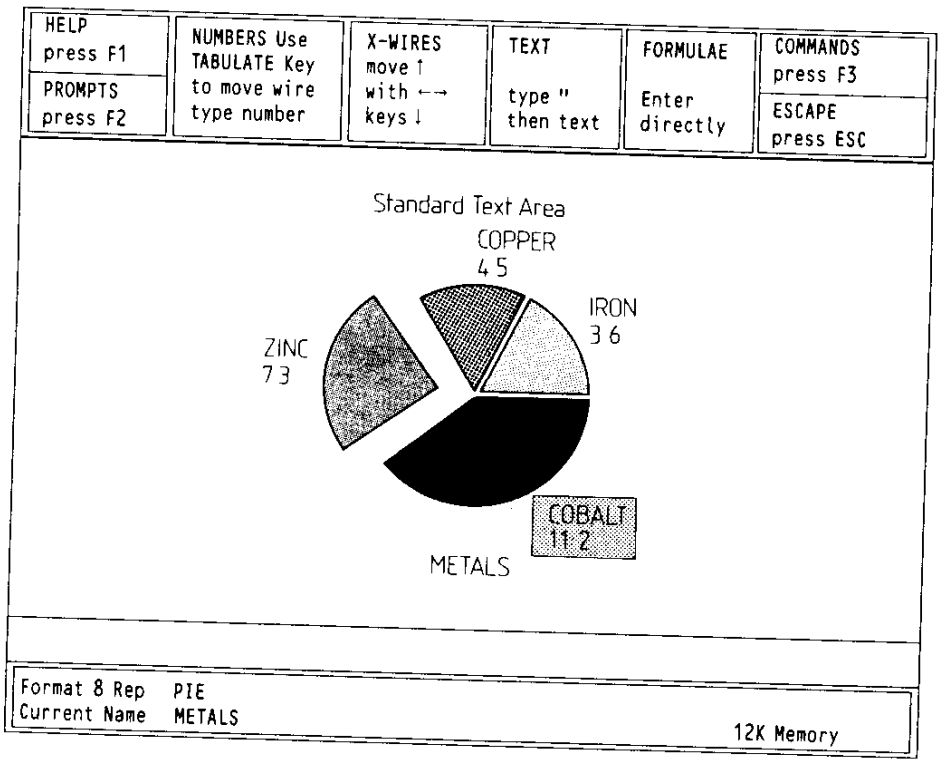


Figure 10.1 A highlighted pie sector.

KILL Deletes one or more sets of figures from the graph and destroys the data. When you select this command you are asked to type in a list of the names of the figures you want to delete, separated by commas and ending with **ENTER**. If you just press **ENTER**, Easel will delete the current figures. You can, if you like, type in the text **all figures**.

LOAD Loads a previously saved graph from a Microdrive cartridge. Easel asks you to type in the name of the file to be loaded. All the Design options are loaded with the data so that the graph of the loaded data has exactly the same appearance as it had when it was saved.

If you do not specify a file name extension, Easel assumes an extension of **_grf**.

NEWDATA Allows you to create a new set of figures, which becomes the 'current figures'. You are asked to type in the name of the new set (no quotation marks are needed). When you press **ENTER** you are returned in data entry mode, ready to type in some values.

OLDDATA The **Olddata** command allows you to make an existing set of figures the 'current figures'. You are asked to type in the name of the old set (no quotation marks are needed). When you press **ENTER** you are returned to data entry mode, ready to change or add to the values.

PRINT

Prints the graph that is currently displayed on the screen.

The command offers three options. Press the P key to print the graph, using the current graphics printer driver.

Press the S key for a screen dump to a Microdrive file. In the Screen dump option Easel asks you to type in a name for the file.

Press the I key if you want to install a different graphics printer driver. Easel will wait for you to type in the name of one of the printer driver files (with an assumed extension of **_prt**) supplied on the Easel cartridge. See Chapter 9 for further information.

QUIT

You use this command to leave Easel and return to SuperBASIC. You are offered the options to press **ENTER** to confirm your choice and return to SuperBASIC, or to press **ESC** to cancel the command and return to Easel's command menu.

RENAME

This command allows you to rename an existing set of figures. Easel asks you to type in the old name, suggesting the current set of figures, and then the new name. Press **ENTER** at the end of each name.

If you do not specify a file name extension for the old file, Easel assumes an extension of **_grf**. The new file is given the same extension as the old one, unless you also type in an extension for the second name.

SAVE

Saves all the sets of figures currently in the computer's memory on a Microdrive cartridge. You are asked to type in a name under which the figures will be saved. If you do not specify a file name extension, Easel assumes an extension of **_grf**.

All the properties of the graph, e.g. the bar colours and style of axes, are saved with the figures.

VIEW

You use this command to redisplay your graph, showing all, or a selected few, of your sets of figures. Easel suggests that all sets of figures are to be displayed and you can either accept this suggestion, by pressing **ENTER**, or type in a list of the names of those sets that you want to be displayed. You should separate the names in the list by commas and end the list by pressing **ENTER**.

