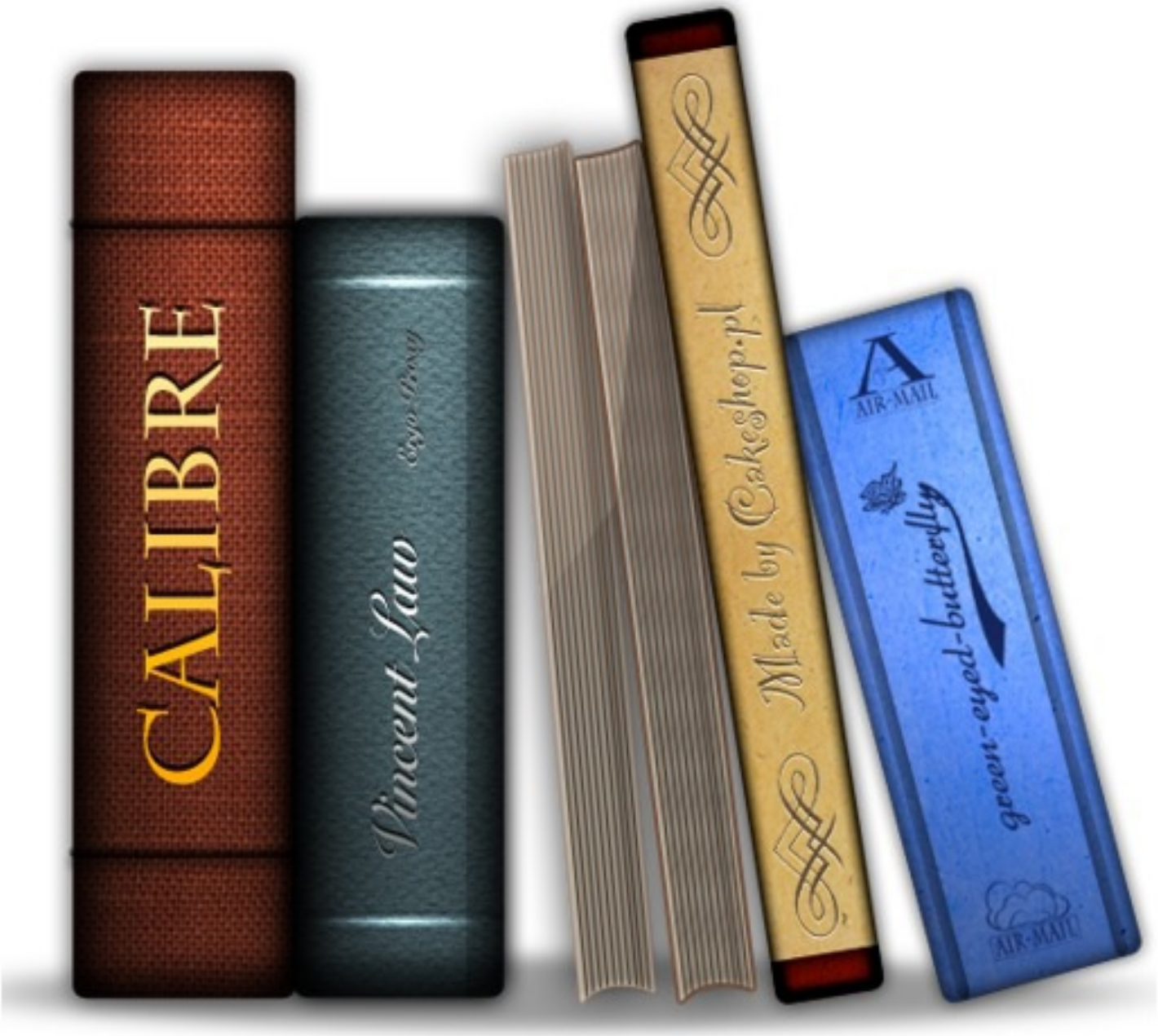
QL User Manual

Paolo Proietti

Libro 1 di Sinclair QL



calibre 0.7.35

In no circumstances will either Sinclair Research Limited or Psion Limited be liable for any direct, indirect, incidental or consequential damage or loss including but not limited to loss of use, stored data, profit or contracts which may arise from any error, defect or failure of the QL hardware or the software supplied with it.
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.
QL User Guide First Edition
Published by Sinclair Research Limited 1984
25 Willis Road, Cambridge.
Edited by Stephen Berry (Sinclair Research Limited)

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

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 QL USER GUIDE
INTRODUCTION
BEGINNER'S GUIDE
REFERENCE GUIDE
KEYWORDS
CONCEPTS
APPLICATIONS SOFTWARE
QL QUILL
QL ABACUS
QL ARCHIVE
QL EASEL
INFORMATION
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.
OL User Guide Second Edition
Published by Sinclair Research Limited 1984
25 Willis Road, Cambridge
Edited by Stephen Berry (Sinclair Research Limited)

Printed and bound in Great Britain by
William Clowes Limited,Beccles and London
Designed and typeset by
Keywords, Manchester

INTRODUCTION

An 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:
4 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
SERI - RS232C serial port
SER2 - RS232C serial port
CTLI - control port for joystick
CTL2 - control port for joystick
ROM - QL ROM cartridge software (use reversed one to 10)
N.B. ZX ROM cartridges are not compatible with QL ROM cartridges and cannot be used with 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

THE POWER SUPPLY

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

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.

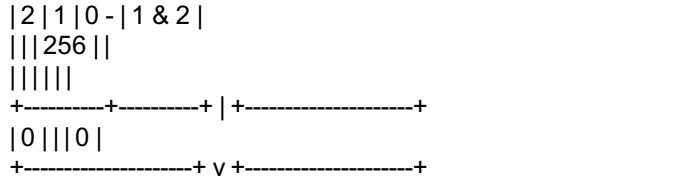
TUNING IN

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

| |
| |
| |
| |
| F1...MONITOR |
| F2...TV |
| |
| (C)1983 Sinclair Research Ltd |
| |
+------------------------------+
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 <- (left arrow) 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 -> (right arrow) 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 Fl or F2, the screen will look like this:
--------0 to 511------> -------0 to 511------->
+----------+----------+ | +--------------------+
| | | | | |
| | | | | |



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 mdvl_ 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.

INTRODUCTION TO THE QL PROGRAMS

This introduction outlines the four programs supplied wlth the OL and

describes their common features.
The four programs are:
QL Ouill - 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 OL 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 mdvl_ 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 removlng the master cartridge from Mlcrodrlve 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 Fl 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 wlth 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 mdvl_boot
press ENTER and loading will proceed as before.
In both cases the program wlll 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 Quill 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. In the table below, <- and -> mean the cursor left and right arrow keys respectively, while the cursor up and down arrow keys are represented by <cursor up> and <cursor down> respectively.

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 & <cursor up> Delete the line to the left of the cursor
CTRL & <cursor down> 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, Ouill 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 mdvl_, 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 cartridge; press Fl 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 cartrldge In Microdrive 1 and load the program by typing
lrun mdvl_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 sending
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.

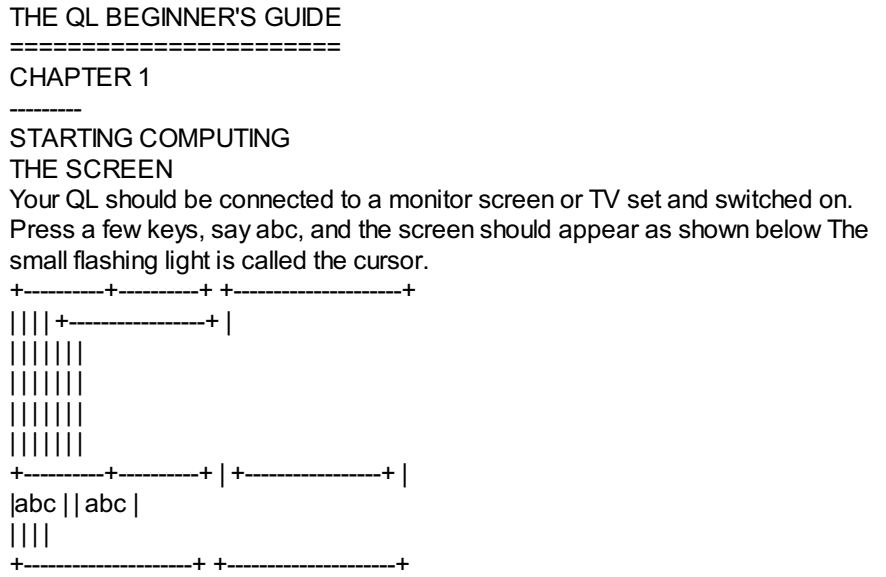
THE QL BEGINNER'S GUIDE

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.



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.

