

QL SuperBASIC

THE DEFINITIVE HANDBOOK

Jan Jones

designer and writer of Sinclair's
QL SuperBASIC language



QUANTA REPRINT LIMITED EDITION

The image features a large, stylized graphic of the letters 'Q' and 'L'. The 'Q' is formed by multiple concentric, slightly irregular yellow and red rings, giving it a sense of depth and movement. The 'L' is composed of vertical yellow and red bars, also with a layered, 3D effect. A black diagonal banner with white text cuts across the middle of the 'Q' and 'L'.

QL SuperBASIC – The Definitive Handbook

Jan Jones

SECTOR SOFTWARE
39 Wray Crescent, Ulnes Walton,
Leyland, Lancashire PR5 3NH
Tel: (0772) 454328
Fax: (0772) 454480

QUANTA – The Independent QL User Group

First printed by McGraw-Hill Book Company (UK) Limited 1985.
This QUANTA LIMITED EDITION was produced in response to an
overwhelming demand by our members. We would like to express our
gratitude to JAN JONES for allowing us to reprint the book.

Published by

QUANTA — The Independent QL User Group
15 Grosvenor Crescent GRIMSBY South Humberside England

British Library Cataloguing in Publication Data

Jones, Jan

QL SuperBASIC : the definitive handbook.

1. Sinclair QL (Computer) Programming
2. SuperBASIC (Computer program language)

I. Title

001.64'24 QA76.8.S625

ISBN 0-07-084784-3

Library of Congress Cataloging in Publication Data

Jones, Jan

QL SuperBASIC.

Includes index

1. Sinclair QL (Computer) — Programming.
2. BASIC (Computer program language)

I. Title. II. Title: QL SuperBASIC. III. Title: QL SuperBASIC.

QA76.8.S6216J66 1985 001.64'2 84-26173

ISBN 0-07-084784-3

This QUANTA LIMITED EDITION printed in July 1989

Copyright 1989 QUANTA . All rights reserved. No part of this publication may be reproduced, stored in a retrieval system, or transmitted in any form or by any means, electronic, mechanical, photocopying, recording, or otherwise without prior written permission of QUANTA , or of the original copyright holder.

Preface to the 2nd Edition 19th July 1989

It is now five years, two children and one extension since I wrote this book. These days it would be more appropriate for me to be writing a set of short snappy articles on DIY for the under-5's or the dichotomy of nursery hygiene in a partly built kitchen.

Looking at the Introduction with hindsight, it strikes me as a bit naive, valiantly high-flying. However, surely that is the spirit in which all new languages ought to be conceived and I am still of much the same opinion regarding SuperBASIC as I was then.

Given a few more years, a few more hours in the day and a few less children, I'm not sure that I'd make that many changes....

Jan Jones

Copyright Notice

As of December 1987 all rights in this book reverted to Jan Jones. No portion of this book may be reproduced without her written permission.



CONTENTS

1. Introduction 1

Or, why we did it.

2. Notes 4

1. on the syntax of the definitions
2. on the examples
3. on the internal storage units

3. Procedures, Methods and Functions 7

What procedures are. How they are defined. How they are used

– DEFine PROCedure – RETurn – END DEFine –

SuperBASIC stages. Parsing and execution

The return stack, where procedures return to

Functions, definition of, return value, use of

– DEFine FuNction – RETurn expression – END DEFine –

Ways of commenting programs, dividing code up

– REMark –

4. The Nametable 15

How names are stored, different types of names

Simple variables, assignment, the variable values area, storage space used

– LET –

Procedure, function names, description of nametable entry. The namepass. Procedure arguments, how they work, nametable entry swapping, temporary entries

Local variables, temporary entries swapped with global ones

– LOCAl –

Arrays, indexed variables, array descriptors, size and storage

– DIM –

Sub-arrays, assigning to numeric arrays, substrings of string

arrays, assigning to string arrays, substrings of string variables

length of a string, size of an index

– LEN – DIMN –

Arrays as parameters

Local arrays

5. The Expression Evaluator 33

What is an expression? Component parts, order of precedence and functions of operators. Intermediate storage, internal representation. Legal constants. Sub-expressions, functions, coercion

Mathematical functions, random number selection, string functions, ASCII conversion

6. Control Structures 43

Simple and complex clauses. Advantages and limitations of both. Simple controlled transfer, method of choice

- IF - THEN - ELSE - END IF -

Multiple controlled transfer, method of choice

- SElect - ON - END SElect -

Uncontrolled transfer

- GO TO - GO SUB - ON .. GO TO - ON .. GO SUB -

7. Loops 51

Continuous loops and exiting from them. The REPEAT index variable and its associated information. Simple or multi-line loops

- REPEAT - NEXT - EXIT - END REPEAT -

Controlled looping. FOR ranges. The FOR index variable and storage of its associated information. Loop epilogue

- FOR - NEXT - EXIT - END FOR -

8. Data Handling, Input, Output and the File System 58

Simple INPUT and PRINT for interactive work, separators, printing arrays. Other forms of simple input

- INKEY\$ - KEYROW - PAUSE -

Constant data held with program

- READ - DATA - RESTORE -

Channels, file system, reading and writing to files, formatting Microdrive media, copying files, deleting files, end of file function

- OPEN - CLOSE - DIR - COPY - DELETE - FORMAT - EOF -

Peripheral devices, defining and using SERIAL ports, CONSOLE windows, SCREEN windows, the NETWORK

- WIDTH - BAUD -

9. Creating and Running Programs 80

Editing a line. Bad line during parsing. Deleting lines one at a time or DLINE. Using the AUTO and EDIT facility. LISTING all or part

of programs. RENUMbering lines. Saving programs onto Microdrive media. Restoring programs from Microdrives. Clearing out programs

- SAVE - LOAD - MERGE - CLEAR - NEW -

Running programs, stopping execution, continuing after any program halt. Error recovery

- RUN - LRUN - MRUN - STOP - CONTINUE - RETRY -

10. Screen Control 94

High resolution, low resolution. Display modes, default windows. Pixels on screens, defining windows, overlaying windows

- MODE - WINDOW - CLS -

Colours, bit patterns, composite colour. Colouring a window, overwriting, window areas

- INK - PAPER - STRIP - AT - OVER -

Defining a border, pixel reference inside window

- BORDER -

Blocks of colour, recolouring

- BLOCK - RECOL -

Changing cursor position by pixel, changing character size, flashing text, underlined text

- CURSOR - CSIZE - FLASH - UNDER -

How to restore default windows without resetting

11. Graphics 114

Units of reference. Scaling

- SCALE -

Drawing lines, current graphics cursor, drawing dots, off-window drawing

- LINE - POINT -

Drawing curves; arcs, circles, ellipses

- ARC - CIRCLE - ELLIPSE -

Annotating a graphics diagram

- CURSOR -

Drawing relative to current position

- LINE_R - POINT_R - ARC_R - CIRCLE_R - ELLIPSE_R -

Filling shapes with colour, non-reentrance

- FILL -

Turtle graphics

- PENDOWN - PENUP - MOVE - TURN - TURNT0 -

Animation

12. Machine Code and Memory Access 137

Loading and saving hexadecimal data

- LBYTES - SBYTES -

Usage of the resident procedure area

- CALL - RESPR -

Executing independent machine code programs

- EXEC - EXEC_W - SEEXEC -

Accessing memory directly

- PEEK - POKE -

13. The Calendar and Clock 142

Setting the clock, adjusting the clock, reading the clock, clock time in seconds, day of week

- SDATE - ADATE - DATE\$ - DATE - DAY\$ -

14. The QL Sound 146

Range of sounds. An explanation of the BEEP parameters. Menu driven BEEP exerciser

- BEEP - BEEPING -

15. The Syntax Graphs 152

The 'railway lines' definition of SuperBASIC

16. The Keywords 165

Rapid guide to all restricted names in the SuperBASIC system

APPENDICES 184

- A. The character set, ASCII codes, hex conversion
- B. Errors and what they mean
- C. SuperBASIC tokens
- D. SuperBASIC storage area
- E. Memory map

INDEX 208

A comprehensive guide to all topics in

1 INTRODUCTION

SuperBASIC is a language designed by programmers for programmers.

The aim of this book is to explain the structure of the language, the principal commands and the reasons behind the design.

WHY A NEW LANGUAGE?

My colleague, Tony Tebby, and I are both experienced programmers. In no one language have we ever found everything to satisfy us. We wanted something easy to use yet with powerful constructions. We didn't want the efficiency to decline with each new variable or each added line. We wanted structures which were elegant and consistent, but also functional. We wanted to be able to implement this language on a micro!

For a machine as innovative and powerful as the Sinclair QL, we decided that we would have to design the language ourselves, so we have incorporated all the features that we feel to be necessary to the average programmer.

SuperBASIC is a total design. Each element has been carefully thought out and merges with the others. None of the features is, I believe, entirely new, though our interpretation and implementation of them certainly may be.

It is obvious that different applications need different tools. While one could use a screwdriver to hammer in a nail, it is not the ideal solution nor is SuperBASIC intended to be all things to all men. It does, however, go a considerable way along the all-purpose road.

WHY A NEW BOOK?

As the writer of SuperBASIC, I am dissatisfied with other texts which give you (often incorrectly) a bald recital of what you can and can't do in SuperBASIC. I feel that it is due to the user to know **why** certain things work in the way that they do and how the information is built up internally. In **my** book at least, the facts will be correct; as I wrote most of the SuperBASIC system, you can be sure of that! There is also a

comprehensive index at the back so that you should be able to find out immediately where any topic is discussed.

WHO SHOULD READ THE 'QL SUPERBASIC HANDBOOK'?

Any competent programmer will find this handbook useful. I won't pretend that it is for the complete novice, though I have attempted to explain the 'computer jargon' as much as possible. However, even a beginner is unlikely to be a beginner for long, and once the initial complications of coming to terms with a computer have been resolved (using the 'Beginners' Guide' supplied with the QL), then this handbook will be invaluable.

HOW TO READ THIS BOOK

Strange concept ? No, not really. I suggest that you make the time to read this handbook properly. It should be read from cover to cover, preferably forwards, since the contents have been designed to follow on from one another in a natural sequence. As far as possible, I have tried not to refer to any topic unless I have previously explained it in detail. Thus certain parts of Chapter 9 assume a familiarity with Chapter 8.

Perhaps it is unreasonable of me to expect that you read the book in this way, but I do feel that having bought (or if you are considering buying) the QL and this book, you owe it to yourself to peruse the contents carefully.

Of course, once read, 'QL SuperBASIC - The Definitive Handbook' will be invaluable as a reference guide; to look up the exact syntax of a command or to find out why a construction appears not to have worked quite in the way that you thought it would.

OTHER USEFUL BOOKS

For machine code programming and interfacing such programs to SuperBASIC, 'QL Assembly Language Programming', by Colin Opie is the one to buy. This is published by the McGraw-Hill Book Company (UK) Ltd, Maidenhead, Berks.

CREDITS

My thanks go to Sinclair Research Ltd, for giving me the job in the first place and for agreeing to the imperative need for a new language; to Tony Tebby, for doing all the difficult bits of the design work and for suggesting that I write this book; to John Watson, Liz Nemecek and Susan Sanders at McGraw-Hill for much invaluable advice and for publishing the book.

Most thanks, however, are due to my husband, Brian, for supporting, encouraging and threatening me; for persuading his boss, Bob Davies of H&M (UK) Ltd, Newmarket, to lend me a word processor at weekends to write the book on; and last but not least, for risking lifelong backstrain by lifting all the equipment in and out of the car every week!

TRADEMARKS

SINCLAIR, QL, SuperBASIC, QDOS, QL Microdrive and QL Toolkit are all trademarks of SINCLAIR Research Ltd, 25 Willis Road, Cambridge, CB1 2AQ, England.

COPYRIGHT

QDOS ©1983 SINCLAIR Research Ltd
SuperBASIC ©1983 SINCLAIR Research Ltd
QL Toolkit ©1983 SINCLAIR Research Ltd

The QDOS and SuperBASIC systems are the property of Sinclair Research Ltd.

Any tables, definitions of storage, etc., in this handbook are correct at the time of writing and appear with the kind permission of Sinclair Research Ltd. Sinclair Research Ltd does not guarantee that they will remain correct for any future releases of the System ROM.

The Sinclair QL Toolkit is available from Sinclair Research Ltd.

2 NOTES

1. ON THE SYNTAX OF THE DEFINITIONS

In all the formal definitions in this book, the following syntax is used :

CAPITAL LETTERS	denotes a SuperBASIC keyword or built-in feature
lower case	gives a general description to be replaced
[anything]	means that the enclosure is optional
{anything}	means that the enclosure is optional and repeatable

For example,

```
DEF[INE] PROC[EDURE] procedure_name [ ( argument {,argument} ) ]
```

would lead to various forms from the least

```
DEF PROC fred
```

through

```
DEFINE PROC david(x)
```

up to

```
DEFINE PROCEDURE jonathan(a,b,c)
```

Most of the keywords may be abbreviated. In order to avoid the tedious construction REP[E[A[T]]] meaning REP, REPE, REPEA or REPEAT, I will instead use REP[EAT]. Note that on the QL display, the optional part of the keyword is written in lower case thus : REPeat.

Common abbreviations used in the syntax are:

lno	for line number
st	for statement
exp	for expression
var	for variable

2. ON THE EXAMPLES

In the examples CAPITAL letters denote keywords or built-in procedures and functions whereas lower case denotes user-defined names. If I refer to a user-defined name in the text, I will usually write it in upper case if that clarifies the subject matter.

SuperBASIC statements are either held in a program or they are typed in directly from the keyboard. If they are to form part of a program, then they must have line numbers so that SuperBASIC knows which order to do them in. Any line typed at the keyboard with a line number in front of it will be inserted into your current program, but anything without a line number will be executed immediately.

For this reason all the examples in this book will be given line numbers where they might form part of a program.

3. ON THE INTERNAL STORAGE UNITS

During the course of this handbook, I will refer several times to the amount of room taken up by various tables, or entries in tables. Items in these entries will always be defined in terms of the number of **bytes** which they occupy.

A QL byte is composed of eight **bits**. A bit is the smallest unit available for independent access. It takes the value zero or one. The bits in a byte are numbered,

7 , 6 , 5 , 4 , 3 , 2 , 1 , 0

Two bytes make up one **word**; two words make up one **long word**. These are terms in general use but, for the sake of simplicity, I will refer to the number of bytes only, in the tables in the rest of the book.

A limitation of the Motorola 68008 chip is that, though it can access single bytes anywhere, words or long words must begin on an even, or word, boundary if they are to be retrieved from memory. For this reason, 'spare' bytes have to be included in certain storage areas in order to get to an even-byte boundary. For instance, in the following, fictional, table, I want to hold a two-byte length, a one-byte type and a two-byte pointer in that order. If the numbers are bytes offsets from some even address,

address + 0	length	(2 bytes)
+ 2	type	(1 byte)
+ 3	spare	(1 byte)
+ 4	pointer	(2 bytes)

you can see that an extra byte is needed at (address+3) so that the pointer, which must be retrieved as a complete word, starts on an even

offset. A long word address now added to the table,
address + 6 address (4 bytes)
would be fine, as it starts at a word boundary.

3 PROCEDURES, METHODS AND FUNCTIONS

SuperBASIC is a procedure-based language. That being the case, you ought to understand about procedures before anything else is brought in to distract you.

Procedures

In real life, a procedure is a set of directions which are followed in order to reach some desired object. The itemized procedure for feeding my cats, for example, is :

- (1) pick the dirty dishes up from the floor;
- (2) wash them and put them on the draining board;
- (3) dry them and put them on the worktop;
- (4) select a tin of cat food and open it;
- (5) if there is a cat on the worktop, then put it back on the floor;
- (6) divide the contents of the tin between the dishes;
- (7) fend off the cats and put the dishes on the floor.

This procedure will be followed without fail every morning at eight o'clock and every evening at seven o'clock. In my own mind it is firmly labelled as FEED_CATS and is unlikely to get mixed up with any of the other procedures involved in day-to-day living. A SuperBASIC procedure is structured in much the same way. It is an enclosed set of statements, labelled with its own name, designed to fulfil a specific task.

Most programs can be broken down into many such tasks. A program for drawing a scene, for example, might consist of a procedure to draw a house, a procedure to draw a tree, a procedure to fill in the sky and a procedure to draw the grass. Each of these procedures would be written separately and tested to make sure that they work, then they would be executed in turn to produce the finished picture.

The advantages of using procedures is immense. Provided that you have defined a procedure somewhere in your current program, you can call it from anywhere just by giving its name.

To define a procedure you need to use certain SuperBASIC keywords. You need to give the procedure a unique name so that when it is called, there

is no confusion over which one you mean.

The formal definition is

```
line number DEF[INE] PROC[EDURE] procedure_name [ ( argument {,argument}) ]
    { statement { :statement } }
```

```
line number END DEF[INE] [proc_name]
```

where the **procedure_name** must be unique.

When **calling** a procedure, the general form is

```
[line number] procedure_name [ parameter (separator parameter) ]
```

SuperBASIC stops whatever it is doing, goes to find the named procedure, executes all the statements inside it and then comes back to where it left off.

A procedure has finished either when SuperBASIC reaches the END DEFine statement or when it executes a RETURN statement. RETURNing may be done from anywhere inside a procedure and SuperBASIC then carries on from just after the original call. A RETURN statement from a procedure is simply

```
[line number] RET[URN]
```

A procedure must be called by name in order to have its statements executed. If SuperBASIC, during the course of a program, suddenly comes across a

DEFine PROCedure

statement, it will immediately start looking for an END DEFine, not taking any notice of the intervening statements. When it finds an END DEFine, it continues execution from there.

For example, the normal flow of execution is downwards, from line 1 to line 32767, so in the following skeletal program where ... indicates missing statements,

```
.....
100 sky
110 house
120 tree
130 grass
.....
200 DEFine PROCedure sky
.....
270 END DEFine sky
.....
300 DEFine PROCedure grass
```

```

.....
340 END DEFine
.....
400 DEFine PROCEDURE tree
.....
520 END DEFine
.....
1000 DEFine PROCEDURE house
.....
1230 END DEFine house
.....

```

when SuperBASIC gets to line 100, it looks for the procedure called SKY and finds it at line 200. It executes the procedure and, at line 270, returns to the call. The next line is a call to the procedure HOUSE, which is at line 1000. This procedure is carried out and at 1230, SuperBASIC is sent back to the call at line 110. The next line directs SuperBASIC to perform procedure TREE, and the one after that to do procedure GRASS. After this, execution would normally continue until it reaches the end of the program or is told to stop in some other way. When it comes to line 200, which is a definition line, SuperBASIC ignores everything until it reaches the first END DEFine at line 270 and then continues from there. It proceeds in this way, skipping all the definition structures as it comes to them.

A procedure can **call** any other procedure, but it should not **contain** another procedure. For example, you could have,

```

900 DEFine PROCEDURE housing_estate
.....
920 house
.....
940 house
.....
960 END DEFine

```

in which HOUSE is called repeatedly. This is acceptable, even desirable, but the structure **should not** be written

```

900 DEFine PROCEDURE housing_estate
.....
.....
1000 DEFine PROCEDURE house
.....
1230 END DEFine house
.....
1300 END DEFine housing_estate

```

This is because, though the procedures would work reasonably well if called, when SuperBASIC is skipping through the definitions, it looks for the **first** END DEFine after a DEFine. The name on an END DEFine is optional and for your convenience only; SuperBASIC does not check it. So, in the skipping through process described earlier, the lines between 1230 and

1300 will be executed when they should not be. There are absolutely no advantages in nesting procedure definitions in this manner.

You cannot have multiple entry points to a procedure. There is only one place to start and that is at the beginning. If you want to make the same basic procedure do different things under different conditions, you will have to use one of the **choice** constructions, explained in Chapter 6, together with a procedure **parameter**.

A parameter is normally used when there is a choice to be made within the procedure, such as whether to draw a tree with or without apples. The procedure TREE is perfectly willing to put apples on the tree, but it needs to know whether they are wanted this time or not. Suppose that particular definition line was updated to

```
400 DEFine PROCedure tree(apples)
```

where statements inside the procedure check the value of the variable APPLES and draw apples on the tree only if that value is not zero. Then the procedure calling line will be

```
TREE 0
```

for an empty tree, or

```
TREE 1
```

(or any other non-zero value) for a full one.

Parameters are essentially **information** for the procedure. When you write a procedure, you will know which factors can vary and what the calling statement will have to specify. The names that you give to any of these items of information in the statements within the procedure block must be included (within brackets), in the definition line. The calling statement will then give the actual values to be used at the time of running the procedure.

As a further example, consider the procedure HOUSE. It would be nice to be able to specify what size the house should be, either by width or by height, and what colour the door should be painted. You could do this by incorporating the parameters

```
1000 DEFine PROCedure house(size,door_colour)
```

in the definition line and then using whatever values are given in the call to determine the size of the house and colour of the door each time that the procedure is called.

The way in which parameters to procedures actually work is a little complicated and so is dealt with in far more detail in Chapter 4 - The Nametable.

Parsing v. execution

Whatever is typed into your QL must be in valid SuperBASIC if it is to be accepted. If it is in SuperBASIC, then it will conform to one of the syntax graphs set up in the SuperBASIC ROM (read-only memory). The process of checking whether the line does conform is called **parsing** and every line entered into the QL, whether typed in by hand or read in from a Microdrive cassette, is parsed before anything else is done to it.

During parsing, each element in the line is checked against the syntax graphs and, if correct, is converted into an internal SuperBASIC token. All leading and trailing spaces are removed. Any 'forced' space (e.g., one that separates a keyword from a name) is also removed.

If a line is wrong, you will be given the error, "bad line", and the line will be echoed for you to correct. Changing a line is detailed in Chapter 9 - Creating and Running a Program.

If a line parses satisfactorily, it is examined to see whether it is intended to form part of a program or to be executed immediately. If the line has a **line number** as its first item, it will be absorbed into the current program at the appropriate place, an indicator of the length of the tokenized line being slotted in front of it for greater efficiency later on; if the line does not have a line number in front of it, then the instructions contained within the line will be carried out straight away. Only lines which have been parsed may be executed.

Executing a procedure

To run a procedure, all that you have to do is to give its name and the values, in order, of any parameters. If you do this directly from the keyboard, the procedure will be executed immediately; if you put the call into a program, then it will not be done until the line is reached during normal program execution. If a procedure which you have called cannot be found in your current program, then you will get the error, "not found". The parameters, which may be variables, arrays, array elements or expressions, must be separated from each other by a valid SuperBASIC **separator** :

, ; \ ! TO

There is no means currently that you can use within a SuperBASIC procedure to indicate which separators have been used, but many of the built-in procedures do check on the separators and treat parameters in different ways accordingly.

THE RETURN STACK

When a procedure has finished, SuperBASIC returns to the calling statement and continues from there. It must therefore keep a record of where to go back to. Since one procedure can call another, there must be a record, in order, of all the calls. This is achieved by building up the **return stack**. This is a table having an entry for each procedure which has been called. As a new procedure is called, an entry is made on the stack. As a procedure is finished, its entry is removed again.

Each entry contains such information as the type of routine, the line number to return to, the statement on that line, the status of that line and information about the parameters in the call. The full list can be found in Appendix D - The SuperBASIC Storage Area.

Functions

Most of the SuperBASIC commands have themselves been implemented as procedures, e.g., PRINT, PAPER, ELLIPSE, etc. Of the rest, all of the 'structural' commands are defined as **keywords** (e.g., FOR, SElect, etc.), and the remainder are **functions**.

A function is similar to a procedure except that, in addition to doing a specific job, it always returns a single value and can only be used in an expression. Expressions are dealt with in Chapter 5. The formal definition of the structure of a user-defined function is :

```
line number DEF[INE] F[UN]N[CTION] function_name [(argument {,argument} ) ]  
        { statement {;statement} }
```

```
line number RET[URN] expression
```

```
line number END DEF[line] [fn_name]
```

where the **function_name** must be unique.

In this case the RETurn statement is obligatory and must be followed by an expression giving the returned value of the function. There may be as many RETurn statements as you like within the body of the function as long as there is at least one. If, in a function, you put a RETurn statement without an expression giving the value of that function, then an error will be detected when the time comes to return and the message, "error in expression", will be printed out.

To use a function, you have to give the name of the function (together with any parameter values enclosed in brackets and separated by valid

SuperBASIC separators), as a term in an expression. A simple example of this is the built-in SuperBASIC function, PI, for returning the numerical value of the mathematical quantity, π . This takes no parameters so you can simply say,

```
PRINT PI
```

to get the result. Another mathematical function, SIN, returns the sine of an angle. The parameter to this is the angle itself, expressed in radians. So the call,

```
PRINT SIN(0)
```

will give the result zero. Instead of being a single figure, the parameter can be a more complicated expression. For example,

```
PRINT SIN(PI/2)
```

gives the result, 1, of the sine of π by 2, or π divided by two.

The functions which you define yourself, called user-defined functions, may be used in exactly the same way as the built-in ones. An entry on the return stack will be created when a user-defined function is called and removed when a value has been returned. As with procedures, functions may call other functions or procedures, but the definition block must not contain any other definitions. The only way to use a function is to call it by name inside an expression, the 'skipping-through' process described earlier for procedure definitions works in exactly the same way for function definition structures.

Remarks or comments

It is a good idea to include comments in your programs to remind yourself of what the different sections do. It may not be immediately apparent in six months' time what you intended to do at the time when you wrote the code! Commenting is achieved by using the REMARK facility. The formal definition is

```
[line number] REM[ARK] text
```

where text is everything up to the line feed.

For example,

```
100 DEFine PROCedure message
200 REMark A tiny procedure to cheer myself up
300 PRINT 'Hiya, gorgeous, you get better looking each day'
400 END DEFine : REMark end of message procedure
```

SPACING OUT THE CODE

SuperBASIC does not allow you to have empty lines in the middle of your program. There are many ways of splitting up sections of your code. I tend to use statements such as

```
nnn REMark -----
```

around procedure definitions to distinguish each from the next, and

```
nnn :
```

the statement separator (:) only, giving an empty statement, whenever I want a free line. You will, I am sure, have your own preference, but remember that it must be legal SuperBASIC.

4 THE NAMETABLE

Names of variables and procedures are held in a tokenized form inside a program. It would be very time consuming and inefficient if, every time a variable was used, the entire list of names had to be checked to see if it existed and, if so, what its value was.

There are three separate areas where the details about the names are kept; the **nametable**, the **namelist** and the **variable values**.

The **namelist** is, as it sounds, merely a list of names in the form

length of name, characters in name

The beginning of this list looks like

```
5PRINT3RUN4STOP5INPUT6WINDOW6BORDER
```

The length of the name is held in a single byte, so the maximum number of characters in a name is 127. A name may be made up of letters, numbers or the underscore character (_), the only restriction being that a name cannot start with a number.

The **nametable** is the hub of the whole SuperBASIC system. Each entry refers to one name and contains the following information :

the type of name	(1 byte)
the type of variable if applicable	(1 byte)
offset of the name in the namelist	(2 bytes)
pointer to associated information	(4 bytes)

where the **nametype** is a code representing variable, procedure, etc;
the **variabletype** is a code representing floating point, string or integer;

the **offset to namelist** is, for example, 0 for PRINT, 6 for RUN, etc;

the **associated information** might be the value of a variable or the position of a procedure, etc.

When a new line is added to the program or a command line is entered, each name in the line is checked against the names stored in the **namelist** during the **parsing** phase. Upper and lower case characters are considered equivalent when comparing names, the name **cat** is the same as **Cat**, **CaT** or **CAT**. The form stored is that of the **first** time that the name was entered.

If a name does not already exist, then it is added to the end of the namelist and a new entry in the nametable is created to point to it. The number of the entry in the nametable which points to the name then forms part of the token used for that name in the internal arrangement of the program. At this stage the variable is unset because no value has been associated with the nametable entry.

A simple variable has nametype 02. It must be floating point, integer or string. The last character of the name indicates which; a dollar (\$) means that this is a string variable, a percentage sign (%) denotes an integer whilst an absence of either is taken to be floating point. The internal variable types are 01 for string, 02 for floating point, 03 for integer.

Assignment

Any variable is treated as unset (nametype 00) until it has been assigned a value. Assigning a value to a variable is done in SuperBASIC with the statement :

```
[ LET ] variable = expression
```

If the variable was unset previous to this statement, a space must be found for the result in the variable values area. After the expression has been evaluated, the value is stored in this space and a pointer to it is added to the nametable entry. If the variable already had a value before the assignment, all that happens is that the old value is replaced by the new. A complete string variable entry in the nametable is therefore :

```
0201.offset of name from base of namelist  
offset of value from base of vv area
```

whereas an unset integer variable is described by

```
0003.offset of name from base of namelist  
FFFFFFFF (no offset into vv)
```

When a variable is used in an expression, the nametable entry is checked to see whether there is an associated value. If there is, then this value is copied over from the vv area; if not, then an "error in expression" is generated. The process is described more closely in the chapter on the expression evaluator.

THE VARIABLE VALUES AREA

In order for you to understand fully the storage of variable values, it is necessary for me to describe the area in which they are held. So far I have shown the **namelist** to be a continuous stream of names and the

nametable to be an orderly sequence of entries. The **vv area**, however, is a heap of values of various sizes. It contains free or unused space as well as assigned space.

A 'free space' pointer points to the first unused space in the table; from this point on, each succeeding free space says how long it is and points to the next one. When a new value is to be stored, SuperBASIC asks the free space manager for a hole of the right size. The position of this hole is returned and the free space pointers around it are adjusted to reflect its status. When an area of the heap is no longer required, SuperBASIC tells the free space manager where and how long this is and the manager absorbs it into the free space system.

Because it may be freed, each entry in the vv area must be long enough to contain its length and the next pointer in that eventuality. The minimum size for this is eight bytes so, even though an integer only takes two and a floating point six, both types are assigned eight bytes. Since it is possible for a partial section of free space to be assigned, the remaining unused space after that portion has been detached must also be at least eight bytes long. This means that each entry must be a multiple of eight bytes long.

Integer and floating point values have fixed lengths, not so strings. Strings are stored as a two-byte length followed by the characters, one byte each. If the number of characters is odd, an unset character is appended. The total of the length plus characters plus any unset character is then rounded up to a multiple of eight bytes.

Reassigning values to floating point and integer variables is simply a matter of changing the entry in the vv table. The pointer to that entry does not change. Because of the flexibility of string variables, reassignment is more complicated. The 'evened-up' length of the new string is compared with that of the old, if it is the same then the entry is simply changed. If it is different, however, the old entry must first be freed, a new space requested, the new string put into the new space and the pointer in the nametable entry updated.

Procedures

Procedures and functions may be written in machine code (68000 series assembler) or in SuperBASIC. As they are executed in different ways they must be given different nametypes in the nametable to distinguish them.

Machine code procedures are type 08, machine code functions 09
SuperBASIC procedures are type 04, SuperBASIC functions 05

The nametable also keeps the information on where the procedures and functions are to be found. Machine code procedures and functions have the actual address of the start of the code included in their entry. SuperBASIC procedures and functions, however, use the line number of the

DEF PROC/FN statement.

A machine code function entry would be

```
0900.offset of name from base of namelist  
address of start of function
```

and a SuperBASIC procedure entry

```
0400.offset of name from base of namelist  
line number of DEFine PROCedure line  
FFFF (FF because last two bytes not used)
```

Whenever a direct command is executed, SuperBASIC does a **namepass** on the single line and, if the current program has been changed since the last direct command, on that as well.

A namepass simply means that all the nametypes in the line or program are checked and reset if necessary. This is also the stage at which the line numbers are stored against the names of user-defined procedures and functions.

PROCEDURE ARGUMENTS

(1) Procedure arguments were only dealt with briefly in the previous chapter. It is not always convenient for procedures to use variable names which may be in use elsewhere in the program. In addition, the procedure will frequently want to be called to work with different sets of variables. A function for calculating the sine of an angle would not be very useful if it could only cope with an angle associated with the variable X. For this reason, the **arguments** in the DEF PROC (or DEF FN) line are always replaced by the actual **parameters** given in the procedure call.

So, if you have a procedure FRED with arguments X and Y

```
100 DEFine PROCedure fred(x,y)  
110 x = x+2: y = y-2  
120 END DEFine
```

and you call it with parameters A and B

```
180 a = 4: b = 10  
200 fred a,b  
210 PRINT a,b,x,y
```

then on returning, A will have increased by two and B will have decreased by two, but X and Y will have reverted to whatever state they were in before the call.

The actual process gone through on calling a procedure or function is as follows :

2) First an entry is made at the top of the nametable for all the actual parameters. Such entries are not permanent, they only exist for the duration of the procedure. If the parameter is a simple variable, the new entry is a copy of the entry for that name with the pointer to the namelist replaced by a pointer to the original nametable entry:

copy of nametype, copy of variabletype, pointer to original entry
 copy of vv offset if one exists

If the parameter is an expression, then it is evaluated, space found for the result in the vv area and a new entry created to point to it:

02.expression type.FFFF (FFFF to show no name)
 offset of expression value from base of vv area

3) Next the nametype, variabletype and vv pointer for each argument are swapped with those for the corresponding parameter. In the example given, within the procedure FRED, any reference to the nametable entry for X will actually be using the vv pointer to the value of A.

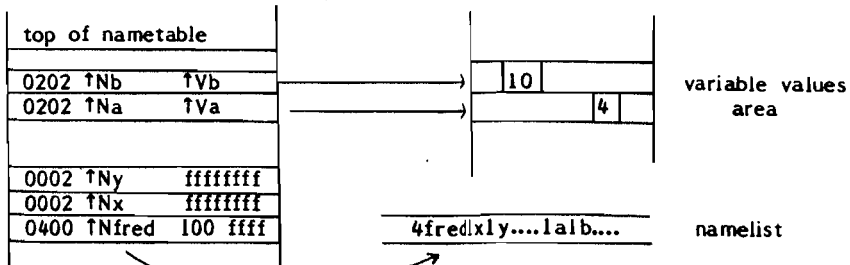
If there aren't enough parameters for the arguments, unset expression entries are created and swapped; if there are too many parameters, the excess are simply ignored.

On RETURNing from a procedure or function the steps are reversed. First the details of the arguments are swapped with those of the new parameter entries. Then those entries are examined. If an entry is one created for an expression, then any value assigned to it is meaningless once the procedure has terminated. The entry in the vv heap is therefore freed and the entry in the nametable deleted. For an entry which is a copy of a simple variable, however, the vv pointer is first transferred to the original nametable entry before the parameter entry is deleted. This copes with the not infrequent situation where a new value has been assigned to a variable inside a procedure or function.

This example shows the process in detail, using the code given earlier :

↑Nx means pointer into the namelist for the name of X;
 ↑Vy means pointer into the variables area for the value of Y;
 ↑A means pointer to the nametable entry for A;
 ffff indicates an unset field.