In the pie chart format Easel suggests only the name of the current set of figures. If you type in a list of names in this format Easel shows a pie chart of the first name in the list and ignores the remaining names.

You are then offered a suggested format number for the display. You can accept the suggested format (which is the last one you were using) by pressing **ENTER**, or you can type in your own choice of format number, followed by **ENTER**.

ZAP

This command erases all text, all sets of figures and all user-defined objects (bars, lines and so on). It also restores the original month labels for the cells. It does not, however, restore the original appearance of the graph formats, but leaves any changes that you may have made.

FUNCTIONS

Think of a function as a kind of recipe which converts a number of initial values, known as the function's *arguments*, into a different value, which is said to be the value that is *returned* by the function.

The functions provided by Easel take one or no arguments. The argument for a function is placed in brackets after its name. You must not leave a space between the name and the opening bracket, but spaces are allowed within the brackets. All function names must be followed by the brackets, even if they take no arguments. The presence of the brackets is a useful reminder that you are referring to a function. They allow you to distinguish between the name of a set of figures and a function, even if they have the same name.

The following functions are provided.

ABS(n) Returns the absolute value, that is the numerical value irrespective of its sign, of the argument. For example, abs(5) and abs(−5) both return the value 5.

ATN(n) Returns the angle, in radians, whose tangent is n.

COS(n) Returns the cosine of the given (radian) angle.

EXP(n)	Returns the value of e (approximately 2.718) raised to the power n. The returned value will be in error if n lies outside the range from -87 to +88, since the result will then exceed the numeric range of Easel
INT(n)	Returns the integer value of the number, by truncating at the decimal point. The truncation always operates towards smaller numbers. Thus; `int(3.7)` returns 3 `int(-4.8)` returns -5
LN(n)	Returns the natural, or base e, logarithm of n. An error results if n is negative or zero.
PI( )	Returns the value of the mathematical constant π.
SGN(n)	Returns +1, −1, or 0, depending on whether the argument is positive, negative or zero.
SIN(n)	Returns the value of the sine of the specified (radian) angle.
SQR(n)	Returns the square root of the number n, which must not be negative.
TAN(n)	Returns the tangent of the specified (radian) angle.

sinclair

QL

Information

©1984 PSION LIMITED

by Dick de Grandis-Harrison *(Psion Limited)*

QL PROGRAM -IMPORT AN EXPORT

You can transfer information between the four QL programs with the **import** and **export** commands.

Stored information in QL Abacus, QL_Archive and QL Easel is similar and can always be represented as a table. Transferring information between them is very simple. In Abacus and Easel the **import** and **export** commands are file command options. In Archive they are two separate commands.

Let us first consider import and export between Abacus, Archive and Easel. The export file structure produced by the three programs is identical in structure and can be imported by any.

For example, suppose we have an Abacus grid containing the following information:

	A	B	C	D
1	cashflow	January	February	March
2	sales	1000	1050	1100
3	costs	500	530	560
4	profits	500	520	540

Abacus grid for export

If this data was imported into Easel, it would be interpreted as three sets of figures, called *costs, sales and profits*. Easel uses the month names as the cell name labels for the graphs. The information would be:

cell labels	January	February	March
sales graph	1000	1050	1100
costs graph	500	530	560
profits graph	500	520	540

Imported into Easel

Easel does not use the first piece of text, *cashflow*. When you export a set of figures from Easel it automatically inserts the text, *labels*, in this position to maintain compatibility.

If we were to import the same set of figures into Archive the result would be a data file containing three records, each of which would have four fields with the field names: *cashflow$* (a text field), *sales*, *costs* and *profits* (numeric fields). The file would be:

Fields	Record 1	Record 2	Record 3
cashflow$	January	February	March
sales	1000	1050	1100
costs	500	530	560
profits	500	520	540

Imported into Archive

RULES

To allow data to be exchanged between the three programs it is necessary to remember a few rules:

1. When you export the contents of a grid from Abacus the section of the grid being exported must have text in the first cell of each row (or each column if exporting in column order).
2. If the first cell of any row (or column) is empty then that row (or column) is not exported.
3. There must be data in the cell immediately following the text cell in each exported row (or column). The type of this data determines the type used for the data in the rest of the row (or column). Each row (or column) must contain either all numeric or all string data.
4. You can export files from Abacus or Archive which contain several sets of text data. Easel can only export a file containing one set of text data - the cell labels.
5. If you import a file containing more than one set of text data into Easel, it uses the first as cell labels and ignores the rest.

FILE STRUCTURE

The export file structure consists of a series of records each terminated by <CR> (ASCII code 13) and <LF> (ASCII code 10). The import commands will, however, accept either of these characters or the two together, in either order. The end of file is marked by a CTRL Z character (ASCII code 26).

Each record consists of a series of values separated by commas. The values are either text (which must be enclosed in quotes) or numbers.