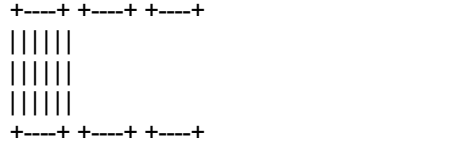
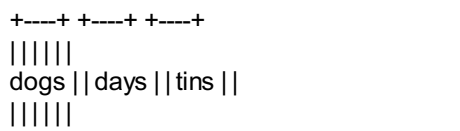
RUBBING OUT

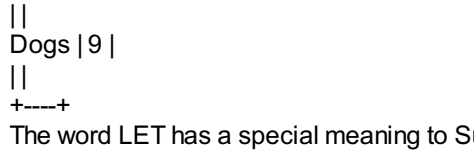
The left cursor together with the CTRL key acts like a rubber. You must hold down the CTRL key while you press the cursor key. Each time you then press both together the previous character is deleted.

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:
[ENTER]
Where we use the [ENTER] symbol to represent a keystroke, we mean press the key labelled ENTER, not type in the letters e n t e r.
We shall use this for convenience, better presentation, and to save space.
Test the [ENTER] key by typing
PRINT "Correct" [ENTER]
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 [ENTER]
cls [ENTER]
clS [ENTER]
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" [ENTER]
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 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.
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 [ENTER]
PRINT 'Hello' [ENTER]
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 [ENTER]
20 PRINT 'HELLO' [ENTER]
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 [ENTER]
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 [ENTER]
causes an internally stored program to be displayed (listed) on the screen or elsewhere.
NEW [ENTER]
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 in the ANSWERS pages at the end of this Beginner's Guide.
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 [ENTER]
9. RUN [ENTER]
10. LIST [ENTER]
11. NEW [ENTER]
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?
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 [ENTER]
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 [ENTER]
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 [ENTER]
The screen should displav 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 [ENTER]
LET days = 28 [ENTER]
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 [ENTER]
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 [ENTER
which will cause the output:
252
to be displayed on the screen.
In summary the program:
LET dogs = 9 [ENTER]
LET days = 28 [ENTER]
LET tins = dogs * days [ENTER]
PRINT tins [ENTER]
causes the internal effects best imagined as three named pigeon holes containing numbers:

```
+----+ +----+ +----+
| | | | | |
dogs | 9 | x days | 28 | = tins | 252|
| | | | | |
+----+ +----+ +----+
```

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 [ENTER]
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[ENTER
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 [ENTER]
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[ENTER
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 [ENTER]
PRINT dogs [ENTER]
LET dogs = dogs + 1 [ENTER]
PRINT dogs [ENTER]
The output should be:
9
10
proving that the final value in the pigeon hole is as shown:

dogs	10

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 [ENTER]
LET days = 28 [ENTER]
LET tins = dogs * days [ENTER]
PRINT tins [ENTER]
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 [ENTER]
PRINT bars [ENTER]
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.

bars	pounds
LET pounds = 7 [ENTER]	7	
LET bars = pounds * 2 [ENTER]	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 [ENTER]
20 LET pens = 7 [ENTER]
30 LET cost = price * pens [ENTER]
40 PRINT cost [ENTER]

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

RUN [ENTER]

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 [ENTER]

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

RUN [ENTER]

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" [ENTER]

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.

Now type:

RUN [ENTER]

and the new output should be:

Total cost 140

DELETE LINE

You can delete line 35 by typing:

35 [ENTER]

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" [ENTER]

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

PRINT bars [ENTER]

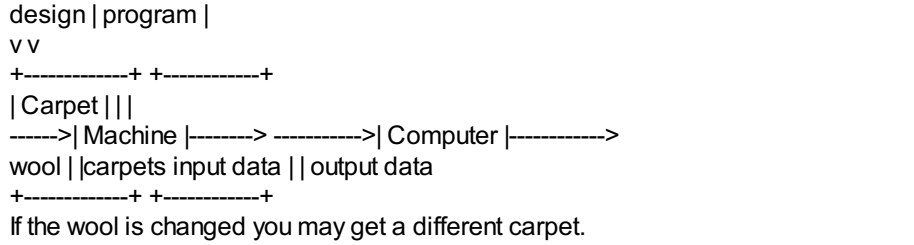
without using quotes.

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 [ENTER]

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 [ENTER]
20 PRINT "How many pens?" [ENTER]
30 INPUT pens [ENTER]
40 LET cost = price * pens [ENTER]
50 PRINT cost [ENTER]
RUN [ENTER]
```

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

4 [ENTER]

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 [ENTER]
10 READ price, pens
20 LET cost = price * pens [ENTER]
30 PRINT cost [ENTER]
40 DATA 15, 4 [ENTER]
RUN [ENTER]
```

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.
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 in "Answers To Self Test" section at the end of this Beginner's Guide.
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?

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 [ENTER]
20 LET rate = 31 [ENTER]
30 LET wage = hours * rate [ENTER]
40 PRINT hours, rate, wage [ENTER]
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).

(This diagram is Chapt3A_pic in the file compilations)

Scale Graphics

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 [ENTER]
10 PAPER 7 : CLS [ENTER]
20 INK 2 [ENTER]
30 LINE 50,60 TO 70,80 [ENTER]
RUN [ENTER]
```

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

RANDOM EFFECTS

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) [ENTER]

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 [ENTER]
10 LET die = RND(l TO 6) [ENTER]
20 PRINT die [ENTER]
RUN [ENTER]

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 [ENTER]
10 PAPER 7 : CLS [ENTER]
20 INK RND(5) [ENTER]
30 LINE 50,60 TO RND(100),RND(100) [ENTER]
RUN [ENTER]

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 [ENTER]

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

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 Fixed part - not repeated
20
30 Changeable part - repeated
40
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 [ENTER]
10 PAPER 6 : CLS [ENTER]
20 BORDER 1,2 [ENTER]
30 REPEAT star [ENTER]
40 INK RND(5) [ENTER]
50 LINE 50,60 TO RND(100),RND(100) [ENTER]
60 END REPEAT star [ENTER]
RUN [ENTER]
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.
(diagram of random length lines emanating from a central point like a broken pane of glass - Chapt3B_pic - caption: "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) [ENTER]
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 [ENTER]
10 PAPER 6 : CLS [ENTER]
20 BORDER 1 ,2 [ENTER]
30 REPeat star [ENTER]
40 LET colour = RND(6) [ENTER]
50 IF colour = 6 THEN EXIT star [ENTER]
60 INK colour [ENTER]
70 LINE 50,60 TO RND(100),RND(100) [ENTER]
80 END REPeat star [ENTER]
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 in the "Answers to self test" section at the end of this Beginner's Guide.
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?

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:

```
+-----+ +-----+
| | | |
weekday$ | | month$ | |
| | | |
+-----+ +-----+
```

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" [ENTER]

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

```
+-----+
| |
weekday$ |FIRST|
| |
+-----+
```

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$ [ENTER]

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$) [ENTER]
```

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

```
NEW [ENTER]
10 LET weekday$ = "FIRST" [ENTER]
20 PRINT LEN(weekday$) [ENTER]
RUN [ENTER]
```

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 [ENTER]
10 LET weekday$ = "FIRST" [ENTER]
20 LET length = LEN(weekday$) [ENTER]
30 PRINT length [ENTER]
RUN [ENTER]
```

The screen should display:

```
5
```

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

```
+-----+ +-----+
| | | |
weekday$ |FIRST| length | 5 |
| | | |
+-----+ +-----+
```

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

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

```
5
```

and there are eight internal pigeon holes involved:

```
+--------+ +-----+
| | | |
weekday$ | FIRST | length1 | 5 |
| | | |
+--------+ +-----+
+--------+ +-----+
| | | |
word$ | OF | length2 | 2 |
| | | |
+--------+ +-----+
+--------+ +-----+
| | | |
month$ |FEBRUARY| length3 | 8 |
| | | |
+--------+ +-----+
+-----+
| |
```

sum | 15 |
| |
+-----+
+-----+
| |
average | 5 |
| |
+-----+
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 [ENTER]
10 LET weekday$ = "FIRST" [ENTER]
20 LET word$ = "OF" [ENTER]
30 LET month$ = "FEBRUARY" [ENTER]
40 LET phrase$ = weekday$ & word$ & month$ [ENTER]
50 LET length = LEN(phrase$) [ENTER]
60 PRINT length/3 [ENTER]
RUN [ENTER]
The output is 5 as before but there are some different internal effects:
+------------------+ +----+
| | | |
weekday$ | FIRST | length | 15 |
| | | |
+------------------+ +----+
+------------------+
| |
word$ | OF |
| |
+------------------+
+------------------+
| |
month | FEBRUARY |
| |
+------------------+
+------------------+
| |
phrase$ | FIRSTOFFEBRUARY |
| |
+------------------+
There is one more reasonable simplification which is to use READ and DATA instead of the first three LET statements. Type:
NEW [ENTER]
10 READ weekday$, word$, month$ [ENTER]
20 LET phrase$ = weekday$ & word$ & month$ [ENTER]
30 LET length = LEN(phrase$) [ENTER]
40 PRINT length/3 [ENTER]
50 DATA "FIRST","OF","FEBRUARY" [ENTER]
RUN [ENTER]
The internal effects of this version are exactly the same as those of the previous one. READ causes the setting up of internal pigeon holes with values in them in a similar way to LET.
IDENTIFIERS AND STRING VARIABLES
Names of pigeon holes, such as:
weekday$
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 [ENTER]
10 LET lettercode = 66 [ENTER]
20 PRINT CHR$ (lettercode) [ENTER]
RUN [ENTER]
```

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

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

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.