(1) initial status before anything has been called



PARTVP (name)
 PARUSE (name)
 PARNAME (parameter number)
 PARSTR# (name, parameter number)

Find type of parameter
 " usage "
 " name of parameter (expressions have no name)
 name of parameter or its value if it is a string
 (so that quotes aren't needed for string parameters)

(2) parameters created

0202	↑B	↑Vb
0202	↑A	↑Va
0202	↑Nb	↑Vb
0202	↑Na	↑Va
0002	↑Ny	fffffff
0002	↑Nx	fffffff
0400	↑Nfred	100 ffff

(3) parameters swapped

0002	↑B	fffffff
0002	↑A	fffffff
0202	↑Nb	↑Vb
0202	↑Na	↑Va
0202	↑Ny	↑Vb
0202	↑Nx	↑Va
0400	↑Nfred	100 ffff

nametype,
 variabletype
 and vv offset
 swapped

A point that cannot be emphasized too strongly is that the nametypes and the variabletypes of the parameter entries are swapped with those of the arguments. Thus, inside a procedure, though the code has been written with names of predetermined types, the type of the names when executed comes from the parameters that the procedure is called with and not from the names used in the code. For example,

```
100 DEFine PROCedure test(a$)
110 a$ = 'test string'
120 END DEFine
.....
200 test x
```

When the call at line 200 is made, an error message will be generated saying
 At line 110 error in expression

This is likely to puzzle you at first because the line appears to be in order, a string expression being assigned to a string variable. However, because this particular assignment is taking place inside a procedure, the argument entry for A\$ has been replaced by the parameter entry for X. The error has occurred because it is impossible to convert the string value given to the floating point value required by the variabletype. The Sinclair QL Toolkit contains functions which test parameter types once inside the procedure.

SEPARATORS

Although any valid SuperBASIC separator may be used to divide the parameters for a user-defined function or procedure, the process of creating parameter entries is the same as that for machine code procedures and functions, where the separator type is often crucial. The separator types must therefore be stored with the parameter entry.

The type of the separator which follows a parameter is incorporated into the nametable entry for that parameter by masking it into the top half of

the variabletype byte. The separator types are

- 1 for a comma ,
- 2 for a semicolon ;
- 3 for a backslash \
- 4 for an exclamation mark !
- 5 for TO

The actual nametype-variabletype of a simple floating point name followed by a comma will be 0212, but the separator type gets removed from a parameter entry very quickly. When we come to discuss channels in the chapter on input and output (Chapter 8), I will be introducing a hash (#) sign which is used to indicate a channel number. If a # is put before a parameter, the top bit in the variabletype is set on, changing the entry in the previous example, say, to 0292. Neither # nor separator types have any effect on the action of parameters to user-defined procedures and functions.

LOCAL VARIABLES

It is often the case that you want to use a variable inside a procedure without affecting its value elsewhere in the program, but where the variable itself is not a parameter of the procedure. This is the purpose of local variables. Swapping the values of two variables, for example, requires the use of a third, temporary, variable:

```
300 DEFine PROCedure swap(m,n)
310 LOCAL temp_var
320 temp_var = m
330 m = n
340 n = temp_var
350 RETurn
360 END DEFine swap
....
1000 swap ja,jb
```

An 'unset-expression' entry for TEMP_VAR is created after those for JA and JB. The nametype, variabletype and vv pointer are swapped with the actual nametable entry for TEMP_VAR in the same way that the copies of JA and JB are swapped with the entries for M and N. At the end of the procedure, the original details are restored, the vv entry, if any, freed and the new nametable entry deleted.

The general form of the LOCAL line is

```
LOC[AL] name {,name}
```

There may be more than one LOCAL statement in a procedure or function, but they must all come before the first executable statement in the procedure. For example,

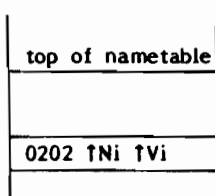
```

100 DEFine FuNction steve(x,y,result)
102 LOCal i,j
104 LOCal temp_var
106 i = x+y
etc.

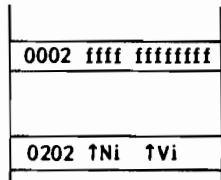
```

Just looking at the local variable, *i*, and assuming that it exists elsewhere,

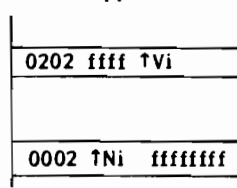
(1) initial



(2) create local entry



(3) swapped



where ↑Ni and ↑Vi are as defined in the earlier example.

Arrays

Whereas a simple variable only has one value at any one time, an array, or indexed variable, is a set of values, one for each element of the array.

Array element indices start at zero and rise to the maximum specified. This specification is done by means of a **dimension** statement. The formal definition is :

```
[line number] DIM dimensioned_array_name {,dimensioned_array_name}
```

where *dimensioned_array_name* is defined as :

```
array_name ( maximum_index {,maximum_index} )
```

For example,

```
10 DIM aa(9),bb(3,4)
```

will create a one-dimensional array AA with ten elements indexed by

```
aa(0),aa(1),aa(2),...,aa(9)
```

and a two-dimensional array BB with twenty elements indexed by

```
bb(0,0),bb(0,1),...,bb(0,4),bb(1,0),bb(1,1),...,bb(3,4)
```

The array names themselves are normal SuperBASIC names as defined earlier.

Floating point and integer arrays are entirely regular. A floating point array contains a given number of elements of six bytes each, the total being rounded up to a multiple of eight bytes as discussed before. An integer array contains elements of two bytes each, the total likewise rounded.

Arrays of strings are not entirely regular. The final dimension of a string array definition is the maximum size of each string in the array. It will always be rounded up to the next even number. Thus the statement,

```
DIM name$(20,10) : DIM age$(20,9)
```

would result in an two arrays being set up with space for 21 strings, each of which cannot exceed ten characters. Similarly,

```
DIM a$(8)
```

provides for one string of eight characters at most.

So much for the **maximum** length. The **actual** length is held in the zero'th element of each string. The length is two bytes long and the characters are one byte each. So after

```
a$='abcde'
```

where a\$ is as dimensioned above, the internal representation would be

```
05abcdeXXX
```

where XXX exist but are undefined. The size of a string array cannot change in the same way as that of a string variable since it would make array indexing very inefficient if this was the case.

STORAGE OF ARRAYS

Obviously an array needs more information held about it than a simple variable. This extra information is called the **array descriptor** and is kept in the vv area. The nametable entry for an array is therefore

```
03.arraytype.offset of name from base of namelist  
offset of array descriptor from base of vv area
```

As soon as a DIM statement has been **parsed**, all the names are marked as being unset arrays. A name cannot, therefore, be treated as both an array and a simple variable.

As soon as a DIM statement is **executed**, the array descriptor is set up for each name and storage space for the elements assigned. The total size is calculated, multiplied by six for floating point and two for integer arrays and a space of that size (rounded up to the nearest multiple of eight) found in the vv area.

Floating point and integer arrays have all their elements initialized to zero. String arrays are initialized to sets of zero length strings.

ARRAY DESCRIPTOR

This resides in the vv area and consists of the following information :

the offset of the array values from the base of the vv area (4 bytes)
the number of dimensions (2 bytes)

{ the maximum size of the next index (2 bytes) }
{ how far apart successive elements of this index are (2 bytes) }

the distance being counted in elements and the last two items being repeated for each index.

Consider, for example, the array descriptor for

DIM fx(1,2,3)

The number of dimensions is clearly three so the first two items in the descriptor are

offset of start of array values
3

The maximum size of each index is also given; 1, 2 and 3, but what about the distance between successive elements ?

Storage of the elements is done sequentially, with the last index varying fastest. The elements of array FX will thus be stored in the order

(000),(001),(002),(003),(010),(011),(012),(013),(020),(021),(022),(023),
(100),(101),(102),(103),(110),(111),(112),(113),(120),(121),(122),(123)

Looking at the last index, successive elements are next to each other, so the distance factor is 1. Searching for successive elements of the middle index, (000),(010),(020),(030),etc., shows them to be 4 away from each other, while successive elements of the first index are 12 apart.

The last six items in the array descriptor are therefore

1,12, 2,4, 3,1

SuperBASIC calculates these distances as

$$\text{distance} = (\max_{i+1} + 1) * \text{distance}_{i+1}$$

in other words, <number of elements in next index> times <distance apart for next index> gives <distance apart for this index>.

The total number of elements in the array is

$$(\max + 1)_1 * (\max + 1)_2 * (\max + 1)_3 \dots \text{ for all indices}$$

so FX in the example above has 24 elements.

The distance factor in string arrays has to take into account the fact that the zero'th element of each string is a two-byte length whereas the characters are only one byte each. In the distance equation given above, the maximum size of the last index must be increased by two rather than by one.

So for the array defined by

```
DIM XY$(2,3,4)
```

the descriptor will be

```
offset of strings from base of vv table
3
2,24, 3,6, 4,1
```

REDIMENSIONING ARRAYS

If, during DIMension statement execution, a name is found which has already been specified as an array, the old array values and descriptor are released before the new ones are assigned.

```
10 DIM a(20)
.....
100 DIM a(5,10)
```

In the above example, A is changed from a one-dimensional array with 21 elements to a two-dimensional array of 66 elements.

SUB-ARRAYS

Sub-arrays are a very powerful SuperBASIC feature. Because of the structure of the array descriptor, it is entirely possible to define several subsets of an array, each with its own array descriptor and all using the same array values. To do this, it is necessary to specify, for each index of the sub-array, the range of the original index which is to be extracted. The general form for index range specification is

```
[initial index] TO [last index]
```

where **initial index** is defaulted to zero and **last index** is defaulted to the maximum specified in the array definition.

Specifying the whole range of an index may be achieved by just giving the keyword **TO**, meaning **0 TO max_index**. In addition, if any ranges are omitted at the **end** of the sub-array, the whole range is assumed. For example, given an array X dimensioned to (2,4,6,8), the sub-array

X(TO, TO 3, 4 TO)

is equivalent to

X(0 TO 2, 0 TO 3, 4 TO 6, 0 TO 8)

Consider the example given earlier while discussing storage of the elements. From the array defined by

DIM fx(1,2,3)

let us extract a smaller array FX(0 TO 1, 0 TO 1, 0 TO 2). This reduced array will only have the elements :

(000),(001),(002),(010),(011),(012),(100),(101),(102),(110),(111),(112)

and so the array descriptor for it must be a modified copy of the original. The spacing between the elements has not changed (though some of them are no longer available), but the maximum size of the indices has. So, from the original array descriptor,

offset, 3, 1,12, 2,4, 3,1

we now get

offset, 3, 1,12, 1,4, 2,1

Suppose we define the array FX(1, 0 TO 1, 0 TO 2). This is half the size of the sub-array in the previous example since we have banished all elements (0,n,n). The values now start at a different place and, because the first index no longer varies, it has become two-dimensional. Because of all this, the amended descriptor is

offset+12, 2, 1,4, 2,1

When a sub-array is defined, each of its indices start again from zero, thus while FX(0 TO 1,1 TO 2,1 TO 3) is a different portion of the original array from FX(0 TO 1,0 TO 1,0 TO 2), most of the descriptor is similar.

FX'	FX''
FX(0 TO 1,0 TO 1,0 TO 2)	FX(0 TO 1,1 TO 2,1 TO 3)
offset, 3, 1,12, 1,4, 2,1	offset+5, 3, 1,12, 1,4, 2,1
The element FX'(1,1,2) is (1*12+1*4+2*1)=18 away from the start.	The equivalent element FX''(1,0,1) is (1*12+0*4+1*1)=13 away from the start but the start has been increased by 5.

Sub-arrays may be used anywhere that full arrays are used. It is not yet possible for mass assignment of one array to another, but it is extremely useful to be able to pass sub-arrays as parameters to procedures.

ASSIGNING VALUES TO ELEMENTS OF NUMERIC ARRAYS

This has the form

```
[LET] array (index {,index} ) = expression
```

where an index must be specified for each dimension defined in the DIM statement.

So, `FX(1,1,1) = 45`

would set the 18th element from the beginning of the array to the value 45. You could also use a sub-array, for instance

```
FX(0,1)(2) = 540
```

sets the third element of the sub-array `FX(0,1,0 TO 2)`, equivalent to the seventh element of the whole array, to the value 540.

If you have failed to define an exact element on the left hand side of an assignment, then you will get the error message "not implemented", because mass array assignment is not available yet.

ASSIGNING STRINGS TO STRING ARRAYS

Slicing string arrays is very similar to slicing numeric arrays.

Since they are defined to be arrays of strings, substring arrays are defined to be arrays of substrings. Given an array `A$(4,6)` containing 'ONE', 'TWO', 'THREE' and 'FOUR', the substring array `A$(0 to 4,3)` would produce the four substrings 'E', 'O', 'R', 'U' and not the single substring 'EORU'.

Because string arrays are defined in this way, assignment to them has the form :

```
[LET] string_array [ (index {,index} ) ] = string_expression
```

where the **final** index is omitted. The length and characters of the string are entered into the vv area as normal. Examples of use are,

```
DIM xy$(2,6): xy$(0) = 'first': xy$(2) = 'last'
```

```
DIM g$(8): g$='abcdefgh'
```

```
DIM tab$(10,12,8): tab$(3,4) = 'yy'
```

You can also, of course, change individual characters. For example,

```
g$(4) = "Q": PRINT g$
```

will give the string "abcQefgh".

If you modify an unset character, the element will be changed but the actual length of the string will not. SuperBASIC assumes that you know what you are doing.

You may access the zero element of a string array to change the apparent length of the string. Changing the length to greater than the maximum dimensioned is not advisable since printing the whole string will give unset characters beyond the actual length of the string.

If you assign too large a string, it will be truncated to the maximum number of characters allowed on the left-hand side. For example, if you tried,

```
g$(4 TO 6) = "12345"
```

only enough characters to fill G\$(4), G\$(5) and G\$(6) would actually be assigned, making the resulting string 'abc123gh'.

SUBSTRINGS OF STRING VARIABLES

The characters of string variables may be indexed in the same way as those of string arrays. The zero element cannot be accessed, though, since the length may **only** be changed by means of an assignment of the **whole string**.

String variables do not have descriptors, so any substrings of string variables have to be treated as expressions rather than as names with associated values because there is nowhere to hold pointers into the reduced string. Consider, for example,

```
100 DEFine PROCedure ex1(a$,b$)
110 a$ = 'abc'
120 b$ = 'def'
130 END DEFine
...
200 DIM x$(8)
210 x$ = '123456'
220 y$ = '123456'
230 ex1 x$(3 TO 5),y$(3 TO 5)
240 PRINT x$,y$
```

Both strings are passed through to procedure EX1 as '345'. The first gets changed to 'abc' and the second to 'def' but, whereas A\$ was swapped with

a substring of an array, B\$ was swapped with a string expression. Thus on return, X\$ will be '12abc6' but Y\$ is unchanged as '123456'.

String variables may be indexed on the left-hand side of an assignment,

```
y$(3 TO 5) = 'def'
```

gives '12def6', but only existing characters may be changed. If you said

```
y$(9 TO 10) = 'xy'
```

you would get an "out of range" error, since this assignment would extend the length of the original string.

It should be noted that you cannot slice string expressions, e.g.,

```
n$=DATE$: PRINT n$(16 TO 17) ... will work
```

but

```
PRINT DATE$(16 TO 17) ..... will not.
```

FINDING THE LENGTH OF A STRING

It is often convenient to know how many characters there are in a string. The SuperBASIC LEN function does this. The general form is

```
LEN (string)
```

where the string can be an expression, a variable or part of an array.

```
PRINT LEN('abcd') ..... gives 4  
x$='t': PRINT LEN(x$) ... gives 1
```

or, from the earlier example

```
PRINT LEN(xy$(0)) ..... gives 5 even though the maximum length  
is 6.
```

RECOVERING THE SIZE OF AN INDEX

A useful function for finding out the maximum size of the indices of an array is DIMN. It takes two parameters :

```
DIMN (array [,index_number] )
```

and returns the maximum size of the index given.

Looking at the array defined by,

```
DIM mat$(2,6,4)
```

If we now

```
PRINT DIMN(mat$,1) ..... we get the answer 2.  
PRINT DIMN(mat$,3) ..... returns the value 4.
```

If the index number is omitted, the first, or major, index is assumed.

```
PRINT DIMN(mat$) ..... gives 2.
```

Any index not dimensioned is necessarily zero :

```
PRINT DIMN(mat$,12) .... produces 0.
```

See the effect of using sub-arrays here.

```
PRINT DIMN(mat$(0 TO 1)) ..... gives 1  
PRINT DIMN(mat$(TO,1 TO 4),2) .. gives 3
```

ARRAYS AS PARAMETERS

If an array is given as a parameter to a procedure call, the swapping process described earlier for variables and expressions is slightly more complicated.

Consider the example,

```
100 DEFine PROCedure john(xarr)  
120 PRINT DIMN(xarr,1)  
140 END DEFine  
...  
200 DIM garr(9)  
220 john garr
```

On processing the call at line 220, a copy of the nametable entry for GARR is added to the top of the table. A copy of the array descriptor is also made and stored in the vv area and the new parameter entry updated to point to it. Note that it is not necessary to make a copy of the actual values. The nametable entries for the argument XARR and the parameter GARR are then swapped as normal.

On returning from the procedure, the entries are swapped back again, the copy of the array descriptor is freed and the parameter entry deleted.

You can also pass sub-arrays as parameters. For example,

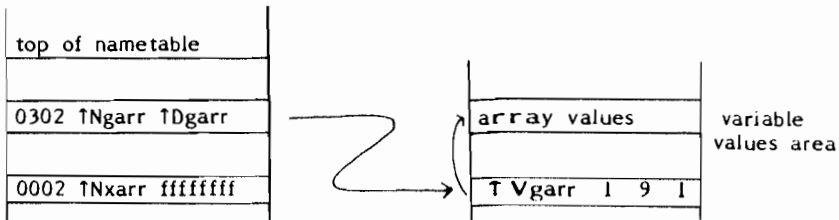
```
250 john garr(4 TO 6)
```

The same process is followed, the temporary array descriptor being for the sub-array rather than the full array.

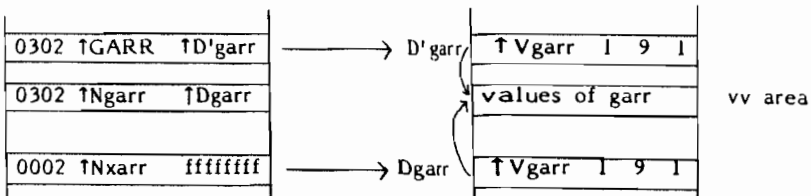
Look at this example where :

↑Ngarr is the pointer into the namelist for the **n**ame GARR;
 ↑Dgarr is the pointer into the vv area for the **d**escriptor for GARR;
 ↑Vgarr is the pointer into the vv area for the **a**rray values for GARR;
 †GARR points to the nametable entry for GARR.

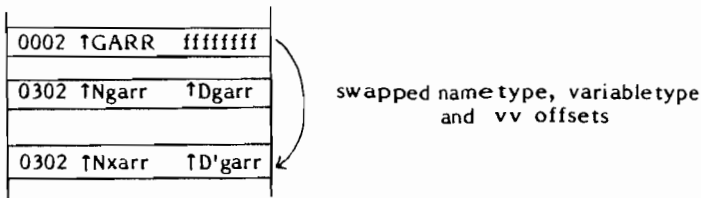
(1) initially



(2) parameter entries created



(3) after swapping



LOCAL ARRAYS

If a LOCAL array is required in a procedure or function, then it must be included in the LOCAL statements. For example,

```
110 LOCAL i,y_arr(4,5),z$(6)
```

A new entry is made on top of the nametable in the form

```
03.arraytype.FFFF           (showing a temporary entry)
offset of array descriptor from base of vv table
```

and a new descriptor and array values created in the vv area. The nametype

and vv offset are then swapped with the actual nametable entry for the local name. This means that you can have a LOCAL array with the same name as a simple variable elsewhere in the program, the only occasion on which this is allowed.

On returning, the real entry for the local name and the temporary array entry are swapped back. The temporary array values are released, followed by the temporary array descriptor. Finally, the nametable entry itself is deleted.

As LOCAL statements can contain indexed, as well as simple, variables, the earlier definition needs to be updated. The formal definition is as follows :

line number LOC[AL] name [(index {,index})] {,name [(index {,index})]}

5 THE EXPRESSION EVALUATOR

An expression is made up of terms and operators in the series

term {operator term }

The permissible operators, grouped into order of precedence, are:

first

&	concatenates (joins) two strings	
INSTR	gets the position of one string inside another string	
^	raise a floating point to the power of a fp or integer	
*	multiply one floating point by another	
/	divide one floating point by another	
MOD	take one integer to the modulus of another	
DIV	perform an integer divide	
+	add two floating points	
-	subtract one floating point from another	
>	greater than	} if both operands are strings, then character comparisons are made, otherwise all floating point
>=	greater than or equal to	
=	equal to	
==	approximately equal (equivalent) to	
<>	not equal to	
<=	less than or equal to	
<	less than	
AND	logical and (floating point)	
&&	bitwise and (integer)	
OR	logical or (floating point)	
	bitwise or (integer)	
XOR	logical exclusive or (floating point)	
^^	bitwise exclusive or (integer)	

last

Permissible terms are:

variables
array elements

functions
strings
values
sub-expressions (expressions in round brackets)

each of which may be preceded by a **monadic operator**

+ positive floating point
- negative floating point

NOT logical not (floating point)
~~ bitwise not (integer)

Values themselves may have an integral positive or negative sign, so it is possible to have an expression such as

1 + + + 2

where the first + is the diadic addition operator
the second + is the monadic positive sign
and the third + is an integral part of the number !

LEGAL CONSTANTS

What constitutes a legal number or string in an expression?

A legal **number** is of the form

[sign] decimal [E [sign] digit {digit}]

where **sign** is + or -

decimal is [digit {digit}] [. [digit {digit}]
and one of the optional [] **must** be present

A legal **string** is a set of characters delimited by a pair of single or double quotes.

For example,

'abcde'
"IA2Bc3"
' "Yes", she said'
"It isn't fair"

but not

'ab'cd'
'ab' 'cd'
" "\$,end"
"quote(" ")"

The expression evaluator

Evaluating terms of expressions has to be done in a given order if a consistent result is to be achieved. For every operator you need two operands. SuperBASIC stores the items in linked lists:

- an entry on top of the **nametable** defines the type of operand;
- an entry on top of the **arithmetic stack** is the value of the operand;
- pending operators are held, for convenience, on the **system stack**.

A marker showing the beginning of the expression is also held on the system stack.

Taking a simple example like

$$b + 2$$

the expression evaluator first makes a copy of the entry for B on top of the nametable. It looks up the value associated with the variable B and stores this on the arithmetic stack, changing the nametable copy to an expression entry by putting FF (meaning no name) in the pointer to the namelist. It then looks at the next item in the expression, which is a +, puts the + on the system stack as a pending operator, and gets the following term. As this is a value, 2, an 'expression' entry is created on top of the nametable and the floating point representation of 2 put on the arithmetic (or RI) stack.

This is the end of the expression so the + is retrieved and the two values on the RI stack are added together. The top entry on the nametable is deleted to leave one expression entry on the nametable, one value (the result) on the RI stack and no pending operators.

In general the process is :

- (1) look at next term;
- (2) if this is not an operator, create an entry on the nametable and a value on the RI stack;
if it is an operator, look at the previous pending operator;
- (3) if the precedence of the old operator is **greater** than the precedence of the new operator, do the previous operation;
- (4) if doing an operation, take both the RI values off the stack, replacing them with the result; take the pending operator off the system stack; delete the top entry on the nametable;
- (5) put the new operator on the stack and repeat from top.

At the end of the expression, do all the pending operators in turn until the beginning-of-expression marker has been reached on the system stack.

Looking at the following example step by step : the **bold** type shows the running order of items on the different stacks

$$a = b \text{ AND } x < 4 * z + y$$

Action at each step	Nametable and RI stack	Pending operator
read term a	1 a	
read operator =		1 =
read term b	2 b	
read op AND,prec(=)>prec(AND)		
do (a=b)	1 (a=b)	1 AND
read term x	2 x	
read op <,prec(AND)<prec(>)		2 <
read term 4	3 4	
read op *,prec(<)<prec(*)		3 *
read term z	4 z	
read op +,prec(*)>prec(+)		
do (4*z)	3 (4*z)	3 +
read term y	4 y	
end of exp. get pending op +		
do (4*z)+y. get pending op <	3 (4*z)+(y)	
do x<(4*z+y). get pend op AND	2 (x)<(4*z+y)	
do (a=b)AND(x<4*z+y)	1 (a=b)AND(x<4*z+y)	

SUB-EXPRESSIONS

Sub-expressions are expressions enclosed in parentheses. They are used to override the normal precedence order. For instance, multiplying the sum of two numbers by a factor is impossible unless the evaluator is forced to generate the sum first and the product second; $4+2*6$ gives 16 but $(4+2)*6$ gives the desired 36.

If, while processing an expression, an opening bracket, (, is found when it is expecting a term, the evaluator immediately puts a new beginning-of-expression marker on the system stack and ignores all previous pending operators until the corresponding) has been located. The sub-expression is then completed and evaluation of the outer expression recommenced. There is no limit to the number of expressions which may be nested in this way.

FUNCTIONS DURING EVALUATION

Functions have much the same effect on the evaluator as sub-expressions do. All current expression evaluation is frozen until the function has finished and produced its result on the RI stack. Function calls may also be nested as many times as you please.

COERCION

In the list of operators at the beginning of this chapter, I also included the **type** of operands with which each operator works. Thus + only adds up two floating point numbers, while & can only concatenate two strings.

It is exceedingly irritating to the user if he has to convert everything to the correct type himself before the expression will evaluate. SuperBASIC makes life easier by forcibly converting the operands, if at all possible, before performing the operation. This makes expressions like

'123' + 4 ... produce 127

and '123' & 4 ... produce '1234'

quite happily.

Converting a number to a string is a fairly straightforward task, but there are some side-effects when the reverse is done. The conversion stops at anything which does not form part of a number but does not give an error unless an **illegal** number has been generated. So, converting 'abcd' to a number gives an error since there is no legal number before the first non-numeric character. '4.5ab', however, produces four and a half without a murmur.

After conversion, the entry in the nametable is updated to reflect the new operand type.

INTERNAL REPRESENTATION

A word should probably be said at this stage about the internal representation of values.

Integers are represented in the Sinclair QL as two-byte hexadecimal numbers. This gives a range of -32768 to 32767.

Floating points are held in two parts, a two-byte exponent and a four-byte mantissa. The top four bits of the exponent are reserved for the SuperBASIC token, so that leaves twelve bits for the exponent itself. This gives a range of 2^{-2048} to 2^{2047} or, more familiarly, about $10^{\pm 616}$. The mantissa is a normalized hexadecimal number. That is to say, the decimal number is converted to hex and stored from bit 30 downwards to bit 0, bit 31 being reserved for the sign. For example,

0 is 0000 00000000

1 is 0801 40000000

-1 is 07FF C0000000 ~~0800 0000 0000~~

2 is 0802 40000000	-2 is 07FE C0000000	0801 80000000
4 is 0803 40000000	-4 is 07FD C0000000	0802 8000 0000
10 is 0804 50000000	-10 is 07FC D0000000	0804 8000 0000

Strings, as has been mentioned before, are stored as the number of characters followed by the characters themselves. The length is a two-byte hexadecimal number and the characters are kept in hexadecimal ASCII. Because the total amount stored must be even (a limitation of the 68008 processor), the length of a string is limited to 32766 which is the size of the integer length.

Operators

Most of the operators will be familiar to you already. Some, however, need a little explanation so, for completeness, I will give the whole list. Any of the operations can fail if one of the operands is an unset variable, or cannot be converted to the type required by the operator.

ARITHMETIC OPERATORS

+, -, *, /, ^

All the above arithmetic operations are carried out in floating point. The only exception is when something is raised to the power of a value which can be exactly represented as an integer. All results, however, are in floating point.

MOD, DIV

These work in integer arithmetic throughout.

n DIV m gives the integer part of n divided by m,

$$5 \text{ DIV } 2 \dots 2$$

n MOD m gives the value of n to modulus m,

$$5 \text{ MOD } 2 \dots 1 \quad (0,1,0,1,0,1,0,1,\dots) \\ (0,1,2,3,4,5)$$

It is the remainder of the operation n DIV m; $n - (n \text{ DIV } m) * m$

$$6 \text{ DIV } 3 \dots 2, 6 \text{ MOD } 3 \dots 0 \quad (0,1,2,0,1,2,0,1,2,\dots) \\ (0,1,2,3,4,5,6)$$

$$5 \text{ DIV } -2 \dots -3, 5 \text{ MOD } -2 \dots -1 \\ -5 \text{ DIV } 2 \dots -3, -5 \text{ MOD } 2 \dots 1 \\ -5 \text{ DIV } -2 \dots 2, -5 \text{ MOD } -2 \dots -1$$

STRING OPERATORS

The **&** operator concatenates two strings. It joins the second set of characters to the end of the first, counts the new length and appends an unset character if necessary. So, internally, given the strings

03abcX and 04wxyz

where X is undefined, the resulting string is 07abcwxyzX

Likewise, 05zyxwvX & 03pqrX produces 08zyxwvqpqr

string1 **INSTR** string2

gives the character position in string2 that string1 starts at.

'fox'	INSTR	"foxglove"	... 1	at beginning of string
'ox'	INSTR	'fox' 2	inside string
'Glove'	INSTR	'foxglove'	... 4	case ignored
'gloved'	INSTR	"foxglove"	... 0	all of string not found

The result produced is always integer.

LOGICAL OPERATORS

Logical operators have to evaluate conditions and decide whether they are true or false. The result, which is always floating point, is set to **zero** if the condition proves false and to **one** if it proves true.

For the **relational** operators,

<, <=, =, ==, >, <=, <

both operands are normally converted to floating point and the condition evaluated numerically. There is an exception to this when both operands are strings. In this case, a character by character comparison is made, except on any embedded numbers.

Arithmetic comparisons are made strictly. $A < B$ is only true if $A - B$ is less than zero; $C = D$ is only true if $C - D$ is exactly zero. The equivalence ($==$) or 'almost equal' operator has a greater tolerance. Here, $X == Y$ will be true if $|X - Y| \leq |Y * 1E-7|$, where $|X - Y|$ means the absolute, or positive, value of $X - Y$.

Note : It may not be immediately obvious to you that no value will ever be $==$ zero. If, in the above equation, Y is zero, then the check reduces to $|X| \leq 0$, clearly impossible when X is non-zero. If you make X zero instead

of Y, then the test becomes $|Y| \leq |Y*1E-7|$, also somewhat infeasible. To test for a quantity being very close to zero therefore, adding one to both sides gives the desired result, e.g., $(X+1)=1$.

If the equivalence operator is applied to strings, case is ignored. Thus 'a<>"A" is true, but 'a=="A" is also true. All other string comparisons treat upper case characters as having less value than their lower case counterparts. Short strings are 'smaller' than long ones. The following are in order :

'Axe','ape','axEhead','axe','axiom','BATH','bat','bath'

and 'a2','a20.0','a24','a3'

Combination logical operators work solely on true and false floating point values. They read zero as false and non-zero as true. They return false as zero and true as one.

L1 AND L2 is true if **both** L1 and L2 are true
L1 OR L2 is true if **either** L1 or L2 or both are true
L1 XOR L2 is true if **either** L1 or L2 **but not** both are true
NOT L1 is true if L1 is false

Bitwise logical operators do the above operations bit by bit on integer operands, setting the corresponding bit in the integer result ON (to 1) for true and OFF (to 0) for false.

Thus,

```
1 && 3 ... 00000001 and-bit 00000011 ... 00000001 ... 1
1 || 3 ... 00000001 or-bit 00000011 ... 00000011 ... 3
1 ^^ 3 ... 00000001 xor-bit 00000011 ... 00000010 ... 2
  ^^ 1 ... not-bit 00000000 00000001 ... 11111111 11111110 ... -2
```

Mathematical functions

There is a comprehensive set of SuperBASIC functions available. All of the trigonometric ones assume that the angle is in radians.

The mathematical functions are :

ABS	gets the absolute (positive) value of a floating point argument
ACOS	calculates the angle, the cosine of which is the fp given
ACOT	gets the angle, the cotangent of which is the fp given
ASIN	returns the angle, the sine of which is the fp given
ATAN	gets the angle, the tangent of which is the fp given
COS	returns the cosine of a given angle
COT	calculates the cotangent of a given angle
DEG	converts radians to degrees (floating point)
EXP	calculates the exponential function of the given fp (e^{arg})
INT	truncates a floating point to an integer and returns it as a fp

LN returns the logarithm to the base e of a floating point number
 LOG10 gets the logarithm to the base 10 of a floating point number
 PI takes no arguments, returns the floating point representation of
 RAD converts degrees to radians (floating point)
 SIN calculates the sine of a given angle
 Sqrt gets the square root of a floating point number
 TAN returns the tangent of a given angle

There are also random number facilities :

RND [([low_range TO] high_range)]

is a function to get a random number in the given range :

RND gets a floating point number in the range 0 to 1
 RND(n) gets an integer between 0 and n inclusive
 RND(m TO n) gets an integer between m and n inclusive

RANDOMISE [seed]

is a procedure to set a new random number seed. If a parameter is given, the sequence of random numbers returned by succeeding RNDs will always be the same for that seed. Different seeds produce different sequences. If no parameter is given then a 'random' random number seed is generated, based on the current clock value.

and string functions :

FILL\$(string,length)

produces a string of the specified length, filled with repeats of the first two characters in the string argument. If the argument has only one character, then all characters are a copy of this.

PRINT FILL\$('oO',9) gives 'oOoOoOoOo'

PRINT FILL\$(8,2) gives '88'

a\$='eric': b\$ = FILL\$(a\$,7): PRINT b\$ gives 'ererere'

CODE and CHR\$

for converting ASCII codes to characters and vice versa.

ascii_code = CODE (string)

gets the code of the first character of the given string, and

character = CHR\$ (integer)

returns a one character string from the ASCII code given.

Thus,

```
PRINT CHR$(CODE('A'))
```

should write out the letter A.

If the CODE of a character is between 48 and 57, the character is a digit; between 65 and 90, the character is in upper case; between 97 and 122, the character is lower case. To convert from upper to lower case therefore, add 32 to the CODE and take the CHR\$ of the result; to convert from lower to upper case, subtract 32.

A full listing of the ASCII equivalents is given in Appendix A.

VER\$

is a string function taking no parameters which just returns the two-character identifier of the System ROM. This identifier is completely random for each new version of the ROM released. The function to return it is included so that, if you do have any problems with your QL, you can let Sinclair know which version of the ROM you have when you report the trouble.

DIMN and LEN

These functions are fully described in Chapter 4 - The Nametable.

All the clock functions and procedures are detailed in Chapter 13 - The Calendar and Clock.

All the machine code functions and procedures are given in Chapter 12.

6 CONTROL STRUCTURES

SuperBASIC is a procedure-based language. Within this framework, the major keywords have their own structure.

CLAUSES

A clause is a set of statements bound together by SuperBASIC keywords. All start with an **opening clause** statement and finish with an **ending clause** statement. Any SuperBASIC keyword which is only meaningful when taken in conjunction with a particular clause is called an **intermediate clause** statement.

We have already met **definition clauses** which opened with DEFINE and closed with END DEFINE. An intermediate clause associated with them was RETURN. Definition clauses are a special case of clauses. They may not be nested and they must always be complex.

Complex clauses are those where the opening clause statement is on a different line to the rest of the structure. They must always finish with an END keyword statement. Another name for them is **multi-line** clauses.