The first value in each record must be text and if its name ends with a dollar sign all the following values must be text.

The export file produced by exporting the original set of example data from Abacus is as follows:

```
"cashflow$","sales","costs","profits"<LF>
"January",1000,500,500<LF>
"February",1050,530,520<LF>
"March",1100,560,540<LF>
```

An export file

An export file can be generated from SuperBASIC. The following program will generate an export file, called *example_exp*, for the standard data.

```
100 OPEN_NEW#4,mdv2_example_exp
120 PRINT #4,'"cashflow$","sales","costs","profits"'
130 PRINT #4,'"January",1000,500,500'
140 PRINT #4,'"February",1050,530,520'
150 PRINT #4,'"March",1100,560,540'
160 PRINT #4, CHR$(26)
170 CLOSE #4
```

SuperBASIC will automatically add a line feed character (ASCII code 10) at the end of each record.

EXPORT TO QUILL

QL Quill works with formatted text and so files exported to Quill must contain formatted text rather than the normal export file structure. Quill will accept any text containing form feeds (ASCII code 12) and line feeds (ASCII code 10) and the printable ASCII characters. Line feeds are interpreted as an end of paragraph marker and form feeds as an end of page. Any other characters in the file are ignored.

Abacus and Archive can produce special files for import by Quill. Archive can export to Quill by producing 'Formatted report' produced by **Iprint**. To export the report you divert the printed output to a Microdrive file using the **export** option of the **spoolon** command (See chapter 12 of the Archive Guide.)

QL PROGRAMS PRINTERS

The master QL program cartridge is write protected and so cannot be put through the printer install process. The cartridge should first be backed up and the subsequent copy installed.

Each of the four Psion QL programs can print text on almost any make of printer that has an RS-232-C interface.

The printer can be set to use either continuous or single sheet paper. If using single sheet paper the printer will stop at the end of the sheet and a message will appear on the display prompting for more paper. Press **ENTER** to continue or **ESC** to abandon the document.

PRINTER DRIVERS

The printer is controlled by a special program called the **printer driver**, which can be modified to use whatever printer you wish.

A non-printable character, other than a line feed and carriage return, must be preceded by an ASCII code 0 (**NULL**) to indicate to the printer driver that it must be output. For example, the Epson FX-80 command to print in bold characters is ASCII code 27 (**ESC**) and E. **ESC** is a non-printable character and must therefore be preceded by a **NULL**. You can send the codes from Archive with the instruction:

```
lprint chr(0) + chr(27) + "E"
```

In Abacus the same task can be performed by putting:

```
chr(0) + chr(27) + "E"
```

into a cell at the point where bold printing is to start.

Adapting QL Quill, QL Abacus and QL Archive to suit other printers is called installing the software and is done using the SuperBASIC install program. The install program (**install_bas**), installation data for various printers (**install_dat**) and the installation data for the current printer (**printer_dat**) are on the QL Quill and QL Abacus program cartridges. You can use the program to install a printer for QL Archive even though the archive cartridge does not contain the installation program or the installation data.

The Abacus, Archive and Quill programs themselves use only the information in **printer_dat**.

INSTALL A SERIAL PRINTER

For example to install Quill to work with an Epson FX-80, fitted with an RS-232-C interface, put the Quill cartridge in Microdrive 1, but do not run it. While in SuperBASIC type:

```
lrun mdv1_install_bas
```

and the installation program will run. The program requires the '**install_dat**' to be on the cartridge in Microdrive 1 so it shouldn't be deleted.

You must first select the Microdrive in which the printer will be installed. In this case press 1, followed by **ENTER**, to install Microdrive 1. Then press **ENTER** to select a serial printer (connected to the computer via serial port **ser1** or **ser 2**).

The program then reads the installation data and displays a list of the names of printers for which a customised driver driver is supplied.

You select a printer from the list with the up or down cursor keys until the required printer is highlighted and then press **F5** to install it. You must confirm the installation by pressing **ENTER**; any other key will cancel the installation and return to the list of printer names.

When the installation is complete you will be returned to SuperBASIC. When Quill is next loaded it will be set up to use the printer you selected, including bold characters, underlining, subscripts and superscripts.

You can remove a printer from the list by pressing **F3**, and save all the printer drivers in the list by pressing **F4**. Since both of these options make irreversible changes to the printer drive information they must be confirmed by pressing **ENTER**.

OTHER SERIAL PRINTERS

If your printer is not included in the list displayed by the install program you have two options:

Do nothing

Leave the installation program by pressing **ESC**. All four programs are set up with a simple printing facility which should be able to print ordinary text on almost any printer.

Install it

Add a new name to the list of printer names. There are three ways of doing this:

1. Use the down cursor key to select the item called 'OTHER'. Press either **F1** or **F2** to create a new item, ready for you to set it up for your printer.
2. Select an existing printer name and press **F1** to create a new printer with the same values as the old one. Use this option if your printer is similar to a printer already in the list.
3. Select an existing printer and press **F2**. This does not make a new copy, but allows the values of an existing printer to be changed. Do not use this option unless you are sure of the changes you intend to make.