(From now on, we shall omit the [ENTER] key symbol at the end of each line of a program to be entered, on the assumption that you are by now familiar with the use of the ENTER key)

SELF TEST ON CHAPTER 4

You can score a maximum of 10 points from the following test. Check your score with the answers in the "Answers To Self Tests" section at the end of this Beginner's Guide.

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.

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
[ENTER] 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:

<-- (left arrow key) one place left
--> (right arrow key) one place right

To delete a character to the left use:

CTRL with <-- (CTRL and left arrow keys)

To delete the character in the cursor position type:

CTRL with --> (CTRL and right arrow keys)

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 with the best machines or devices there 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 in the "Answers To Self Tests" section at the back of this Beginner's Guide.
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 = O
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

```
+-----+ +-----+ +-----+ +-----+
| VAL | | HAL | | MEL | | DEL |
+-----+ +-----+ +-----+ +-----+
```

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"
```

```
westlea$ lakeside$ roselawn$ oaktree$
|  |  |  |
|  |  |  |
v  v  v  v
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)
    |  |
    |  +------ maximum length of string
    |
    +--------- number of string variables
```

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.

```
+--+ ------- ------- ------- -------
|  | / 1 \ / 2 \ / 3 \ / 4 \
+----------+ +----------+ +----------+ +----------+ +----------+
| high_st$ > | ++ ++ | | ++ ++ | | ++ ++ | | ++ ++ |
+----------+ | ++ +-+ ++ | | ++ +-+ ++ | | ++ +-+ ++ | | ++ +-+ ++ |
|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
|  | +----+-+----+ +----+-+----+ +----+-+----+ +----+-+----+
|  |
|  |
```

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$(4,3)
110 LET high_st$(l) = "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:

```
+-----+ +-----+ +-----+ +-----+
high_st$ | 1 | | 2 | | 3 | | 4 |
+-----+ +-----+ +-----+ +-----+
```

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(l TO 6)
150 LET die2 = RND(l TO 6)
160 LET score = diel + 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
21O 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
26O 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 2 and 12. As every dice player knows, the reflects the frequency of the middle range of scores (six,seven,eight) and the rarity of twos or twelves.

```
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 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.

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:
procedure definition procedure call

```
+--------------------------------+
| DEFine PROCedure square |
| REMark code to draw square | <---------- square
| END DEFine |
+--------------------------------+
|
|
v
draws a square
```

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$(13,15)
140 store_data
150 FOR phrase = 1 TO 10
160 get_random
170 make_phrase
180 END FOR phrase
190 REMark ***************************
200 REMark * Procedure Definitions *
210 REMark ***************************
220 DEFine PROCedure store_data
230 REMark *** procedureto store the buzzword data ***
240 RESTORE 420
250 FOR item = 1 TO 13
260 READ adjec1$(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 ! adjec!$(ad1) ! adjec2$(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

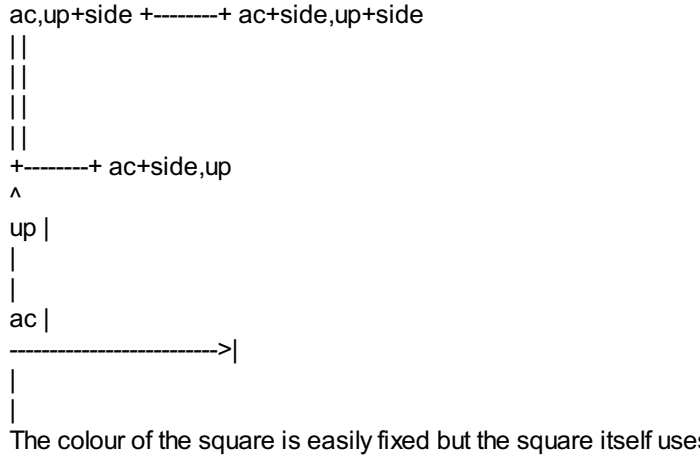
PASSING INFORMATION TO PROCEDURES

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
| | |
| | |
v v v
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 they '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 advantages 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 TESTON CHAPTER 7

You can score a maximum of 14 points from the following test. Check your score with the "Answers To Self Tests" section at the back of this Beginner's Guide.

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?

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 andlor 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$ > item2$ 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: count string: 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
```

---
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 MODE 256
8 Colour Mode MODE 8
256 pixels across, 256 down

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

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 bv 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.

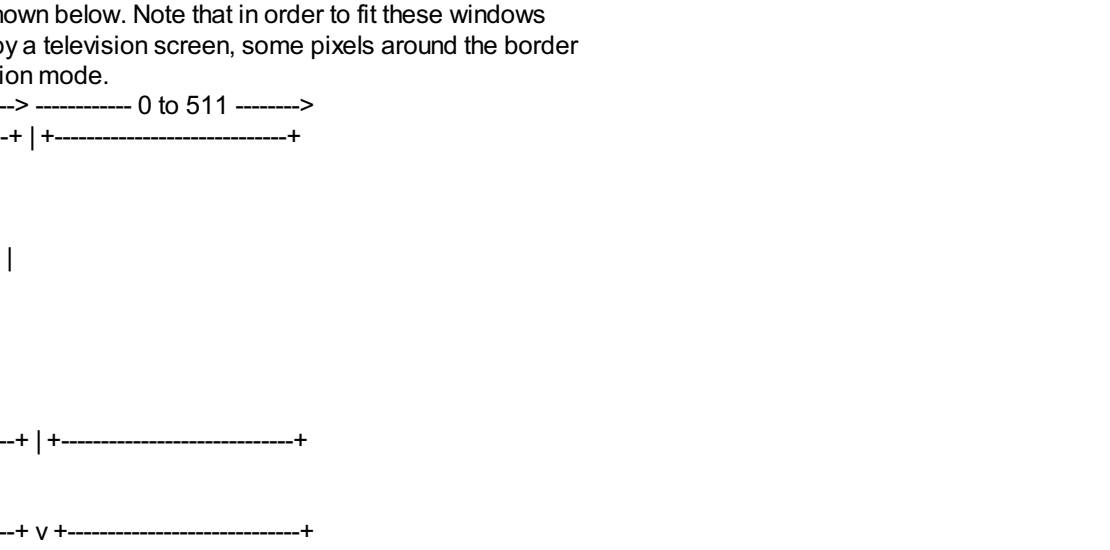
If you are using a television then not all the pixels may be visible.

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 that 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.



This windows are indentified 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 and #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 details 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
```

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 = die1 + 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

DATATYPES 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:
-615 +615
+ or - 10 to + or - 10 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)	7
		ABS(-4.3)	4.3
INT	Integer part of a floating point number	INT(2.4)	2
		INT(0.4)	0
		INT(-2.7)	-3
SQRT	Square root	SQRT(2)	1.414214
		SQRT(16)	4
		SQRT(2.6)	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 may be. 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 thanthe square root then 8/2 must be greater than it.

It follows that if we take any guess and computer 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 THE 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 th eusual mathematical operations. You may notice that they ar elike 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	6.3	
		2.1 * (-3)	-6.3	
Divide	/	7/2	3.5	Do not divide by zero
		-17/5	-3.4	
Raise to power	^	4^1.5	8	
Integer divide	DIV	-8 DIV 2	-4	Integers only
		7 DIV 2	3	Do not divide by zero
Modulus	MOD	13 MOD 5	3	
		21 MOD 7	0	
		17 MOD 8	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.

(N.B. Some of these mathematical expressions are a little hard to represent using standard ASCII notation -DJ)

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:

2
ax + bx + c = 0

(due to ASCII limitations, the above line reads: a x-squared plus bx + c = 0)

One solution in mathematical notation is:

/ 2
x = - b + /\/ b - 4ac

2a

If we start with the equation:

2
2x - 3x + 1 = 0

Example 1

The following program will find one solution.

```
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.

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 of 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

NOT

The word NOT has the obvious meaning.

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:

```
+------+ +------+ +------+
| | | | | |
lit$ | 62 | hist$ | 56 | geog$ | 71 |
| | | | | |
+------+ +------+ +------+
```

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:

```
+--------+
| |
mark$ | 625671 |
| |
+--------+
```

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.

```
: :
: string slice :
+---+---+---+---+---+---+---+---+---+---+---+---+---+---+
jumble$| A | P | Q | O | L | L | I | O | N | A | T | S | U | Z |
+---+---+---+---+---+---+---+---+---+---+---+---+---+---+
1 2 3 4 5 6 7 8 9 10 11 12 13 14
```

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"
160 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 iumble$
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
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$ = " " moves print position
110 PRINT word$
100 LET num = 13 13
110 PRINT num
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.