Simple, or **in-line** clauses have all their statements contained within the same SuperBASIC line. They do not need an ending statement because the end of the line is always judged to terminate an in-line clause.

In general, simple clauses execute quicker than complex because SuperBASIC knows in advance that when exiting, it does not have to search for an END but can go straight to the next line.

Complex clauses, on the other hand, are clearer and more easily understood than simple ones. They are also easier to edit. It is recommended that contents of complex clauses are indented so that nesting of structures is easier to keep track of. It makes no difference to the efficiency of execution but it increases legibility and aids in debugging your program. Indentation is used in all the examples following.

It should, perhaps, be pointed out that while simple clauses can either be embedded in a program or entered as a direct command at the console, complex clauses make no sense unless they are used within a program.

Simple controlled transfer

Frequently, when writing a program, you will want a certain piece of code to be executed only if some condition is met.

The IF - THEN - ELSE - END IF construction caters for this eventuality. The formal definition of the multi-line IF clause is as follows :

```
line number IF expression [THEN]
    {statement {:statement} }
[ line number ELSE [statement] {:statement} ]
    {statement {:statement} }
line number END IF
```

The sequence of events when SuperBASIC reaches an IF statement is very simple. The expression is evaluated and if it gives a **true** (non-zero) result, then the statements immediately following are carried out. On reaching an ELSE, control is transferred to the statement after the ENDIF.

If the expression is **false** (zero), then SuperBASIC must skip this code and do the alternative statements if any have been given. To this end, an ELSE or an ENDIF are searched for. If an ELSE is found before an ENDIF, execution continues at the statement following the ELSE. If there is no ELSE, control is transferred to the statement after the ENDIF.

Since clauses can be nested, SuperBASIC keeps a count any intervening IFs on its search for the alternative code, to be sure of always finding the right ELSE or ENDIF.

Let us look at an example,

```
100 DEFine PROCedure ex2(a,b,c)
102 REMark Silly example with A as input, B as output, C as in and out
110 IF a<=5
120   b = a*a
130   IF b<10
140     c = c+4
150   ELSE
160     c = c*2+4
170   END IF
180 ELSE
190   b = a^3: c = c+2
200 END IF
210 END DEFine ex2
```

Take three cases

(1) a=1.

The lines actually executed are

110,120,130,140,150,180,210
at line 150, look for an ENDIF; at line 180, look for an ENDIF

(2) a=5.

Lines executed are
110,120,130,160,170,180,210
at line 130, look for an ELSE or ENDIF; at line 180, look for ENDIF

(3) a=10.

This time the lines executed are
110,190,200,210
at line 110, look for ELSE or ENDIF. Ignore the ELSE at 150, since
it is embedded in a deeper level IF.

As soon as an IF line has been processed and control has been passed to the appropriate place, SuperBASIC forgets that it is executing an IF clause. When it finds an ELSE, it assumes that it must have been processing a conditional piece of code and immediately searches for an ENDIF.

THE SIMPLE OR INLINE IF CLAUSE

This is defined as:

```
[line number] IF exp THEN statement [:statement] [:ELSE [st] {st} ]
```

alternatively, the THEN keyword may be replaced by a colon, so

```
[ln] IF exp : st {st} [:ELSE [st] {st} ]
```

Inline IFs work in the same way as multi-line IFs, except that the end of the line is always taken to be the ENDIF, e.g.,

```
IF n>limit THEN PRINT "Limit reached"  
IF a<100: a = a+1: ELSE a = 1: b = b+1
```

If you insert an ENDIF yourself, the rest of the line is normal.

```
IF a*b=1 THEN PRINT 'a*b=1': ELSE PRINT 'a*b<>1': ENDIF: PRINT "DONE"
```

whichever choice is made, 'DONE' will always be printed.

Inline IFs may be nested satisfactorily.

```
IF a+b>10 THEN IF a>5: PRINT 'b<5': ELSE PRINT 'b>=5'
```

An elegant solution to the factorial process

```
x! = x(x-1)(x-2)...1
```

can be shown by

```
100 DEFine FuNction factorial(x)
110 IF x=1: RETurn 1: ELSE RETurn x * factorial(x-1)
120 END DEFine
```

Multiple controlled transfer

The SELECT - ON - END SELECT construction gives a wide choice of options. The formal definition of the complex clause is :

```
line number SEL[ECT] [ON] variable
{
  line number [ON variable] = range {,range} {;statement}
  {statement ;statement} }
[
  line number [ON variable] = REMAINDER {;statement}
  {statement ;statement} ]
line number END SEL[ECT]
```

where **variable** must be a simple floating point variable

and **range** is of the form

expression [TO expression]

For example,

```
990 PRINT 'choice is ';
1000 SElect ON choice
1010 ON choice = 1 TO 4,-1,9 TO 11,7
1020 PRINT '-1,1 to 4,7 or 9 to 11'
1030 ON choice = 5: PRINT '5'
1040 =13 TO 15
1050 PRINT "13 to 15"
1060 =REMAINDER: PRINT 'remainder'
1070 END SElect
```

The **process** by which SuperBASIC determines which part of the SElect clause to do runs thus :

An 'ON var =' clause is searched for. When one is found, the value of the variable is checked against the ON ranges. If none of them match, then the next ON is looked for. This process continues until either a range match has been located or a closing ENDSElect has been found. In the latter case, no selection is made and execution continues after the ENDSElect statement. This is not an error, it avoids the annoying step of having to skip around the construction if the variable is out of range.

If a match on range is found, however, control is transferred to the statement after the relevant ON. As with IF, once the initial decision regarding which statement to do next has been made, SuperBASIC forgets that it is executing a SElect clause. Encountering another ON var = , it assumes that it is and skips all the statements up to the corresponding ENdSElect, taking any nested SElects into account.

ON var = REMAINDER is a catch-all. It is **always** true. It should be clear from the above narrative that, even if it is possible for the value of a variable to match with more than one ON range, only the first such match will ever be found. For this reason you are strongly advised to position the =REMAINDER range at the **end** of the set of ONs! Again, REMAINDER cannot be combined with any other ranges within the same ON since everything is included in it anyway.

INLINE SELECTS

These are of the form :

```
[!no] SEL[ECT] [ON] var = range {,r} {:st} {:[ON var]=r {,r}{:st } }
```

The beginning of the clause has been 'closed up' to cut down the simplest clauses, e.g.,

```
SElect x=14,27: y=100  
(very similar to IF x=14 OR x=27 : y=100)
```

When executing an inline SElect, the ON ranges are searched as before but the line feed takes the place of the ENdSElect statement.

The **criteria** for deciding whether a value is in a range other than REMAINDER is slightly different for the two cases

```
ON v = L  
and  
ON v = L1 TO L2
```

If the match is to be made with a single value, then the target and the value are checked for being approximately equal. That is $|L-v| \leq |v*1E-7|$ where $|L-v|$ means the absolute, or positive, value of $L-v$.

For a range, the value must be greater than or equal to the lower limit and less than or equal to the upper limit.

This means that when using an internally calculated value, it is possible for = L to match, but = L TO L not to match.

Looking at an example

```
1000 DEFine PROCedure sel_chk(a,b,c)  
1010 b=0
```

```

1020 SElect ON a
1030 ON a= -4 TO 14
1040   b = a MOD 5
1050   SElect ON b
1070     ON b=0,2
1070       c = c^4
1080     ON b=3: c = c^3
1090     =REMAINDER: c = c^3*4
1100   END SElect
1110 = -16 TO -5,15 TO 20
1120   b = 10
1130   c = 20
1140 END SElect
1150 END DEFine

```

(1) a = 5.

This executes lines

1010,1020,1040,1050,1070,1080,1110,1150

at line 1020, look for range; at line 1050, look for range; at line 1080, look for ENDSElect; at line 1110, look for ENDSElect

(2) a= 15

This executes lines

1010,1020,1120,1130,1140,1150

at line 1020, look for range

(3) a=100

This executes lines

1010,1020,1150

at line 1020, look for range; no match found

Uncontrolled transfer

However carefully structured a language is, there will always be something that you want to do which does not fit in with the design. With this in mind, SuperBASIC permits the use of a GO TO statement which transfers control, unconditionally, to a given line number. The formal definition is

[line number] GO TO expression

The expression is evaluated and converted to an integer. The result must be in the range 1 to 32767 since these are the allowable line numbers. The end of the current line is found, then, utilizing the line length increments stored with each tokenized SuperBASIC line, SuperBASIC travels upwards or downwards very quickly until a line number is located which is either equal to or greater than that required. So, if the line number given does not exist, it is not an error since the next highest one available will be transferred to instead, e.g.,

```
10 GOTO 100
50 x = 1
90 y = x^3
200 PRINT x,y
```

On executing line 10 it is found that line 100 does not exist so control is passed to line 200.

Another facility is

[line number] GO SUB expression

where the expression is also evaluated as a line number. As before, control is transferred to that line, or the next available, with the difference that, on encountering a RETURN statement, execution resumes at the statement following the original GOSUB. It is, in fact, a limited form of procedure, lacking names, parameters and local variables. When a GOSUB is executed the line to return to is placed on the procedure return stack with a different type to mark it as a GOSUB.

When using both GOTO and GOSUB, if the line number which has been given is past the bottom of the program, it is not deemed to be an error, simply an untidy way of finishing execution. SuperBASIC will return control to the console channel.

ON..GOTO/GOSUB

A facility which might be useful is the capability to GO TO different line numbers depending on the integer value of an index expression. The formal definition is

[Ino] ON index_expression GO TO Ino_expression (,Ino_expression)

The index expression is evaluated as an integer, then each line number expression in turn is evaluated until the count of them has reached the index. Control is then passed to the last line number evaluated.

Consider the line,

```
10 ON a+b GOTO 100,1100,105,4000,200,200
```

When this is executed, (a+b) is evaluated to give an integer. If it is less than 1, an "out of range" error is generated. Suppose the result is 3. Then each line number expression up to the third will be evaluated, not skipped, and control passed to, in this case, line 105. If you have not given enough line number expressions for the index value, an "out of range" error is again generated. In this example, results of 7 and over would be illegal.

There is a corresponding definition involving GO SUB

[Ino] ON index_exp GO SUB lno_exp {,lno_exp}

The procedure followed is exactly the same except that when a RETURN is executed, control is passed back to the statement after the ON...GOSUB.

It cannot be stressed too strongly that performing uncontrolled jumps into other structures may have strange side-effects. It is entirely your own fault if this occurs.

7 LOOPS

Frequently, when designing programs, you will want to be able to execute a piece of code several times over. SuperBASIC provides two methods of doing this.

Continuous loops

Repeating a piece of code indefinitely is easy. You simply insert the REPEAT keyword and a loop identifier at the top of the section of code, and an END REPEAT plus identifier at the bottom. The formal definition is

```
line number REP[EAT] loop_identifier_name
           {statement {statement} }
line number END REP[EAT] loop_identifier_name
```

There are two intermediate loop clauses

```
           NEXT identifier
and
           EXIT identifier
```

both of which may appear anywhere in the body of the loop any number of times.

ENDREPEAT sends you straight back to the statement after the REPEAT. NEXT enables you to go back to the beginning before reaching the end of the loop. EXIT, as the name suggests, removes you from the loop altogether, depositing you at the statement after the ENDREPEAT.

WHAT IS A LOOP_IDENTIFIER_NAME?

The loop identifier or index must be a simple floating point variable. The reason is that several pieces of information need to be kept about the

loop and the most convenient place to store them is in the vv area. In all other respects, though, the name is still a simple variable. It can be assigned a value and used in an expression. This means that the value, if any, must come first in the vv entry and the other information at known offsets to it.

The vv entry associated with a REPEAT loop index is 12 bytes long :

floating point value	(6 bytes)
REPEAT line number	(2 bytes)
ENDREPEAT line number	(2 bytes)
statement on REPEAT line	(1 byte)
statement on ENDREPEAT line	(1 byte)

Because the vv entry is bigger, the nametable entry of a loop identifier has to be distinguished from that of a simple variable by giving it a different nametype. A REPEAT loop index has nametype 06.

The process by which a REPEAT loop is executed is as follows :

On encountering a REPEAT statement, the loop identifier is examined. If it is not already a REPEAT loop index, the old vv entry, if it exists, is released and a new one of the right size assigned. The nametype in the nametable entry is updated and the loop information can now be stored.

The index value is always cleared. The line number of the REPEAT line is filled in, also which statement on the line this is. These are necessary so that, when the time comes actually to repeat the loop, SuperBASIC knows where to restart. The spaces for the ENDREPEAT line number and statement are cleared for the moment, the entry will be updated to include them when the ENDREPEAT line is actually reached.

If the identifier name was already a REPEAT loop index, it is possible that we are repeating this very line. The current line number and statement are therefore checked with those stored in the vv entry and, if they correspond, none of the information is replaced. There is a particular reason for this which will be made clear later.

Having set all the information up, SuperBASIC no longer needs to remember that it is looping. When it arrives at an ENDDREPEAT statement it simply reads the identifier name, locates it in the name table, and looks up the loop details in the vv area. A "not found" error is generated if the identifier given has not been set up as a repeat loop index. If all is well, however, the line number and statement are inserted into position in the loop information entry. SuperBASIC goes back to the start line and statement given in that entry and continues from the next statement.

A NEXT statement has much the same effect. The identifier is examined and the loop information located. Execution recommences after the line number and statement given.

An EXIT statement is used to leave the loop. SuperBASIC looks at the identifier and finds the loop information in the vv table. If that

information includes the line and statement number of the ENDREPeat, then control is passed to the statement **after** that given. If the ENDREPeat has not yet been executed, however, the entries in the loop information will be empty. In this case, SuperBASIC has no option but to search for an ENDREPeat with the correct identifier. On finding it, the line number and statement are stored in case they are needed again and execution continues as normal. This is the reason for not blanking out the endline number and statement if the other details of the loop information match. The second EXIT that is executed, if there is one, will be faster than the first because SuperBASIC already knows where to go.

There is no limit to the number of nested loops in a program.

EXIT identifier

will transfer you out of a loop no matter how deeply you may be embedded inside it. NEXT has the same property.

For example,

```
80 DEFine PROCedure print_2da(arr,nrow,ncol)
82 LOCal i,j
90 i = 0
100 REPeat outer
110   j = 0: PRINT
114   i = i+1: IF i>nrow: EXIT outer
120   REPeat inner
130     j = j+1: IF j>ncol: NEXT outer
140     PRINT arr(i,j),
150   END REPeat inner
160 END REPeat outer
170 END DEFine
```

An interesting side-effect of this method of handling loops is that, providing you have not redefined the loop identifier, it is possible to EXIT out of a loop which you left some time ago! I cannot think of a use for this at present, but it is worth bearing in mind when debugging programs which use the same names for different loops, or which use very similar names for different loops.

More importantly, you can also do a NEXT after you have left a loop. This is occasionally useful, especially if you are addicted to GOTOless programming. Indeed, provided that you never want to leave a loop, the ENDREPeat is not even necessary, though it plays havoc with your indentation.

For example,

```
190 min = 0 : max = 0
200 REPeat gobbledegook
210   a$ = CHR$(RND(65 TO 90))
220   PRINT a$;
230   IF a$="A": min = min+1: NEXT gobbledegook
240   IF a$="Z": max = max+1
250   NEXT gobbledegook
```

INLINE REPEAT LOOPS

The formal definition of a simple REPEAT loop is :

```
[Ino] REP[EAT] name : statement {,statement}
```

EXITs and NEXTs may appear among the statements and the line feed is taken as an implicit ENDREPEAT name.

Inline repeats are very useful for simple looping but contain a serious limitation in that they cannot be nested on a line. This was an oversight and ought to be remedied. If you do nest them, only the inner one will be repeated.

Controlled looping

While REPEAT loops are very useful, it is also necessary to be able to do a piece of code a fixed number of times, or with the index variable taking a particular value each time through the loop, or both. FOR loops were designed with this in mind.

The formal definition for the complex clause is :

```
line number FOR index = range {,range}
                    {statement {;statement} }
line number END FOR index
```

where **range** is of the form

```
expression [TO expression [STEP expression] ]
```

Again the intermediate clauses

```
NEXT index
```

and

```
EXIT index
```

can be used anywhere in the body of the FOR code any number of times.

The major difference between FOR and REPEAT loops is that the index variable takes a value each time through the loop and, when it has run out of values, the loop has finished.

THE INDEX VARIABLE AND THE FOR RANGE

The FOR loop index variable must also be a simple floating point name. Each time through the loop it takes the next value in the FOR range. A range, as has already been said, is of the form

expression [TO expression [STEP expression]]

The expressions are evaluated at the time when the range is started. If STEP is not present, the increment between successive values in the range is defaulted to 1. If TO is not present, the step size is zero. The information which has to be associated with a FOR index variable is

the current floating point value	(6 bytes)
the line number of the FOR line	(2 bytes)
the line number of the ENDFOR line	(2 bytes)
which statement on the FOR line	(1 byte)
which statement on the ENDFOR line	(1 byte)
the end value of the current range	(6 bytes)
the step value of the current range	(6 bytes)
the position on the line of the current range	(2 bytes)

Thus 26 bytes are needed and so the nametype must be different again. A FOR index has nametype 07 in the nametable. If this has not been set when the FOR execution is started, the old value, if one exists, is released, space for the FOR information assigned in the vv area and the nametype and vv pointer in the nametable entry updated.

HOW DOES SUPERBASIC DECIDE ON THE NEXT VALUE?

The first time through the loop, the value assigned is the first one in the first range. On subsequent passes, the process is:

- (1) get current step value, if this is zero then go on to the next range;
- (2) add current step to current value;
- (3) test the new value against the end-of-range value:
 - if it is less, carry on;
 - if it is almost equal (within $1E-7$ of stepvalue), set it to the actual end value and continue (this test is necessary since an internally calculated value can be slightly inaccurate);
 - if it is beyond the end value, the range is exhausted. Move to the next range.

Starting a new range :

- (1) check to see whether the range is possible at all. 1 TO 0 STEP -1 is fine, but just 1 TO 0 is not since it has a default step of +1. If the range is exhausted already, move to the next one. (It is not an error

- since it is occasionally useful to be able to force the skipping of a range.)
- (2) if the range is OK, set the index to the first value, enter details of the new range and update the position in the vv entry.

So, in the example

```
1000 FOR j= -14, 1 TO 2, 7 TO 9 STEP 0.5, 2 TO -2 STEP -2
```

the value of J would be successively

```
-14,1,2,7,7.5,8,8.5,9,2,0,-2
```

Reading carefully through the steps given above, you will notice that it is possible to change the number of times that a range is done, but not the number of times that a single value is done, nor which range is done next. If, while running the above example, you put in a line such as

```
1050 IF j==8 THEN j=12
```

the value of J would be successively

```
-14,1,2,7,7.5,8 (changed to 12),2,0,-2
```

plus, perhaps

```
1060 IF j==1 AND flag=0 THEN j=-4: flag=1
```

you would get

```
-14,1 (changed to -4),-3,-2,-1,0,1,2,7,7.5,8 (changed to 12),2,0,-2
```

WHEN IS THE INDEX VALUE UPDATED?

When executing an **END FOR index** statement, SuperBASIC looks at the index, locates the loop information and gets the next value in the range. If the value is OK, control is passed to the statement after the FOR line number and statement held in the vv entry. If the last range has been exhausted, however, execution continues from after the ENDFOR.

NEXT index has a similar effect. If there is a next value, the index entry is updated to hold it and the loop is repeated from just after the FOR. If the FOR range is exhausted, execution continues from the statement after the NEXT.

At this point it is worth pointing out that if the whole FOR range is exhausted before it is even started, right at the beginning of loop execution, then the index variable has no value and execution continues from the statement after the FOR. Any NEXTs or EXITs now will give the error "not found".

Leaving a FOR loop is either done naturally, by exhausting the last range given in the FOR statement, or can be achieved by means of the EXIT index statement. This works in exactly the same way as exiting out of a REPEAT loop except that it is an ENDFOR rather than an ENDREPEAT which is searched for. The index holds whatever value it was given last.

If you have a missing ENDREPEAT or ENDFOR, SuperBASIC gets to the bottom of your program on its search and stops execution normally. Of course, if you have another loop with the same name further down your code, an EXIT will find it. This is something else to remember when debugging programs.

THE INLINE FOR LOOP

A simple FOR loop is defined as

```
[No] FOR index = range [,range] : statement [: statement]
```

where NEXT and EXIT may appear among the statements and the end of the line is taken as an implicit ENDFOR index.

For example,

```
DIM a(9): FOR i=0 TO 9: a(i) = i+1
```

There is no difference in the method of execution of an inline FOR and they are frequently faster in performance.

Inline FOR loops are a remarkably useful construction but unfortunately, as with inline REPEAT loops, simple FOR loops cannot be nested on a line. Only the innermost FOR or REPEAT will be iterated if inline nesting is attempted.

LOOP EPILOGUE

It is often useful for a programmer to know when a loop has finished naturally and when it has been exited out of. A loop epilogue is inherent in the SuperBASIC FOR structure. Look at this example :

```
300 DEFINE PROCEDURE check_str(name$,arr$,row)
310 REMARK Return which row, if any, name$ is in inside arr$
320 LOCAL i
330 FOR i = 0 TO DIMN(arr$,1)
340   row = i: IF arr$(row)=name$: EXIT i
350   NEXT i
360   row = -1
370 END FOR i
390 END DEFINE
```

Line 360 is only ever processed if the range has been exhausted. This is called the loop epilogue.

8 DATA HANDLING, INPUT OUTPUT AND THE FILE SYSTEM

Simple output

We have already seen many examples of the PRINT command. It may be used anywhere in a SuperBASIC line and its form is

```
PRINT [#channel_number separator] { [expression] separator}
```

The **channel number** refers to the place where the parameters of the command are to be printed. It must always be preceded by a hash (#) sign.

```
#0 is the console window  
#1 is the execution window  
and #2 is the program listing window
```

These are the only reserved channel numbers, others may be assigned to user-defined windows, peripheral devices or Microdrive files. There are examples of their use later on in this chapter. If the channel number is left out of the PRINT command, then the execution window, #1, is assumed.

Separators tell the PRINT statement how to format the output. They do this by advancing the print position, the position where the **next** character is to be put :

, spaces to the **next** eighth column, provided that that still leaves a gap of eight columns before the edge of the window. A new line is started if it is otherwise. When tabbing, at least one space is always made.

; does not move the print position at all.

\ sets the print position to the first column of the next line.

! only has any effect if there is a parameter following it. If there is room on the line to print the next parameter, then a single space is left. If there is not enough room, then a new line is started. If the print position is already at the beginning of a line, then no action is taken.

TO should be followed by the character position on the current line where the next item is to be printed. A single space is always made. If that takes the print position to beyond the column specified, the print

position rests there, otherwise more spaces are printed until the character position has been reached. The separator after the column number is not taken as a formatting separator.

If there is no separator at the end of the list, a new line is put out. So

```
PRINT
```

by itself, prints a blank line. The separator after the channel number, if one has been used, is taken as non-formatting.

The following procedure prints column numbers across the screen and then uses separators to position the output :

```
80 DEFine PROCEDURE sep_example
90 REMark print column numbers
100 FOR j=1 TO 3
110   FOR i=0 TO 9: PRINT i;
120 END FOR j
130 REMark print separators
140 PRINT '\,',';','\''!\!at beg of line!!in middle of line!!but
far too much to fit on previous line. Nothing at the end of this'
150 PRINT TO 5, 'TO col 5 works' TO 10, 'but TO col 10 fails'
160 END DEFine
```

The **parameters** which are to be printed must evaluate to either floating point or string values. Because PRINT is a procedure, the parameters must follow the rules for procedures. Unset variables are allowed in the parameter list, but non-evaluating expressions are not. If you try to PRINT an unset variable, an asterisk (*) will be printed instead. For example,

```
400 DEFine PROCEDURE ex3(a,b)
410 LOCAL c
420 a = 1: b = 2
430 PRINT a,b,c,a+b
440 END DEFine
```

will, when you call

```
ex3 x,y
```

produce

```
1      2      *      3
```

If one of the parameters is a non-trivial expression which will not evaluate, an error will be generated. For example,

```
432 PRINT a+c
```

included in the example above, would cause the message

At line 432 error in expression
to be produced.

PRINTING ARRAYS

Arrays and sub-arrays may be printed without having to specify all the elements. The elements are printed in the order in which they are stored. The separator which follows the array name is used to separate all the elements printed. For example,

```
990 DEFine PROCEDURE set_up
1000 DIM ax(7),a24(2,4),a2$(2,8)
1010 FOR i=0 TO 7: ax(i) = 2*i
1020 FOR i=0 TO 2
1030   FOR j=0 TO 4: a24(i,j) = i&j
1040   a2$(i) = i&i&i&i&i&i&i&i&i
1050 END FOR i
1060 END DEFine
```

Now, after running procedure SET_UP on the default television execution screen,

```
PRINT ax,          ..... 0      2      4      6
                   ..... 8      10     12     14

PRINT ax(4 TO 6);  ..... 81012

PRINT a24(TO,3)    ..... 3
                   ..... 13
                   ..... 23

PRINT !a24!        ..... 0 1 2 3 4 10 11 12 13 14 20 21 22 23
                   ..... 24

FOR i=0 TO 2: PRINT \!a24(i)!

                   ..... 0 1 2 3 4
                   ..... 10 11 12 13 14
                   ..... 20 21 22 23 24

PRINT a2$(TO,3),\a2$ ..... 0      1      2
                   ..... 00000000
                   ..... 11111111
                   ..... 22222222
```

FREEZING THE SCREEN

Often, when items are being printed quickly, there is a danger that they will have scrolled out of the top of the window before you have had a chance to assimilate them. The printout can be halted at any time by pressing CTRL and the function key F5 at the same time. This will 'freeze' the whole screen until you press CTRL and F5 again. Pressing any other key while the screen is frozen will also unfreeze it, but the key pressed will remain in the input buffer to be used when the screen output has finished.

Simple input

It is very often the case that you want to be able to set variables to different values each time that you run the program. This is the purpose of the INPUT command. It has much in common with the PRINT procedure.

```
INPUT [#channel_number separator] { [parameter] separator}
```

Again, if the channel number is not given, the execution window, #1, is assumed.

If a parameter is an expression, the INPUT command acts in the same way as PRINT, writing the value of the expression to the channel. Any separators are also treated as they would be by PRINT.

If a parameter is a simple variable, an array element or an array string, the cursor will flash at the next print position and wait for the data to be entered. Each item of data must be terminated by an ENTER keystroke. The input value is converted to the correct type and assigned to the variable specified. INPUT then moves on to the next item.

A couple of examples,

```
100 INPUT 'Width of room (ft) ?',wide,'Length of room (ft) ?',long
110 PRINT 'You need '!long*wide/9!'sq.yds of carpet'
```

```
200 INPUT 'Price including VAT ?',total
210 PRINT 'Price excluding VAT is'total/1.15 \'VAT is'!15*total/115
```

You may INPUT data into anything that you can normally assign to. If, when waiting for input, just the ENTER key is pressed (no text in front of it), this is taken to be an empty string. If a numeric value was expected, the conversion will therefore fail and an "error in expression" message will be generated.

INPUTING SINGLE CHARACTERS

INPUT needs an ENTER keystroke to finish each item. This is not very helpful if you want to write a program which requires constant input from the user, a game where the cursor movement arrows are to be used to move something around, for example. A function which returns a single character from the keyboard is obviously required.

INKEY\$ does just that. It is a string function having the form,

```
INKEY$ [ ( [#channel] [time] ) ]
```

where **channel** is where the input is to come from, default 1;

and **time** is the number of frames to wait for that input before returning, a frame being one-fiftieth (1/50) of a second. If the number of frames is zero (the default), the function will return immediately; if the number of frames is -1, INKEY\$ will wait indefinitely.

To see the difference between an immediate return and an indefinite one, first enter the line,

```
REPeat in: PRINT INKEY$(-1)
```

Now any character that you type at the keyboard is echoed on succeeding lines of the execution window until you BREAK (CTRL and space together) out of it. See the difference when you execute the line

```
REPeat in: PRINT INKEY$(0)
```

The function is constantly checking the input channel, finding nothing and printing nothing followed by a line feed. When you do find anything, it will be echoed as before, but for the most part there is nothing in the input queue. Again BREAK (CTRL and space together) to finish. If you retype the line as

```
REPeat in: PRINT INKEY$;
```

and type some random characters, although it looks as if it is waiting indefinitely, the function is still constantly checking and finding nothing but this time you have told it not to advance the print position each time that it samples the keyboard, so it doesn't.

You will have noticed during these examples that several of the keys, including the cursor movement ones, print blotches on the screen. A blotch indicates a non-printing character. To check them, therefore, you need to use the CODE function which returns their decimal ASCII values. Try the line,

```
REPeat in: PRINT !CODE(INKEY$(-1))!
```


to find out that

← is 192	CTRL ← is 194	ALT ← is 193
→ is 200	CTRL → is 202	ALT → is 201
↑ is 208	CTRL ↑ is 210	ALT ↑ is 209
↓ is 216	CTRL ↓ is 218	ALT ↓ is 217

and many more!

TWO OR MORE KEYS PRESSED SIMULTANEOUSLY

In the above example, I cited the value of CTRL combined with a key. This changes the key values because it does not have an intrinsic value of its own.

Other keys, when pressed together, do not combine to make a single value; but there are applications where it is advantageous to know if a particular combination of keys has been selected. For example, some games allow the user to give the directions 'left', 'up' and 'fire' at the same time by simultaneous depression of ←, ↑ and the space bar. SuperBASIC provides a function, KEYROW, which allows direct interrogation of the keyboard in a reasonably limited fashion.

The keys are divided up into the following **keyboard matrix**. Each row is numbered and each key assigned to a row sets a defined bit in the integer value of that row. The matrix is,

Row 0	7	4	F5	F3	F2	5	F1	F4
Row 1	↓	SPACE	\	→	ESC	↑	←	ENTER
Row 2	~	m	ℓ	b	c	.	z]
Row 3	;	g	=	f	s	z	CAPLOCK	[
Row 4	j	d	p	a	1	h	3	l
Row 5	o	y	-	r	TAB	i	w	9
Row 6	u	t	0	e	q	6	2	8
Row 7	,	n	/	v	x	ALT	CTRL	SHIFT
bits	7	6	5	4	3	2	1	0

Each row can be interrogated using the function KEYROW, each bit in the result being set on (to 1) if the corresponding key is depressed, or off (to 0) if the corresponding key is idle.

The form of the function is

KEYROW (row_number)

where row_number is 0 to 7 as defined above and is which row to check.

For example, suppose that you have asked for the value of KEYROW(1) and that you are holding down

← , ↑ and SPACE

all at the same time.

The result will be

0 1 0 0 0 1 1 0 = 64 + 4 + 2 = 70
↓ sp \ → esc ↑ ← ent

or suppose that you press all of the function keys at once, then the value of KEYROW(0) will be

0 0 1 1 1 0 1 1 = 32 + 16 + 8 + 2 + 1 = 59
7 4 F5 F3 F3 5 F1 F4

The following procedure will demonstrate the function. It uses two screen commands, CLS and AT, that will not be explained until Chapter 10 - Screen Control.

```
100 DEFine PROCedure row_val
110 CLS: FOR i = 0 TO 7: PRINT 'Row!'i
120 REPEAT rows
130 FOR i = 0 TO 7: AT i,8: PRINT KEYROW(i): CLS 4
140 END REPEAT rows
150 END DEFine
```

KEYROW calculates the value of each row in turn and the result is printed out against its row number. You may press any combination of keys and check the result. As soon as the keys are released, KEYROW finds a value of zero.

There is a failing in the action of this function in that, when three keys which form three of the corners of a rectangle within the keyboard matrix layout are depressed simultaneously, the KEYROW values of both the affected rows are the same, clearly untrue, e.g.,

D and A produce 80 from KEYROW(4)
D and A and B also produce 80 from KEYROW(2) even though M is not being held down. If A is released, the KEYROW values return to 16 for row 2 and 64 for row 4.

This limitation should be born in mind when making use of the function.

PAUSING DURING EXECUTION

There is one more form of input to be considered. We have already seen that the user may freeze the screen at any stage by pressing CTRL and F5 together, then CTRL and F5 again to continue the display. The program itself may also halt execution temporarily, before it goes on to produce a new diagram say, by using the PAUSE command. This causes execution to stop for a specified length of time before continuing. At any point during this wait, the user can press any key on the keyboard to force the program to carry on. If an indefinite period of time has been specified, this is the **only** way to continue execution.

The form of the procedure is

```
[line number] PAUSE [time]
```

where **time** is the number of frames (a frame is 1/50th of a second) to wait for input before proceeding. The default is -1, wait indefinitely; if a time length of zero is specified, no pause is made.

The following procedure will give you an idea of the action on execution different PAUSEs.

```
100 DEFine PROCedure pausing
110 CLS
120 PRINT 'Pause indefinitely (default)': PAUSE
130 PRINT 'Pause 100 frames (2 seconds)': PAUSE 100
140 PRINT 'Pause 200 frames (4 seconds)': PAUSE 200
150 PRINT 'No pause': PAUSE 0
160 PRINT 'Pause indefinitely (-1)': PAUSE -1
170 INPUT 'Pause for how many frames ?!p;
180 IF p>=0: PRINT !('p/50!'seconds)'
190 PAUSE p
200 PRINT 'finished'
210 END DEFine
```

Constant data

DEFINING

It is often the case that certain variables and arrays are required to take values sequentially from an unvarying set of data. It is convenient, therefore, to be able to keep these values somewhere in the program. The mechanism used is the DATA statement which simply marks out sets of

expressions to be assigned at some stage during execution of the program. The formal definition is

```
line number DATA expression {,expression}
```

e.g.,

```
100 DATA 'Mon','Tues','Wednes','Thurs','Fri','Satur','Sun'
```

ASSIGNING

The data items are normally selected in turn and to assign an item of this data to a variable, you must use the READ command. The general form is

```
[line number] READ parameter {,parameter}
```

where a read-parameter may be a simple variable, an array element or an array string, e.g.,

```
1000 READ dayname$: PRINT 'Today is!'dayname$;'day'
```

When SuperBASIC executes a READ statement, it first checks to make sure that the parameter is valid for input, then it gets the next item in the whole DATA sequence for the assignment.

A **data pointer** is kept up to date in SuperBASIC's storage area. It holds

the line number of the data item read last	(2 bytes)
the statement number on that line	(1 byte)
the item number along that line	(1 byte)

When the next data item is requested, SuperBASIC looks to see if there is another data item on the current data line. If so, the value of that item is returned and the item count incremented; if not, the next DATA statement is searched for and, when found, the first item on that line is returned and the data information updated.

If an item of data cannot be converted to the required type for the variable, an "error in expression" is generated. If there is no data left when a READ statement is processed, an "end of file" error is given. If all is well, though, the value read is assigned to the variable in the normal way. You can find out whether you are about to run out of data by using the function EOF. This returns **true** if there is no data left, **false** otherwise, e.g.,

```
500 IF EOF: PRINT 'Run out of DATA statements': EXIT read_loop
```

DATA statements themselves do not have any effect on normal execution. They may be put almost anywhere. It is most efficient, however, to put them at the very beginning of a program. This is because moving to a data item always starts from the first line of a program so, if all the DATA statements are up there, it takes less time to find them.