In each case you are shown a list of printer parameters to alter. Press the up and down cursor keys to select an item and the left and right cursor keys to change it.

There are two types of item in the list:

- those with a variety of possible values, such as the DRIVER NAME, and END OF LINE CODE,
- and those with a limited range of values, such as the PARITY.

The values of each type are changed in different ways. The diagram below shows the values given to the DEFAULT printer. At the right of the diagram are other possible values (for those with limited range).

	Default	Other options
DRIVER NAME	: DEFAULT	....
PORT	: ser1	ser2
BAUD RATE	: 9600	75, 300, 600, 1200, 2400, 4800
PARITY	: NONE	SPACE, MARK, ODD, EVEN
LINES/PAGE	: 66	0 to 255
CHARACTERS/LINE	: 80	0 to 255
CONTINUOUS FORMS	: YES	NO
END OF LINE CODE	: CR, LF	....
PREAMBLE CODE	: NONE	....
POSTAMBLE CODE	: NONE	....
BOLD ON	: NONE	....
BOLD OFF	: NONE	....
UNDERLINE ON	: NONE	....
UNDERLINE OFF	: NONE	....
SUBSCRIPT ON	: NONE	....
SUBSCRIPT OFF	: NONE	....
SUPERSCRIPT ON	: NONE	....
SUPERSCRIPT OFF	: NONE	....
TRANSLATE1	: NONE	....
TRANSLATE2	: NONE	....
TRANSLATE3	: NONE	....
TRANSLATE4	: NONE	....
TRANSLATE5	: NONE	....
TRANSLATE6	: NONE	....
TRANSLATE7	: NONE	....
TRANSLATE8	: NONE	....
TRANSLATE9	: NONE	....
TRANSLATE10	: NONE	....

For each of the items that has a limited number of options, the value changes each time the left or the right cursor key is pressed.

For the other items pressing one of these cursor keys erases the existing value; you then type in your own value and press **ENTER**. All these items, except for the DRIVER NAME, will accept lists of up to ten codes separted by commas. Each code can be typed in several ways:

1. A number between 0 and 255
2. A hexadecimal number, preceded by a dollar sign, between $0 and $FF
3. Any single character, preceded by a quote symbol (" or ')
4. A standard ASCII control code mnemonic, in upper or lower case:

NUL	SOH	STX	ETX	EOT	ENQ	ACK	BEL
BS	HT	IF	VT	FF	CR	SO	SI
DLE	DC1	DC2	DC3	DC4	NAK	SYN	ETB
CAN	EM	SUB	ESC	FS	GS	RS	US

5. The text DEF (or def) causes the printer to use a default action making the printer backspace to produce the desired effect. It should only be used for emphasis and underlining. These items must be set in pairs, for example, if UNDERLINE ON is set to DEF then so must UNDERLINE OFF. The printer must be able to respond to the ASCII backspace code.

Alternatively you may just press **ENTER** to select NONE. You are free to mix the different methods in any way you choose.

The DRIVER NAME contains the name of the manufacturer, or of the model, of the printer. It is the name by which you can identify the printer driver. The name must not be more than 16 characters long. To change this item press the left or right cursor key, type in the name you want and press **ENTER**.

The PORT is either ser1 or ser2 and selects one of the two standards serial ports.

The BAUD RATE determines the speed at which characters are via a serial interface, in terms of the number of bits that are transmitted per second. 110 baud is approximately equivalent to 10 characters per second, 300 baud to 30 characters per second, and so on. The baud rate of the printer driver must match that of the serial interface of your printer.

The PARITY item depends on the way your printer handles the most significant bit (binary digit) in the data sent from the computer. All ASCII codes lie between 0 and 127 and can be represented by a 7-digit binary number. Many serial printers expect a character to be sent as a seven bit value. Other printers may expect eight bit values, accepting codes between 0 and 255. The extra codes, between 128 and 255, may be printed as graphics or as accented characters. Your printer may interpret the eighth bit of an 8-bit code as a *parity bit*, used to check if there has been an error during transmission of a character. Your printer may use EVEN parity (the parity bit is set to 0 or 1, so that the total number of 1s in each character code is even) or ODD parity (the total number of 1's is odd). If your printer does not check the parity you can select SPACE (the eighth bit is always 0) or MARK (the eight bit is always 1). A setting of NONE allows the full eight bits to be sent to the printer.

LINES/PAGE and CHARACTERS/LINE specify the maximum number of lines of text (including the blank lines if you are printing double – or triple – spaced text) on each page, and the maximum number of characters on any one line. The values used in the printer drivers supplied are suitable for use with A4 stationery.

CONTINUOUS FORMS specifies whether your printer uses continuous stationery (YES) or separate sheets (NO). If you are printing on single sheets of paper, the printer will stop at the end of each page. A message appears on the screen, asking you to insert a fresh sheet of paper. Press **ENTER** to start printing again, or press **ESC** to abort the print out.