```
+----+----+
|CCCC|RRRR|
|CCCC|RRRR|
|CCCC|RRRR|
+----+----+
|RRRR|CCCC|
red--> |RRRR|CCCC| <--contrast
|RRRR|CCCC|
+----+----+
```

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:

```
+----+----+
|RRRR|RRRR|
|RRRR|RRRR|
|RRRR|RRRR|
+----+----+
|CCCC|CCCC|
|CCCC|CCCC|
|CCCC|CCCC|
+----+----+
```

The possible effects are shown using red [R] and contrast [C]

Code	Name	Effect
0	Single pixel of contrast	RC RR
1	Horizontal Stripes	RR CC
2	Vertical Stripes	CR CR
3	Chequerboard	CC RC

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 #6, 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_360x50a80x40
| | | |
| | | +---- down value
| | |
| | +------- across value
| |
| +---------- height
|
+------------- width
```

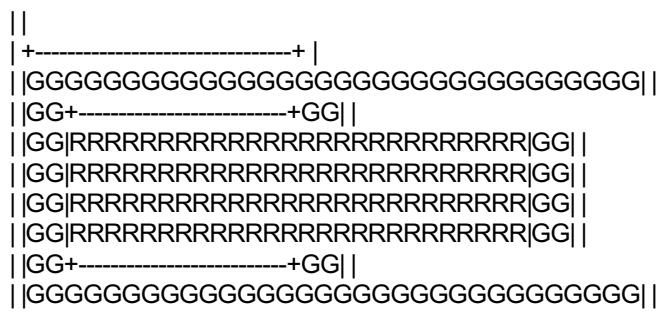
The following program creates a window with the channel number 5 and fills
it with green (code 4) and then closes it:

```
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
```

```
|
|
| +---+
| 60 across | |
| ..... +---+
| :
| :
| : 50 up
| :
+-----------------------------
```

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 (due to the limitations of ASCII characters, this is the best possible representation, or see Chapt11e_PIC):

```
|
|
```

+----------
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. (Chapt11f_PIC)

+----------
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 fith 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 (or in Chapt11g_PIC)

+----------
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_100x100a100x50
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

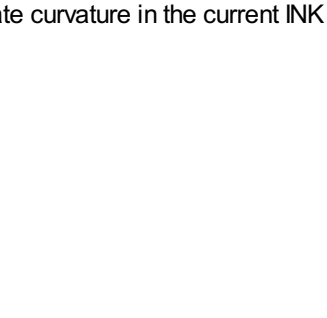
If you want to draw an arc you need to decide:

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. (Chapt12h_PIC)



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:
```
FILL0
```
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.

```
+---+---+ +---+---+ +---+---+ +---+---+ +---+---+
| | | | | | | | | | | | | | | | |
| | | | | | | | | | | | | | | | |
| | | | | | | | | | | | | | | | |
+---+---+ +---+---+ +---+---+ +---+---+ +---+---+
west(1) west(2) west(3) west(4) west(5)
```

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.

```
+---+---+ +---+---+ +---+---+ +---+---+ +---+---+
| | | | | | | | | | | | | | | | |
| 5 | 0 | | 3 | 7 | | 8 | 6 | | 4 | 1 | | 3 | 2 |
| | | | | | | | | | | | | | | | |
+---+---+ +---+---+ +---+---+ +---+---+ +---+---+
west(1) west(2) west(3) west(4) west(5)
```

The complete program is shown below:

```
100 REMark Prisoners
110 DIM west(5)
120 FOR cell 1 = 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':

```
flat$(1) flat$(2) flat$(3) ..... flat$(10)
+----------+ +----------+ +----------+ ..... +----------+
| | | | | | | |
| Tom | | Graham | | Sevvy | | Hal |
| | | | | | | |
+----------+ +----------+ +----------+ ..... +----------+
```

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$(num)
190 END DEFine
200 DEFine PROCedure exchange
210 LET temp$ = f1at$(1)
220 LET flat$(1) = f1at$(3)
230 LET flat$(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:

```
flat$(2,0) flat$(2,1) flat$(2,2) ..... flat$(2,9)
+---------+ +---------+ +---------+ ..... +---------+
| | | | | | | |
| | | | | | second(2) | |
| | | | | | | |
+---------+ +---------+ +---------+ ..... +---------+
flat$(1,0) flat$(1,1) flat$(1,2) ..... flat$(1,9)
+---------+ +---------+ +---------+ ..... +---------+
| | | | | | | |
| | | | | | first(1) | |
| | | | | | | |
+---------+ +---------+ +---------+ ..... +---------+
flat$(0,0) flat$(0,1) flat$(0,2) ..... flat$(0,9)
+---------+ +---------+ +---------+ ..... +---------+
| | | | | | | |
| | | | | | ground(0) | |
| | | | | | | |
+---------+ +---------+ +---------+ ..... +---------+
```

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 2,4 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 0 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
1 2 3 4 5 columns
+---+---+---+---+---+
1|||||
+---+---+---+---+---+
2||||XXX|
+---+---+---+---+---+
row 3|||||
+---+---+---+---+---+
4|XXX|||||
+---+---+---+---+---+
5|||||
+---+---+---+---+---+
(The squares represented by XXX above are black squares, the file
Chapt13a_pic contains a better diagram)
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 bandit
170 END REPeat bandit
```

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 EXTension 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:

```
----------> ----------> ----------->
| | |
| ----------> | |
| | | |
| | ---------- ----------
| | | |
| | Right ----------> | ----------->
| | (nested) | |
| ---------- | Right | Wrong
| | (sequence) | (Meshed)
---------- ---------- ----------
```

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 = vowels + 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 "Consonant 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:

```
North 1
|
|
|
|
West 4 ----------+---------- East 2
|
|
|
|
South 3
```

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(l 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 : 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.

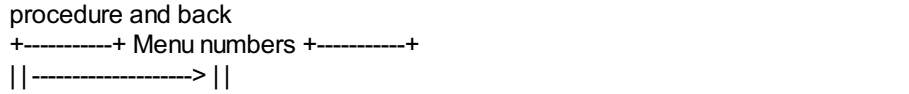
Program
```
100 REMark Variable parameter
110 LET num = 70
120 item 3,cost_1
130 item 4,cost_2
140 LET bill = num + cost_1 + cost_2
150 PRINT bill
160 DEFine PROCedure item(num,price)
```

```
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.

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.

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 whlch
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
```

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. We place the eight colours in an array "colour$" which we divide into two parts:

```
+----------------------+
| |
v |
+-----+-----+-----+-----+- - -+ +-----+-----+-----+-----+-----+
| | | ---|--> | : | | | | |
| | | | | : | | | | |
+-----+-----+-----+-----+- - -+ +-----+-----+-----+-----+-----+
SORTED PART UNSORTED PART
```

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 2 3 4 | 5 6 7 8
black blue magenta red | green cyan yellow white
sorted part | 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:

```
|
```

1 2 3 4 | 5 6 7 8
black blue magenta red | red cyan yellow white
|
^ |
|
4. We now repeat the process but this time we are comparing green with magenta and we get:
|
1 2 3 4 5 | 6 7 8
black blue magenta magenta red | cyan yellow white
|
^ |
|
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.
|
1 2 3 4 5 | 6 7 8
black blue green magenta red | cyan yellow white
|
| ^
|
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:

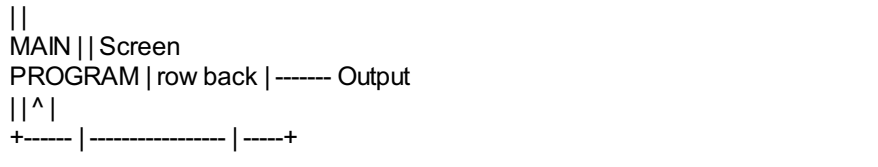
```
DELETE mdv2 phone
COPY mdv1_phone TO mdv2_phone
COPY mdv1_phone TO scr
DELETE mdv1_phone_temp
```

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.