REPEATING

It is likely that circumstances will arise in which you want to skip certain items of data or repeat some of them. There is a facility for doing this called RESTORE. It has the form

```
[line number] RESTORE [line_number_expression]
```

If there is no parameter, RESTORE sets the current data pointer to the beginning of the program file, otherwise the expression is evaluated as a line number and the current data pointer set to statement one and item zero on that line. It doesn't matter if the line given does not contain a DATA statement because, when the next data item is requested, the first one after that line will be found.

This is an example using READ, DATA and RESTORE statements

```
100 DATA 'January',31,'February',28,'March',31,'April',30,'May',31
110 DATA 'June',30,'July',31,'August',31,'September',30,'October',31
120 DATA 'November',30,'December',31
124 :
130 DEFine PROCedure print_diary
132 REMark Print set of diary sheets
140 :
150 INPUT 'Start at year ?'start_year'\E nd at year ?'end_year'\Print
to channel ?'chan$
160 IF chan$="" : chan$ = 1
170 a$ = FILL$("-",80)
180 FOR i=start_year TO end_year
190   RESTORE 100
200   FOR j=1 TO 12
210     READ month$,ndays
220     IF j=2 AND i MOD 4 = 0 THEN ndays = 29
230     PRINT #chan$,\a$\\month$,i\\a$\\
240     FOR k=1 TO ndays: PRINT #chan$,k
250   END FOR j
260 PRINT #chan$,\a$\\a$\\
270 END FOR i
280 END DEFine print_diary
```

The file system

Although READ-DATA-RESTORE and simple interactive PRINT-INPUT are very useful, they cannot be said to satisfy the whole spectrum of input and output needs. Some method of holding data separately from the program is necessary.

WRITING TO FILE

The QDOS operating system supports a full file handler. The files may be kept on Microdrive. To write information to a new file from a program, the process is:

- (1) OPEN the new file on an unused channel;
- (2) PRINT the data to that channel;
- (3) CLOSE the channel.

A **file name** may be any sequence of letters, numbers or underscores. Within each Microdrive cassette, the name must be unique. When referring to a file in SuperBASIC, the name must be prefixed by the device. This will be **mdv1_** or **mdv2_** depending on which Microdrive slot you have inserted the medium into, e.g.,

```
200 OPEN_NEW #3,mdv1_data1
210 FOR i=1 TO 10: PRINT #3,i
220 CLOSE #3
```

will create a new file called DATA1 on the cassette in Microdrive 1.

Files are likely to be wanted for different purposes. The OPEN command has variations to cater for different needs.

OPEN_NEW will open a new file for input and output. Currently there is no way to rewind a file back to the beginning so the input facility on a new file is somewhat superfluous. It is expected that rewinding will be implemented in future releases of the ROM.

OPEN_IN will open an existing file for input only. This is not an exclusive assignment, so the same file may be OPEN_INed on more than one channel. The input pointers are independent for each such channel.

OPEN will open an existing file for input and output. If a file is OPENed on one channel, it may not be opened on any other until it has been closed on the original channel.

The parameters for all three commands are

#channel , name

The **channel** is a number which can only be in use for one channel at any one time. If the channel number given has already been assigned, it will be closed automatically before being re-opened.

Any number up to 32767 may be used, but details about the channels are kept sequentially, each entry in the channel table being 40 bytes long, so using a high number when there are lower ones unused is very wasteful of space.

The name is a string expression, a string variable or a name. If it is a string expression or string variable, the resulting string will be used. If it is any other name, however, then the characters of that name are converted to the string required, e.g.,

```
OPEN #4,mdv2_dat2           is equivalent to
OPEN #4,'mdv2_dat2'        and
hf$='dat2': OPEN #4,"mdv2 "&hf$  and
m$='mdv2_dat2': OPEN #4,m$
```

Great care should be taken if reassigning channels 1 or 2. Certain SuperBASIC commands use these automatically and expect them to have various attributes. Reassigning the console channel, channel zero, is particularly hazardous and not to be recommended because you will not be able to enter any more commands from the keyboard. Think about it.

CLOSING A CHANNEL

Closing an output file or channel is very important since the action of an explicit or implicit CLOSE is to write out anything still in the buffer and to put an end-of-file marker on a file. The formal definition is :

```
[line number] CLOSE #channel
```

If you have been writing data to a Microdrive file and remove the medium before a CLOSE has been processed on the relevant channel, the last part of the file including the end-of-file marker will be missing. An implicit CLOSE is done whenever a new OPEN is carried out on a channel which is already open. Don't leave it to chance though, do an explicit CLOSE to be safe!

READING FROM FILE

This is done in the same way as writing to a file. The file should be OPEN_INed for maximum protection against unwitting PRINTs, e.g.,

```
390 DIM a(10)
400 OPEN IN #3,mdv1_dat1
410 FOR i=0 to 9: INPUT #3,a(i)
420 CLOSE #3
```

When using a console window channel, the INPUT command will write expressions and perform separator formatting. When using any other channel, all formatting and printing of expressions are suppressed. The method used is to enquire whether character positioning is allowed on the channel from which you are INPUTing, e.g.,

```
490 DEFine PROCedure roots
500 INPUT #1,'Name of input device ?',inp$
```

```

510 OPEN #3,inp$
520 REPEAT coeff
530 INPUT #3,'give terms a,b,c of ax^2+bx+c=0',a,b,c
540 root_term = b^2-4*a*c:
545 IF root_term<0: PRINT 'imaginary': NEXT coeff
550 root = SQRTRoot_term
560 PRINT 'Roots of 'a;'x^2 +'b;'x +'c!'are!(-b+root)/(2*a)!'and!
(-b-root)/(2*a)
570 END REPEAT coeff
580 END DEFine roots

```

Thus, when asked to give the name of the input device, if you opt for a window on the screen, all the prompts will be given. If data is to come from a file, however, the prompts will be suppressed. Specifying new windows on the screen is dealt with later on in this chapter.

When INPUTing from a file, every character up to and including a line feed will be read as a single item. The item will then be converted into the type required for the corresponding parameter. If the conversion fails, then an "error in expression" will be generated.

If INKEY\$ is used to read from a file, each character is read separately. To see the difference, execute the following statements one at a time,

```

OPEN NEW #3,mdv1_exin
PRINT #3,' 67'           to set the data file up
PRINT #3,'not number'

OPEN #3,mdv1_exin
INPUT #3,a$,b$          to show that what goes in must come out
PRINT a$,b$

OPEN #3,mdv1_exin
INPUT #3,a,b            to show what happens converting a non-
                        numeric string to a floating point number

PRINT a,b              result of last INPUT

OPEN #3,mdv1_exin
REPEAT in: PRINT 'next',INKEY$(#3)  to prove that INKEY$ reads one
                                    character at a time.

```

Notice that this last line finished by displaying the "end of file" message on the console window. To find out whether the end of a data file has been reached, use the EOF (end of file) function with the appropriate channel number.

```
420 IF EOF(#3): CLOSE #3
```

The general form of the EOF function is

```
EOF [(#channel number)]
```


where the default is DATA statements in the program itself. The result will be true if the end of the file has been reached, but false otherwise.

USING A NEW MICRODRIVE CASSETTE

Before a new Microdrive medium can be used, it must be **formatted**, divided up into manageable units of length called **sectors** which are then used to direct the file handling system around the tape. This is done by using the FORMAT procedure, giving which Microdrive the medium is in and the name to use for the tape, e.g.,

```
FORMAT mdv1_aaron4
```

The size of the name is limited, and will be truncated, to ten characters, otherwise any normal SuperBASIC name will do. The FORMAT command fills the tape with blank sectors, writing the name of the tape, in this case, AARON4, at the start of every sector. This enables QDOS to tell very quickly whether the tape in a Microdrive has been changed so, even though an invisible random number is incorporated into the tapename, it is not a good idea to have different media with the same name.

When it has finished this formatting, QDOS writes out a message into the execution screen telling you how many sectors are on the tape and how many of them were able to be verified properly.

In the case of the above example, the message might be

```
119/122 sectors
```

This means that the tape has 122 sectors on it and 119 of them have been verified as good. If you want the message to be written into a different place, put the desired channel number into the command. The formal definition is

```
[line number] FORMAT [#channel,] name
```

where **name** is of the form

```
mdv1_tapename or mdv2_tapename.
```

FORMAT is also the command to use if you want to scratch or completely overwrite an old tape. **It will destroy** any previous contents.

LISTING FILENAMES

To get a list of all the files on a particular medium, you need to use the DIRectory command. The formal definition is

```
[line number] DIR [#channel,] device
```

where **channel** is the place where you want the catalogue to be printed out. The default, as before, is the execution window;

and **device** is either **mdv1_** or **mdv2_** depending on which drive you have put the cassette in.

The directory listing is in the form

```
name of medium
no of vacant sectors/total no of good sectors
{filename}
```

For example, after creating files **LOG_OUTPUT** and **INPUT_DATA**, say, the directory listing of **AARON4** above, produced by

```
DIR mdv1_
```

might look like,

```
aaron4
114/119 sectors
log_output
input_data
```

showing you that you have now got 114 sectors left out of the 119 good ones that you started with.

COPYING FILES

It is possible to make copies of files on Microdrives. The normal form is

```
COPY name TO name
```

where **name** can be **mdv1_filename** or **mdv2_filename** and the only restriction is that the first file must exist but the second file must not exist.

```
COPY mdv1_dat1 TO mdv1_dat33
COPY mdv2_sink TO mdv1_sink
COPY mdv1_afile TO mdv2_bname
```

DELETING FILES

This is done by simply saying

```
DELETE mdv1_filename
```

or **DELETE mdv2_filename**

No error is given if the filename isn't found; an error is given if the

file you are trying to delete is currently open on a channel.

The only way to overwrite a file is to delete it and then re-save it or copy to it. For example, the set of statements to update the library versions of a particular program presently in memory might be,

```
DELETE mdv1_last_version
COPY mdv1_current_version TO mdv1_last_version
DELETE mdv1_current_version
SAVE mdv1_current_version
```

MICRODRIVE BACKUP

There is no built-in facility for copying entire Microdrive cartridges, but the following procedure will copy each file in turn from the medium in mdv1 to the pre-formatted medium in mdv2 :

```
100 DEFine PROCedure copy1_to_2
110 REMark copies all files from mdv1_ to mdv2_
120 REMark tape in mdv2_ must be formatted already
130 OPEN NEW #3,mdv1_allfiles
140 DIR #3,mdv1_
150 OPEN #3,mdv1_allfiles
160 INPUT #3,a$,a$ : REMark skip 1st 2 lines
170 REPEAT all
180 IF EOF(#3): EXIT all
190 INPUT #3,a$
200 IF a$='allfiles': NEXT all
210 PRINT 'copying',a$
220 COPY 'mdv1 '&a$ TO 'mdv2_ '&a$
230 END REPEAT all
240 CLOSE #3
250 DELETE mdv1_allfiles
260 END DEFine
```

Facilities for default directories and the use of 'wild cards' in file names as well as direct access to files are provided in the Sinclair QL Toolkit, an addition to the standard SuperBASIC system.

Using peripheral devices

THE SERIAL PORTS

There are two RS-232-C jack-plug sockets at the back of the Sinclair QL. They are marked

SER1 and SER2

These are referred to in SuperBASIC as the serial ports and are used to transfer data to or from external devices, printers for example.

The device names must be in the following form :

SER [portnumber] [parity type] [handshaking status] [protocol]

where **the port number** is either 1 (default) or 2 depending on which socket the cable is plugged into.

The **parity** is either the default of eight-bit data
e ... for even
o ... for odd
m ... for mark
or s ... for space.

The best bet is to set the parity to whatever the device at the other end of the RS-232 cable is expecting.

The **handshaking status** is either h (default) or i.

If it is **h**, the transmitting software will only send data when the receiving hardware is ready. It uses the DTR (data terminal ready) line on SER1 or the CTS (clear to send) line on SER2.

If you specify **i**, handshaking is ignored. This means that data bytes are sent without waiting for the ready signal. If you have sent data with the handshaking option over the RS-232 line to a device which isn't ready and you wish to clear the input buffer, BREAKing (by pressing CTRL and space together) and re-opening the channel with the ignore option will cause any data waiting to be sent to be sent. This loses the data but clears the input buffer, i.e.,

```
OPEN #4,SERh: PRINT #4,a$ ... then BREAK out of it and  
OPEN #4,SERi: OPEN #4,SERh
```

The **protocol required** may be r (default), z or c.

r just sends raw or unmodified data.

z appends a CONTROL Z to the end of the file sent. This enables SuperBASIC to detect EOF on a text file.

c changes all 'line feed' (new line) characters to 'carriage return' characters as well as putting a CONTROL Z at the end of a file.

So, SER by itself is equivalent to SER1hr.

The **SERIAL device names** can be used in the same way as the Microdrive filenames earlier. It is as valid to say

```
COPY mdv1_dat1 TO SER
```

as it is to say

```

500 OPEN #3,mdvl_datl
510 OPEN #1,SER
520 REPEAT copy_line
530 INPUT #3,a$
540 PRINT #4,a$
550 IF EOF(#3): EXIT copy_line
560 ENDREPEAT copy_line
570 CLOSE #3: CLOSE #4

```

When you are printing to an output device like this, you may want to specify the width of a line for formatting purposes. This is done by using the following command

```
[line number] WIDTH #channel, number of characters
```

where the width of the device linked to a channel is taken to be 80 characters until it is reset. This does not apply to channels attached to screen windows where the size of the window itself determines its width.

You will have noticed by now that the COPY command is not limited to Microdrive files. Any device names may be used. The formal definition is

```
[line number] COPY device TO device
```

where **device** is MDVn_filename or any other peripheral device such as SER, CON, SCR or NET, all of which are defined in this chapter.

It is possible to connect two QLs together via the SERIAL ports. If the RS-232 cable is plugged into SER1 on one QL, it must be connected to SER2 at the other end (unless you have a crossover connector). So you can say

```
COPY MDVn_name TO SER
```

on the first machine and

```
COPY SER2 TO MDVn_name
```

on the other.

COPY copies a file with its header if at all possible. For this reason you should only use COPY for copying a file across SERIAL ports if there is a QL at the other end also COPYING the file. With anything else, the file should be copied without its header by using the copy-no-header command. This is of the form

```
COPY_N file TO device
```

If you are COPYING anything other than a file, it will realize that there isn't a header and won't try to copy it.

The header is only noticeable over the serial ports. If a file is copied over a serial line without the Z option, the header is used to determine

the length of the file.

If you are interested in communication with other computers, the file header is in the form

file length	(4 bytes)
file access	(1 byte)
file type	(1 byte)
type dependent information	(8 bytes)
length of filename	(2 bytes)
filename	(36 bytes)
date information	(12 bytes)

The **BAUD rate** is the number of characters per second that are being transmitted and received. The default baud rate is 9600, but this can be changed using the SuperBASIC procedure

[line number] BAUD baud_rate

where **baud_rate** may be 75, 300, 600, 1200, 2400, 4600, 9600 or 19200

When receiving at 9600 baud, at least two stop bits must be sent by the transmitting device. Receiving at 19200 baud is not yet possible.

CONSOLE DEVICES

A CONsole device defines a window on the screen with an attached keyboard queue. Each CONsole window effectively acts as an independent terminal. Channels 0, 1 and 2 are all defined as CONsole devices. The form of a CONsole device definition is

CON [size of window] [position of window] [length of keyboard queue]

where the optional parameters are defined as follows :

The **size of the window** is

_ width of window in pixels X height of window in pixels

the default being

_448x180

A monitor screen is approximately 512 pixels across and 256 pixels deep. A TV screen is approximately 448 pixels wide and 240 pixels deep. Thus the 32 pixels at each side of the TV screen are invisible, as are the 16 pixels at the top.

The **position of window** is the pixel coordinate position of the **top left-hand** corner of the window. Coordinates start at (0,0) at the top left-hand corner of the **monitor** screen, the x-coordinate increasing towards the

right-hand edge and the y-coordinate increasing downwards towards the bottom of the screen. The position option has the form

A x-coordinate X y-coordinate

the default being

a32x16

The length of keyboard queue is simply the number of characters which can be accumulated in the input buffer for the window. Its form is

_ number of characters

and the default is

_128

So CON defaults to CON_448x180a32x16_128

CON_50	CON_50x180a32x16_128
CONx20	CON_448x20a32x16_128
CON_x30	CON_448x30a32x16_128
CONa100	CON_448x180a100x16_128
CONax40	CON_448x180a32x40_128
CON_50	CON_448x180a32x16_50
CONa_60	CON_448x180a32x16_60

CONsole devices may be used in the same way as SERIAL devices, e.g.,

```
COPY mdv1_dat33 TO CON
OPEN #3,CON_50x50: INPUT #3,a$
```

If you have a printer attached to SER1, try

```
COPY CON TO SER
```

and anything which you now type at the keyboard is immediately re-routed to the printer until you BREAK (CTRL and space) out of it.

The input buffer on channel zero expands automatically when it is necessary. **There is therefore no need at all to re-open #0.** If you want to change the size and position of of the console channel, the WINDOW procedure, described in Chapter 10 on Screen Control, is perfectly safe.

SCREEN DEVICES

A SCReen device is exactly the same as a CONsole device except that there is no attached keyboard queue. It is an output window only. Setting up the device name uses the same form and defaults as for CON but the final '_length' is omitted,

SCR [size of window] [position of window]

where **size** and **position** are as defined for CONSOLE.

```
OPEN #4,SCR_100x100a224: PRINT #4,!table$!  
COPY mdv2_log TO SCRax180
```

The NETWORK

Another method of communicating with other QLs is to use the NETWORK. A set of QLs can be connected together via the NETWORK ports at the back of the machine (which are both alike), and each is given a different **NETWORK station number** to differentiate it from the others. When it is switched on, a QL is automatically assumed to be NETWORK station one. Obviously, if two are linked together, they cannot both be station one. To assign yourself a different station number, you simply enter

```
NET station_number
```

where **station_number** is evaluated to an integer which must fall in the range 1 to 64.

To send information to another QL, that QL must be expecting to receive it. You must therefore have a NETWORK Output channel to the second QL open, and that QL must have a NETWORK Input channel from you open.

To **set up** as station 4 and
send data to station 5

```
NET 4  
OPEN #3,NETO_5  
PRINT #3,'Hello station 5'
```

To **set up** as station 5 and
accept data from station 4

```
NET 5  
OPEN #6,NETI_4  
INPUT #6,a$: PRINT a$
```

So NETI_n means accept input from station N; NETO_n means send output to station N. It is a collaborative system, you cannot do a NETO if the other station hasn't done a NETI.

Any data sent is transmitted in blocks of 255 bytes. If the data is less than 255 bytes long it will not be sent immediately, but closing the channel causes any remaining data to be sent. If the NETWORK station on the receiving end of the block does not respond within 15 seconds, supposing the channel to have been closed, for example, then the transmission is aborted and control returned to the console.

The **above system** describes communications between **two** QLs only. A station may set itself up to receive input from **any** other network device by the following statement

OPEN_IN #channel,NETI_n

where N is **its own** station number. For example,

OPEN #6,NETI_5

in the example above, will **receive data** from any station transmitting to station 5 whenever it executes an INPUT #6 statement.

To **transmit** data to any station that may be **listening**, you have to open a channel on the **NETwork broadcasting** device. This is achieved by executing the statement

OPEN #channel,NETO_0

and then any number of stations can say

OPEN #channel, NETI_0

and ask for input on that channel. Zero is **always** the NETwork station used for broadcasting. A QL should not **define itself** to be station zero, just open a channel on the broadcast device.

Since the transmitting station can **send data** to any number of other stations, there is no handshaking **when broadcasting**. Thus, as there is no confirmation that data has been **received** it is possible to lose it. It is a sensible precaution, therefore, to **limit broadcasting** to less than 255 bytes at a time.

9 CREATING AND RUNNING PROGRAMS

Creating a program

Entering lines into a program is easy. You simply type the line of SuperBASIC prefixed by a line number and the editor inserts or replaces that line. If a line with no statements after it is entered, that line will be deleted from the program.

Changing the line that you are currently typing can be achieved by means of the left and right cursor keys. These are on the left-hand side of the space bar and are marked ← and →. Pressing the cursor-left key moves the cursor one character position to the left. The cursor continues to move for as long as the key is held down.

Deleting the character on the left of the cursor is done by pressing CTRL and ←; deleting the character on the right of the cursor by pressing CTRL and →. The character that the cursor is flashing over is always taken by SuperBASIC to be on the **right** of the cursor. New characters are inserted in front of this character. The line editor is always in **insert mode**. To overwrite a character, you have to position the cursor on top of it, press CTRL and →, then type in the new character.

To **enter** a line of SuperBASIC, the cursor does not have to be at the end of the line. The line being input is held in an input buffer which has an implicit line feed at the end of it. The cursor can therefore be anywhere on the line when the ENTER key is depressed and the whole input buffer will be accepted. The ENTER key is not the only one which will send the input buffer to SuperBASIC, the up-arrow and down-arrow keys (↑, ↓) on the right-hand side of the space bar will do so as well. They have a side-effect, however, which will be detailed later on in this chapter.

As soon as a line of SuperBASIC, whether preceded by a line number or not, has been entered, it is **parsed**. This means that it is checked for the correct syntax and converted to internal SuperBASIC tokens. If the line cannot be parsed, the error "bad line" will be printed on the console channel, channel zero, and the line will be echoed, leaving the cursor flashing after the last character. You can now edit the line and re-send it, or press CTRL and the space bar together in the BREAK sequence to clear the input buffer completely.

If you are **deleting** a large chunk of a program, there are more convenient

methods than by entering one line number at a time. DLINE deletes a set of SuperBASIC lines. The definition is

```
DLINE range of lines {,range of lines}
```

where **range** is

```
[line_number] [ TO [line_number] ]
```

For example,

```
DLINE 1 TO 10,30,100 TO 200
```

would delete lines 1 to 10 inclusive, 100 to 200 inclusive, and line 30.

If an empty range is given (e.g., DLINE ,,), no action is taken. Beware, though, DLINE TO is expanded to DLINE 1 TO 32767 so your whole program will vanish!

For entering a **substantial** portion of a program, the AUTOMATIC line number generator is most useful. Its definition is

```
AUTO [startline] [,increment ]
```

where **startline** is the line number at which you want to start;

and **increment** is the difference between subsequent line numbers.

If a starting line is not given, line number 100 is assumed; if no increment is given, a step of 10 is generated, e.g.,

```
AUTO          gives you lines 100,110,120,130,...  
AUTO 20       gives you lines 20,30,40,50,...  
AUTO ,3       gives you lines 100,103,106,109,...  
AUTO 400,5    gives you lines 400,405,410,415,..
```

If a line number does not exist already, then SuperBASIC prints it in the console window, channel zero, at the bottom of the screen, and leaves the cursor flashing ready for you to enter the body of the line. When you have finished the line, press the ENTER key. The line will be absorbed into the current program and SuperBASIC will give you the next line number in the sequence.

Should a line in the sequence already exist, the whole line will be printed in the console window and the cursor left flashing at the end of it. This allows you to edit the line if you wish or just press ENTER to move on to the next line.

The lines are treated exactly as if you had entered the line numbers manually. If there is no text after a line number, that line will not be included in the program but AUTO will continue to generate the next. If you delete all the characters except for the line number, the line will be deleted from the program and AUTO continues.

To stop the automatic line number generation completely, press CTRL and the space bar at the same time. This is the BREAK sequence which has been referred to before. SuperBASIC then ignores whatever is in the input buffer and turns off the AUTO command.

As was mentioned earlier, the up and down arrows can be used to send the line instead of the ENTER key. If the down arrow is used, after the input buffer has been passed to the editor, the next line in the program is written to the console and waits to be edited. The AUTO sequence will now continue from this line. So given the initial program

```
100 a=1
120 b=2
140 c=3
160 d=4
```

and the command

```
AUTO 120,5
```

the first line to appear is

```
120 b = 2
```

If the ENTER key is pressed, the next line is

```
125
```

Suppose you type E=5 ↓ . This will insert the whole line 125 e=5 and produces the next existing line

```
140 c=3
```

An ENTER here gets you

```
145
```

and so the sequence continues. The use of the up arrow is similar; after interpreting the input buffer, the previous line is recalled. So, continuing the example above

```
145 f=6 ↑
```

inserts line 145 and writes

```
140 c=3
```

Another up-arrow here produces

```
125 e=5
```

whereupon ENTER gives

that being the current line plus the current increment.

An up arrow terminates AUTO if it would take you beyond the top of the program. A down arrow beyond the end of the program, however, continues to increment for new lines. The up and down arrows may be used outside of the AUTO command, where they produce the previous or next existing lines as normal. In this case, if the down arrow is used on the last line of the program, that **same** line is repeated, because the current increment is zero.

There is another command, EDIT, which is often of use. It is defined in exactly the same way as AUTO,

```
EDIT [startline] [,increment]
```

with a default **startline** of 100 but the default **increment** is zero. If you only want to edit one line, it is much easier to say

```
EDIT line_number
```

make the change and be finished, than to say

```
AUTO line_number
```

make the change, wait for the next line to be printed and then BREAK out of it. Even remembering to put ',0' after an AUTO is a nuisance, so EDIT is very useful.

Listing programs

Displaying the program currently in memory is done by using the LIST command. This has the form

```
LIST [#channel,] range of lines {,range of lines}
```

where **channel** is the place to write the program lines to

and **range of lines** is

```
[line number] [ TO [line number] ]
```

```
LIST alone or LIST TO , translates to LIST 1 TO 32767
LIST line TO line       lists from startline to endline inclusive
LIST line                just lists the single line
```

If a channel number is not given, the program listing window, #2, is used.

LISTing can only be done on the program currently in memory. It takes the

internal SuperBASIC tokens and detokenizes them, one at a time, into the print buffer. When a line feed token is reached and converted, the whole line is written to the channel at the next print position.

When a line number token is reached, LIST checks to see whether it is past the endline of the current range. If this is the case, LIST looks for the next range to print. When listing a new range, all the lines from the top of the program down to the first line of the range are skipped, making the LIST command reasonably quick, whichever set of lines is required.

If you would like a hard-copy of your program, connect a printer to SERIAL port 1 and say

```
OPEN #3,SER: LIST #3: CLOSE #3
```

or whichever channel number is most convenient.

The last range of lines printed to the listing window, #2, is stored in a table. This set is called the **list range** and is made use of whenever a line of the current program is changed. If the altered line is in the list range, i.e., displayed on the screen, or within one line either side of this range, then the set of lines is relisted in the window to reflect the change. If the line is not in the list range, the window is not redrawn. Thus if you want to see the lines you are changing updated, you must make sure that they are in the list window to start with.

Taking an example. Suppose you have a program starting with line 100 and rising in steps of 10. When you LIST 100 TO 350, some lines will scroll off the top of the window. Say you are left with the line at the top of your window being 160, and the line at the bottom being 350. There are two other lines which are of interest; the line before the top one displayed and the line after the bottom one displayed. These are called the invisible top line and the invisible base line.

```
150 invisible top
-----
160 top
170
... the list range          window
...
350 base
-----
360 invisible base
```

Now if you **edit**, say, line 250, the range 160 to 350 will be relisted on the screen to show the change. If you alter line 100, the line in the program will change, but it will not be visible because it is outside the list range.

Inserting a line inside the list range, 255 say, will cause the lines following it to be scrolled down, so the new range is 160 to 340.

Deleting a line inside the list range, 270 say, will cause the lines

following it to be scrolled up, so the list range now includes the previous invisible base line.

Inserting a line between the top and the invisible top line will cause the list window to be redrawn to include it, as will changing the invisible top line. Similarly, inserting a line between the base and invisible base line or changing the invisible base line redraws the screen.

Now, the list window might not be full when relisting is to be done. If that is the case, printing continues until the window is full. This naturally increases the list range. Since the LIST command writes the lines at the next print position, scrolling when the bottom of the window is reached, SuperBASIC has no idea **where** on the window the lines are. Relisting is therefore **always** done from the top left-hand corner and the remainder, if any, of the window is always cleared to avoid confusion.

Renumbering a program

It is obviously necessary to be able to renumber the lines in a program. At the same time, there ought to be the capability of only renumbering certain parts of programs if you so wish. To this end, the SuperBASIC RENUMBER command was designed in the following form

```
RENUM [top_oldline [TO [base_oldline] ] ; ] [new_startline] [,increment]
```

If none of the optional parameters are given

```
RENUM 1 TO 32767;100,10
```

is assumed. Thus,

```
RENUM 200      means renumber everything starting at 200 in steps of 10
RENUM ,30     means renumber everything starting at 100 in steps of 30
RENUM 150;    means renumber from 150 onwards, starting at 100 step 10
RENUM TO 300; means renumber up to line 300 inclusive, from 100 step 10
```

This format makes the norm (renumbering everything) simple to use, but is flexible without being confusing.

Renumbering will relist the lines in the list window if they have changed.

There are a few rules associated with renumbering.

You cannot renumber beyond 32767 or before 1.

You cannot renumber a range if, by doing so, you would take it out of position. For example, given lines

```
100,200,300,400
```

you cannot

```
RENUM 200 TO 300;500
```

because that would give lines

```
100,500,510,400
```

which is not allowed. Renumbering out of range is not allowed for two reasons. First, it dramatically increases the possibility of error; wiping out portions of the program, for example, would be ridiculously easy. Secondly, even assuming that the command given was always correct, the length of time that rearranging the lines in the program would take would be disproportionate to its usefulness.

In order to know when lines would be renumbered out of sequence, the actual line before the first one in the range and the actual line after the last one in the range are also included in the renumber set, being 'renumbered' to their existing values. A table is created in the vv area to hold all the line numbers in the range and what they will be changed to. The form of this table is

size of table	(4 bytes)
{ current line number value	(2 bytes) }
{ new line number value	(2 bytes) }

RENUM then goes through all the lines in the range, filling in the new line number in the table and editing the program line to reflect it at the same time.

The next stage is to scan through the whole program, looking for any line number expressions after GOTO, GOSUB and RESTORE. In any such line number expression, it is only the first term, if that is a floating point constant, which will be renumbered.

If a line number is less than or equal to the first one in the table, it is ignored. Similarly, if a line number is greater than or equal to the last one in the table, that too is ignored. If a line number is within the range, however, the renumber table is scanned until a line number is found which is greater than or equal to the target. The line number expression within the program is then updated to the new value given in the table, and the search through the program continues.

RENUM can only deal with GOTO, GOSUB and RESTORE because these are SuperBASIC **keywords** and as such have an unchanging, easily identifiable internal token. Other commands which have line number parameters are **procedures** which cannot be singled out at all easily. To do it, all the procedures which might take line numbers would have to have their names set aside in a separate list, then each and every name token in the program would have to be detokenized and checked against the list. Again, could any parameter be a line number, or would the particular one have to be specified? RENUM would be very slow indeed.

As an example of all this, consider what happens when the skeleton program below is changed by the statement

RENUM 140 TO 300; 200, 2

original program	after RENUM	reason
120 GO TO 305 130	120 GO TO 310 130	(1)
140 150 160 GOSUB 218 170 180 m = 200 195	200 202 204 GOSUB 216 206 208 m = 200 210	(2)
200 RESTORE 132 210 220	212 RESTORE 200 214 216	(3)
300 ON x GOTO 180+k, nlin+200,110,200,m	218 ON x GOTO 208+k, nlin+200,110,212,m	(4)
310	310	

- (1) 305 is between 300 and 310
- (2) 218 is between 210 and 220, 220 has become 216
- (3) 132 is between 130 and 140, 140 has become 200
- (4) 180 has become 208; only the first term of a line number expression is changed; 110 is not in the renumber range; 200 has become 212; only floating point constants can be changed

RENUM will renumber GOTO, GOSUB and RESTORE at the time. It will renumber the line number associated with a DEFinition line at the time when the next line is executed, i.e., when the next namepass is done. It **will not** renumber the line numbers associated with REPEAT or FOR indexes. Thus

```
start=100: REPEAT ren: RENUM start: start = start+100
```

will be perfectly safe if entered as a direct console command (though it is a little single minded and might not let you BREAK for a while), but the same line within a program is likely to become remarkably confused.

Saving a program

It is clear that there needs to be some method of saving a program onto a Microdrive medium. SAVE takes your current program and does just that. The formal definition is

SAVE name {,range of lines}

where **name** is MDVn_filename or a peripheral device

and **range of lines** is [start_line] TO [end_line] as defined for LIST.

If no ranges are given, then the whole program will be saved with the specified name onto the tape in the given Microdrive. The filename chosen must not already exist on that tape.

SuperBASIC takes the internal program, detokenizes it as it does for LIST, and writes it out in hexadecimal ASCII.

Restoring a program

There are two methods of retrieving a program from Microdrive,

LOAD name and MERGE name

where **name** in both cases is MDVn_filename or a peripheral device.

The SuperBASIC **LOAD** procedure first locates the file on the given device. A "not found" error is generated if it doesn't exist, otherwise the current program, if any, is deleted and the new one read in line by line. The process is exactly the same as if you were typing the lines in at the keyboard (but less strenuous). Each line is accepted, parsed, converted to internal tokens and stored in memory in the program file area.

The only difference will be noticed in the case where a line will not parse properly. To facilitate entry of the rest of the file, an error is not generated but the **MISTAKE** keyword is inserted into the line, just after the offending line number. Since it won't parse, the rest of the line is treated as text (similar to **REMark**). If a line beginning with **MISTake** is executed, the error "bad line" will appear at the console and the program will halt.

MERGE is similar in method to **LOAD**, but it does not clear out the current program and variables before commencing to read the new one. This means that each line will be inserted into the original program if it has a new line number, but will replace a line with a matching line number. The **MERGE** facility is very useful for incorporating standard procedures into a program.

Neither of the above commands executes any part of the **program**, they merely read in the ASCII lines and convert them into SuperBASIC tokens.

If the lines of SuperBASIC do not have line numbers, then they will be executed as if they had been typed directly at the keyboard. For example, try this exercise, line by line,

```

OPEN_NEW #3,mdv1_loadtest
PRINT #3,'a=1'\b=2'\print a,b'
CLOSE #3
LOAD mdv1_loadtest

```

You should have the values 1 and 2 displayed in the execution window. To see what the file looks like, enter

```
COPY mdv1_loadtest TO SCR
```

If you have a non-line-numbered file like this which you want to read in as a line-numbered SuperBASIC program, you can say,

```
AUTO
```

at the keyboard, then delete the line number which is printed at the console (don't BREAK) and enter

```
LOAD filename
```

AUTO will obligingly put a line number in front of every line. Try it with the above file like so,

```
AUTO
```

delete four characters to the left and type

```

LOAD mdv1_loadtest
CTRL and space      .... to BREAK out of AUTO
LIST

```

Lo and behold!

You can also achieve the same effect by having AUTO as the first line of your non-line-numbered file, for example ,

```

OPEN #3,mdv1_loadtest : OPEN_NEW #4,mdv1_autotest
PRINT #4,'auto'
REPEAT p : INPUT #3,a$ : PRINT #4,a$
CLOSE #3 : CLOSE #4
LOAD mdv1_autotest
CTRL and space      ....to BREAK out of AUTO
LIST

```

Clever innit?