The END OF LINE CODE is the code sequence to be sent to the printer to indicate the end of a line. Most printers will accept a carriage return followed by a line feed. Select a line feed as the end of line marker if you want to print a SuperBASIC program to a file.

The PREAMBLE and POSTAMBLE CODES may be needed if your printer requires an initialisation sequence before you first use it. You may, for example, want to set the printer margin positions, or select a particular character set. You may also want to restore these settings to their original values when you have finished using one of the QL programs. The preamble and postamble items allow you to specify a sequence of up to 10 characters to be sent to the printer for these two purposes.

The BOLD ON and OFF items contain the codes to turn bold (emphasised) printing on and off. If your printer cannot print emphasised characters you can use the DEF value, described earlier, provided the printer will respond to a backspace character.

UNDERLINE ON and OFF turn underlining on and off, provided your printer has an automatic underlining facility. If your printer cannot print underlined characters you can use the DEF value, described earlier, provided the printer will respond to a backspace character.

Use the SUBSCRIPT ON/OFF and SUPERSCRIPT ON/OFF items for the sequence of codes needed by your printer to turn subscript and superscript printing on and off.

Each of TRANSLATE1 to TRANSLATE10 accepts up to ten characters. The first character specified is translated into the following sequence of characters before being sent to the printer. The first character must not be a control character (its ASCII code must be in the range 32 to 255). The translation can contain any character. The result must appear as a single character when printed.

As an example, let us create a second printer driver for the Epson FX-80. Start by loading and running the installation program from SuperBASIC. Select the driver named OTHER and press either **F1** or **F2**. The initial values displayed are listed below, the column at the right showing the values needed for the FX-80:

	Default	Other options
DRIVER NAME	: OTHER	....
PORT	: ser1	: ser2
BAUD RATE	: 9600	: 9600
PARITY	: NONE	: NONE
LINES/PAGE	: 66	: 66
CHARACTERS/LINE	: 80	: 80
CONTINUOUS FORMS	: NO	: YES
END OF LINE CODE	: CR, LF	: CR, LF
PREAMBLE CODE	: NONE	: esc,@,ESC,R,NUL
POSTAMBLE CODE	: NONE	: NONE
EMPHASIZE ON	: NONE	: ESC,E
EMPHASIZE OFF	: NONE	: ESC,F
UNDERLINE ON	: NONE	: ESC - 1
UNDERLINE OFF	: NONE	: ESC - 0
SUBSCRIPT ON	: NONE	: ESC,S,1
SUBSCRIPT OFF	: NONE	: ESC,T
SUPERSCRIPT ON	: NONE	: ESC,S,0
SUPERSCRIPT OFF	: NONE	: ESC,T
TRANSLATE1	: NONE	: £,ESC,R,ETX, # ,ESC,R,NULL
TRANSLATE2	: NONE	: NONE
TRANSLATE3	: NONE	: NONE
TRANSLATE4	: NONE	: NONE
TRANSLATE5	: NONE	: NONE
TRANSLATE6	: NONE	: NONE
TRANSLATE7	: NONE	: NONE
TRANSLATE8	: NONE	: NONE
TRANSLATE9	: NONE	: NONE
TRANSLATE10	: NONE	: NONE

First change the driver name, press the right cursor key to erase the existing text, and type

FX-80 ENTER

If you make a mistake you can repeat the process.

Press the down cursor key until the CONTINUOUS FORMS entry is highlighted. There are only two options; select YES by pressing the right or left cursor key.

A suitable PREAMBLE sequence for the Epson FX-80 is **ESC** which initialises the printer and clears its print buffer. The printer should also be set to use the American character set (to print both the hash symbol, #, and the pound sign – see later). The FX-80 code to do this is **ESC R NUL**.

Use the cursor keys to select the PREAMBLE and press the right (or left) cursor key to erase the current value. The following three options all produce the same result and initialise the printer:

```
ESC,"@,ESC,"R,NUL
27,64,27,82,0
$1B,$40,$1B,$52,$0
```

You could use this item to set other printer properties, such as the line spacing or italic characters. If your printer doesn't require any initialisation then you can leave the initial setting at NONE.

The FX-80 doesn't need a POSTAMBLE so the setting can be kept at NONE.

EMPHASIZE ON and EMPHASIZE OFF codes for the Epson FX-80 are **ESC E** and **ESC F** respectively. You can set them by typing:

```
esc,"E
esc,"F
```

The remaining codes can be set by typing:

Item	You type
UNDERLINE ON	`esc,"-,"1`
OFF	`esc,"-,"0`
SUBSCRIPT ON	`esc,"S,"1`
OFF	`esc,"T`
SUPERSCRIPT ON	`esc,"S,"0`
OFF	`esc,"T`
TRANSLATE1	`£,esc,R,ETX,#,ESC,R,NUL`

In the above example TRANSLATE 1 enables the Epson FX-80 to print a pound sign, which is only available in the English chracter set. The QL pound sign is translated to:

Switch to the English character set
print a hash symbol (which appears as a pound sign)
switch back to the American chracter set

When you have finished editing the printer codes you can install the printer by pressing **F5**. Alternatively you can return to the list of printers, ready to make mode changes.