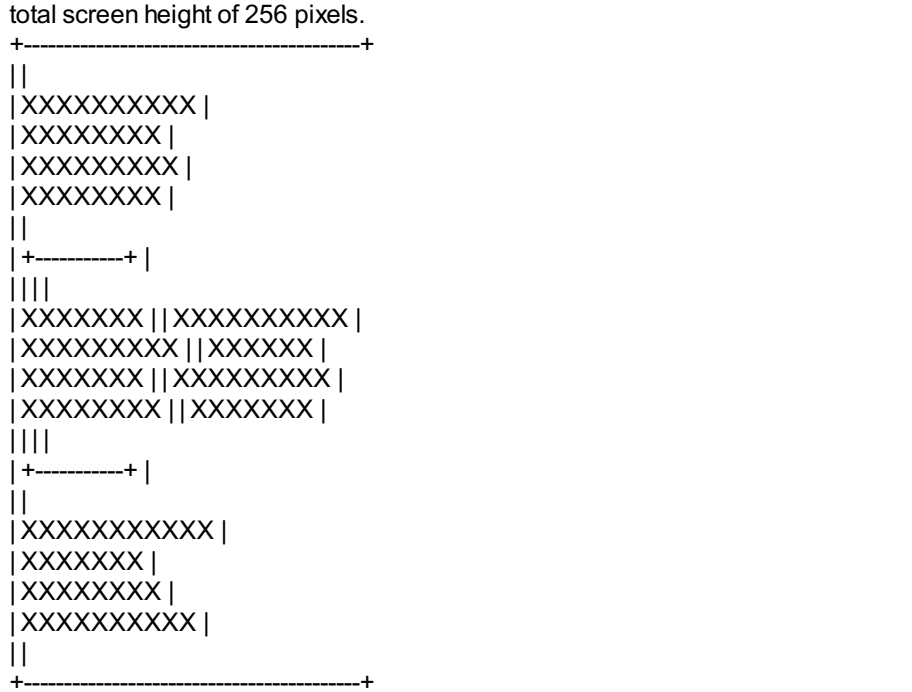
You now have enough understanding of techniques and syntax to handle a more complex screen layout. Suppose you wish to represent the hands of four card players. A hand can be represented by something like:

```
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:

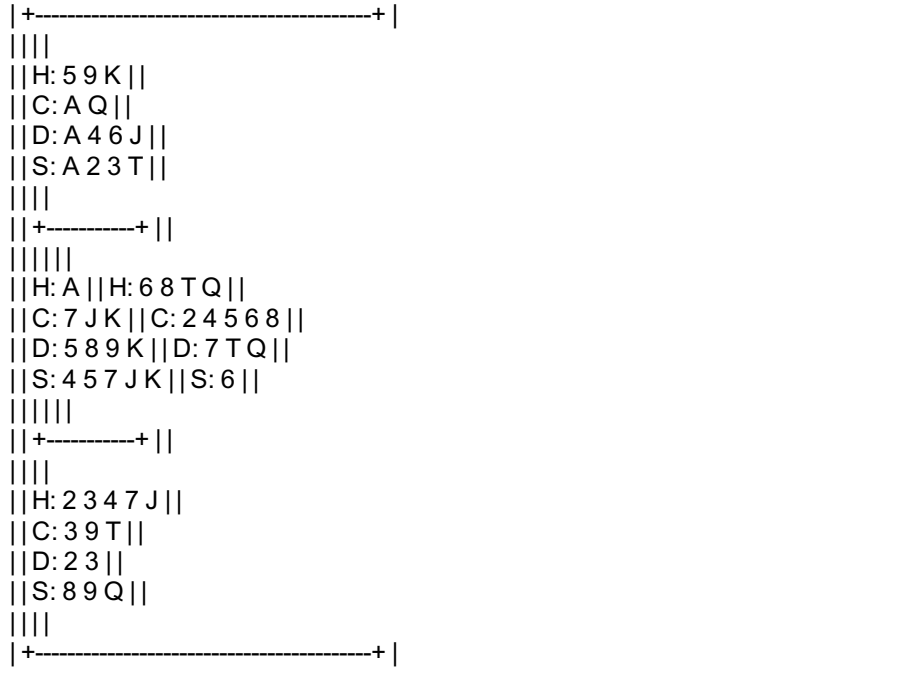
west hand + table width + east hand

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
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 #6,1in$
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 <- (CTRL left arrow) 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 [ENTER] for the ENTER key
8. CLS [ENTER] causes part of the screen to be cleared.
9. RUN [ENTER] causes a stored program to be executed.
10. LIST [ENTER] causes a stored program to be displayed on the screen.
11. NEW [ENTER] 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.

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 [ENTER] (1 point)
A READ statement which uses a name for the first time will have the same effect, for example:
READ count [ENTER] (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 [ENTER] 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.
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.
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.

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:= nunieric_expression {in radians}
ACOS (angle)
ACOT (angle)
ASIN (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 are 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 are 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 are from 15,10 to 40,40 turning through PI/2 radians}
ii. ARC TO 50,50,PI/2
{draw an are 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 are, starting 10,10 from the last point plotted to 55,45 from the start of the are, 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 O, column O) 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_expressian {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 {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(O 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+l, bottom-height+l,colour
190 END FOR bar

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
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, AO 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 #O, DO
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$(numeric_expressen)
example: i. PRINT CHRS(27) {print ASCII escape character}
ii. PRINT CHR$(65) {print A}
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_expession
radius:= numeric_expression
eccentricity:= numeric_expression
angle:= numeric_expression {range 2 to PI}
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 = O - 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. mdvl
is a directory device; serl 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 mdvl_data_file TO con_ {copy to default window}
ii. COPY neti_3 TO mdvl_data {copy data from network station to mdv_data.}
iii. COPY_N mdvl_test_data TO ser1_ {copy mdvl_test_data to serial port 1 removing header information}
COT
COS math 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 C0S(3.141592654/2)
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 C0T(3.141592654/2)
CSIZE window
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·
6 pixels
8 pixels
12 pixels
16 pixels
10 pixels
20 pixels
syntax: width:= numeric_expression {range 0..3}
height:= numeric_expression {range 0..11}
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$(l2,9)
110 RESTORE
120 REMark Data statement example
130 FOR count=l TO 12 :
READ month$(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.
DATE$
DATE clock
DATE$ is a function which will return the date and time contained in the QLls 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 {forma_parameters}
[LOCal identifier x[, 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_parameter:= (expression *[, expression]*)

actual_parameters:= expression *[, expression]*
DEFine PROCedure identifier {forma_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 O : 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
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 math 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/Z) {will print 90}
DELETE micrdrives
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 mdvl_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 x[, 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,Z) {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 Microdrives 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 = O
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 x[: 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>lE20 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.

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

Comment: The control structures available in SuperBASIC make the GOSUB statement redundant.

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
In all three forms of the IF statement the THEN is optional. In the short form it must comment 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.
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 windows 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 forever. 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}
examples:
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 ISHIFT(CTRL I ALT I X I V I N
8121610 EIOITIU
41L5
WII ITABIRI-Iv
31HIIIAIPIDIJ
CAPS
LOCK
KISIFI=IG
ZI .IC B f M
ENTER I t up ESC I I ISPACE down
F4 I 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 ,startaddress
example:
i. LBYTES mdvl_screen, 131072 {load a screen image}
ii. LBYTES mdvl_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 ength 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
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 XO 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}
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 "mdvl_test_program"
ii. LOAD mdvl_guess
iii. LOAD neti_3
iv. LOAD serl_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 (numenic_expression) {range greater than zero}
LN (numeric_expression) {range greater than zero}
example: i. PRINT LOG10(20)
ii. PRINT LN(3.141592654)
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
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 "sti LI 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
If NEXT is used inside a REPeat - END REPeat construct it will force processing to continue at the statement following the matching REPeat statement.
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_expressicn
OPEN channel, device
example: i. OPEN #5, f_name$
ii. OPEN_IN #9,"mdv1_filename" {open file mdvl_file__name}
iii. OPEN_NEW #7,mdv1_datafile {open file mdvl_datafile}
iv. OPEN #6,con_10x20a20x2032
{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..i]
OVER [channel,] switch
where switch = 0 - print ink on strip
switch = 1 - print in ink on transparent stnp
switch = -1 - XORs the data on the screen
example: i. OVER 1 {set "overprinting")
ii. 10 REMark Shadow Writing
11 PAPER 7 : INK O : 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}
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 tie. 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 cent rest
18 END FOR colour
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
PEEICW
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 function
PI is a function which returns the value of x.
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
POKL_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 O}
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"
! - 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_expressio
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 O 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, cl, 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.
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: startline:= 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];] [startnumber] [,step]

example: i. RENUM {renumber whole program from 100 by 10}
ii. RENUM 100 TO 200 {renumber from 100 to 200 by 10}
Comment: 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}
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 them RETurn statement is used to return the function's value.
syntax: RETern [expression]
example:
i.
```
100 PRINT ack (3,3)
110 DEFine FuNction ack(m,n)
120 IF m=0 THEN RETurn n+l
130 IF n=0 THEN RETurn ack (m-l,l)
140 RETern a c k (m-l ,a c k (m, n-l ) )
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
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 ieturn from a GOSUB.
RND maths function
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}
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 }
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_screendata,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 mdvl_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.l}
ii. SCALE 10,0,0 {set scale to 10 with the origin at 0,0}
iii. SCALE 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 specifiedto 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 QCs clock to be reset.
syntax: year:= numeric_expression
month:= numeric_expression
day:= numeric_expression
hours:= numenc_express,on
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 selectvariable. 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-', 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 selectvariable] = 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

The Qdos system documentation should be read before attempting to use this command.

SIN maths function

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 function

will compute the square root of the specified argument. The argument must be greater maths functions than or equal to zero.

syntax: SORT (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 BASIC 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.

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 PI/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 pointing at 0 degrees, 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 degrees}
ii. TURNTO 0 {turn to heading 0 degrees}
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 oi default channel.
syntax: switch:= numeric_expression {range 0..1}
UNDER [channel,] switch
example: i. UNDER 1 {underlining on}
ii. UNDER 0 {underlining off}
WIDTH
WINDOW windows
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}

QL CONCEPTS MANUAL
==========
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 limits to 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:
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
1,3
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.