Clearing out a program

NEW is the command for clearing out most of the SuperBASIC storage area. There are no parameters, so its form is simply

NEW

and on execution the current program will disappear, the vv area is all released and the nametable is left with only machine code procedures and functions. In addition, the arithmetic stack, return stack, data status and line number table are cleared and all channels apart from 0, 1 and 2 are closed. These default channels are not reset. If you do not want to take such drastic measures, then

CLEAR

just attacks the vv area, return stack, data status and RI stack. It does, however, go through the current program on a name-search pass resetting all the nametypes.

Because NEW clears out everything that it can find, any other statements following NEW on the same line will be destroyed as well.

Running a program

So far, we have been running programs by defining them as procedures and then entering the procedure names to execute them. If you have not set your program up as a procedure, then you can use the RUN command instead. The formal definition is

RUN [line number]

where **line number** is an expression giving the line number at which you would like execution to start. If this is omitted, execution starts from the top of the program.

Because you will frequently want to run a program with different sets of data, the values of the variables used and the current DATA item pointer **are not reset** on RUN.

To load a program from Microdrive and execute it immediately, you can say

LRUN name

where **name** is MDVn_filename or a peripheral device as defined for LOAD. This is exactly equivalent to the two commands "LOAD name" and "RUN". There is a similar facility for MERGE. Its form is

[line number] MRUN name

and, when used as a direct command, it first merges the named program and then executes from the top of the program. When used within a program, it merges the named program and then continues execution from the line after the MRUN command. Thus MERGE and MRUN, when used inside a program, have the same effect.

If you have a program called `BOOT` on the medium in `MDV1_`, then on starting up your QL, SuperBASIC performs an `LRUN` on it automatically. It is exactly the same as if you reset and then entered the command,

```
LRUN mdv1_boot
```

Although it is possible to incorporate `LOAD` and `LRUN` into programs, you should remember that they destroy the original code before reading in the new, so that there is no way of returning to the initiating program. Both of them actually do a `NEW` before they start reading. This also means that you cannot put any statements after `LOAD` and `LRUN` and expect them to be done.

In other words,

```
LOAD name : LIST
```

will not write out the program because any statements after `LOAD` are lost, but

```
MERGE name : LIST
```

will display the whole program after the new lines have been read in. You should also note that if you `LOAD` a file as a direct command from another file currently being `LOADed`, it will work, but it will not return to the original file when the second one has finished.

Passing programs between QLs

It is possible to `LOAD` and `SAVE` programs across QLs. If two QLs are connected together by an RS-232 cable plugged into the `SER1` port of one machine and the `SER2` port of the other, you can say

```
SAVE SERz
```

on the first and

```
LOAD SER2z
```

on the second, and the current program will be transferred across. The `Z` option is necessary to mark the end of the text file.

The current program can also be transferred across network lines.

```
NET 5 : SAVE NETO_6
```

on one QL and

```
NET 6 : LOAD NETI_5
```

on the other one does the same job.

The actual ranges of lines required can be specified on this sort of SAVE command in the same way as they can on a normal SAVE.

MERGE, MRUN and LRUN can also be used across the SERIAL ports and the NETWORK in the same way as described here for LOAD. It should be noted that they will all only work on the program currently in memory.

Stopping execution

When SuperBASIC reaches the bottom of the program, execution will stop of its own accord. To stop it earlier than that, use the procedure

```
[line number] STOP
```

anywhere in the program. Execution will halt and the cursor will flash in the console window. If you find that you want to carry on from where you left off, enter

```
CONTINUE
```

and execution will recommence from the statement after the STOP.

The way that SuperBASIC achieves this is to keep the **continuation status** in its storage area. The information held is

line number halted at	(2 bytes)
statement on that line	(1 byte)
whether inline or not	(1 byte)
index if an inline loop	(2 bytes)

As this information is updated every time that a halt is made, all that CONTINUE has to do is to retrieve it, go to the line number and statement given, and carry on from the statement after that.

You can also CONTINUE after BREAKing (CTRL and space) into a program and getting the message "not complete".

When a program is renumbered, the continuation status data remains unchanged, so the line number contained therein might start you off somewhere unexpected if you do a CONTINUE.

You cannot CONTINUE after a halt has been made while processing a direct command, i.e., a line that you have entered from the keyboard rather than one which is embedded in a program. The reason is obvious when you think about it, because processing any new direct command (i.e., the CONTINUE), overwrites both the input buffer and the internal tokens of the previous command line.

RECOVERING AFTER ERRORS

When an error is generated, an error message is printed out on the console window, together with the line number where it occurred, and execution stops. SuperBASIC treats this as a program halt and updates the continue status accordingly so, to carry on after an error, you can simply enter CONTINUE as before. If you are able to make a correction, you may wish to re-execute the line. This can be achieved by using

RETRY

which retrieves the continue status, decrements the statement number, and does a CONTINUE.

For example, suppose you have a small procedure

```
100 DEFine PROCedure ex4(x,y)
120 x = x^2+4*y+16
140 END DEFine
```

and you have mistyped the calling line, for example

```
q=4: r=5: ex4 q
```

As you haven't provided a second argument, an unset expression will be generated for Y. The expression evaluator therefore cannot calculate a result at line 120, so the message

At line 120 error in expression

is printed at the console and execution stops.

You now have the chance to, say, LIST 120 to see what the erring statement contains and you can then enter

```
y=5: RETRY
```

to set the value of Y and re-evaluate the expression.

If an error occurs inside a procedure or function and you do not CONTINUE or RETRY, you will occasionally get the message

PROC/FN cleared

This means that the return stack has been cleared and it happens when you CLEAR, RUN, edit the program or put a new name into the namelist. It means that you can no longer CONTINUE from inside that procedure or function.

If an error occurs while processing a direct command line, you will not be able to RETRY it since parsing the statement overwrites the buffer.

10 SCREEN CONTROL

Screen modes

There are two QL screen modes; high resolution and low resolution. At high resolution, 512 separate pixels can be identified across the screen; in low resolution mode the pixels are taken in pairs so that a low resolution character is twice the width of a high resolution character. At low resolution, often called 256 mode, the full eight colours can be used, but high resolution, or 512 mode, only supports four colours. This is because the double width is necessary to display all the physical colours, one pixel in each pair being used for red and green, and its partner taking care of blue and flash.

To change display modes, use the procedure

```
MODE mode_number
```

where **mode_number** is 8 or 256 for low resolution mode
and 4 or 512 for high resolution mode

This command not only changes modes, it also clears all the windows on the screen, resetting them to their **current** size and colour. MODE resets any printing characteristics which may have been changed.

Windows

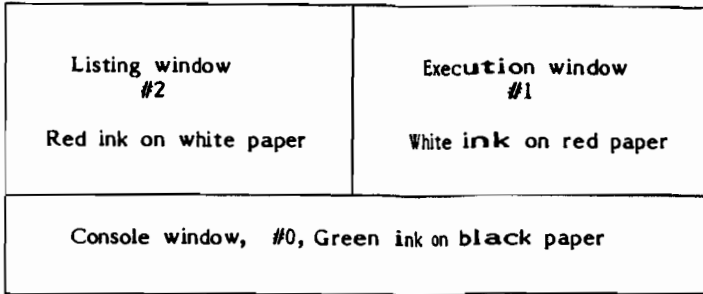
Windows are an important feature of the Sinclair QL. The only way to write or draw anything to the screen is to direct the output to a window.

When you switch on, you are asked to choose whether you want a monitor display or a TV display. The reason that the displays are different is that a television overscans on both the vertical edges and on the top of the screen, so pixels in those areas are lost. In addition, the convex rounded corners produce a lot of distortion so need to be avoided.

The television option automatically selects low resolution mode whereas the monitor option automatically selects high resolution mode. You can, of course change these by use of the MODE command once the initial selection

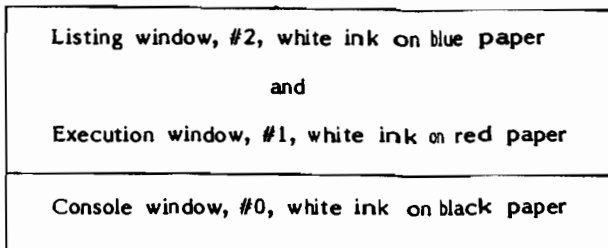
has been made.

Choosing the monitor option, you are immediately confronted with three distinct windows on the screen,



Both the execution and listing windows are surrounded by a black and white checkered border. You can fit 20 lines into each of them with 42 monitor-sized characters on a line. The console has room for 5 lines with 85 characters to a line.

For a television display, all the characters are printed as double the width of a high resolution character, so it is more readable to have the listing and execution windows running the whole width of the screen. To do this, these two windows are laid on top on one another, so that there are still three windows even though it only looks like two.



Both the listing and execution windows are 37 characters wide and 20 lines deep. The console window is 37 characters wide and 4 lines deep.

The windows are actually defined in terms of pixels. A monitor screen is 512 pixels wide and 256 pixels deep whereas a television screen is 448 pixels wide and 240 pixels deep. The 32 pixels on either side of the television screen are virtually invisible, as are the 16 pixels at the top. The relative positions of the screen windows to one another are shown in Fig. 10.1.

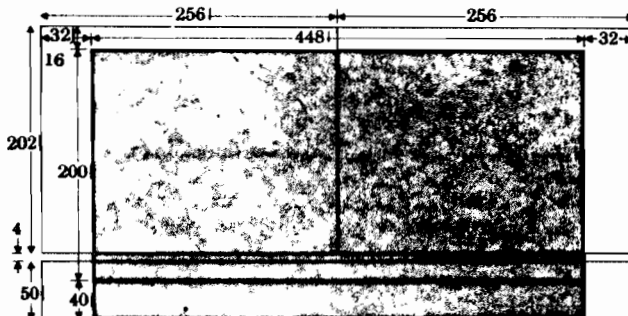


Figure 10.1

DEFINING A WINDOW

Redefining the window for a channel is achieved by use of the procedure

```
[Ino] WINDOW [#channel,] size of window, position of window
```

where **size of window** is

width of window in pixels , height of window in pixels

and **position of window** is given by the x and y pixel coordinates of the top left-hand corner of the window

number of pixels along , number of pixels down

All pixel counting starts at the top left-hand corner of the monitor screen, so remember to add 32 to the x-coordinate and 16 to the y-coordinate if you are using a television.

The channel used must already have been opened.

```
OPEN #3,CON_30x50a100x100
```

is the same as

```
OPEN #3,CON : WINDOW #3,30,50,100,100
```

```
OPEN #4,SCR_60x60a200x100
```

is equivalent to

```
OPEN #4,SCR : WINDOW #4,60,60,200,100
```

or just

```
WINDOW 400,200,32,16
```

to change the size of the execution window.

OVERLAYING WINDOWS

The positioning of channels 1 and 2 on the initial television display may cause some confusion at first. If you enter something like

```
100 PRINT 'Hello world'
```

the line is entered into a program and is printed in white on a completely blue background. If you now enter

```
PRINT 'hello\'','hello\'','hello'
```

the words are written in white on a red background on top of what was there before. If you now LIST, the second 'hello' will be overwritten. This is because the two windows are truly independent and each has its own pointer to where the next character in that window should go.

LIST always fills in the rest of the line with the paper colour to make it absolutely clear what a program line contains. When something is PRINTed to a window, though, only that section of the window is affected. The red paper behind 'hello' stops after the final character, it is not continued to the end of the line. This is one of the things that makes screen handling so quick. It also makes it easier to highlight your screen output without having to put up with irritating bands of colour.

There are times, though, when you don't want this irregular appearance. In that case you can completely cover the window with the colour of the paper first. The simple form of the clear screen command is

```
[line number] CLS [#channel]
```

where **channel** is the channel attached to the window. If it is omitted, the execution window will be cleared.

So, on a TV screen,

```
CLS or CLS #1
```

fills the upper window with red. When you now PRINT something, the background colours blend in perfectly.

The colours

BIT PATTERNS

The colour of each pixel in the window is a combination of red, blue and green. The numerical value of it can be calculated consistently if we use certain bit values for each of these colours and set the bits on or off depending upon whether the primary colour is a component of the final colour. The bits used are

bit 0	(value 1)	blue
bit 1	(value 2)	red
bit 2	(value 4)	green

so the full range of colour values is

000	no colour	... B	black	0
001	blue	... L	blue	1
010	red	... R	red	2
011	blue+red	... M	magenta	3
100	green	... G	green	4
101	green+blue	... C	cyan	5
110	green+red	... Y	yellow	6
111	green+red+blue	... W	white	7

In four colour mode, the colour values are

0,1	... black
2,3	... red
4,5	... green
6,7	... white

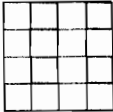
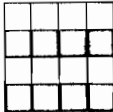
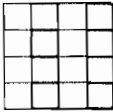

COMPONENTS

A QDOS colour is made up of three components:

a **main colour** whose numerical value is one of those defined above;

a **contrast colour** also of numerical value 0 to 7;

the **pattern** in which the two are combined. There are four patterns available, defined by

- 0 dots :  the top right-hand pixel in a block of four is contrast
- 1 horizontal stripes  the bottom two pixels in a block of four are contrast
- 2 vertical stripes  the two right-hand pixels in a block of four are contrast
- 3 checker board  the top right- and bottom left-hand pixels in a block of four are contrast

When you are asked for a **colour** parameter to a SuperBASIC procedure, you can actually give,

```
main [,contrast [,pattern] ]
```

where the default **contrast** colour is the same as the main colour and the default **pattern**, or **stipple** effect, is a checkerboard.

COMPOSITE

In order to pass the colour information to QDOS, SuperBASIC has to combine the three components into one **composite** colour. This composite colour is one byte long :

bit 0	value	1	bit pattern of main colour
1		2	
2		4	

How to get the colours onto the screen

When you open a new CONsole or SCReen channel, the window defaults to black paper and green ink. To change the colour of the **paper** associated with a window, use

```
[line number] PAPER [#channel,] colour
```

where the **channel** defaults to the execution screen and the **colour** may be main [,contrast [,stipple]] or a composite colour as defined above. You won't notice any change in the colour until you write something to the window.

The colour of the **ink** with which you write can be changed by the command

```
[line number] INK [#channel,] colour
```

where channel and colour are as above. Be wary of using the same colour for your ink as you have on the background!

You can **highlight** your output by using the SuperBASIC procedure

```
[line number] STRIP [#channel,] colour
```

with channel and colour as before. This writes any characters, including spaces, which you now print on a different colour background to the paper. When a PAPER command is executed, the STRIP colour is automatically reset to the new colour of the paper.

Positioning the cursor without spacing to it can be done by using the procedure

```
[line number] AT [#channel,] row_number , column_number
```

where **channel** is the window in which to change the print position, if omitted it will default to the execution window, #1;

row_number is the number of lines down the window, taking 0 as the top line and not going past the bottom line;

column_number is the number of character positions along the window, taking 0 as the left-hand edge and not going past the right-hand edge.

For example,

```
REPeat tick_tock: AT 0,0: PRINT DATE$
```

shows the time ticking away at the top left hand corner of the execution window. Or try this on a television display,

```

100 DEFine PROCedure pretty
110 REPEAT ugh
120   row = RND(0 TO 19)
130   column = RND(0 TO 36)
140   colour = RND(0 TO 255)
150   AT row,column : STRIP colour : PRINT " "
160 END REPEAT ugh
170 END DEFine pretty

```

It eventually covers the window with a kaleidoscope of colour. Put in

```
102 PAPER 2: CLS: FOR i=1 to 295: PRINT i;
```

as well to see how long it takes to cover every position.

OVERWRITING

If you do not want a solid strip of colour at the back of your characters, you can make use of the different **overwriting** modes. The procedure call is

```
[line number] OVER [#channel,] overwriting_mode
```

where **channel** is the window in which to change the overwriting characteristic, default 1;

and the **overwriting_mode** is 0, 1 or -1 as defined below,

OVER 0 is the normal overwriting mode where characters are written in the current INK colour on the current STRIP colour background;

OVER 1 writes characters in the current ink colour but on a transparent strip;

OVER -1 also has a transparent strip but, as the characters are printed, the colour of the ink is XORed with the colour of the existing background for each pixel written.

The overwriting mode remains in force until it is reset. MODE will reset the overwriting characteristic to zero.

To illustrate the different effects, run the following procedure in eight colour mode :

```

100 DEFine PROCedure print_over
110 PAPER 1: CLS
120 PAPER 7: FOR i = 0 TO 19: PRINT ,\
130 STRIP 6: INK 3
140 AT 0,0
150 FOR i = 0,1,-1: OVER i: PRINT 'aaaaaaaaaaa'
160 END DEFine

```


Line 110 covers the window in blue;
 line 120 runs a broad white band down the left-hand side;
 line 130 sets a yellow STRIP with magenta INK;
 line 140 resets the print position;
 line 150 prints a row of 'a's in the three overwriting modes;

The row written with OVER 0 has been printed in magenta ink on a yellow strip clean across the the white and blue area.

The row written with OVER 1 has been printed in magenta ink on a transparent strip so that the white and blue are clearly seen.

The row written with OVER -1 also has a transparent strip but the ink colour on the white background is given by magenta XOR white = $011 \wedge 111 = 100 = \text{green}$. The ink colour on the blue background is magenta XOR blue = $011 \wedge 001 = 010 = \text{red}$.

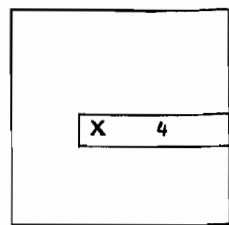
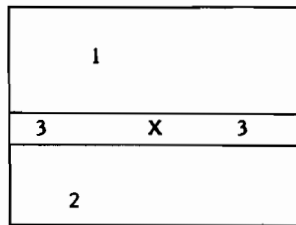
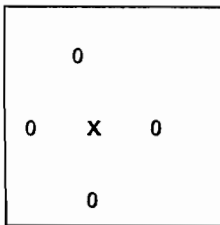
AREAS OF A WINDOW

A window is defined as having five areas. They are :

- 0 - the whole window
- 1 - above the cursor line
- 2 - below the cursor line
- 3 - the whole cursor line
- 4 - from the cursor position to the right-hand end of the cursor line

where the cursor line is the row of the window in which the current cursor position is.

If X marks the position of the cursor, the area codes are



If you only want to clear one of these areas, the area code can be incorporated into the CLS parameter list. The formal definition is

[line number] CLS [#channel,] [area of window]

where **channel** is the window to clear, default #1;

and **area** is 0 to 4 as defined above, the default being zero, the whole

window.

If any area except zero is specified, the new cursor position after clearing the window area is at the **beginning** of the **next** line after the current cursor position.

For example, enter each line in turn at the console to see the relevant areas being cleared :

```
PAPER 2: CLS : FOR i=1 TO 245: PRINT i;  
STRIP 0: AT 10,16: PRINT 'abcdef'  
CLS 1: PRINT ',,xx'  
CLS 4: PRINT 't'  
CLS 2: PRINT 'again'  
CLS 3: PRINT 'fin'
```

Defining a border

To make windows stand out, it is possible to specify the edge of them in a different colour. The BORDER command is of the form

```
[line number] BORDER [#channel,] [depth [,colour] ]
```

where **channel** is the window around which the border is to go;

depth is the size of the border in pixels. The number of pixels specified will be doubled at the vertical edges;

and **colour** may be main [,contrast [,pattern]] or a composite.

If the channel number is omitted, then the execution window, #1, is assumed. If the depth is not given, then the border no longer exists. If a colour is not specified, then the border is deemed to be transparent.

A border is always drawn **inside** the window. As soon as the command is executed, the border is drawn and, if its width has changed, the cursor is set to the new character position (0,0) just inside the border. If the border will not fit inside the window, "out of range" will be generated.

To illustrate the transparent border, try the following example in eight colour mode:

```
100 DEFine PROCEDURE bord  
110 FOR k=1 TO 10  
120   BORDER k*5  
130   CLS  
140   FOR j=1 TO 5  
150     FOR i=1 TO 5: PRINT i;  
160     PRINT  
170   END FOR j
```

```
180 END FOR k
190 END DEFine
```

You will see that the border, though transparent, is definitely in existence. To see the sequence of events more clearly, append

```
,k
```

to line 120 and run the procedure again. For a better effect, delete line 130, which clears the screen, and rewrite line 110 as

```
110 FOR k=10 TO 1 STEP -1
```

leaving line 120 as

```
120 BORDER k*5,k
```

Now rerun the procedure. Pretty, isn't it ?

REFERENCE POINTS

When defining windows, all the pixel coordinates have to be given relative to the top left-hand corner of the monitor screen because there is no other frame of reference available. It would be nonsensical to continue to use that reference position after a window has been defined, since the whole point of using a window as a framework for your output would be lost.

Within a window, therefore, all pixel coordinates and character positioning are relative to the top left-hand corner of that window. A border is defined as running round the inside edge of a window so, if a border has been specified, the top left-hand pixel available for output on the window changes again.

Blocks of colour

For solid areas of colour within a window, the SuperBASIC BLOCK command is defined as,

```
[line number] BLOCK [#channel,] size of block, position of block, colour
```

where **channel** is the window in which to draw the block, default 1;

block size must fit inside the window and is

width of block in pixels , height of block in pixels

block position is the pixel coordinates of the top left-hand corner

of the block relative to the top left-hand corner of the window :

number of pixels along , number of pixels down

colour is the colour of the block. This may be

main [,contrast [stipple]] or a composite colour

If the overwriting mode is -1, then the block colour will be XORed with the existing colour for each pixel in the block. If there is a border on the window, blocks are positioned relative to inside the border. Try the following, one step at a time,

```
CLS                to set the execution window all red
BLOCK 100,100,0,0,4  draws a green block in the top left-hand corner
BORDER 10          sets a transparent border inside the window
BLOCK 100,100,0,0,1  draws a blue block in the new top left-hand corner
```

Using blocks is a very fast way of drawing horizontal or vertical lines on the screen. For example, you could say,

```
100 DEFine PROCedure sql
110 OVER 0: PAPER 7: CLS
120 BLOCK 100,2,20,20,1
130 BLOCK 2,50,120,20,2
140 BLOCK 100,2,20,70,3
150 BLOCK 2,50,20,20,4
200 END DEFine
```

then run SQL for a fast multi-coloured square. To really appreciate the power of the screen driver, amend the procedure to,

```
100 DEFine PROCedure sql
110 OVER 0: PAPER 7: CLS
114 FOR i=0 TO 445
120  BLOCK 100,2,20+i,20+i,1
130  BLOCK 2,50,120+i,20+i,2
140  BLOCK 100,2,20+i,70+i,3
150  BLOCK 2,50,20+i,20+i,4
160 END FOR i
200 END DEFine
```

and now run it!

If OVER -1 is specified when you execute a BLOCK command, the block colour is XORed with what is already there. Add the following line to your procedure,

```
180 OVER -1: FOR i=0 TO 7: BLOCK 154,104,16,16,i: PAUSE
```

and run it again. Whenever the execution seems to have stopped, press any key on the keyboard for the next colour wash.

RECOLOURING

Using BLOCK with OVER -1 is a very fast way of recolouring part of a window, but the same relative colours will always be generated and working out what they will be is a little time consuming. If you want to reset any colour in a window to any other colour then you will have to use the RECOLour procedure. This takes the form

[line number] RECOL [#channel,] colour changes

where **channel** is the window to recolour, default 1;

and **colour changes** are eight numerical values defined as follows

the colour to change black	(0) pixels to
the colour to change blue	(1) pixels to
the colour to change red	(2) pixels to
the colour to change magenta	(3) pixels to
the colour to change green	(4) pixels to
the colour to change cyan	(5) pixels to
the colour to change yellow	(6) pixels to
the colour to change white	(7) pixels to

For example,

```
RECOL 3,7,5,0,6,4,4,4
```

would change the square printed in the above example to having a black top, white base, cyan left edge and yellow right edge, the whole on a green background.

Because it has to look up every pixel, RECOL is quite slow compared with blocking over, but on the other hand, you do know what to expect!

Changing the cursor position

If you are laying out a window, it is not always convenient to write the text at particular character positions as required by AT. You can instead define the next print position by using

[line number] CURSOR [#channel,] pixel position

where **channel** is the window to move the cursor around in, default 1;

and the **pixel position** is the x and y pixel coordinate pair of the top left-hand corner of the print position to move to. The coordinates are, as before; relative to the top left-hand corner of the window (minus any

border).

So, for example,

```
CURSOR #4,0,0
```

positions the cursor at the top left-hand corner of the window attached to channel 4;

```
CURSOR 224,100
```

puts the cursor in the middle of the default television screen execution window.

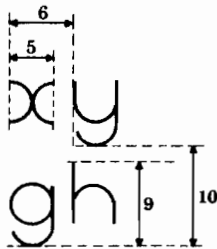
A nice little example is

```
100 CLS: OVER 0: FOR i=0 TO 200: CURSOR 2*i,i: PRINT i
```

Try running it - ahhh!

Changing the character size

The normal character size at high resolution is 5 pixels wide by 9 pixels high in a total space of 6 pixels by 10, i.e.,



Low resolution characters are double width, 10 pixels by 9 in a space of 12 by 10. The character size can be changed by

```
[line number] CSIZE [#channel,] width_mode , height_mode
```

where **channel** is the window in which to change the character size attribute, default 1;

and **width_mode** is

- 0 for single width (5) , 6 pixel spacing
- 1 for single width (5) , 8 pixel spacing
- 2 for double width (10) , 12 pixel spacing
- 3 for double width (10) , 16 pixel spacing

and **height_mode** is

- 0 for single height (9), 10 pixel line spacing
- 1 for double height (18), 20 pixel line spacing

All the widths may be specified in high resolution mode, but only the double widths may be used in low resolution mode. If 0 or 1 are used at low resolution, they will be converted to 2 and 3 respectively.

Changing the size of characters alters the character positioning which AT uses. MODE will reset character size to the default.

Moving the contents of windows

The contents of windows can be moved up and down or left and right. To move the contents up and down, use

```
[line number] SCROLL [#channel,] distance [,window area]
```

where **channel** refers to the window in which you want to scroll, default 1;

distance is the number of pixels to move the contents **down** by (so a negative distance moves the contents up);

- and **window area** is 0 for the whole window (default)
- 1 for the area above the cursor line
- 2 for the area below the cursor line

Vacated rows of pixels are filled with the current PAPER colour and, once the contents have been scrolled out of the top or bottom of a window, it is impossible to retrieve them.

To move left and right, use

```
[line number] PAN [#channel,] distance [,window area]
```

where **channel** is the window to pan in (default #1);

distance is the number of pixels to move the contents **right** by (so a negative distance moves the contents left);

- and **window area** is 0 for the whole window (default)
- 3 for all of cursor line
- 4 for right-hand side of cursor line, including the character at the cursor position.

The vacated pixels will be filled with the current PAPER colour and it should be noted that in low resolution, panning can be done only in steps of two.

Once anything has been panned past the edge of a window, it cannot be retrieved again.

To see what can be achieved with scroll and pan :

```
100 DEFine PROCedure roll
110 OPEN #3,SCR_40x120a40x120
120 OPEN #4,SCR_120x20a40x120
130 OPEN #5,SCR_40x120a120x120 : FOR i=3 TO 5: CLS #i
140 PRINT #3,'*'
150 FOR i=120 TO 130 STEP 2: SCROLL #3,2
160 FOR i=40 TO 120 STEP 2: PAN #4,2
170 FOR i=140 TO 230 STEP 2: SCROLL #5,2
180 FOR i=120 TO 140 STEP 2: PAN #5,2
190 FOR i=230 TO 130 STEP -2: SCROLL #5,-2
200 FOR i=140 TO 66 STEP -2: PAN #4,-2
210 FOR i=120 TO 20 STEP -2: SCROLL #3,-2
220 FOR i=66 TO 40 STEP -2: PAN #3,-2
230 CLOSE #3: CLOSE #4: CLOSE #5
240 END DEFine
```

Flashing text

In low resolution mode, characters can be flashed on and off. The command to use is

```
[line number] FLASH [#channel,] flash_mode
```

where **channel** is the window in which to change the flash attribute, the default being #1;

and **flash_mode** is 0 to turn flash off
1 to turn flash on.

Turning flash on and off does not affect anything that is already in the window. Switching flash on, means that any characters now printed will flash; switching flash off, means that anything printed from now on will not flash.

When a character is flashing, it is first written as it would normally be. Each row of pixels in the character square then alternates between its current form and the colour of the background. The background colour for a row of pixels is taken from the colour of the pixel at the left hand edge of the row. So if a character has been written on a bi-colour background (Fig. 10.7)



the character square will alternately flash between



which looks somewhat irregular. To see this, run the following procedure

```
100 DEFine PROCedure flash1
110 PAPER 2: INK 7: MODE 8
120 CSIZE 3,1: CLS
130 FOR i=0 TO 30 STEP 2: BLOCK 2,10,90+i,110+i,1
140 CURSOR 100,100: OVER 1: FLASH 1: PRINT 'a'
150 END DEFine
```

To prove that flash doesn't work in high resolution, change line 110 to MODE 4 and rerun the procedure. MODE resets the flash mode to 0.

Underlining text

Characters may be underlined using the procedure

```
[line number] UNDER [#channel,] underline_mode
```

where **channel** is the window in which to underline the characters, the default being #1 as usual;

and **underline_mode** is 0 to turn underlining off
1 to turn underlining on.

When UNDERlining is switched on, anything now printed will be underlined in the current INK colour. When underlining is switched off, no further text will be underlined. If FLASH 1 has been set, the underscores themselves do not flash. MODE resets the underlining attribute to off, e.g.,

```
100 DEFine PROCedure under_ex
110 PAPER 2: INK 7: MODE 8
120 PRINT 'Default is not underlined'\\
```

```
130 UNDER 1: PRINT 'This is underlined';
140 UNDER 0: PRINT '!\"but this isn't\"\\
150 UNDER 1: FLASH 1: PRINT \"Underlining doesn't flash\"
160 UNDER 0: FLASH 0: END DEFine
```

Redefining the default windows

It may be that while experimenting, all your windows have got themselves tangled up. This is a comprehensive procedure for resetting the default channels without having to reset the machine. Note that channel zero only has its window redefined, it is not re-opened. If you have been misguided enough to re-open #0, then there is no alternative but to reset the QL using the button on the right hand edge.

(1) on a monitor screen

```
100 DEFine PROCedure redef_mon
110 WINDOW #0,512,50,0,206
120 PAPER #0,0: INK #0,4: BORDER #0
130 OPEN #1,CON 256x202a256x0 128
140 PAPER #1,2: INK #1,7: BORDER #1,1,255
150 OPEN #2,CON 256x202a0x0 128
160 PAPER #2,7: INK #2,2: BORDER #2,1,255
170 MODE 4
180 END DEFine
```

(2) on a television screen

```
200 DEFine PROCedure redef_tv
210 WINDOW #0,448,40,32,216
220 PAPER #0,0: INK #0,7: BORDER #0
230 OPEN #1,CON 448x200a32x16 128
240 PAPER #1,2: INK #1,7: BORDER #1
250 OPEN #2,CON 448x200a32x16 128
260 PAPER #2,1: INK #2,7: BORDER #2
270 MODE 8
280 END DEFine
```

Animation

The screen organization on the QL is for high resolution graphics rather than for games. The emphasis is on scalable, off-window graphics, and cursor positioning of text rather than on individual pixel movement. SCROLL and PAN can be used for shifting the contents of windows, and you can redefine windows over and over again to give the illusion of movement. User-defined character generation is possible using a command in the Sinclair QL Toolkit.

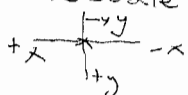
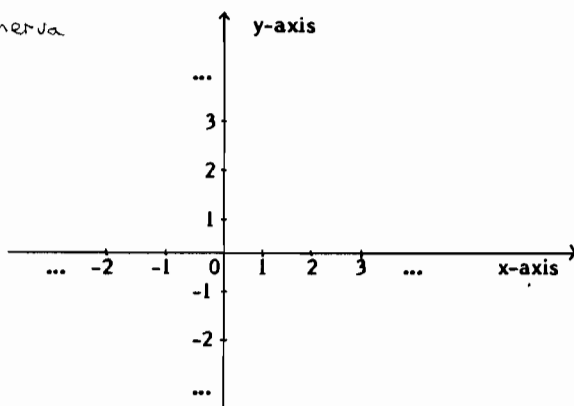
High quality animation is far better done neatly in machine code than messily in SuperBASIC. Colin Opie's book 'QL Assembly Language Programming', published by McGraw-Hill, describes animation techniques in detail. Since SuperBASIC is a totally extendible language, it is possible to create procedures of your own in machine code for animated graphics; the machine code interface is well defined and details can be found in the afore-mentioned 'QL Assembly Language Programming'.

11 THE GRAPHICS

The units

The QL SuperBASIC graphics system works in a conventional cartesian coordinate framework, with an x-axis and y-axis at right angles, x increasing regularly towards the right and y increasing steadily upwards.

Note - Minerva
-ve Scale

For every window, the state of the graphics initially is that the origin, (0,0), is at the bottom left-hand corner and that the height of the window is 100 units. The number of pixels in each graphics unit therefore depends on the size of the window, and the number of units wide that the window is is deduced from the height. Both the height and the position of the origin can be changed by using the SCALE procedure,

[line number] SCALE [#channel,] scale_factor, x-value, y-value

where **channel** is the window in which to change the scale, default 1;

scale_factor is the number of units high that the window is to be;

x-value is the coordinate unit value of x at the left-hand edge of the window;

and **y-value** is the coordinate unit value of **y** at the bottom of the window.

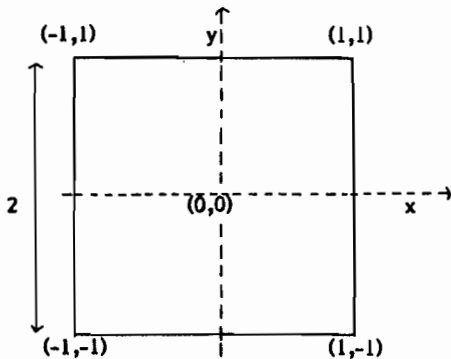
So, given a square window created by, say,

```
OPEN #3,SCR_137x100
```

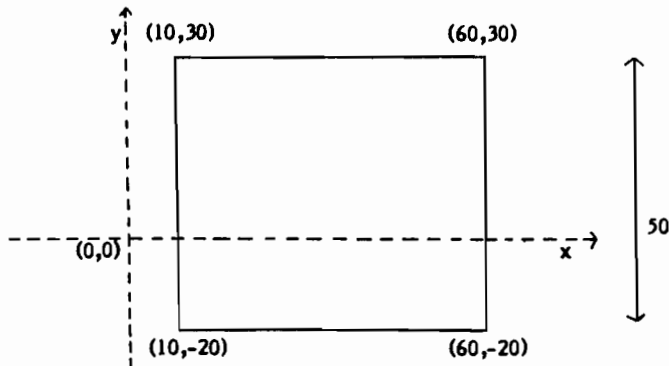
the command,

```
SCALE #3,2,-1,-1
```

would place the origin, or (0,0), right in the middle of the window; (-1,-1) at the bottom left-hand corner; (1,-1) at the bottom right-hand corner; (-1,1) at the top left-hand corner and (1,1) at the top right-hand corner of the window.



or SCALE #3,50,10,-20 would be



The graphics cursor for every window is initially set to (0,0). This

changes as soon as something is drawn.