INSTALL FOR QL ARCHIVE

Put a QL Quill or QL Abacus cartridge in Microdrive 1 and a QL Archive cartridge in Microdrive 2. Load and run **install_bas** from Microdrive 1 but then press 2, followed by **ENTER**, to indicate that you want to install a printer to Microdrive 2.

Follow the installation procedure as normal. The installation data will be read from Microdrive 1 but the printer will be installed to the cartridge in Microdrive 2.

PARALLEL PRINTERS

The installation program allows the installation of a printer connected to the QL via ports other than **ser1** or **ser2**. You would use this option if, for example, you have added an optional parallel interface. Load and run **install_bas** as described earlier. After you have selected installation to Microdrive 1 or 2, press the space bar to select the parallel port option.

The list of printers appears as before but when you press **F1** or **F2** the list of parameters appears as shown in the following table.

	Default	Other options
DRIVER NAME	: DEFAULT	....
PORT	: NONE	....
LINES/PAGE	: 66	0 to 255
CHARACTERS/LINE	: 80	0 to 255
CONTINUOUS FORMS	: YES	NO
END OF LINE CODE	: CR, LF	....
PREAMBLE CODE	: NONE	....
POSTAMBLE CODE	: NONE	....
EMPHASIZE ON	: NONE	....
EMPHASIZE OFF	: NONE	....
UNDERLINE ON	: NONE	....
UNDERLINE OFF	: NONE	....
SUBSCRIPT ON	: NONE	....
SUBSCRIPT OFF	: NONE	....
SUPERSCRIPT ON	: NONE	....
SUPERSCRIPT OFF	: NONE	....
TRANSLATE1	: NONE	....
TRANSLATE2	: NONE	....
TRANSLATE3	: NONE	....
TRANSLATE4	: NONE	....
TRANSLATE5	: NONE	....
TRANSLATE6	: NONE	....
TRANSLATE7	: NONE	....
TRANSLATE8	: NONE	....
TRANSLATE9	: NONE	....
TRANSLATE10	: NONE	....

You are not given the option to select the baud rate or parity since they are only relevant for a serial interface via **ser1** or **ser2**. The PORT section is also different. Change this item by pressing either the left or right cursor key and then typing any valid *device name*, of up to sixteen characters. Refer to the *Devices* section of *QL Concepts*, or the manual accompanying an add on interface.

Apart from these differences, the remainder of the installation is exactly the same as described for a serial interface.

THE CONVERT UTILITY

Version 2.0 of the **install_bas** program has been modified to offer a wide range of printer options. This means that it is not compatible with **install_dat** files created with version 1. A conversion program, **convert_bas**, is supplied to convert version 1 **install_dat** files so that they are readable by the version 2.0 installation program.

First copy **convert_bas** to another cartridge. Put the cartridge containing the copy of **convert_bas** in Microdrive 1 and a cartridge containing your version 1 **install_dat** file in Microdrive 2. Run the program by typing:

```
lrun mdv1_convert_bas
```

The program reads the **install_dat** file in Microdrive 2 and writes the new version to Microdrive 1. Note that the new version will replace any **install_dat** file on this cartridge. You can then, if necessary, copy the new **install_dat** file to another cartridge.

QL PROGRAM - CONFIG

The program **config_bas** allows you to specify alternative default devices for the QL programs and to modify the sort order in the Order commands of Abacus and Archive.

As supplied, the programs expect to use Microdrive 2 for storing data, and Help information and the installed printer driver are on Microdrive 1. You may wish to modify these to make use of additional Microdrives, disk drives, and so on.

You may also wish to modify the order in which Archive records, or rows of an Abacus grid are sorted. This might be useful, for example, if you want to sort text which includes accented characters from a foreign language.

USING CONFIG

You can run **config_bas** from any Microdrive, and modify a QL program on a cartridge in either Microdrive 1 or Microdrive 2. Suppose you want to run **config_bas** from Microdrive 2 to modify a copy of a QL program in Microdrive 1. Run the program by typing:

```
lrun mdv2_config_bas
```

When prompted, type the name of the program you want to modify (Quill, Abacus, Easel or Archive) and press **ENTER**. Then enter the value **1** when asked which drive contains the program.

The program waits for you to press the space bar after you have made sure that the program cartridge is in the correct Microdrive. When you have done so the program shows you the main menu of options which are:

Select new default devices
Modify the sort order
Leave the program

Sort Order

To select the option to modify the sort order press **ENTER**. When prompted press the space bar.

The largest area of the screen shows a block of 256 characters which define the sort order. The *position* in the block, reading from left to right and top to bottom, determines the character being sorted; the *contents* at that position shows how the character will be tested by the Order command. The right hand side of the screen shows more information about the character marked by the cursor. Move the cursor from character to character with the cursor keys.