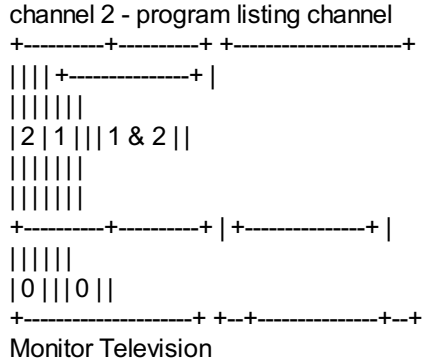
LPRINT use PRINT #
LLIST use LIST #
VAL not required in SuperBASIC
STR$ not required in SuperBASIC
IN not applicable to 68008 processor
OUT not applicable to 68008 processor

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]
keys
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 activated,
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
concepts DEVICE and PERIPHERAL EXPANSION.
Data can be output to a channel by PRINTing 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

CHARACTER SET AND KEYS

The cursorncontrols 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 |) 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
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 / /
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
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 €
129 81 CTRL SHIFT 1
130 82 CTRL SHIFT ' ,
131 83 CTRL SHIFT 3 ƒ
132 84 CTRL SHIFT 4 „
133 85 CTRL SHIFT 5 …
134 86 CTRL SHIFT 7 †
135 87 CTRL ' ‡
136 88 CTRL SHIFT 9 ˆ
137 89 CTRL SHIFT 0 ‰
138 8A CTRL SHIFT 8 Š
139 8B CTRL SHIFT = ‹
140 8C CTRL , Œ
141 8D CTRL _
142 8E CTRL .
143 8F CTRL /
144 90 CTRL 0
145 91 CTRL 1 ‘
146 92 CTRL 2 ’
147 93 CTRL 3 “
148 94 CTRL 4 ”
149 95 CTRL 5 ·
150 96 CTRL 6 –
151 97 CTRL 7 —
152 98 CTRL 8 ˜
153 99 CTRL 9 ™
154 9A CTRL SHIFT ; š
155 9B CTRL ; ›
156 9C CTRL SHIFT , œ
157 9D CTRL =
158 9E CTRL SHIFT .
159 9F CTRL SHIFT / Ÿ
160 A0 CTRL SHIFT 2
161 A1 CTRL SHIFT A ¡
162 A2 CTRL SHIFT B ¢
163 A3 CTRL SHIFT C £
164 A4 CTRL SHIFT D ¤
165 A5 CTRL SHIFT E ¥
166 A6 CTRL SHIFT F ¦
167 A7 CTRL SHIFT G §
168 A8 CTRL SHIFT H ¨
169 A9 CTRL SHIFT I ©
170 AA CTRL SHIFT J ª
171 AB CTRL SHIFT K «
172 AC CTRL SHIFT L ¬
173 AD CTRL SHIFT M
174 AE CTRL SHIFT N ®
175 AF CTRL SHIFT O ¯
176 B0 CTRL SHIFT P °
177 B1 CTRL SHIFT Q ±
178 B2 CTRL SHIFT R ²
179 B3 CTRL SHIFT S ³
180 B4 CTRL SHIFT T ´
181 B5 CTRL SHIFT U µ
182 B6 CTRL SHIFT V ¶
183 B7 CTRL SHIFT W ·
184 B8 CTRL SHIFT X ¸
185 B9 CTRL SHIFT Y ¹
186 BA CTRL SHIFT Z º
187 BB CTRL [»
188 BC CTRL \ ¼
189 BD CTRL] ½
190 BE CTRL SHIFT 6 ¾
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 right
209 D1 ALT Up Scroll up
210 D2 CTRL Up Search backward
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 CAPS LOCK Toggle CAPS LOCK function
225 E1 ALT CAPS LOCK
226 E2 CTRL CAPS LOCK
227 E3 ALT CTRL CAPS LOCK
228 E4 SHIFT CAPS LOCK
229 E5 SHIFT ALT CAPS LOCK
230 E6 SHIFT CTRL CAPS LOCK
231 E7 SHIFT CTRL ALT CAPS LOCK
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
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 CAPS LOCK generates the code FF, followed by a byte indicating what the keycode would have been if ALT had not been depressed.

Note that CAPS LOCK 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

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

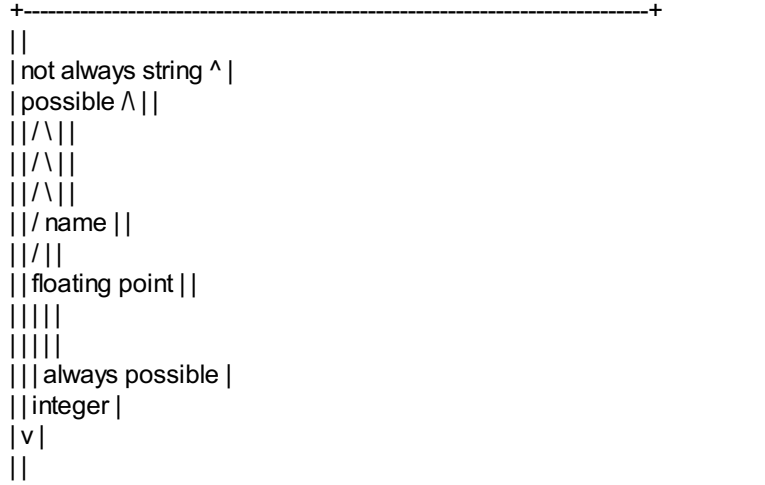
COERCION

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 natural ordering of data types on the QL, see figure below. String is the most general type since it 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)

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.
i. Single:
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.

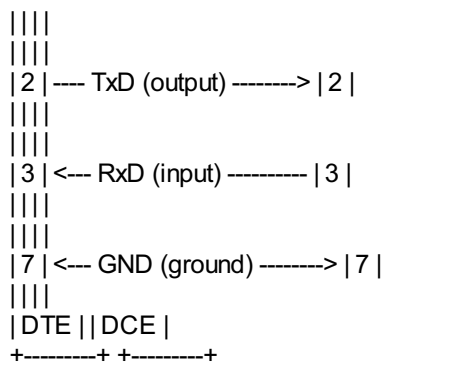
bit 7 6 5 4 3 2 1 0
By specifying only the bottom three bits (i.e. the required colour) no stipple will be requested and a single solid colour will be used for display.
ii. Double
colour: = background, contrast
The colour is a stipple of the two specified colours. The default checkerboard stipple is assumed (stipple 3)
iii. 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 background 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:
XO XX OX OX
XX OO OX XO
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.
COMMUNICATIONS 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 configurd 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		SER2	
pin	name	function	pin	name	function
1	GND	signal ground	1	GND	signal ground
2	TxD	input	2	TxD	output
3	RxD	output	3	RxD	input
4	DTR	ready input	4	DTR	ready output
5	CTS	ready output	5	CTS	ready input
6	+12V		6	+12V	

TxD Transmit Data DTR Data Terminal Ready
RxD Receive Data 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 one and a half stop bits are required.
The 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 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 FROMT HE 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 +/- (10^-615 to 10^615), with 8 signiflcant 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"
Type name has the same form as a standard SuperBASIC identifier and is used by the name 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
---------- Select Console Device
|
| ---------- Underscore
| |
| | ---------- Window Width
| ||
| || ---------- Separator
| |||
| ||| --------- Height
| ||||
| |||| -------- Separator - read as AT
| |||||
| ||||| ------- Window X coordinate
| ||||||
| |||||| ------ Separator
| |||||||
| ||||||| ----- Window Y coordinate
| ||||||||
| ||||||| ---- Separator
| |||||||||
| ||||||||| --- length of keyboard type ahead buffer
| ||||||||||
| |||||||||
con_wXhaxXy_k
CON wXhaxXy_k Console I/O
|wXh| - window, wldth, 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
SCR_wXhaxXy Screen Output
[wXh] - window, width, height

[AxXy] - window X, Y coordinate
default: scr_448x180a32x16
xample: PEN #4, scr _0x10a20x50
OPEN #5, scr_10x10
SERnphz Serial (RS-232-C)
n port number (1 or 2)
[p] parity [h] handshaking [z] protocol
e - even i - ignore r - raw data no EOF
o - odd h - handshake z - control Z is EOF
m - mark c - as z but converts
s - space ASCII 10 (Qdos
newline character)
to ASCII 13
<CR>)
default: ser1rh (8 bit no parity with handshake)
example: OPEN #3, serle
OPEN #4, serc
COPY mdv1_test_file TO ser1c
NETd_s Serial Network I/O
[d] indicates direction [s] station number
i - input 0 - for broadcast
o - output own station - for general listen
(input only)
default: no default
example: OPEN #7, neti_32
OPEN #4, neto_0
COPY ser1 TO neto_21
MDVn_name Microdrive File Access
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 nurnber. 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
(2O) 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 nextstatement 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
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.