Drawing lines

A line can be drawn on the execution screen from the current graphics cursor position to another point by using the LINE command in the following simple form :

```
LINE TO end_position
```

where **end_position** is the pair of cartesian coordinates ,

x-coord of the end of the line , y-coord of the end of the line

The line will be drawn in the current INK colour from the current position to the given end point, but the final pixel is not inked in. The reason for this will become apparent later. The current graphics pointer is then updated to hold the end coordinates.

To change the current graphics position without drawing a line to it can be done by saying

```
LINE new_position
```

where **new_position** is the graphics coordinate pair of the new cursor position. The pointer will be changed but nothing will actually be drawn.

If the two forms are combined, the method of drawing a line from any position to any other position can be clearly seen,

```
LINE x1,y1 TO x2,y2
```

first changes the graphics cursor to (x1,y1) then draws a line in the current ink colour to (x2,y2) and leaves the cursor there. The formal definition is

```
[Ino] LINE [#channel,] [new_position] { [,new_pos] TO end_position }
```

where **channel** is the window in which to draw the line, default 1;
omit comma if it is the case before TO

new_position is the position to **move** the cursor to;

and **end_position** is the point to **draw** a line to.

So the following line,

```
SCALE 100,0,0: LINE 15,75 TO 115,75 TO 115,15 TO 15,15 TO 15,75
```

draws a square.

If the printing characteristic OVER -1 has been specified, the colour of the ink will be XORed with the colour of the background for each pixel. So now the line,

```
OVER -1: LINE 15,75 TO 115,75 TO 115,15 TO 15,15 TO 15,75
```

will wipe the above square out again. This is the reason mentioned earlier for why LINE doesn't draw the last pixel on a line. If it did, when drawing over the top of a set of linked lines, dots would be left at all the intersections.

If you want to draw dots, use the procedure,

```
[line number] POINT [#channel,] position (,position)
```

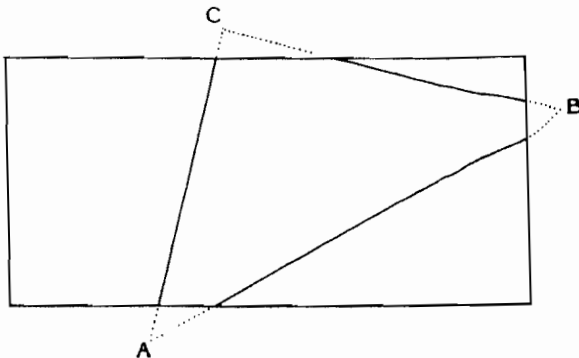
where **channel** is the window to draw the dots in, default 1;

and **position** is the graphics coordinate pair of the point to ink in.

POINT uses the current INK colour and updates the current graphics cursor position.

Off-window drawing

Changing the scale does not affect the coordinate position of the current graphics cursor. Suppose you start off with a scale of 100 units with the coordinates of the bottom left-hand corner of the window being (0,0), draw a line to (50,50), then change the scale to 1,0. The graphics cursor will still be at (50,50) - somewhere near the ceiling of the room next door! Drawing a line now to (0.5,0.7) will illustrate this, the line comes diving in from nowhere. This is something to watch out for but it does make a very interesting point, namely that the graphics system continues to operate outside the window. This has great advantages, one is that for a diagram like



you do not have to work out the positions where AB, AC and BC intersect the edge of the window, you can just use the actual coordinates of A, B and C even though they don't appear in the window. Another advantage is that you can examine part of a diagram in more detail just by changing the number of vertical units in the window and where the window is to start. It will seem a bit slow because SuperBASIC is still drawing the whole diagram, even though only a portion of it is visible.

The fact that the graphics system works in this manner is especially useful for the next topic, curves, but be warned, if nothing seems to be happening when you are drawing, it could well be that the wrong scale or axes have been set!

Drawing curves

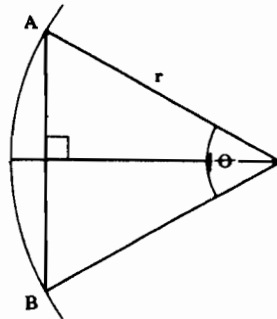


Figure 11.1

In Fig. 11.1, it is possible to specify the arc AB subtended by the angle θ (in a circle of radius r) by using the Super BASIC procedure,

ARC x_A, y_A TO x_B, y_B, θ

The course of action taken is to move the current graphics pointer to the coordinates for A, work out the radius (r) of the circle, then draw the arc anticlockwise in the current ink colour to the position given for B. The graphics pointer is then updated.

Like LINE, you can continue to draw ARCs to other points. For example,

ARC 10,50 TO 50,50,PI/4 TO 90,50,PI/4 TO 130,50,PI/4

The formal definition is

[no] ARC [#channel,] [new_position] { [,new_pos] TO end_position,angle}

where **channel** is the window in which to draw, default 1;

new_position is the graphics coordinate pair of the point to move to;

end_position is the coordinate pair of the point to draw to;

and angle is the value in radians of the subtended angle required at the centre of the 'circle'.

The arc is always drawn in an anticlockwise direction. If you want it to be drawn in a clockwise direction, you have to either reverse the end points or specify a negative angle.

For example, given point A at (70,70) and B at (60,20), the command

```
ARC 70,70 TO 60,20,PI/2
```

gives the arc



whereas both

```
ARC 60,20 TO 70,70,PI/2
```

```
and ARC 70,70 TO 60,20,-PI/2
```

give the arc



The angle must be given in radians; the larger the subtended angle that is specified, the curvier the arc that will be produced. Generally, arcs are specified in terms of π (SuperBASIC function PI), 2π radians being equal to 360 degrees, a complete circle. An angle of π gives half a circle, the chord AB being its diameter. There are two SuperBASIC functions, RAD and DEG, to convert degrees to radians and vice versa if you need them.

Lines and arcs can be combined of course; look at the following example :

```
100 DEFine PROCedure club(ch$)
110 REM Call with 'f' for flower shape,
120 REM      's' for club with straight stem (default)
130 REM      'c' for club with curved stem
140 SCALE 100,0,0
150 ARC 120,40 TO 120,65,3*PI/2 TO 92,65,3*PI/2 TO 92,40,3*PI/2
160 IF ch$(1)='f': ARC TO 120,40,3*PI/2: RETURN: ELSE LINE TO 106,52.5
170 IF ch$(1)='c': ARC TO 115,20,PI/4: ELSE LINE TO 115,20
```

```

180 LINE TO 97,20
190 IF ch$(1)='c': ARC TO 106,52.5,PI/4: ELSE LINE TO 106,52.5
200 LINE TO 120,40
210 END DEFine club

```

It would seem as if a complete circle could be drawn with the command

```
ARC 20,20 TO 20,20,2*PI
```

but thinking about it for a couple of seconds should convince you that this isn't so. There is no way to tell what size the circle should be!

Whenever a graphics procedure is insoluble in this way, nothing at all is drawn on the window and no error is given, control just passes to the next statement.

Circles and ellipses

To draw a circle, the procedure

```
[line number] CIRCLE [#channel,] centre_position , radius
```

should be used, where

channel is the window in which to draw the circle, default 1;

centre_position is the coordinate pair of the centre of the circle,

x value at centre , y value at centre

and **radius** is the length in graphics units of the radius of the circle.

The circle is drawn in the current ink colour. If OVER -1 has been set, the ink colour is XORed with the background colour at each pixel on the circumference of the circle. The current graphics pointer will be updated to the origin (centre) of the circle.

Try this for a nice effect :

```

100 DEFine PROCedure circ
110 SCALE 100,0,0
120 FOR i=1 TO 70 STEP 2: CIRCLE i,i,i
130 END DEFine

```

and expand it thus for a very soothing vision,

```

140 DEFine PROC rep_circ
150 PRINT #0,' *** BREAK (CTRL and space) to stop ***'
160 PAPER 1: INK 7: CLS: OVER 0

```

In coordinate geometry, the equation of a circle with centre (0,0) is

$$x^2 + y^2 = r^2$$

where r is the radius. This is a special case of the equation for an ellipse with centre (0,0)

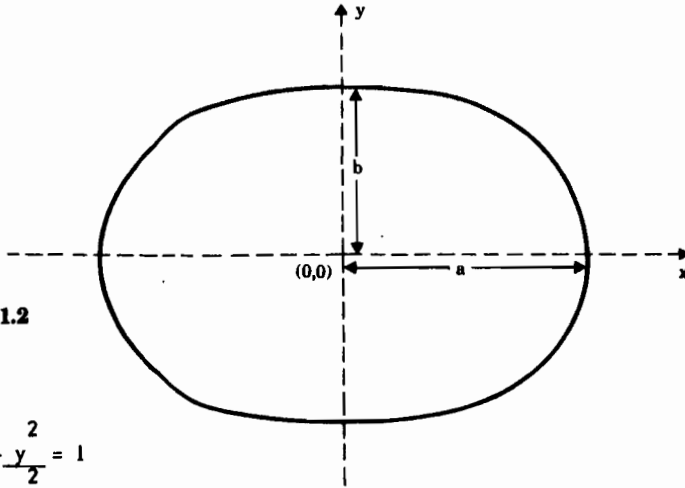


Figure 11.2

$$\frac{x^2}{a^2} + \frac{y^2}{b^2} = 1$$

where a is the radius along the x-axis;

and b is the radius along the y-axis.

An ellipse is an oval, symmetric about its major and minor axes. These bisect one another at right angles, the major axis being longer than the minor axis. A SuperBASIC ellipse can be drawn anywhere in the graphics coordinate system, at any angle. It is defined by

[Ino] ELLIPSE [#channel,] centre_position, major_radius, ratio, rotation

where **channel** is the window in which the ellipse is to be drawn, default 1

centre_position is the coordinate pair of the centre of the ellipse,

x value at centre , y value at centre

major_radius is the length in units of the longest radius;

ratio is the ratio of the minor radius to the major radius. This will normally be in the range 0.1 to 1;

and **rotation** is the angle in radians at which the major radius is inclined away from the vertical in an anticlockwise direction. A rotation of zero is vertical, 0 , $\pi/2$ is horizontal, 0 and $\pi/4$ is at 45 degrees 0 .

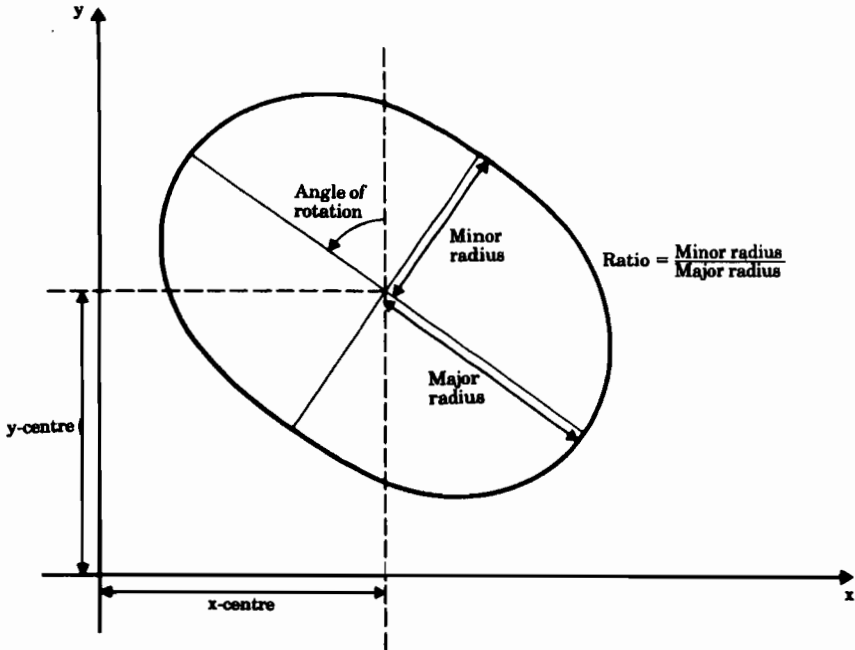


Figure 11.3

An example of the different shapes and sizes of ellipses can be seen if you run the following procedures, ELL or REP_ELL :

```

90 DEFine PROCedure ell
100 FOR i=1 TO 70 STEP 2: ELLIPSE i,i,,5,PI/2-i*PI/70
110 END DEFine
120 REMark -----
130 DEFine PROCedure rep_ell
140 PRINT #0,'*** BREAK (CTRL and space) to stop ***'
150 PAPER 1: INK 7: CLS : OVER 0
160 ell: OVER -1: REPeat c; ell
170 END DEFine

```

Varying the ratio instead of the major radius produces a very strange effect :

```

90 DEFine PROCedure star
100 FOR i=1 TO 70 STEP 2: ELLIPSE i,i,5,i
110 END DEFine
120 REMark -----
130 DEFine PROCedure rep_star
140 PRINT #0,'*** BREAK (CTRL and space) to stop ***'
150 PAPER 1: INK 7: CLS : OVER 0
160 star: OVER -1: REPEAT c: star
170 END DEFine

```

Quite a firework display!

Incidentally, to see how well the ELLIPSE command matches the geometric equation, and to appreciate the speed of it, the following procedures first draw the ellipse or circle using the SuperBASIC graphics commands, then painstakingly blot it out using coordinates generated from the equations given earlier.

```

100 DEFine PROCedure make_ell(a,b)
110 PAPER 1: INK 7: SCALE 4,-2,-2
120 ELLIPSE 0,0,a,b/a,PI/2
130 INK 0
140 FOR i=0 TO a STEP a/200
150 j=SQRT((1-i^2/a^2)*b^2)
160 POINT i,j, i,-j, -i,j, -i,-j
170 END FOR i
180 END DEFine
190 REMark -----
200 DEFine PROCedure make_cir(a)
210 PAPER 1: INK 7: SCALE 4,-2,-2
220 CIRCLE 0,0,a
230 INK 0
240 FOR i=0 TO a STEP a/200
250 j=SQRT(a^2-i^2)
260 POINT i,j, i,-j, -i,j, -i,-j
270 END FOR i
280 END DEFine

```

Now call MAKE_ELL with two parameters, major_radius, minor_radius (bearing in mind the scale of four), e.g., MAKE_ELL 1.8,1.1, and call MAKE_CIR with one parameter, the radius, e.g., MAKE_CIR 1.5 .

Note that calling MAKE_ELL with both parameters equal is the same as calling MAKE_CIR. In fact the SuperBASIC procedures, CIRCLE and ELLIPSE, are themselves interchangeable, three parameters being necessary to generate a circle and five to create an ellipse.

Annotating a graphics diagram

You will often want to mix text and graphics. We saw in the previous chapter how the procedure, CURSOR, was used to move the print, or text, cursor around the window, but it needed pixel coordinates specified, which are not very easy to relate to graphics units. What you really want to be able to do, in order to annotate drawings, is to combine the CURSOR command with the graphics units system.

When using the CURSOR procedure earlier, it had two parameters, the x-pixel coordinate and the y-pixel coordinate. There is an alternative form of CURSOR using four parameters.

The formal definition of the full form is :

```
[line number] CURSOR [#channel,] [graphics_position,] pixel_position
```

where **channel** is the window in which to move the text cursor, default 1;

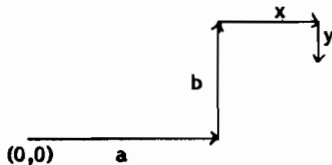
graphics_position is the graphics units coordinate pair of the position to move the text cursor to;

pixel_position is either the x and y pixel offsets from the graphics position, if given;
or the absolute x and y pixel position if the graphics position has been omitted.

In diagrammatic form, the command

```
CURSOR a,b,x,y
```

moves the cursor,



where A and B are counted across and up in graphics units, and X and Y are counted across and down in pixels. So,

```
CURSOR 200,100
```

moves the text cursor to a position 200 pixels along from the left hand edge of the window and 100 pixels down from the top of the window;

moves the text cursor to a position 200 graphics units to the right of the graphics origin and 100 graphics units above the graphics origin;

```
CURSOR 100,50,8,10
```

moves the text cursor to 8 pixels along and 10 pixels down from the graphics position given by (100,50).

The CURSOR command only affects the position of the text cursor. It has no effect at all on the current graphics cursor. As an example of text cursor movement within the graphics system, run the following procedure :

```
100 DEFine PROCEDURE axes
110 WINDOW 448,200,32,16: SCALE 4,-2,-2
120 PAPER 2: INK 7: CLS
130 LINE 0,-2 TO 0,2 , -2,0 TO 4,0
140 INK 0: FOR i=-2 TO 3: POINT i,0, 0,i
150 INK 7: FOR i=-2 TO 3
160   IF i<>0: CURSOR i,0,0,2: PRINT i
170   IF i<0: y=30: ELSE y=16
180   CURSOR 0,i,-y,0: PRINT i
190 END FOR i
200 CURSOR 2.5,0,0,20: PRINT 'x-axis'
210 CURSOR 0,1.5,-100,0: PRINT 'y-axis'
220 END DEFine
```

Relative drawing

All of the graphics commands we have examined so far,

LINE, POINT, ARC, CIRCLE and ELLIPSE

have used **absolute** coordinates. That is, graphics coordinates relative to the origin of the graphics system.

It is often advantageous to draw parts of diagrams relative, not to the origin, but to the current graphics cursor position. If this facility existed, it would be possible to write a procedure using **relative** coordinates to draw a picture and then, by moving the graphics cursor around, to reproduce the drawing anywhere in the window.

Well, the facility for relative drawing does exist, and it is very simple to use. All of the above commands may be suffixed by

_R

and take **exactly the same** parameters except that all the coordinate pairs are taken as the **offsets** from the current graphics cursor position. After

drawing each stage, the graphics cursor is updated to reflect its new position.

The relative graphics commands are :

[Ino] LINE_R [#channel,] [xr,yr] { [,xr,yr] TO xr,yr}

[Ino] POINT_R [#channel,] xr,yr {,xr,yr}

[Ino] ARC_R [#channel,] [xr,yr] { [,xr,yr] TO xr,yr,angle}

[Ino] CIRCLE_R [#channel,] xr,yr,radius

[Ino] ELLIPSE_R [#channel,] xr,yr,major_radius,rotation

where all the pairs of coordinates, xr,yr , are the offsets from the current cursor position, the value of the current cursor being updated at each stage of the commands.

In other words,

x	=	x	+	x	new x
absolute		current		relative		current
y	=	y	+	y	new y
absolute		current		relative		current

You can see how easy this makes drawing a square. Wherever the current cursor position is, you can say,

```
LINE_R TO 20,0 TO 0,20 TO -20,0 TO 0,-20
```

to draw a square of side 20 units with the current graphics position at the bottom left-hand corner. The following procedure illustrates this,

```
100 DEFine PROCedure rep_square
110 SCALE 100,0,0
120 PAPER 7: CLS
130 REPEAT sq
140   POINT RND(0 TO 170),RND(0 TO 100)
150   square_r
160 END REPEAT sq
170 END DEFine
180 REMark -----
190 DEFine PROCedure square_r
194 REMark Draw a relative square
200 INK RND(0 TO 7)
210 LINE_R TO 20,0 TO 0,20 TO -20,0 TO 0,-20
220 END DEFine
```


Filling shapes with colour

All of the graphics commands draw outlines only. If you want to draw solid shapes then you will have to utilize the FILL attribute.

FILL fills non-reentrant shapes with the current ink colour spectacularly fast. The general form is

```
[line number] FILL [#channel,] fill_mode
```

where **channel** is the window in which to change the fill attribute, the default being channel 1;

and **fill_mode** is 0 to turn area filling off
1 to turn area filling on

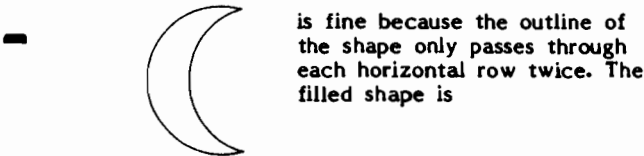
The FILL command does not affect anything that is already on the screen. FILL 1 means that any non-reentrant shape drawn from now on will be filled in. FILL 0 turns further area filling off.

It is a good idea to turn FILL off as soon as you have finished drawing your shape otherwise you might find unexpected areas being flooded with colour. Fill starts to paint as soon as you have defined a closed shape.

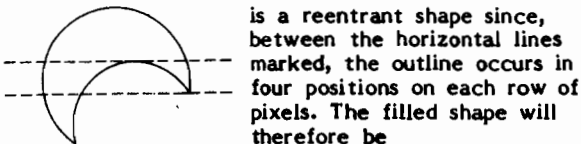
NON-REENTRANCE

Any shape must be non-reentrant in the vertical plane. FILL works by keeping the start pixel to fill and the end pixel to fill for each row of pixels in the window. This means that the outline of a shape must pass through no more than two points on each horizontal row of pixels. If the outline crosses a row in more than two places, the outside ones mark the filling limits.

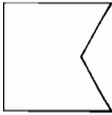
For example,



but



Similarly



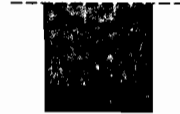
will fill inside the outline



but



will be filled right up to the top dotted line



To illustrate non-reentrance on the QL, you can run the following program. There is a pause before each new figure, hit any key to continue.

```
100 DEFine PROCedure fill_ex
110 REM illustrate FILL problems
120 PAPER 2: CLS: SCALE 4,-2,-2
130 FILL 1: INK 0: fillin
140 FILL 0: INK 7: fillin
150 PAUSE: CLS
160 FILL 1: INK 0: fillin2
170 FILL 0: INK 7: fillin2
180 PAUSE: CLS
190 INK 0: fillsq 1
200 INK 7: fillsq 0
210 END DEFine
220 REMark -----
230 DEFine PROCedure fillin
240 ARC 2,1 TO 1,-1,3*PI/2
250 ARC 2,1 TO 0,0,PI
260 ARC 1,-1 TO 0,0,-5*PI/6
270 END DEFine
280 REMark -----
290 DEFine PROCedure fillin2
300 ARC -1,2 TO 2,1,3*PI/2
310 ARC -1,2 TO 2,1,PI
320 END DEFine
330 REMark -----
340 DEFine PROCedure fillsq(fill_mode)
350 FILL fill_mode
360 POINT -.5,1
370 LINE_R TO 0,-1 TO 1,0 TO -.5,.5 TO .5,.5 TO -1,0
380 FILL 0: FILL fill_mode
390 POINT 2,1
400 LINE_R TO 0,-1 TO 1,0 TO 0,1 TO -.5,-.5 TO -.5,.5
410 FILL 0
420 END DEFine
```

If you understand FILL, you should be able to work out why line 380 is

necessary. If not, try inserting 'REMark' after the line number and rerun it. In actual fact, you do not need both FILL 0 and FILL fill_mode, since FILL 1 automatically turns an old fill off before it turns a new one on.

As long as you treat it with respect, FILL is a very powerful feature. Here is another example which takes a key, or combination of keys, as input and produces various swirly patterns. Especially good are those produced on receipt of

F4, d, e, space, CTRL and F3, b, p, ALT and ENTER, SHIFT and h

```

100 DEFine PROCedure rep_swirl
110 WINDOW 448,240,32,16
120 PAPER 0: MODE 8
130 SCALE 300,-200,-150
140 PRINT '\\Press any key or combination of keys each time the figure
finishes\\' BREAK (CTRL and space) to stop.'
150 REPEAT s: a=KEYROW(0): RANDOMISE CODE(INKEY(-1)): CLS: swirl
160 END DEFine
170 REMark -----
180 DEFine PROCedure swirl
190 loop = RND(100 TO 200)
200 col = 8+(248*RND(0 TO 1))
210 rot = RND(1 TO 10)
220 ratio = 1/RND(2 TO 10)
230 bit = RND(1 TO 5)
240 FOR n=loop TO 5 STEP -bit: INK n MOD 2: FILL 1: ELLIPSE 0,0,n,ratio,
n/rot: FILL 0
250 END DEFine swirl

```

You ought to be able to modify this to produce endless different patterns without the need for input. It can be watched for hours!

Turtle graphics

In order to make the graphics system even simpler, it would be nice, when drawing diagrams, to be able to give the length of the line to draw (rather than specifying its end-coordinates), or the angle to turn through relative to the direction of the last line drawn.

The turtle graphics commands maintain a current direction as well as the normal current position. MOVE moves the cursor by a given number of units in the current direction. Whether the line is drawn as the cursor moves along it is decided by which of PENUP or PENDOWN is currently set. The formal definitions of these commands are :

[line number] PENDOWN [#channel]

where channel is the window in which to set the pen attribute to on,

default 1;

PENDOWN does not affect anything which is already in the window. Any MOVES done from now on will be drawn in the current ink colour. PENDOWN has no effect on any of the other graphics commands;

[line number] PENUP [#channel!]

where **channel** is the window in which to set the pen attribute to off, default 1;

PENUP is the state of the pen on first opening a window. When PENUP is set, all MOVE commands move the cursor but their action is invisible. PENUP does not affect anything already in the window, nor does it have any effect on the other graphics or printing commands;

[line number] MOVE [#channel,] distance

where **channel** is the window in which to move the cursor, default 1;

and **distance** is the number of graphics units to move by.

MOVE moves the graphics cursor by a given number of units in the current direction. The graphics cursor is then updated. If PENDOWN is currently set, the line moved will be drawn in the current ink colour. If OVER -1 has been set, then the colour of the ink will be XORed with the colour of the background for each pixel drawn. If PENUP has been specified, the move is transparent.

Changing the current direction can be done only with the two commands, TURN and TURNT0. Their form is

[line number] TURN [#channel,] angle

where **channel** is the window in which to change the current direction, default 1;

and **angle** is the number of degrees to turn, anticlockwise from the current direction.

There is no visible effect of this command, the current direction is modified and when the next MOVE is done, it will be in the new direction. When the window is first opened, the turtle graphics direction is initially horizontal and pointing towards the right-hand side of the screen. Even though the other graphics commands draw lines at different angles, they do not update the current turtle graphics direction. Only TURN and TURNT0 do this. There are two SuperBASIC functions, DEG and RAD, which convert radians to degrees and vice versa if you need them.

TURNT0 is used when you want to change the current direction to a specific absolute angle from the horizontal. Its form is

[line number] TURNT0 [#channel,] angle

where **channel** is the window in which to **C**hange the direction, default 1;

and **angle** is the angle in degrees **t**o set the current direction pointer, counted anticlockwise from due East.

TURNT0 0 sets the current direction **s**quarely towards the right-hand edge; TURNT0 90 points vertically towards the top and TURNT0 270 points towards the bottom of the window. A negative angle is turned clockwise from the horizontal, an angle of greater than 360 degrees is taken as (360-angle) degrees.

Like TURN, TURNT0 does not have any visible effect until the next MOVE is done.

The following procedure is a stunningly beautiful example of MOVE and TURN

```
100 DEFine PROCedure daisy_wheel
110 SCALE 4,-2,-2: PAPER 7: MODE 8: P ENDOWN
120 r = 2.2
130 FOR j=0 TO 6
140   INK j: n = 2: r = r - 2
150   disc n,r
160 END FOR j
170 END DEFine
180 REMark -----
190 DEFine PROCedure disc(no_deg,radius)
200 LOCAL i
210 FOR i=1 TO 360/no_deg: POINT 0,0: TURN no_deg: MOVE radius
220 END DEFine
```

Include lines

```
124 REPEAT always
```

and

```
164 END REPEAT always
```

to watch the figure disappear again then gradually get larger in single colour steps.

As another example, consider the problems of drawing regular polygons. Given a fixed length side and a specified number of sides, you can write the procedure,

```
100 DEFine PROCedure poly(no_sides,len_side)
110 LOCAL i
120 PENDOWN
130 FOR i=1 TO no_sides
140   MOVE len_side
150   TURN 360/no_sides
160 ENDFOR i
170 END DEFine
```

If you now say

```
SCALE 100,0,0: POINT 50,50: POLY 3,20
```

you get a triangle. Now,

```
POLY 4,20
```

draws a square on top of it and

```
POLY 5,20
```

adds a pentagon. The final direction is always the same as the initial direction, so the first lines of these figures coincide. I find this slightly displeasing, but adding the lines

```
124 TURN 180/no_sides - 90
```

and

```
164 TURN 90 - 180/no_sides
```

to the procedure means that the original direction now bisects the angle made by the first and last sides and is still restored after the last line. Redrawing the three figures above is now much more pleasant.

A very simple extension to this procedure can give multiple polygons equally spaced about the original point :

```
200 DEFine PROCedure mpoly(no_poly,no_sides,len_side)
210 LOcal k
220 FOR k=1 TO no_poly
230   poly no_sides,len_side
240   TURN 360/no_poly
250 END FOR k
260 END DEFine
```

Now run

```
mpoly 5,5,20
```

```
mpoly 6,3,20
```

or, very pretty this one,

```
CLS: LINE 70,50: mpoly 10,10,15
```

As a final example of what can be achieved with the QL SuperBASIC graphics system, I offer a set of scenic procedures together with three ways in which you might combine them. You will, I am sure, be able to modify and improve them to your own satisfaction.

Run the procedures SCENE1, SCENE2 and SCENE3 for the sample composites.

```
100 DEFine PROCEDURE scene1
110 MODE 8: SCALE 1,0,0
120 sky
130 mountains
140 rainbow .7, 2e-2, .8, .15
150 house .6, .8, 3e-2, 6
160 tree 1.5, 1e-2, .5, 2
170 grass 5
180 END DEFine
190 REMark -----
200 DEFine PROCEDURE scene2
210 MODE 8: SCALE 1,0,0
220 PAPER 7,0,0: CLS
230 house .1, 0, .9, 2
240 house .2, .1, .8, 4
250 house .3, .3, .65, 6
260 house .5, .6, .35, 5
270 house .6, 1.1, 0, 0
280 END DEFine
290 REMark -----
300 DEFine PROCEDURE scene3
310 MODE 8: SCALE 1,0,0
320 PAPER 7,0,3: CLS
330 FOR ii = .9 TO 0 STEP -.01
340   IF RND>.2: NEXT ii
350   house .2, RND*1.6, ii, RND(0 to 6)
360   jj = RND: IF jj>.5: NEXT ii
370   tree jj*2*1.6, ii, RND(1 TO 2)/10, RND(0 TO 4)
380 END FOR ii
390 END DEFine
1000 REMark -----
1010 DEFine PROCEDURE rainbow(x,y,h,w)
1020 REMark x,y - coords of mid-point of base
1030 REMark h - height of rainbow
1040 REMark w - total width of bands
1050 LOCAL colour,r,s
1060 s=-w/7: r=h
1070 FOR colour = 2,226,6,4,1,201,209
1080   INK colour: FILL 1
1090   ARC x, y+r TO x-r, y, PI/2
1100   LINE x, y+r TO x, y+r+s
1110   ARC TO x-r-s, y, PI/2
1120   FILL 0: FILL 1
1130   ARC x+r, y TO x, y+r+s, PI/2
1140   LINE TO x, y+r+s
1150   ARC x+r+s, y TO x, y+r+s, PI/2
1160   r=r+s: FILL 0
1170 END FOR colour
1180 END DEFine
2000 REMark -----
2010 DEFine PROCEDURE sky
```

```

2020 REMark pale blue sky with fluffy white clouds
2030 LOCAL i
2040 PAPER 5: CLS: INK 7
2050 FOR i=0 TO 30
2060   FILL 1
2070   ELLIPSE RND*1.8, .5+RND*.5, 3e-2+RND*3e-2, 1+RND, 0
2080   FILL 0
2090 END FOR i
2100 END DEFine
3000 REMark -----
3010 DEFine PROCedure tree(x,b,h,apples)
3020 REMark x,b - position of base of tree
3030 REMark h - height of trunk (half of whole tree)
3040 REMark apples - 0 for none, > 1 for lots
3050 LOCAL t,w,o,s,i
3060 t=b+h: w=h/6: o=-w/2: s=-o/2
3070 REMark ---- trunk ----
3080 FOR i = 0,16,192+16+2,16+2,2
3090   INK i: FILL 1
3100   LINE x+o, b TO x+o/2, t
3110   o = o + s
3120   LINE TO x+o, b TO x+o/2, t
3130 END FOR i
3140 REMark ---- leaves ---
3150 INK 4
3160 FOR i = 0 TO 55
3170   FILL 1
3180   ELLIPSE x+(RND-.5)*h, t+(RND-.5)*h, (RND+.4)*w
3190 ENDFOR i
3200 REMark ---- apples ----
3210 IF apples
3220   INK 2
3230   FOR i = 1 TO 10*apples
3240     FILL 1
3250     ELLIPSE x+(RND-.5)*(7*h/8), t+(RND-.5)*(7*h/8), .25*w
3260   END FOR i
3270 END IF
3280 END DEFine
4000 REMark -----
4010 DEFine PROCedure grass(flowers)
4020 REMark flowers - 0 for none, 20 for masses
4030 LOCAL i,h
4040 INK 4: FILL 0
4050 FOR i=0 TO 1.7 STEP 3.5e-3: LINE i, 0 TO i, 2e-2+RND*5e-2
4060 IF flowers
4070   FOR i=0 TO 1.7 STEP 5e-3
4080     IF RND>5e-2*flowers: NEXT i
4090     INK 2+4*RND(0 TO 1)
4100     h=RND*3e-2+1.5e-2
4110     FILL 1
4120     ELLIPSE i, h, 1e-2
4130   END FOR i
4140 END IF

```



```

5000 REMark -----
5010 DEFine PROCEDURE mountains
5020 LOCAL h,x,col
5030 FOR col = 3,1
5040 h=RND*.2+col*1.5e-2+3e-2: x=0
5050 FILL 1: INK col
5060 LINE 0,0 TO 0,h
5070 REPEAT loop
5080 h=RND*.2+col*1.5e-2+3e-2
5070 x=x+5e-2+RND*.1
5100 LINE TO x,h TO x,0
5110 FILL 1
5120 LINE TO x,h
5130 IF x>2 THEN EXIT loop
5140 END REPEAT loop
5150 END FOR col
5160 END DEFine
6000 REMark -----
6010 DEFine PROCEDURE house(w,x,y,door)
6020 REMark w - width
6030 REMark x,y - bottom left hand corner
6040 REMark door - front door and window frame colour
6050 LOCAL ww,wl_5,w2_5,h_5,wh,wr
6060 ww = w/5: wl_5 = 1.5*ww: w2_5 = 2.5*ww
6070 h = .6*w: h_5 = .5*w: wh = ww + y
6080 rect w,h,x,y,226
6090 windw ww, wl_5, x+.5*ww, wh, 201, door, 7
6100 windw ww, wl_5, x+3.5*ww, wh, 201, door, 7
6110 windw ww, w2_5, x+.4*w, y, door, 7, -1
6120 FILL 1: INK 1
6130 ELLIPSE_R .1*ww, wl_5, .1*ww
6140 FILL 1: INK 82
6150 POINT x-.5*ww, h+y
6160 wr = .5*(w+ww)
6170 LINE_R TO wr, h_5 TO wr, -h_5 TO -w+ww, 0
6180 FILL 0
6190 END DEFine
6200 REMark -----
6210 DEFine PROCEDURE windw(w,h,x,y,i,bord,cross)
6220 REMark w,h - width,height
6230 REMark x,y - bottom left hand corner
6240 REMark i - main colour
6250 REMark bord - border colour if +ve, else none
6260 REMark cross - crossbars colour if +ve, else none
6270 rect w,h,x,y,i
6280 IF bord>0
6290 INK bord
6300 LINE_R TO 0,h TO w,0 TO 0,-h TO -w,0
6310 END IF
6320 IF cross>0
6330 INK cross
6340 LINE_R w/2,0 TO 0,h , w/2,-h/2 TO -w,0
6350 END IF

```

```
6360 END DEFine
6370 REMark -----
6380 DEFine PROCedure rect(w,h,x,y,i)
6390 INK i
6400 FILL i
6410 POINT x,y
6420 LINE_R TO 0,h TO w,0 TO 0,-h TO -w,0
6430 FILL 0
6440 END DEFine
```

12 MACHINE CODE AND MEMORY ACCESS

This handbook is on SuperBASIC, not machine code. McGraw-Hill publishes an excellent book, 'QL Assembly Language Programming' which deals exhaustively with interfacing machine code programs to SuperBASIC. All that I am going to do in this chapter, therefore, is to explain the various SuperBASIC procedures and functions which enable you to use pre-written machine code procedures and functions.