The block of characters at the bottom of the screen is used for modifying the order. It also has a cursor, which you move with **SHIFT** and the cursor keys. This block only shows half of the full set of characters – press **F1** to switch between the two halves.

The best way of describing how to modify the sort order is by means of examples. As supplied, the lower case characters will be sorted to come after all the upper case ones, that is, "a" will come after "Z". Suppose you want to make the order independent of upper or lower case so that, for example, "A" and "a" are not distinguished.

To make "a" be sorted as though it were "A", move the cursor in the main block of characters to the letter "a" and press the "A" key (make sure you type an upper case character). The "a" in the upper block changes to "A" and the information on the right of the screen shows that the character "a" will now be regarded as equivalent to "A" for the purpose of sorting.

Repeat this process for each lower case letter, making "b" equivalent to "B", "c" to "C", and so on.

An alternative way of changing a character is to move the cursor in the lower block of characters, using **SHIFT** and the cursor keys, until it marks the character you require and then press **F2**. This method is particularly useful for the characters, such as foreign accented characters, that are not marked on the keys. This method is used in the following example.

Suppose you want to reverse the normal sort order for the upper case letters, leaving the rest of the ordering unchanged. To do this you must change the part of the main block that reads "A B C . . . X Y Z" so that it reads "Z Y X . . . C B A". Move the main cursor to "A" and the lower cursor to "Z" and press **F2** to enter the new character. The character "A" will then sort as though it were "Z". Repeat this for each upper case letter, changing "B" to "Y", "C" to "X", and so on.

When you have completed specifying the sort order, press **F5** to save the new order in the QL program, replacing the old one. Press **ESC** to return to the main menu.

Device Selection

As supplied, the QL programs use Microdrive 1 for system information (the installed printer driver, for example) and for Help. They all use Microdrive 2 for their data.

From the main menu press the space bar to choose new default devices. Press the space bar again when prompted.

After reading the current settings from your program cartridge, the program shows you these values and waits for you to type in your new choices. Press **ENTER** to keep an old value, or type in your new selection and press **ENTER**.

Having made your selection you may save your new devices, reselect the devices or cancel this option and return to the main menu.

If you save your device selection, the QL program will use these devices until you use **config_bas** to change them again.

QL PROGRAM ARITHMETIC

Except for Easel the valid range for numbers in the QL programs is:

$\pm\ 2.9*10^{39}$ to $\pm\ 1.7*10^{38}$

All calculations are accurate to sixteen significant digits but only a maximum of fourteen characters can be displayed.

In easel the range of valid numbers is

$\pm\ 1.0^{-35}$ to $\pm\ 1.0 \times 10^{+36}$

The following arithmetic operators are provided in Abacus, Archive and Easel:

Operator	Function
+	Addition on numbers or concatenation on strings
-	Subtraction
*	Multiplication
/	Division
^	Raise to a power
=	Equal
>	Greater than
<	Lesser than
< =	Lesser than or equal to
> =	Greater than or equal to
< >	Not equal to

There is no automatic coercion between data types. Therefore, operands must be of the same type. The result is always a number, 1 if the comparison is true and 0 if it is false.

Functions and operators have the following precedence:

Operation	Precedence
Subscripting and slicing	12
All functions	11
^	10
Unary minus	9
*, /	8
+, -	6
=, >, <, <=, >=, < >	5
not	4
and	3
or	2

FORMAT PROCEDURE

Formatting a cartridge will result in overwriting any data that was previously stored on the cartridge. This data cannot be recovered so ensure that you only format blank cartridges or cartridges that have no useful information on them.

First decide on a name for the cartridge, using not more than ten characters. With the QL switched on and displaying the flashing cursor, place the cartridge to be formatted in Microdrive 1. Let us assume that the cartridge name is to be 'data'. Then type

```
FORMAT mdv1_data
```

Do not confuse the underscore symbol (_) with the minus sign (-), since they are on the same key. The underscore symbol is the upper one and so **SHIFT** must be held down while the key is pressed.

Press **ENTER** and the left hand Microdrive light will glow for about thirty seconds. The QL will output a message on the screen indicating how much space is available on that cartridge. The **FORMAT** command is described in full in the *Keywords* section.

It is good practice to format a new cartridge several times. This will help the tape to run smoothly and may result in a greater capacity.

The cartridge could equally well have been formatted in Microdrive 2 by substituting **mdv2_** for **mdv1_**.

BACKUP PROCEDURE

A backup is made by copying all the files contained on the cartridge to be backed up onto a blank cartridge. Preferably the blank cartridge will be newly formatted and named to reflect that it is a backup.

Choose a blank cartridge or a cartridge that holds no useful information and place it in Microdrive 1. Decide on a name for the cartridge, for example, if the name of the cartridge to be backed up is 'QL_data' then 'QL_data_bak' would be a good name for the backup cartridge. Then type:

```
FORMAT mdv1_data_bak
```

followed by **ENTER**. The left hand Microdrive light will glow for about thirty seconds.