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 te 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

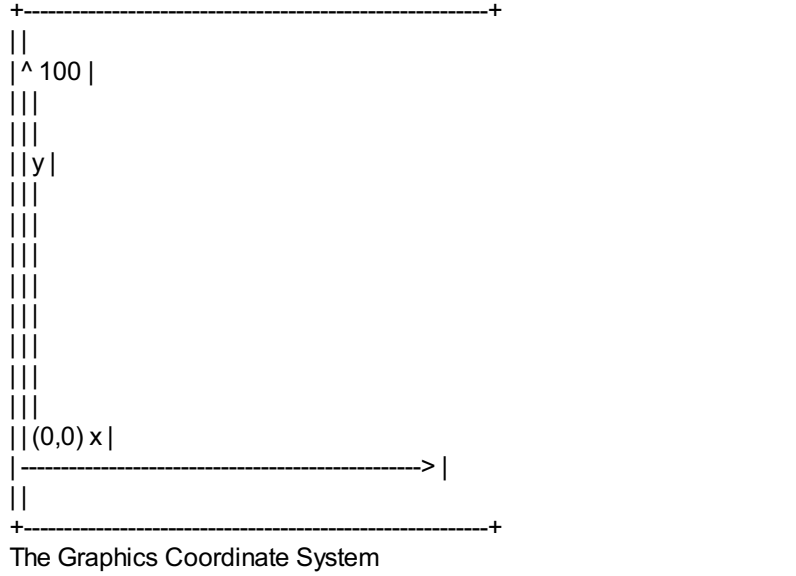
assigning data to the corresponding formal parameter will set the variable specified as the actual parameter.
Variables can be defined to be local to a function or procedure with the LOCal statement. Local variables have no effect on similarly named variables outside the function or procedure in which they are defned 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 screen mode will change the shape of the pixels. Thus if the grapics 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 the 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.

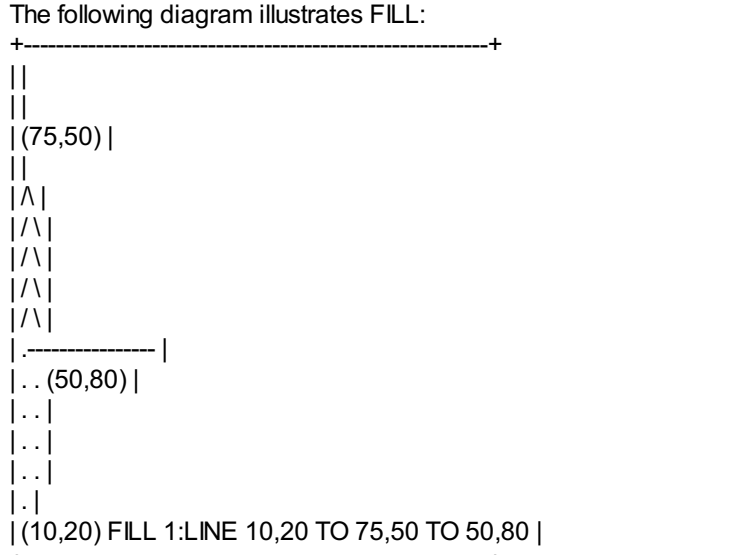
Command Function

CIRCLE draw an ellipse or a circle }
LINE draw a line } absolute
ARC draw an arc of a circle }
POINT plot a point }

CIRCLE_R draw an ellipse or a circle }
LINE_R draw a line }
ARC_R draw an arc of a circle } relative
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.
IDENTIFIER

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
moe	key key
up	cursor up F4
down	cursor down F2
left	cursor left F1
right	cursor right F3
fire	space F5

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 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

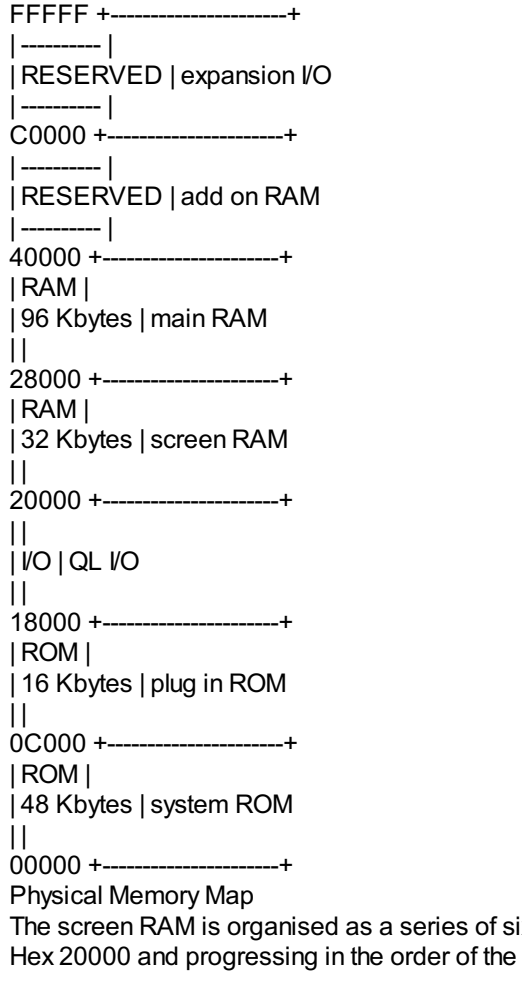
SuperBASIC has the standard trigonometrical and mathematical functions.

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 pi ±
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 eachbword 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	low byte	mode
A0=0	A0=1	
GGGGGGGG	RRRRRRRR	512 mode (high res)
GFGFGFGF	RBRBRBRB	256 mode (low res)

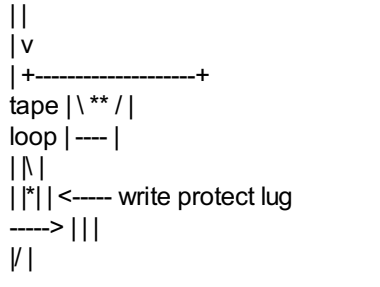
G-Green B-Blue R-Red F-Flash
Setting the Flash bit toggles the flash state and freezes the background colour for the flash to the value given by R, G and B for that pixel. Flashing is always reset at the beginning of each display line.
In high resolution mode, red and green specified together is interpreted 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. Pokng 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.

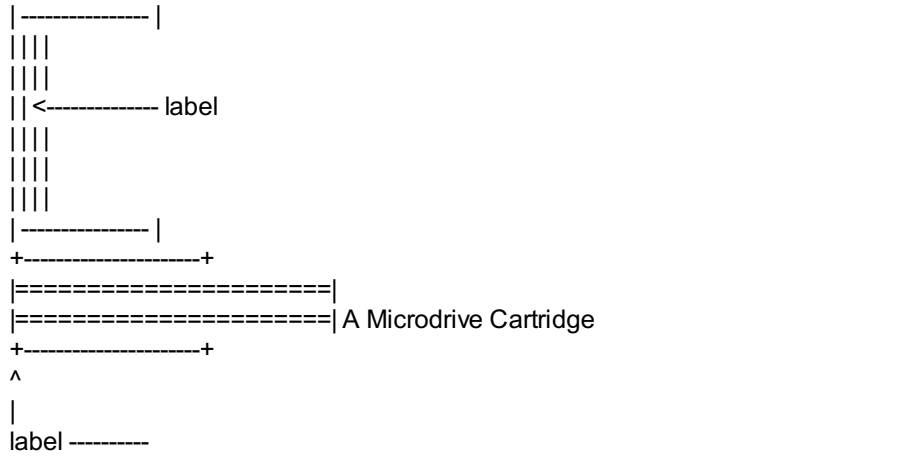
MICRODRIVES

Microdrives provide the main permanent storage on the QL. Each Microdrive cartridge has a capacity of at least 100Kbytes. 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 be removing the small lug on the right hand sie.
On receiving new blank 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 7.5 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 tocuh the tape with your fingers for this or any reason.





Command Function

FORMAT prepare a new cartridge for use
DELETE delete a file from a cartridge
DIR list the files on a cartridge
SAVE
SBYTES saves data from a cartridge
SEXEC
LOAD
LBYTES
EXEC loads data from a cartridge
MERGE
OPEN_IN
OPEN_NEW
OPEN opens and closes files
CLOSE
PRINT
INPUT SuperBASIC file I/O
INKEY$

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.

MONITOR

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.

---

sleeve colour pin	function	signal on QL RGB colour lead
1	PAL composite	PAL (4) orange
2	GND	ground green
3	VIDEO	composite monochrome 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 an 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 available from audio shops.

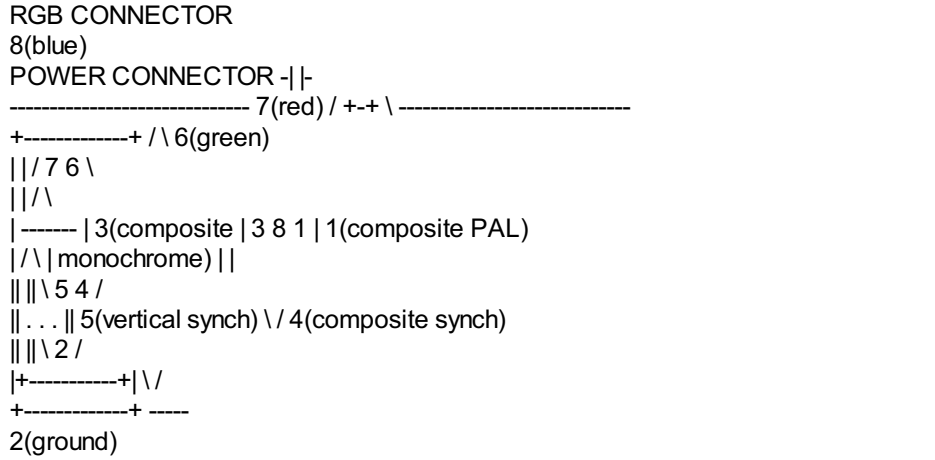


Diagram of Monitor Connector as viewed from rear of QL, showing pin numbers and functions.