Saving and loading hexadecimal data

To **load** raw hexadecimal bytes into the memory of the QL, use

```
[line number] LBYTES device, start_address
```

where **device** is the place where the data is to come from. This can be a Microdrive file, e.g., mdv1_test_proc, or any other peripheral device, a network channel for example.

and **start_address** is the absolute address in memory where the data is to go to.

To **save** bytes from memory, use the command,

```
[line number] SBYTES device, start_address, length
```

where **device** is where the data is to go. It can be a file on Microdrive (e.g., MDV2_test2) or any other peripheral device, e.g., SERz ;

start_address is the start of the data to save;

and **length** is the number of bytes to save.

For example,

```
SBYTES NETO_3, 131072, 32768
```

saves the contents of the screen over the network.

Using the resident procedure area

CALL is used to transfer execution to a machine coded area. It is effectively a JSR, or 'jump to subroutine', instruction. Its form is

[line number] **CALL** address [, registers]

where **address** is the absolute address in memory to transfer execution to;

and **registers** are the new contents of any 68000 registers. These will normally be used as parameters to the machine code program and should be given in the order

data registers

D1,D2,D3,D4,D5,D6,D7

then address registers

A0,A1,A2,A3,A4,A5

The machine code routine will need an RTS (return statement) to transfer control back to SuperBASIC. On returning, the contents of the data register, D0, are checked. This should have been set by the machine code routine to **zero** for a good return, anything else is an error and will be treated accordingly.

All extra machine code procedures and functions should be kept in the **resident procedure** area. Because it is resident, once any space has been reserved in it, that space cannot be released again until the QL is reset. To allocate space, use the function,

RESPR (length)

where **length** is the number of bytes of room required in the resident procedure area;

and the **return value** is the address at which the allocated space begins.

Space should be allocated in blocks of 512 bytes.

Machine code extensions to SuperBASIC should first be loaded into the resident procedure area and then linked with CALL to create entries for them in the nametable. For example,

np = RESPR(512)

to reserve 512 bytes in the resident procedure area

LBYTES mdv1_boot_code,np

to load the raw bytes from the file on the tape in mdv1 into the space just allocated

CALL np

to link the procedures in

Executing independent machine code programs

CALL executes a machine code subroutine in memory from a SuperBASIC program. To execute an **independent** machine code program, i.e., one that will not interfere with the normal SuperBASIC program and which is not in memory, you must use the command :

```
[line number] EXEC device
```

where **device** can be the Microdrive file where the program is held, or any other peripheral device from where the program is coming.

```
EXEC mdv2_clock  
EXEC NET1_2
```

If you want to run a machine code program in this way and suspend SuperBASIC processing while you do so, then use :

```
[line number] EXEC_W device
```

where **device** can be a Microdrive file or any other input peripheral device.

This command starts up the independent machine code program as it does for EXEC, but does not allow you to continue with any other SuperBASIC commands until the job has finished.

Using EXEC means that you can have up to 56 jobs running on your QL at once. Quite a lot of these could be requiring input from you. Obviously you can only put data into one channel at any one time. Changing which input buffer you are attached to can be done by pressing CTRL and C together. This key combination moves the cursor to the next channel awaiting input. The cursor flashes when input is expected so, if each of the jobs wanting input has a window open on the screen, then you can see which input buffer you are attached to now by whereabouts the cursor is flashing. Continue to press CTRL and C together until you reach the channel that you want.

If you want to **save** a machine code program to be executed with EXEC or EXEC_W, use

```
[line number] SEEXEC device,start_address,length, data space
```

where **device** is the new Microdrive file or other output peripheral device where the program is to be written;

start_address is the absolute address in memory where the machine code program starts;

length is the number of bytes long that the program is;

and **data space** is the number of bytes required on the user stack plus the number of bytes required by the program for data.

Accessing memory directly

To read the contents of memory directly, you can use the functions

PEEK (address)

PEEK_W (even_address)

and PEEK_L (even_address)

where **address** is the absolute address in memory that you want to look at.

PEEK returns the integer value, range 0-255, of the **byte** at the address given;

PEEK_W returns the integer value, range 0-32767, of the **word** (two bytes) starting at the even address given;

PEEK_L returns the floating point value, with no loss of precision, of the **long word** (four bytes) starting at the even address given.

Words and long words can only be accessed on word, or even, boundaries.

To change the contents of memory, you can use the procedures

[!no] POKE address,expression

[!no] POKE_W even_address,expression

and [!no] POKE_L even_address,expression

where **address** is the absolute address in memory in which you are interested;

and **expression** is the value which you want to put into that address.

POKE puts a **byte**, range 0-255, into the contents of the address given;

POKE_W puts a **word** (two bytes), range 0-32767, into memory starting at the even address given;

POKE_L puts a **long word** (four bytes), range $10^{2^{616}}$, into memory starting at the even address given.

Words and long words can only be accessed on word, or even, boundaries.

The usage of PEEK and POKE is not to be particularly recommended. The machine code interface is well defined and is described in detail in Colin Opie's 'QL Assembly Language Programming'. Any further comment is beyond the scope of this Handbook.

Other commands for running machine code programs are collected together in the Sinclair QL Toolkit, available as an extension to the SuperBASIC system.

13 THE CALENDAR AND CLOCK

The QL contains a real-time clock chip and SuperBASIC provides a set of procedures and functions to access it. The most familiar is the string function, `DATE$`. This returns a string made up from

- a four digit year number
- space
- a three character month number
- space
- a two digit day number
- space
- two digits for the hour
- colon
- two digits for the minute
- colon
- two digits for the second

There is no battery backup for the clock, so the date set when you switch your QL on is random.

```
PRINT DATE$
```

might give you something like

```
2019 Feb 08 00:54:39
```

To set the clock to a specific date, you will have to use the procedure, `SDATE`, with a numeric parameter for each component of the date. The full form of the command is :

```
[line number] SDATE year,month,day,hour,minute,second
```

where **year** is the non-abbreviated year number, 0 to 32767;

month is the month number, 1 to 12;

day is the day in the month, 1 to 31;

hour is the 24-hour-clock number, 0 to 23;

minute is the number of minutes past the hour, 0 to 60;

and **second** is the number of seconds past the minute, 0 to 60.

So,

```
SDATE 1984,10,6,19,20,15
```

sets the time to twenty minutes and fifteen seconds past seven in the evening on the 6th October 1984. It doesn't have any visible effect but

```
PRINT DATE$
```

now gives you

```
1984 Oct 06 19:20:21
```

because six seconds have passed since you set the date.

You can **adjust** the date and time by using the procedure

```
[line number] ADATE seconds
```

where **seconds** is the number of seconds to add to the current time. A negative parameter will turn the clock back.

So

```
ADATE 3600
```

advances the clock by an hour and

```
ADATE 3*24*3600 - 30*60
```

puts it forward by half an hour short of three days.

DATE\$ returns the current date and time in a **string**; to get the value of the current date and time in **seconds**, use the function

```
DATE
```

which has no parameters. Thus, having set the date as above,

```
PRINT DATE
```

will give you something like

```
7.499352e8
```

- an awesome number of seconds!

A more serious use for DATE is in timing tests. You can say

```
d_beg = DATE
```

at the beginning of a piece of code and

```
d_end = DATE
```

at the end, then the statement

```
d_diff = d_end - d_beg
```

gives the number of seconds taken. Since the date count is only incremented once per second, fractions of seconds will not be included. Timing runs should therefore be done repeatedly and the average taken for a more accurate result.

By using the expanded form of the DATE\$ function, you can find out the date in an intelligible form from any number of seconds. The general form is :

```
DATE$ [ (seconds) ]
```

where **seconds** is the total number of seconds in the whole date. As we have already seen, if the number of seconds is omitted, the current date is used.

For example,

```
PRINT DATE$(8e8)
```

produces

```
1986 May 09 06:13:20
```

A function is also provided to find out the current day of the week. The formal definition of the string function, DAY\$, is :

```
DAY$ [ (seconds) ]
```

where **seconds** is the total number of seconds in the date and, if omitted, the current date is used.

So,

```
PRINT DAY$
```

when the date has been set as it was earlier, gives

```
Sat
```

Or using other values for the total number of seconds,

```
PRINT DAY$(7e8)
```

gives

```
Tue
```

and

```
PRINT DAY$(8e8)
```

gives

```
  Fri
```

To find out what day of the week a particular day was, you will have to set the date and then print the day. For instance,

```
SDATE 1955,12,24,0,0,0 : PRINT DAY$
```

tells me that I was born on a Saturday (works hard for a living). To restore the original date afterwards is a little more tricky and not terribly accurate. This is an approximate method.

```
100 PRINT DATE$  
110 d_orig = DATE  
120 SDATE 1950,11,6,0,0,0  
130 PRINT DATE$,DAY$  
140 SDATE 0,0,0,0,0,0  
150 ADATE d_orig + 10  
160 PRINT DATE$
```

14 THE QL SOUND

The QL has an astonishingly varied repertoire of sounds and a powerful speaker through which to play them. The formal definition of the command is :

```
[line number] BEEP [beep_parameters]
```

where all of the `beep_parameters` should be omitted to turn the sound off (an extremely necessary option), but otherwise are in the form :

```
duration, pitch [, range, time_step, pitch_step [, repeats [, fuzz [, random]]]]
```

Each of these parameters will be dealt with in turn, but the only real way to judge for yourself how to use the BEEP command is to spend a whole morning varying the parameters and listening to the result. At the end of the parameter explanations therefore, I have appended a fairly simple procedure which you would be wise to type into your QL and keep on a Microdrive cassette. It is an uncomplicated menu-driven method of varying the parameter values, certain of which have been adjusted to make experimentation easier for you. When you do find a pleasing combination of sounds, write down the parameter values somewhere safe or you are unlikely to be able to reproduce it for a while.

Do note the adjustments which I have made to the parameter values, they are for convenience only and are in no way obligatory. It is always possible to improve upon a program and to tailor it to your individual needs. All that I have done is to give you a starting point.

So, first of all, read the parameter explanations, then enter and save the BEEP_MENU procedure, then use it in conjunction with the parameter explanations.

DURATION

This is the length of time, in units of 72 microseconds, that the note, or sequence of notes, will be held for. The range is 0 to 32767.

A duration of zero means that the sound will be continued indefinitely. It can be turned off by the command BEEP with no parameters at all.

To get a burst of sound one second long, the duration must be $1000000/72$. Half a second is $500000/72$. The smallest detectable length is about 10, but short bursts do distort the tone of the note.

Since it is an integer parameter, the longest finite length of time that you can specify with one command is 32767 lots of 72 microseconds, or 2.36 seconds. There are ways around this problem, however.

The sound produced is independent of the SuperBASIC system. Once the BEEP parameters have been assimilated and the command passed to the IPC (inter-process communication) controller to be executed, SuperBASIC carries on immediately with the next statement. You can make use of this to produce the length of sound you require by giving the BEEP command an infinite duration, putting a PAUSE in of the right length, then turning the sound off,

```
BEEP 0,30 : PAUSE 200 : BEEP
```

for example, produces a sound four seconds long, the pause-time-rate being in fiftieths of a second.

In the exerciser example, I have assumed steps of half a second, so that the actual parameter value passed to BEEP is the value you give multiplied by $500000/72$.

PITCH

This is the tone of the main sound produced. The pitch range is 0 to 255, cyclic, so that a parameter value of 256 is equivalent to one of 0, 257 is the same as 1 and -1 gives the same pitch as 255. A pitch value of zero is the highest tone produced, descending gradually to the lowest at 255.

When just two parameters are given to the BEEP command, duration and pitch, a single pure note is produced. Use of the next three parameters gives a sequence of notes tied to the main pitch parameter. These three parameters are secondary pitch, time-interval step and pitch-interval step.

SECONDARY PITCH

This is a second pitch below the main pitch, defining a range across which pitch steps can be made. The range of the parameter is 0 to 255, cyclic, as with the main pitch parameter. A secondary pitch value less than or equal to the main pitch will not produce a range, otherwise the range achieved depends on the interval between the main and secondary pitches, and the size of the pitch step given.

TIME INTERVAL BETWEEN STEPS

This is the length, in units of 72 microseconds, of each note in the sound sequence. The permitted range for this parameter is 0 to 32767. The larger the time steps, the more distinct each separate note becomes. Small steps only produce a buzz.

In the exerciser program, the actual parameter for time interval is adjusted so that the input value is in hundredths of a second. So, parameter value = input value * 1e4/72.

PITCH INTERVAL BETWEEN STEPS

This has a range of 0 to 15, cyclic, and gives the size and direction of the pitch interval between each note in the sound sequence. A new note is made at each time interval specified.

A pitch step of zero produces no step.

A pitch step of 1 means that each note will be one pitch below the last.

A pitch step of 2 means that each note will be two pitches below the previous one.

Pitch steps of 3 to 7 produce increasingly larger steps in a downwards direction.

A step of 8 gives a complete ripple upwards.

Pitch steps of 9 to 15 give increasingly smaller steps in an upwards direction corresponding to the step sizes 7 to 1. Since the range is cyclic, you could use -7 to -1 instead if that makes the step size and direction clearer.

When running through a sequence of notes, the pitch step is added to the previous pitch and the resulting note sounded at the next time interval. If that resulting pitch is **below** the secondary pitch specified in the command, then it is not sounded. Instead the step direction is reversed and the new note is produced one pitch step above the old one. Likewise, if the new note would be **above** the top of the range, the direction is reversed again. The sequence continues to 'bounce' between the upper and lower range markers until the length of time specified for the complete sound has been covered.

Pitch steps of 1 (smallest step possible) to 7 (largest step possible) always start the sequence on the highest note of the range; pitch steps of 9 to 15 (or -7 to -1) always start the sequence on the lowest note. A pitch interval of 8 is not really a step size at all; when it is specified BEEP fits as many notes as possible (in sequence) into the range.

REPEATS

A normal sequence has the notes rippling up and down between the range markers. The **repeat** parameter allows each 'sweep' of notes to be repeated before going on to the next.

By a sweep, I mean from the first note in a particular direction to the last-but-one note in that direction (the last note being taken as the first note in the next direction). So, if your sequence is

```
x           x           x
 x         x x         x
  x       x   x       x
   x     x x     x x
    x   x   x
      x     x
```

then a repeat of one will give the sequence

```
x           x           x           x           x
 x         x x         x x         x x         x x         x
  x       x   x       x   x       x   x       x   x       x
   x     x x     x x     x   x   x   x   x   x   x   x
    x   x   x
      x     x
```

each sweep being repeated once. A repeat of two produces

```
x           x           x           x           x
 x         x x         x x         x x         x x         x
  x       x   x       x   x       x   x       x   x       x
   x     x x     x x     x   x   x   x   x   x   x   x
    x   x   x
      x     x
```

etc.

The range of the repeat parameter is 0 to 15, cyclic, the default being no repeats, just the ordinary range.

FUZZ

This parameter affects the purity of tone of each note in the sequence. The range is 0 to 15, cyclic, but since values of zero to seven have no effect on the tone, the effective range is 8 to 15.

Each note becomes slightly 'fuzzy' or blurred with a parameter value of 8 down to an indistinguishable (fairly unpleasant) buzz at 15. The default value is zero, no fuzz.

RANDOM

Specifying a random factor in a sequence of notes directs BEEP to find a note a random step away from the next note in the sequence. Again, though the range of parameter values is 0 to 15, cyclic, the effective range is actually 8 to 15. As the value increases from 8 to 15, a greater proportion of random, rather than true, notes is selected. At a parameter value of 15, very little of the original sequence is discernible.

The BEEP exerciser

```
100 DEFine PROCedure beep_menu
110 REMark BEEP exerciser
120 PAPER 2: INK 7: WINDOW 448,200,32,16
130 CLS
140 PRINT 'Duration      in half-seconds (0-4)\'Pitch      (0-255)\'
'Pitch 2      (0-255)\'Time step      in hundredths (0-235)\'Pitch step'
      (0-15)\'Repeats      (0-15)\'Fuzz      (8-15)\'Random
(8-15)'
150 At 12,0:PRINT 'Cursor up, cursor down to change the menu item selected
\'Cursor left to decrease the current value\'Cursor right to increase
the value\'SPACE to stop the noise\'ENTER to stop noise and leave menu'
160 DIM parm(7)
170 FOR iparm = 0 TO 7: print_param
180 STRIP 0: iparm = 0: print_param
190 REPEAT in
200   inc = CODE(INKEY$(-1)): REMark read keyboard
210   SELEct ON inc
220     ON inc = 208: IF iparm>0: change_param -1: REMark up
230     ON inc = 216: IF iparm<7: change_param 1: REMark down
240     ON inc = 192: parm(iparm) = parm(iparm)-1: rebeep: REMark left
250     ON inc = 200: parm(iparm) = parm(iparm)+1: rebeep: REMark right
260     ON inc = 32: BEEP: REMark space
270     ON inc = 10: BEEP: EXIT in: REMark enter
280   END SELEct
290 END REPEAT in
300 END DEFine
310 REMark -----
320 DEFine PROCedure rebeep
330 print_param
340 BEEP parm(0)*500000/72,parm(1),parm(2),parm(3)*10000/72,parm(4),
parm(5),parm(6),parm(7)
350 END DEFine
360 REMark -----
370 DEFine PROCedure change_param(change)
380 STRIP 2: print_param: REMark print old selection on red
390 iparm = iparm + change
```



```
400 STRIP 0: print_param: REMark print new selection on black
410 END DEFine
420 REMark -----
430 DEFine PROCedure print_param
440 AT iparm,11: PRINT parm(iparm) TO 14
450 END DEFine
```

CONTROLLING THE SOUND

There is, unfortunately, no way of turning down the volume control on the loudspeaker. I find the application of a woolly hat to the grill on the front edge of the QL an improvement, or possibly a strip of sticky tape, but the vents at the rear of the QL must be left clear to prevent overheating.

If the QL is beeping and you want to stop it doing so, you can either :

BREAK (CTRL and space together) then enter BEEP with no parameters as fast as possible;

or you can reset the QL completely. This method is drastic, but fast.

There is a SuperBASIC function which enables a program to detect whether the QL is producing a noise or not. Its form is simply

BEEPING

and it has a true or false return. Thus you can have lines like

```
nnn IF BEEPING : PRINT "Noisy, isn't it?"
```

A good idea is to have plain BEEP commands scattered around programs to turn any sound off, just in case!

15 THE SYNTAX GRAPHS

The SuperBASIC syntax definitions were originally drawn in the shape of 'railway line' diagrams, which is a marvellous way to resolve any lurking ambiguities.

The railway line concept is that, when travelling along it, you can run along curves or straight lines but you must not reverse or make any sharp turns. So, in Fig. 15.1, taking A as the starting point,

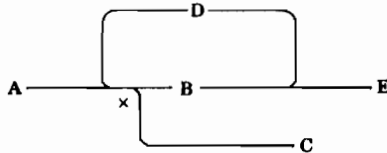


Figure 15.1

you **can** travel from A to B along the straight, or from A to C along the curve;
 you **cannot** travel directly from A to D because that would involve a sharp turn.

From B, you **can** go either to E or along the curve to D;
 you **cannot** reverse back and go to A or C.

From D, you **have** to go through junction X and then to B or C as before;
 you **cannot** make the sharp turn back to A;
 you **cannot** reverse back to B.

To show how the graphs were designed, look at this example of the definition of a SuperBASIC name. This can be any combination of letters, numbers and underscores, not starting with a number and optionally finishing with a percentage sign or a dollar.

(1) must start with a letter or an underscore (Fig. 15.2)

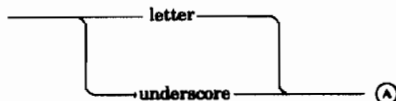


Figure 15.2

(2) then can be letter, number, underscore or none (Fig. 15.3)

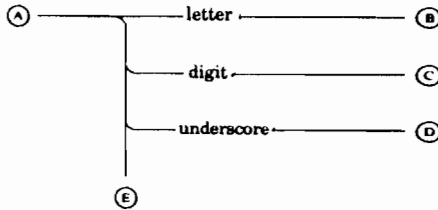


Figure 15.3

(3) repeated indefinitely (Fig. 15.4)

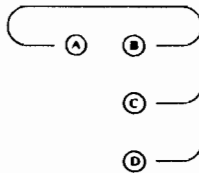


Figure 15.4

(4) optionally finishing with percentage sign or dollar (Fig. 15.5)

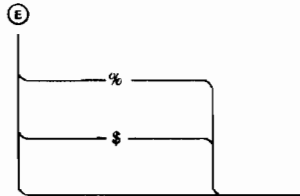


Figure 15.5

So the whole diagram is as shown in Fig. 15.6

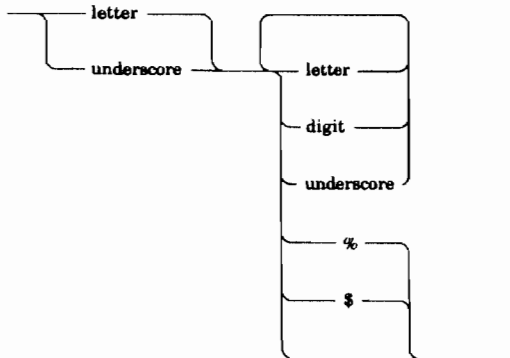


Figure 15.6

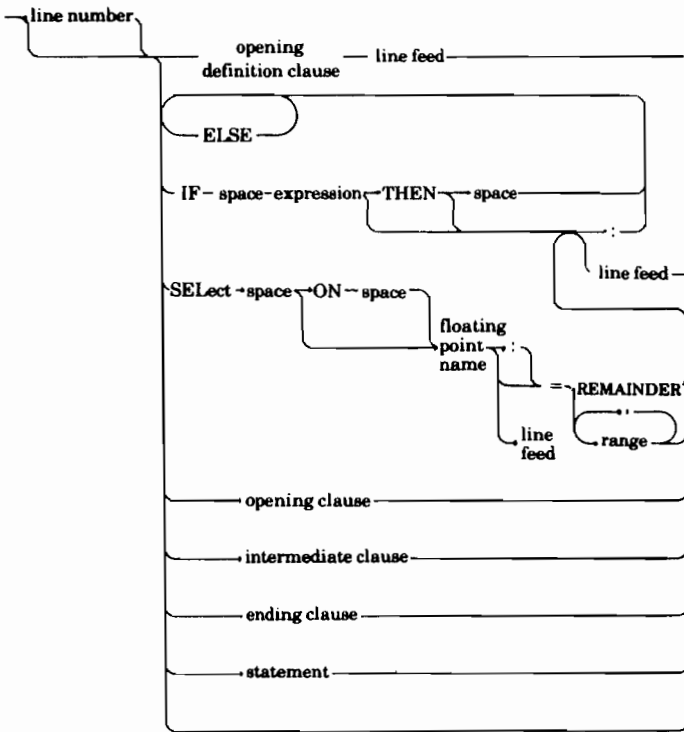


Figure 15.7 Syntax graph for a complete SuperBASIC line

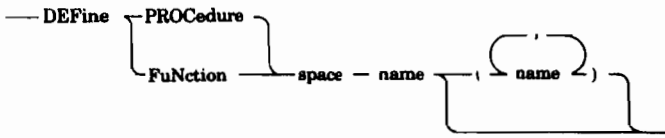


Figure 15.8 Syntax graph for an opening definition clause

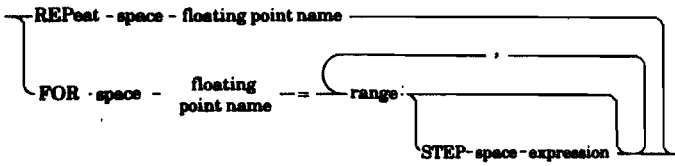


Figure 15.9 Syntax graph for an opening clause

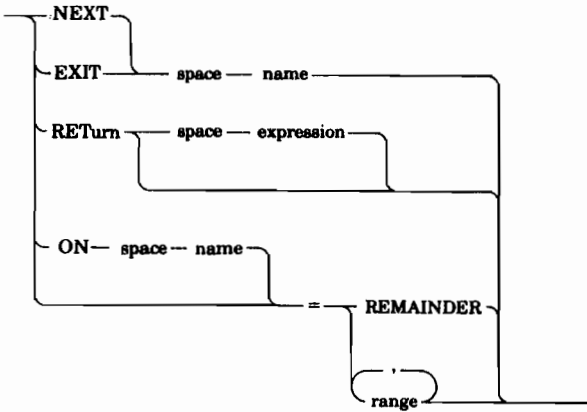


Figure 15.10 Syntax graph for an intermediate clause

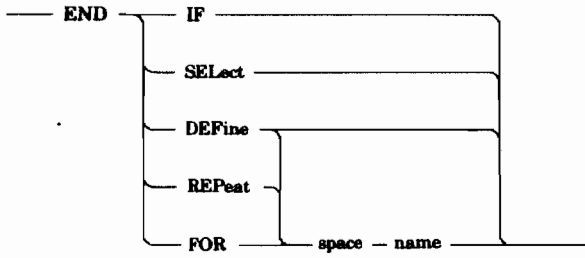


Figure 15.11 Syntax graph for an ending clause

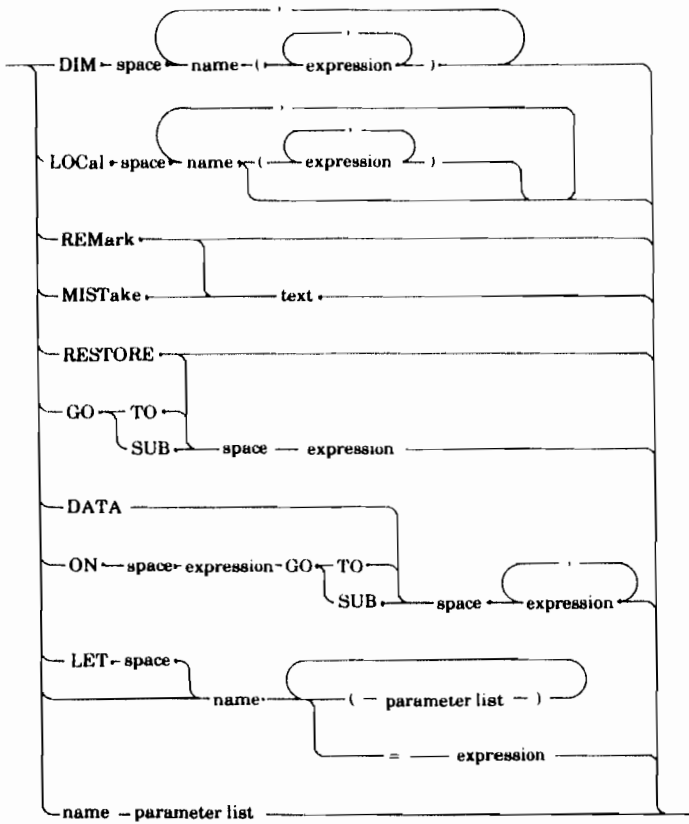


Figure 15.12 Syntax graph for a statement

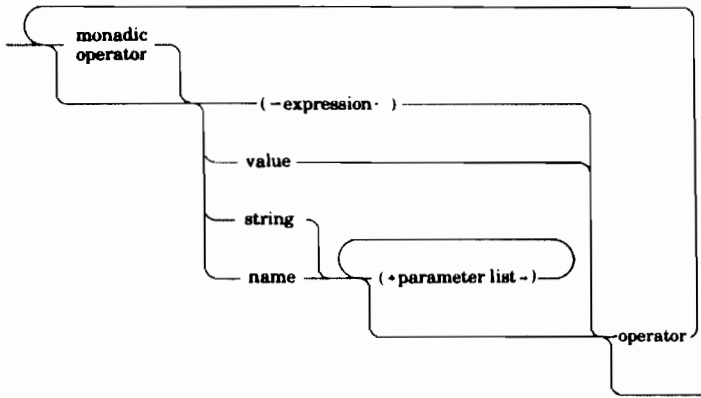


Figure 15.13 Syntax graph for an expression

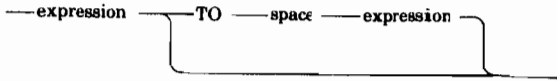


Figure 15.14 Syntax graph for a range

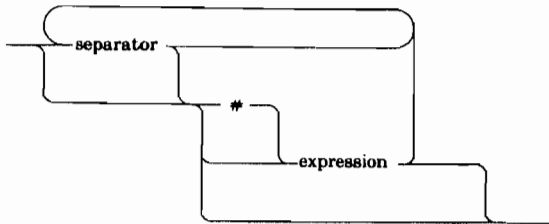


Figure 15.15 Syntax graph for a parameter list

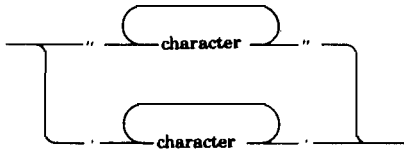


Figure 15.16 Syntax graph for a string

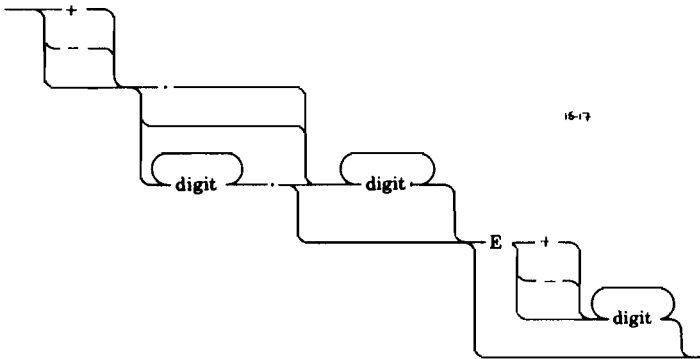


Figure 15.17 Syntax graph for a value

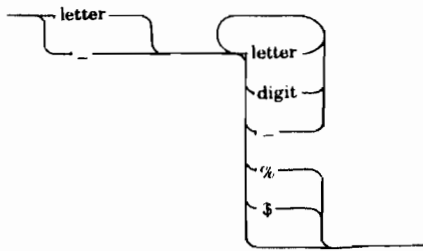


Figure 15.18 Syntax graph for a name

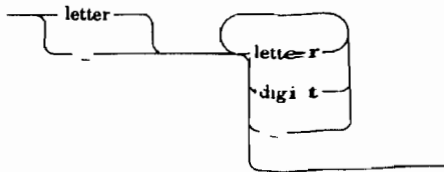


Figure 15.19 Syntax graph for a floating point name

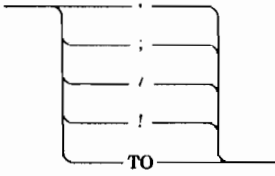


Figure 15.20 Syntax graph for a separator

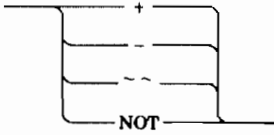


Figure 15.21 Syntax graph for a monadic operator

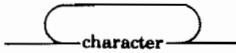


Figure 15.22 Syntax graph for text

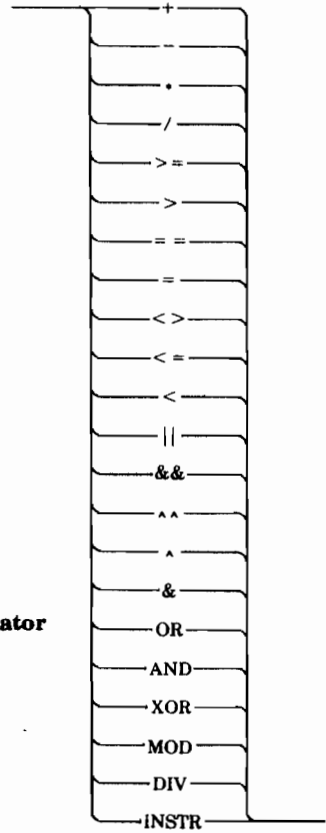


Figure 15.23 Syntax graph for an operator

16 THE KEYWORDS

'Keywords' is a loose term generally taken to mean some name which has a predetermined meaning. In the SuperBASIC system, such names might be built-in procedures or functions or actual structural keywords.

The following pages contain an alphabetical list of all the restricted names in the SuperBASIC system, giving their type, action and syntax. Only a brief explanation is made, you are referred to the chapter concerned for more information on the names in particular, or the group to which they belong in general.

ABS (expression)

Function

returning the absolute (positive) floating point value of the expression given. Chapter 5.

ACOS (expression)

Function

returning the floating point angle in radians of which the given expression is the cosine. Chapter 5.

ACOT (expression)

Function

returning the floating point angle in radians of which the given expression is the cotangent. Chapter 5.

ADATE seconds

Procedure

to adjust the total value of the clock-calendar by the given number of seconds. Chapter 13.

AND

Combination logical operator

working on two operands which are floating point true or false quantities, returning a floating point true or false result. Chapter 5.

ARC [#channel,] [x,y] ([,x,y] TO x,y,angle)

Procedure

to reset the current graphics cursor and draw an arc of subtended angle given anticlockwise between the two absolute graphics points specified. Default channel 1; default point is current graphics position. Chapter 11.

ARC_R [#channel,] [xr,yr] ([,xr,yr] TO xr,yr,angle)

Procedure

to reset the current graphics cursor and draw an arc of subtended angle given anticlockwise between two graphics points specified relative to the current graphics cursor. Default channel 1; default point is the current graphics position. Chapter 11.

ASIN (expression)

Function

returning the floating point angle in radians of which the given expression is the sine. Chapter 5.

AT [#channel,] row, column

Procedure

to move the text cursor invisibly to the character position given. Default channel 1. Chapter 10.

ATAN (expression)

Function

returning the floating point angle in radians of which the given expression is the tangent. Chapter 5.

AUTO [startline] [,increment]

Procedure

to start the automatic line number generator and editor. Default startline is 100; default increment is 10. Chapter 9.

BAUD baud_rate

Procedure

to reset the baud rate to that specified for transmitting and receiving. Chapter 10.

BEEP [beep_parameters]

Procedure

for producing sound. Lack of any parameters stops the sound sequence. Chapter 14.

BEEPING

Function

returning floating point true if the QL is beeping, floating point false otherwise. Chapter 14.