Place the cartridge to be backed up into Microdrive 2 and type

```
DIR mdv2_
```

this will list all the files contained on this cartridge.

For each file listed type:

```
COPY mdv2_filename TO mdv1_filename
```

substituting the relevant file names where marked. This command will copy each specified file from Microdrive 2 to Microdrive 1. The speed of this operation depends on the sizes of the files being copied: the operation could take some time.

Repeat the **COPY** command for each of the listed files. When complete, the backup cartridge (the one in Microdrive 2) should be marked with the data and the name of the cartridge for which it is a backup and then put in a safe place.

Normally for each cartridge that you work with and which contains data you may have one, two, or more backup cartridges depending on how important the data is. If you use this system then always backup onto the oldest backup cartridge in the set.

QL Users Bureau

You may find that you would like some assistance in using and getting the most out of your four supplied QL programs. The QL Users Bureau (QLUB) has been founded to enable you to do just this and keep you in touch with developments on the QL.

Membership is only available to QL owners and entitles you to a number of benefits:

- Regular newsletters will be published giving technical information and tips. Information will be given on new software and peripherals with an opportunity to buy new products in advance of the general public.
- Sinclair Research has made arrangements for QLUB members to obtain software assistance from PSION. If you have any queries concerning the use or application of QL Abacus, Archive, Quill or Easel, all you have to do is write to PSION direct using a QLUB Query Form and quoting your membership number. PSION will usually reply within 48 hours.
- When updates to the four QL programs (QL Abacus, Archive, Quill and Easel) are issued they will be offered to QLUB members at special discount prices.
- Membership of the QLUB is FREE and for LIFE.

To join the QLUB please fill in the registration form in the *Information* section of your QL User Guide.

Psion can assist members solely on the QL programs supplied with your QL. QL software which you buy subsequently will, where appropriate, offer separate service and update arrangements.

If you feel you need help in using the QL itself, rather than the QL programs, please refer first to the relevant section(s) in your QL User Guide. Numerous publishers are selling books about all kinds of different aspects of the QL. You may find these not only helpful but also very interesting.

Companies independent of Sinclair Research are offering QL user courses and symposia and we are encouraging such developments. We will keep all members informed of such activities through the newsletter.

ORDER FORM

	Quantity	Unit Price	Code	Total
Monitor lead (2 metre)*		7.95	6030	
RS-232-C lead (2 metre)		14.95	6040	
Joystick adaptor		5.95	6060	
		SUB TOTAL		
POST AND PACKING – Orders	under £90	2.95	0028	
	£90 to £390	4.95	0029	
	over £390	7.95	6999	

TOTAL

Please tick box if VAT receipt required: ☐

* I enclose a cheque/postal order payable to Sinclair Research for £

* Please charge my Access/Barclaycard/ Trustcard account number

* please delete as applicable

Signature: _____

PLEASE PRINT

Mr/Mrs/Miss _____

Address: _____

_____

_____

_____

Please send this form and your remittance (if paying by cheque or postal order) to:

Sinclair Research Limited
FREEPOST, Camberley, SURREY. GU15 3PS
Telephone: Camberley (0276) 685311

Please allow up to 28 days for delivery

GUARANTEE

Your Sinclair QL is covered by a 12 month comprehensive guarantee valid in the UK only and effective from the date of dispatch. It is not transferable. The guarantee is invalidated if the product is opened, modified, repaired or tampered with by any party other than Sinclair Research Limited or their agents. This guarantee does not affect your statutory rights.

A guarantee card is enclosed with your QL. Please read it straight away if you have not already done so.

If you have a problem then it may not be immediately clear whether this is caused by the QL itself or one of the QL program cartridges.

Please follow the instructions below.

IF none of the four software cartridges will load

THEN your QL is probably faulty.

OR

IF all the software packages run successfully but all display a similar fault when running your QL is probably faulty. If you believe your QL to be faulty, please take your complete QL package (including the QL programs) in its original polystyrene box back to the shop from which you originally purchased it, together with your proof of purchase.

THEN If you bought the QL by mail order from Sinclair Research Limited then please send the QL package to:

Sinclair Research Limited,
Stanhope Road, Camberley,
Surrey, GU15 3BR

Please use recorded delivery or registered post and keep proof of postage. Please send with the package a letter indicating your present address (and your old address if you have moved since ordering the QL), and giving any details you can about the nature of the fault which it has developed.

If your QL is faulty it will be repaired or replaced at Sinclair's option, free of charge within the guarantee period.

IF one or more of the software packages fails to load, but at least one cartridge loads and runs successfully

THEN the cartridges which you have are probably faulty, and not the QL. Whether you bought the QL from a shop or by mail order from Sinclair Research Limited please return the faulty QL program cartridge(s) in its protective case to

Sinclair Research Limited,
Stanhope Road, Camberley,
Surrey, GU15 3BR