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 VCRs may work witht he 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 2 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 receving 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) s 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	open a network channel
CLOSE	close a network channel
PRINT	
INPUT	network I/O
INKEY$	
LOAD	
SAVE	
LBYTES	
SBYTES	
EXEC	load and save via network
SEXEC	
LRUN	
MRUN	
MERGE	

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 lads 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

**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.

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 | RESEYCPUL
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' appended 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 Functional

SP0..SP3 Select peripheral 0 to 3

VIN 9V DC (nominal) - 500mA max.
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 allocate 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.

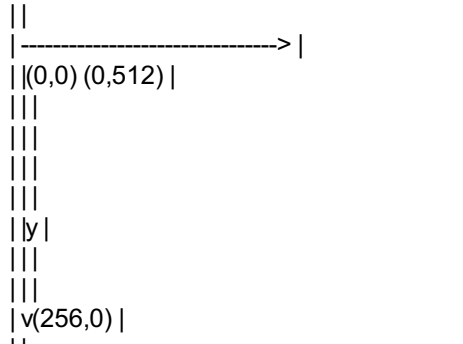
QDOS includes facilities for queue management and simple serial I/O which may be of use when writing device drivers.

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 defien 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 REMark 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:
SV_RAMT-1
+--------------------+ QDOS MEMORY MAP
Resident	
Procedures	
SV_RESPR | | v fills
+--------------------+
Transient	
Programs	
SV_TRNSP | | v fills
+--------------------+
SuperBASIC command	
interpreter data	
and	
SV_BASIC | SuperBASIC programs | v fills
+--------------------+
| |
Filing subsystem
slave block
SV_FREE | |
+--------------------+
| | ^
| Channels and other | |
| heap items | |
SV_HEAP | | | fills
+--------------------+
| |
| System tables |
| and |
| System Variables | 28000 Hex
+--------------------+
| |
| Display memory |
| |
| |
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 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 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 environemtn 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 cannot be written to. QL devices are processed by the SERIAL I/O SYSTEM. As its name suggests any data output by 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 runnign 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 rel-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
terms 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 folliwng the NEXT.
Prcoessing 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:

```
FOR identifier = for_list
statements <-------------| exit--
NEXT identifier --next-------| |
epilogue |
END FOR identifier <--------------------
```

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:

```
REPeat identifier <-----------------
statements |
IF condition THEN NEXT identifier ----
epilogue
```

END REPeat identifier
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 Cartrdge 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
+----------+
_ | a 1 b | VDD
A12 | a 2 b | A14
A7 | a 3 b | A13
A6 | a 4 b | A8
A5 | a 5 b | A9
SLOT | a 6 b | SLOT
A4 | a 7 b | A11
A3 | a 8 b | ROMOEH
A2 | a 9 b | A10
A1 | a 10 b | A15
A0 | a 11 b | D7
D0 | a 12 b | D6
D1 | a 13 b | D5
D2 | a 14 b | D4
GND | a 15 b | D3
+----------+
Side b is the upper side of the connector; side a is the lower.

Signal Function

A0..A15 Address lines
D0..D7 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 and white can be displayed in this mode.
256 MODE
Low resolution mode also has a hardware flash. The screen is 256 pixels across and 256 pixels deep. The full set of solid colours is available in this mode:
black, blue, red, magenta, green, cyan, yellow and 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$, MID$, RIGHTS. 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)	RIGHTS (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.

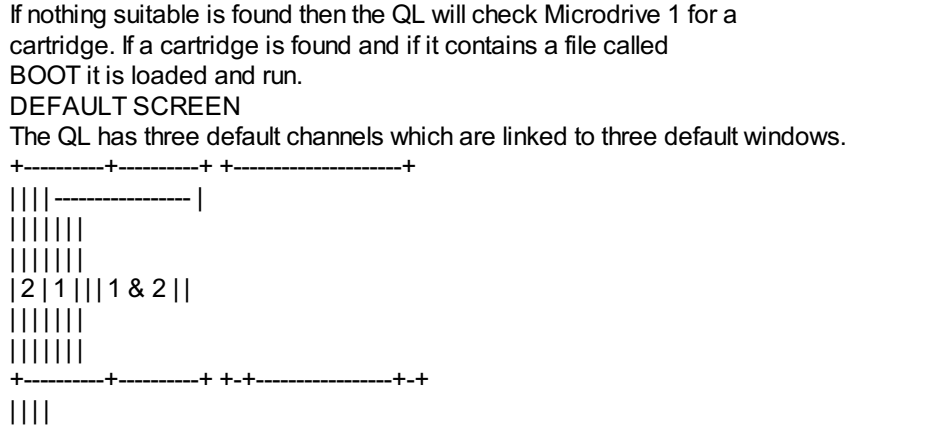
Press F1 if you are using a monitor and F2 if you are using a television set.

The QL has the ability to 'boot' itself up from programs contained in either the ROM cartridge slot or in Microdrive 1. 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.



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

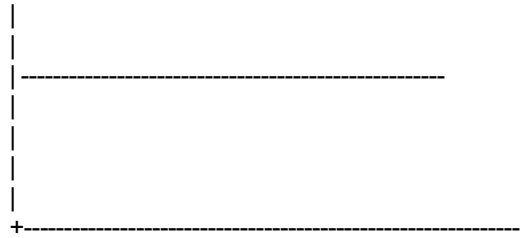
if any fuzziness should be built into the sound. i.e. deviations on every cycle of the sound

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

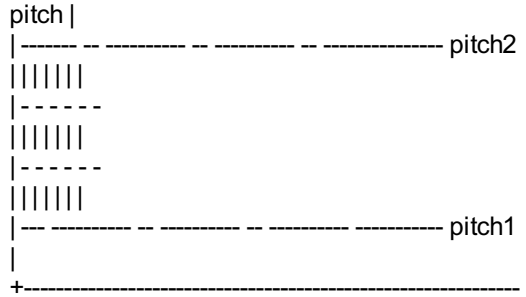


This is the simplest sound command, other than the command to stop the sound, on the QL.

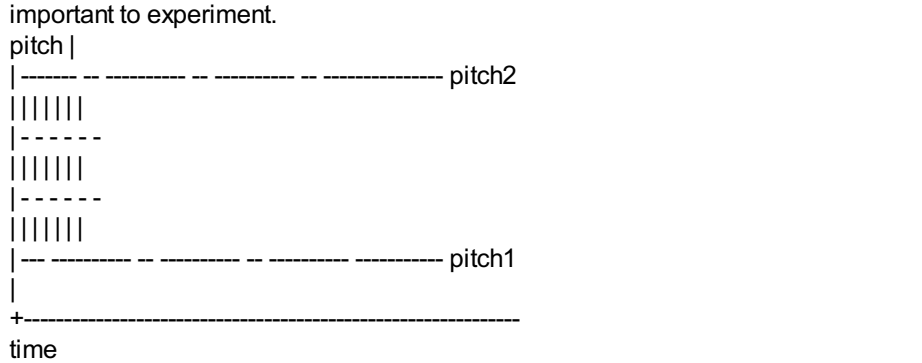
LEVEL 2

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.

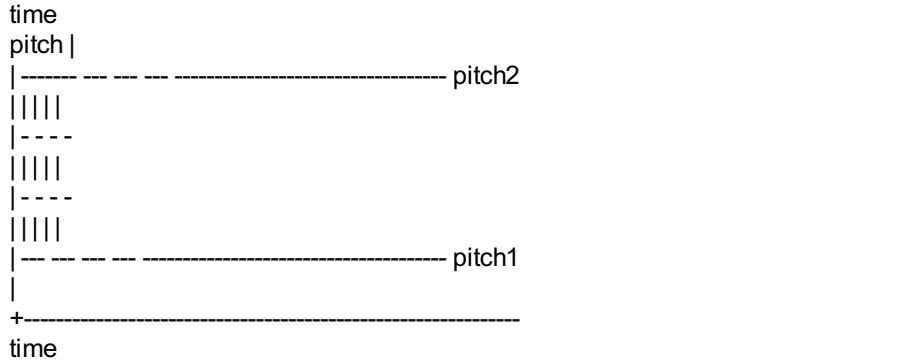
The sounds produced at this level can vary between: semi musical beeps, growls, zaps and moans. It is best to experiment.



time
LEVEL3
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.



LEVEL4
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

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

type 0 not normally used by the SuperBASIC system.

USAGE

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
TURTLE GRAPHICS

SuperBASIC has a set of turtle graphics commands:

Command Function

PENUP stop drawing
PENDOWN start drawing
MOVE move the turtle
TURN turn the turtle
TURNTO turn to a specific heading

The set of commands is the minimum and normally would be used within another
procedure 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
```
These will define some of the more famous turtle graphic commands.
Initially the turtle's pen is up and the turtle is pointing at 0 degrees
which is to the right hand side of the window.
The FILL command will also work with figures drawn with turtle graphics. Also
ordinary graphics and turtle graphics can be mixed, although the direction of
the turtle is not modified by the ordinary graphics commands.
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