BLOCK [#channel,] width,height,x,y,colour

Procedure

for painting a block in the specified colour of the given size, with the top left-hand corner at the point given. All sizes and coordinates are in pixels relative to the window. Default channel 1. Chapter 10.

BORDER [#channel,] [pixel_depth [,colour]]

Procedure

to set a border of the given depth in a given colour around the specified window. Default channel 1; default depth is zero; default colour is transparent. Chapter 10.

CALL address [,register values]

Procedure

to transfer execution to machine code at the absolute address given. If further parameters are given, they will be put into the registers D1 to D7, A0 to A5. D0 should be 0 or error code on return. Chapter 12.

CHR\$ (code)

Function

to return a one-character string from the ASCII code given. Chapter 5.

CIRCLE [#channel,] x,y,radius

Procedure

to draw a circle of given radius about a given absolute centre point. Radius and position given in graphics units. Graphics cursor updated. Default channel 1. Chapter 11.

CIRCLE_R [#channel,] xr,yr,radius

Procedure

to draw a circle of given radius around a mid-point given relative to the current cursor position. Radius and position offsets given in graphics units. Graphics cursor updated. Default channel 1. Chapter 11.

CLEAR

Procedure

to clear out the variable values area, the arithmetic stack, the return stack, the data status and reset name types. Chapter 9.

CLOSE #channel

Procedure

to close the data stream open on the given channel and to release the channel number. Chapter 8.

CLS [#channel,] [window_area]

Procedure

to cover the given area of the specified window with the current paper colour for that channel. Default channel 1; default window area is the entire window. Chapter 10.

CODE (string)

Function

to return the integer ASCII code of the first character in the given string. Chapter 5.

CON [_size] [Aposition] [_bufferlength]

Device name

for a console window with an attached input buffer. Default name is CON_448x200a32x16_128. Chapter 8.

CONTINUE

Procedure

to continue execution from the statement after that which caused the last halt. Chapter 9.

COPY device TO device

Procedure

to copy information from an existing Microdrive file or other input peripheral device to a new Microdrive file or other output peripheral device. Chapter 8.

COPY_N file TO device

Procedure

to copy a Microdrive file without its header to an output peripheral device. Chapter 8.

COS (angle)

Function

returning the floating point cosine of an angle given in radians. Chapter 5.

COT (angle)

Function

returning the floating point cotangent of an angle given in radians. Chapter 5.

CSIZE [#channel,] with_mode , height_mode
Procedure

to select a different character size to be used in the specified window. Default channel 1. Chapter 10.

CURSOR [#channel,] [graphics_position,] pixel_position
Procedure

to move the text cursor in the specified window to the pixel offset required from the graphics position given. Default channel 1; default graphics position is the top left hand corner of the relevant window. Chapter 10. Chapter 11.

DATA expression {,expression}
Keyword

defining a set of constant data expressions within a program. Chapter 8.

DATE
Function

returning the total number of seconds (floating point) in the current date. Chapter 13.

DATE\$ [(seconds)]
Function

returning a string containing an intelligible date and time from the total number of seconds given. Default is the current date. Chapter 13.

DAY\$ [(seconds)]
Function

returning a string containing the abbreviated name of the day of the date given in seconds. Default is the current date. Chapter 13.

DEF[INE]
Keyword

starting off an opening definition clause. Also used in conjunction with END. Chapter 3.

DEF[INE] F[UNCTION] function_name [(argument {,argument})]
Keyword structure

of a complete opening definition clause for a user-defined function. Chapter 3.

DEF[INE] PROCEDURE procedure_name [(argument {,argument})]
Keyword structure

of a complete opening definition clause for a user-defined procedure. Chapter 3.

DEG (radians)
Function

returning the number of degrees (floating point) equivalent to the number of radians given. Chapter 5.

DELETE MDVn_filename
Procedure

to remove the specified file from the medium in the Microdrive slot given. Chapter 8.

DIM name(expression [,expression]) [,name(expression [,expression])]

Keyword

defining a set of dimensioned names, or arrays, together with the maximum size of each index allowed. Chapter 4.

DIMN (array_name [,index_number])

Function

returning the integer maximum value of the index required for the given array. Default index is the first. Chapter 4.

DIR [#channel,] MDVn_

Procedure

writing out on the specified channel a list of all the filenames on the medium in the specified Microdrive slot, together with the number of good sectors used and remaining. Default channel 1. Chapter 8.

DIV

Operator

performing an integer divide on two integer operands and producing an integer result. Chapter 5.

DELETE [startline] [TO [endline]] [,startline] [TO [endline]]

Procedure

to delete ranges of lines from the SuperBASIC program currently in memory. Default startline is 1; default endline is 32767. Chapter 9.

EDIT [startline] [,increment]

Procedure

to initiate the automatic line number generator and editor. Default startline is 100; default increment is 0. Chapter 9.

ELLIPSE [#channel,] x,y,major_radius,ratio,rotation

Procedure

to draw in the specified window an ellipse with the given major radius and minor to major ratio, around the central absolute graphics position specified at the given angle in radians anticlockwise to the vertical. All lengths are in graphics units. Graphics cursor updated. Default channel 1. Chapter 10.

ELLIPSE_R [#channel,] xr,yr,major_radius,ratio,rotation

Procedure

to draw in the specified window an ellipse with the given major radius and minor to major ratio, around the central position specified relative to the current graphics cursor, at the given angle in radians anticlockwise to the vertical. All lengths and offsets are in graphics units. Graphics cursor updated. Default channel 1. Chapter 10.

ELSE [statement]
 Keyword used **once** only in a clause as an intermediate **IF** clause to indicate the start of the alternative section of code. If the alternative code is not required, then it transfers control to the corresponding **END IF** instead. Chapter 6.

END
 Keyword used to start all ending clauses.

END DEF[INE] [name]
 Keyword structure of an ending definition clause. Also acts as a return from a procedure call. Chapter 3.

END FOR index_name
 Keyword structure of an ending **FOR** loop clause. Transfers control back to the beginning of the **FOR** loop. Chapter 7.

END IF
 Keyword structure of an ending **IF** clause. Also marks end of alternative code. Chapter 6.

END REP[EAT] index_name
 Keyword structure of an ending **REPEAT** loop clause. Transfers control back to the beginning of the **REPEAT** loop. Chapter 7.

END SEL[ECT]
 Keyword structure of an ending **SELECT** clause. Also marks end of all choices. Chapter 6.

END WHEN
 Keyword structure of non-implemented ending **WHEN** clause.

EOF [(/#channel)]
 Function returning floating point true if the end of the file on the specified channel has been reached, floating point false otherwise. Default is for **DATA** items within the current program. Chapter 8.

ERR[OR]
 Keyword used in the non-implemented **WHEN** construction.

EXEC device
 Procedure to initiate the independent machine code program coming from the device given. This can be **MDVn_filename** or any other input peripheral device. Chapter 12.

EXEC_W device
Procedure

to initiate the independent machine code program coming from the device given and to suspend SuperBASIC execution while it does so. The device can be MDVn_filename or any other input peripheral device. Chapter 12.

EXIT index_name
Keyword structure

of an intermediate FOR or REPEAT loop clause transferring execution to the statement after the end of the loop indicated by END FOR index_name or END REPEAT index_name. Chapter 7.

EXP (expression)
Function

returning the floating point value of the mathematical quantity e raised to the power of the expression given. Chapter 5.

FILL [#channel,] fill_mode
Procedure

to turn area filling on or off in the window given. Default channel 1. Chapter 10.

FILL\$ (string,length)
Function

returning a string of the specified length, filled with repeats of the first two characters in the string given. If there is only one character, repeats are all of that. Chapter 5.

FLASH [#channel,] flash_mode
Procedure

to turn the text flashing attribute on or off in the specified window. Default channel 1. Chapter 10.

FOR
Keyword

used to start an opening or inline FOR loop clause. Also used in conjunction with END. Chapter 7.

FOR index_name = range {,range} [: statements]
Keyword structure

of an opening FOR loop clause. If followed by any statements on the same line, it is an inline FOR loop. Chapter 7.

FORMAT [#channel,] MDVn_name
Procedure

to format the medium in the Microdrive slot defined, giving it the name specified and directing any message to the channel given. Default channel 1. Chapter 8.

[FUNCTION]
Keyword

used in conjunction with DEFINE. Chapter 3.

GO

Keyword starting an uncontrolled transfer statement. Chapter 6.

GO SUB line_number

Keyword structure transferring further execution to the line number given, returning later to the statement after this. Chapter 6.

GO TO line_number

Keyword structure transferring further execution to the line number given. Chapter 6.

IF

Keyword starting off an opening or inline IF clause. Also used in conjunction with END. Chapter 6.

IF expression [THEN] [:] [statements]

Keyword structure of an opening IF clause. If there are any further statements on the line, this is an inline IF clause. Chapter 6.

INK [#channel,] colour

Procedure to set the colour of the ink used in the given window to that specified. Default window 1. Chapter 10.

INKEY\$ [([#channel] [timeout])]

Function returning a single character (string) taken from the given channel during the specified time in fiftieths of a second. Default channel 1; default timeout 0. Chapter 8.

INPUT [#channel,] { [parameter] separator }

Procedure to input values from the specified channel. If channel is a console window, parameter expressions will be printed and separator formatting done, otherwise ignored. Default channel 1. Chapter 8.

INSTR

Operator returning integer position of first string operand inside second string operand. Chapter 5.

INT (expression)

Function returning the integer part of the floating point expression given. Chapter 5.

KEYROW (row_number)

Function returning the integer value of the state of the keys in the specified row (0 to 7) of the keyboard matrix. Chapter 8.

LBYTES device,address,length
Procedure

to load the given number of raw hexadecimal bytes from the Microdrive file or other input peripheral device specified into QL memory starting at the absolute address given. Chapter 12.

LEN (string)
Function

returning the integer length of the string expression given. Chapter 5.

[LET] variable = expression
Keyword structure

to assign the value of the expression given to a simple variable, array element or array string. Chapter 4.

LINE [#channel,] [x,y] [,x,y] TO x,y)
Procedure

to reset the current graphics cursor and to draw a straight line from one absolute graphics position to another. Default channel 1; default point is current graphics cursor. Graphics cursor updated. Chapter 11.

LINE_R [#channel,] [xr,yr] [,xr,yr] TO xr,yr)
Procedure

to reset the current graphics cursor and to draw a straight line from one graphics position relative to the current graphics cursor to another. Default channel 1; default point is current graphics cursor. Graphics cursor updated. Chapter 11.

LIST [#channel,] [startline] [TO [endline]] [, [startline] [TO [endline]]]
Procedure

to list the specified ranges of lines of the SuperBASIC program currently in memory to the channel given. Default channel 2; default startline is 1; default endline is 32767. Chapter 9.

LN (expression)
Function

returning the floating point logarithm to the base e of the floating point expression given. Chapter 5.

LOAD device
Procedure

to load a completely new SuperBASIC program into memory from the Microdrive file or input peripheral device given. Chapter 9.

LOCAL name[(expression (,expression))] [,name[(exp (,exp))]]
Keyword

defining a set of simple or dimensioned variables (arrays) to be used locally to a procedure or function. Chapter 4.

- LOG10** (expression)
Function returning the floating point logarithm to the base 10 of the floating point expression given. Chapter 5.
- LRUN** device
Procedure to load a completely new SuperBASIC program into memory from the Microdrive file or other input peripheral device given, and then to run from the top of the program immediately. Chapter 9.
- MDV**number_ [name]
Device name of a Microdrive. Optionally device name of a file on the medium inside the specified Microdrive slot. Chapter 8.
- MERGE** device
Procedure to merge a SuperBASIC program from the Microdrive file or other input peripheral device given with the SuperBASIC program currently in memory. Chapter 9.
- MISTAKE**] text
Keyword defining a line of illegal SuperBASIC. Chapter 9.
- MOD**
Operator returning the integer result of the first integer operand in the modulus of the second integer operand. Chapter 5.
- MODE** mode_type
Procedure to reset the mode of the screen to high or low resolution. Also clears various screen attributes. Chapter 10.
- MOVE** [#channel,] distance
Procedure to move the graphics cursor in the given window by the specified graphics units distance from the current graphics position in the current turtle graphics direction. Line will be visible or not according to PEN status. Default channel 1. Chapter 11.
- MRUN** device
Procedure to merge a SuperBASIC program from the Microdrive file or other input peripheral device given with the SuperBASIC program currently in memory and then run it immediately either from the top of the program or from the statement after the MRUN. Chapter 9.
- NET** number
Procedure to reset the network station number of the QL to the value given. Chapter 8.

NETI number	
Device name	of a network input channel from the network station with the station number given. Chapter 8.
NETO number	
Device name	of a network output channel to the network station with the station number given. Chapter 8.
NEW	
Procedure	to clear out almost everything. Chapter 9.
NEXT index_name	
Keyword structure	of an intermediate FOR or REPEAT loop clause directing execution back to the beginning of the FOR or REPEAT loop with the given name. Chapter 7.
NOT	
Monadic operator	working on a floating point true or false value and returning a floating point true or false answer. Chapter 5.
ON	
Keyword	starting an intermediate SELECT clause or a multiple uncontrolled transfer. Can also be combined with SELECT in an opening or inline SELECT clause.
[ON name] = range {,range} [:statements]	
[ON name] = REMAINDER	
Keyword structure	of an intermediate SELECT clause. Marks end of previous ON block. If unexpected will transfer execution to the ENDSELECT. Chapter 9.
ON expression GOTO/GOSUB expression {,expression}	
Keyword structure	of a multiple uncontrolled transfer. Chapter 6.
OPEN #channel, device	
Procedure	to open exclusively a device for input or output on the channel given. If the device is a Microdrive filename, that file must exist on the medium in the specified slot. Chapter 8.
OPEN IN #channel, device	
Procedure	to open a shared file for input only on the channel given. The file must exist on the medium in the specified Microdrive slot. Chapter 8.

OPEN_NEW #channel, device
Procedure

to open a new file for input or output on the channel given. The file must not exist already on the medium in the specified Microdrive slot. Chapter 8.

OR
Combination logical operator

returning a floating point true or false result from the two floating point true or false operands given. Chapter 5.

OVER [#channel,] overwriting_mode
Procedure

to change the overwriting mode in the window given. Default channel 1. Chapter 10.

PAN [#channel,] distance [,window area]
Procedure

to move the contents of the window area specified in the window given rightwards by the number of pixels defined. Vacated columns are filled with current paper colour. Distance must be even in low resolution mode. Default channel 1; default window area is entire window. Chapter 10.

PAPER [#channel,] main_colour [,contrast [,pattern]]
Procedure

to reset the colour of the paper in the given window to that specified. Colour may be composite instead. Also resets STRIP colour. Default channel 1; default contrast as main; default pattern is checkerboard. Chapter 10.

PAUSE [(timeout)]
Procedure

to halt execution until the time limit (in fiftieths of a second) is reached or until any key on the keyboard is pressed. Default timeout is -1 or indefinite. Chapter 8.

PEEK (address)
Function

returning the integer value of the byte at the absolute address in memory given. Chapter 12.

PEEK_L (even_address)
Function

returning the floating point value of the long word (four bytes) starting at the absolute address in memory given. Chapter 12.

PEEK_W (even_address)
Function

returning the integer value of the word (two bytes) starting at the absolute address in memory given. Chapter 12.

PENDOWN [#channel]

Procedure turning the turtle graphics pen on in the given window. MOVES will now be visible. Default channel 1. Chapter 11.

PENUP [#channel]

Procedure turning the turtle graphics pen off in the given window. MOVES will now be invisible. Default channel 1. Chapter 11.

PI

Function returning the floating point value of the mathematical quantity π . Chapter 5.

POINT [#channel,] x,y, {,x,y }

Procedure drawing a dot at the absolute graphics position given in the specified window. Graphics cursor updated. Default channel 1. Chapter 11.

POINT_R [#channel,] xr,yr, {,xr,yr }

Procedure drawing a dot at the graphics position given relative to the current graphics cursor in the specified window. Graphics cursor updated. Default channel 1. Chapter 11.

POKE address, expression

Procedure putting the value of the integer expression given into the byte at the specified absolute address in memory. Chapter 12.

POKE_L even_address, expression

Procedure putting the value of the floating point expression given into the long word (four bytes) starting at the specified absolute address in memory. Chapter 12.

POKE_W even_address, expression

Procedure putting the value of the integer expression given into the word (two bytes) starting at the specified absolute address in memory. Chapter 12.

PRINT [#channel,] [[parameter] separator]

Procedure to print the values of the expressions and do the separator formatting specified to the given channel. Default channel 1. Chapter 8.

PROCEDURE

Keyword used with definition clauses. Chapter 3.

- RAD** (expression)
Function returning the number of radians (floating point) equivalent to the number of degrees given. Chapter 5.
- RANDOMISE** [expression]
Procedure using an optional parameter to set a new random number seed. Default is a random seed. Chapter 5.
- READ** variable {,variable}
Procedure to assign the next DATA item to the variable, array element or array string specified in the read-list. Chapter 8.
- RECOL** [#channel,] list of eight colours
Procedure to change each pixel in the specified window to the colour determined by the new list of colours given. Default channel 1. Chapter 10.
- REMAINDER**
Keyword used as a range in the intermediate SElect clause. Chapter 6.
- REM[ARK]** [text]
Keyword defining a non-executable line of text. Chapter 3.
- RENUM** [[startline] [TO [endline]];] [new startline] [,increment]
Procedure to renumber the SuperBASIC program currently in memory, renumbering the startline given to the new starting line number specified, each succeeding line until the endline given being at the defined offset to the last. Default is RENUM 1 TO 32767;100,10. Chapter 9.
- REP[EAT]**
Keyword starting an opening or inline REpeat loop clause. Also used in conjunction with END. Chapter 7.
- REP[EAT]** index_name {statements}
Keyword structure of an opening REpeat loop clause. Any further statements on the line indicate an inline REpeat loop. Chapter 7.
- RESPR** (length)
Function to allocate the specified number of bytes in the resident procedure area and to return the absolute address in memory where they start. Chapter 12.
- RESTORE** [line_number]
Keyword resetting the current DATA pointer to the beginning of the line given. Default is top of current program. Chapter 8.

RETRY

Procedure

to restart execution at the beginning of the statement which caused the last halt. Chapter 9.

RET[URN] [expression]

Keyword structure

of intermediate definition clause. Transfers execution to the statement after a procedure call, the term after a function call, or the statement after a GOSUB statement. If returning from a function, the given expression is the value of that function, else ignored. No default allowed on a function return. Chapter 3. Chapter 4. Chapter 6.

RND [(lower_limit) TO upper_limit]

Function

returning a random floating point value between 0 and 1 if no parameters given, or a random integer between the lower and upper limit inclusive specified. Default lower limit is 0. Chapter 5.

RUN [line_number]

Procedure

to start execution of the current program at the line number given. Default is top of program. Chapter 9.

SAVE device [,range of lines]

Procedure

to save the specified ranges of lines from the SuperBASIC program currently in memory onto the given device. This may be an output peripheral channel or MDVn_filename. The filename must not already exist on the medium in the Microdrive slot given. Chapter 9.

SBYTES device,address,length

Procedure

to save the specified number of raw hexadecimal bytes starting from the absolute address in memory given onto the specified device. This may be an output peripheral channel or MDVn_filename. The filename must not already exist on the medium in the Microdrive slot given. Chapter 12.

SCALE [#channel,] height,coordinates of bottom left-hand corner

Procedure

to set the number of graphics units along the vertical edge of the window given, also the x and y graphical coordinate pair of the bottom left-hand corner of the window. Default channel 1. Chapter 11.

SCR [_size] [Aposition]

Device name

of a screen window. Default is SCR_448x200a32x16. Chapter 8.

SCROLL [#channel,] distance [,window area]

Procedure

to move the contents of the given window area of the specified window downwards by the given number of pixels. Vacated rows of pixels are filled with the current paper colour. Default channel 1; default window area is the entire window. Chapter 10.

SDATE year,month,day,hour,minute,second

Procedure

to reset the clock to the date and time given. Chapter 13.

SELECT

Keyword

starting an opening or inline Select clause. Also used in conjunction with END. Chapter 6.

SELECT [ON] name [= range {,range} {statements}]

Keyword structure

of an opening Select clause. The optional statements indicate an inline Select clause. Chapter 6.

SER [port] [parity] [handshaking] [protocol]

Device name

of a serial channel for RS-232-C use. Default SER1hr. Chapter 8.

SEXEC device,address,size,length of data space

Procedure

to save the specified length of executable machine code starting from the absolute address in memory given onto the specified device so that it may be later used with EXEC. The device may be an output peripheral channel or MDVn filename. The filename must not already exist on the medium in the Microdrive slot given. Chapter 12.

SIN (angle)

Function

returning the floating point sine of the angle in radians given. Chapter 5.

SQRT (expression)

Function

returning the positive floating point square root of the positive floating point expression given. Chapter 5.

STEP

Keyword

used in a FOR range defining the increment to be added to each succeeding value of the index variable. Chapter 7.

STOP

Procedure

directing SuperBASIC execution to halt and to return control to the QL console channel. Chapter 9.

STRIP [#channel,] main_colour [,contrast [,pattern]]

Procedure

to define the background, or highlighting, to text in the given window. Colour may be a composite instead. Default channel 1; default contrast is as main colour; default pattern is checks. Channel 10.

SUB

Keyword

used with GO. Chapter 6.

TAN (angle)

Function

returning the floating point tangent of the angle in radians given. Chapter 5.

THEN

Keyword

optionally used in an opening or inline IF clause. Chapter 6.

TO

Keyword

used in conjunction with GO. Chapter 6;
as part of a value range. Chapter 6. Chapter 7;
as a separator, in general, Chapter 3
- in particular for PRINT, Chapter 8
- as part of a line number range, Chapter 9
- with graphics procedures, Chapter 11.

TURN [#channel,] angle_r

Procedure

to turn the turtle graphics angle in the specified window by the given number of degrees anticlockwise from the current direction. Default channel 1. Chapter 11.

TURNTO [#channel,] angle

Procedure

to turn the turtle graphics angle in the specified window to the absolute number of degrees given anticlockwise from due East. Default channel 1. Chapter 11.

UNDER [#channel,] underlining_mode

Procedure

to set the text underlining attribute on or off in the given window. Default channel 1. Chapter 10.

VER\$

Function

returning a two character string giving the version letters of the current system ROM. Chapter 5.

WHEN

Keyword

used in the non-implemented WHEN construction.

WIDTH [#channel,] number of characters

Procedure

to set the width of the given channel to the specified number of characters. Default channel 1, but `_screen` windows will be ignored. Chapter 8.

WINDOW [#channel,] width, height, position of top left hand corner

Procedure

to change the size and position of the window attached to the given channel. All sizes and coordinates to be given in pixels. Default channel 1. Chapter 10.

XOR

Combination logical operator

producing a floating point true or false result from two floating point true or false operands.

Appendix A – THE CHARACTER SET, ASCII CODE AND CONVERSION

Decimal	ASCII code	Character	Keys pressed	Hexadecimal	ASCII code
0			CTRL ␣	0	
1			CTRL a	1	
2			CTRL b	2	
3		ch chan	CTRL c	3	
4			CTRL d	4	
5			CTRL e	5	
6			CTRL f	6	
7			CTRL g	7	
8			CTRL h	8	
9			CTRL i, tab	9	
10		line feed	CTRL j, enter	a	
11			CTRL k	b	
12			CTRL l	c	
13			CTRL m	d	
14			CTRL n	e	
15			CTRL o	f	
16			CTRL p	10	
17			CTRL q	11	
18			CTRL r	12	
19			CTRL s	13	
20			CTRL t	14	
21			CTRL u	15	
22			CTRL v	16	
23			CTRL w	17	
24			CTRL x	18	
25			CTRL y	19	
26			CTRL z	1a	
27			CTRL sh [(CTRL {),escape	1b	
28			CTRL sh \ (CTRL)	1c	
29			CTRL sh] (CTRL)	1d	
30			CTRL sh ␣ (CTRL ~)	1e	
31			CTRL sh esc (CTRL @)	1f	
32		blank	space	20	
33		!	sh 1 (!)	21	
34		"	sh ' (")	22	
35		#	sh 3 (#)	23	
36		\$	sh 4 (\$)	24	
37		%	sh 5 (%)	25	
38		&	sh 7 (&)	26	

Decimal	ASCII code	Character	Keys pressed	Hexadecimal	ASCII code
39		'	'		27
40		(sh 9 (()		28
41)	sh 0 ())		29
42		*	sh 8 (*)		2a
43		+	sh = (+)		2b
44		,	,		2c
45		-	-		2d
46		.	.		2e
47		/	/		2f
48		0	0		30
49		1	1		31
50		2	2		32
51		3	3		33
52		4	4		34
53		5	5		35
54		6	6		36
55		7	7		37
56		8	8		38
57		9	9		39
58		:	sh ; (:)		3a
59		;	;		3b
60		<	sh , (<)		3c
61		=	=		3d
62		>	sh . (>)		3e
63		?	sh / (?)		3f
64		@	sh 2 (@)		40
65		A	sh a (A if caps lock on)		41
66		B	sh b (B if caps lock on)		42
67		C	sh c (C if caps lock on)		43
68		D	sh d (D if caps lock on)		44
69		E	sh e (E if caps lock on)		45
70		F	sh f (F if caps lock on)		46
71		G	sh g (G if caps lock on)		47
72		H	sh h (H if caps lock on)		48
73		I	sh i (I if caps lock on)		49
74		J	sh j (J if caps lock on)		4a
75		K	sh k (K if caps lock on)		4b
76		L	sh l (L if caps lock on)		4c
77		M	sh m (M if caps lock on)		4d
78		N	sh n (N if caps lock on)		4e
79		O	sh o (O if caps lock on)		4f
80		P	sh p (P if caps lock on)		50
81		Q	sh q (Q if caps lock on)		51
82		R	sh r (R if caps lock on)		52
83		S	sh s (S if caps lock on)		53
84		T	sh t (T if caps lock on)		54
85		U	sh u (U if caps lock on)		55
86		V	sh v (V if caps lock on)		56
87		W	sh w (W if caps lock on)		57
88		X	sh x (X if caps lock on)		58
89		Y	sh y (Y if caps lock on)		59

Decimal ASCII code	Character	Keys pressed	Hexadecimal ASCII code
90	Z	sh z (Z if caps lock on)	5a
91	[[5b
92	\	\	5c
93]]	5d
94	^	sh 6 (^)	5e
95	_	sh - (_)	5f
96			60
97	a	a	61
98	b	b	62
99	c	c	63
100	d	d	64
101	e	e	65
102	f	f	66
103	g	g	67
104	h	h	68
105	i	i	69
106	j	j	6a
107	k	k	6b
108	l	l	6c
109	m	m	6d
110	n	n	6e
111	o	o	6f
112	p	p	70
113	q	q	71
114	r	r	72
115	s	s	73
116	t	t	74
117	u	u	75
118	v	v	76
119	w	w	77
120	x	x	78
121	y	y	79
122	z	z	7a
123	{	sh [(])	7b
124		sh \ ()	7c
125	}	sh] (})	7d
126	~	sh  (~)	7e
127		sh esc ()	7f
128		CTRL esc	80
129		CTRL sh 1 (CTRL !)	81
130		CTRL sh ' (CTRL ")	82
131		CTRL sh 3 (CTRL #)	83
132		CTRL sh 4 (CTRL \$)	84
133		CTRL sh 5 (CTRL %)	85
134		CTRL sh 7 (CTRL &)	86
135		CTRL ' ()	87
136		CTRL sh 9 (CTRL ())	88
137		CTRL sh 0 (CTRL))	89
138		CTRL sh 8 (CTRL *)	8a
139		CTRL sh = (CTRL +)	8b
140		CTRL , ()	8c

Decimal	ASCII code	Character	Keys pressed	Hexadecimal	ASCII code
141		à	CTRL -		8d
142		á	CTRL .		8e
143		â	CTRL /		8f
144		ã	CTRL 0		90
145		ä	CTRL 1		91
146		å	CTRL 2		92
147		æ	CTRL 3		93
148		ç	CTRL 4		94
149		è	CTRL 5		95
150		é	CTRL 6		96
151		ê	CTRL 7		97
152		ë	CTRL 8		98
153		ì	CTRL 9		99
154		í	CTRL sh ; (CTRL :)		9a
155		î	CTRL ;		9b
156		ï	CTRL sh , (CTRL <)		9c
157		ï	CTRL =		9d
158		ï	CTRL sh . (CTRL >)		9e
159		ï	CTRL sh / (CTRL ?)		9f
160		Ï	CTRL sh 2 (CTRL @)		a0
161		Ï	CTRL sh a (CTRL A)		a1
162		Ï	CTRL sh b (CTRL B)		a2
163		Ï	CTRL sh c (CTRL C)		a3
164		Ï	CTRL sh d (CTRL D)		a4
165		Ï	CTRL sh e (CTRL E)		a5
166		Ï	CTRL sh f (CTRL F)		a6
167		Ï	CTRL sh g (CTRL G)		a7
168		Ï	CTRL sh h (CTRL H)		a8
169		Ï	CTRL sh i (CTRL I)		a9
170		Ï	CTRL sh j (CTRL J)		aa
171		Ï	CTRL sh k (CTRL K)		ab
172		Ï	CTRL sh l (CTRL L)		ac
173		Ï	CTRL sh m (CTRL M)		ad
174		Ï	CTRL sh n (CTRL N)		ae
175		Ï	CTRL sh o (CTRL O)		af
176		Ï	CTRL sh p (CTRL P)		b0
177		Ï	CTRL sh q (CTRL Q)		b1
178		Ï	CTRL sh r (CTRL R)		b2
179		Ï	CTRL sh s (CTRL S)		b3
180		Ï	CTRL sh t (CTRL T)		b4
181		Ï	CTRL sh u (CTRL U)		b5
182		Ï	CTRL sh v (CTRL V)		b6
183		Ï	CTRL sh w (CTRL W)		b7
184		Ï	CTRL sh x (CTRL X)		b8
185		Ï	CTRL sh y (CTRL Y)		b9
186		Ï	CTRL sh z (CTRL Z)		ba
187		Ï	CTRL [bb
188		Ï	CTRL \		bc
189		Ï	CTRL]		bd
190		Ï	CTRL sh 6 (CTRL ^)		be
191		Ï	CTRL sh - (CTRL _)		bf

Decimal	ASCII code	Character	Keys pressed	Hexadecimal	ASCII code
192			←		c0
193			ALT ←		c1
194			CTRL ←		c2
195			ALT CTRL ←		c3
196			sh ←		c4
197			ALT sh ←		c5
198			CTRL sh ←		c6
199			ALT CTRL sh ←		c7
200			→		c8
201			ALT →		c9
202			CTRL →		ca
203			ALT CTRL →		cb
204			sh →		cc
205			ALT sh →		cd
206			CTRL sh →		ce
207			ALT CTRL sh →		cf
208			↑		d0
209			ALT ↑		d1
210			CTRL ↑		d2
211			ALT CTRL ↑		d3
212			sh ↑		d4
213			ALT sh ↑		d5
214			CTRL sh ↑		d6
215			ALT CTRL sh ↑		d7
216			↓		d8
217			ALT ↓		d9
218			CTRL ↓		da
219			ALT CTRL ↓		db
220			sh ↓		dc
221			ALT sh ↓		dd
222			CTRL sh ↓		de
223			ALT CTRL sh ↓		df
224			(caps lock)		e0
225			ALT caps lock		e1
226			CTRL caps lock		e2
227			ALT CTRL caps lock		e3
228			sh caps lock		e4
229			ALT sh caps lock		e5
230			CTRL sh caps lock		e6
231			ALT CTRL sh caps lock		e7
232			F1		e8
233			CTRL F1		e9
234			sh F1		ea
235			CTRL sh F1		eb
236			F2		ec
237			CTRL F2		ed
238			sh F2		ee
239			CTRL sh F2		ef
240			F3		f0
241			CTRL F3		f1
242			sh F3		f2

Decimal	ASCII code	Character	Keys pressed	Hexadecimal ASCII code
243			CTRL sh F3	f3
244			F4	f4
245			CTRL F4	f5
246			sh F4	f6
247			CTRL sh F4	f7
248			F5	f8
249			CTRL F5	f9
250			sh F5	fa
251			CTRL sh F5	fb
252			sh space	fc
253			sh tab	fd
254			sh enter	fe
255			ALT	ff

Except where stated, combining the ALT key with any other key on the keyboard produces a two character code. The first character is always 255 and the second is the code generated by the normal key press. ALT can thus be used to generate a completely alternative character set; each keystroke could be treated as a new command just by checking the first character of an INKEY\$ for 255, then using the second as an offset to a table, for example.

Appendix B – ERRORS AND THEIR MEANING

There are a finite number of SuperBASIC and QDOS error messages. In most cases, due to lack of space, the messages are a trifle terse. In several cases the messages are multi-purpose, the cause of the error could be a number of things. In one or two cases, the error message generated has nothing to do with what you have originally done wrong, but the fault has caused spin-off effects which eventually result in the error message.

Apart from the message, "out of memory", when a message is printed in the form

At line_number message

it means that the error has occurred while processing that line in the current program. You can LIST the line to get an idea of what the problem might be.

If the message is printed without an "At line_number" in front of it, it means that the error occurred on the direct keyboard command which you have just typed in. The line should still be visible on the console window, but you will have to retype it in order to execute it again.

"not complete"

This message only occurs when you have broken out of program execution, program listing or the AUTOMATIC line number generator. BREAKing is achieved by pressing the CTRL key and the space bar together. This is not an error message (unless you have managed to BREAK accidentally!), it just lets you know what's going on.

"out of memory"

My, you have been working hard. This message means exactly what it says. Something that you have done has exceeded the memory available. SuperBASIC immediately does a CLEAR to clear some space. This error can happen on any operation but favourite causes are too large or unnecessary dimensioning of arrays, too high a channel number when there are lower ones unused, too much resident procedure area assigned. Cures? Use low (starting at #3)

channels, check arrays carefully, buy an expansion board.

If you get an "out of memory" message when you are creating a program, there will be enough space to parse a couple of commands but probably not any more. If you don't want to have to reset the QL, use those commands wisely (like saving the program). You have been warned.

"out of range"

This error normally occurs when you are referring to elements of arrays or strings which either don't exist, or which you are not allowed access to (e.g., element zero of a string - Chapter 4). It is also produced by the RENUM command, or if any other line ranges used are not sensible. If pixel or character positioning would take you outside the relevant window, this error will be generated to inform you of the fact.

"buffer full"

Aha, you've reached the end of an input buffer. If this occurs while you are doing normal console input, get in touch with Sinclair Research Ltd. If it occurs while you are inputting to a different channel, assign a larger buffer (Chapter 8).

"channel not open"

Normally means that the channel which you are trying to write to, read from or even close, hasn't been opened. Could be a spelling mistake or a missing OPEN command.

"already exists"

Another file handling message. A file which was supposed not to exist has been found. You will have to delete it then re-try the command.

"not found"

This message is generated when SuperBASIC cannot find a file or device which it expects to exist (e.g., if you leave the 'MDVn_' off the front of a filename). It also occurs when an END FOR, END REPEAT or NEXT doesn't have a corresponding FOR or REPEAT line to go back to, or if you happen to delete a procedure definition line before calling the procedure. The name exists but cannot be found in the current program.

This message **does not** occur when an EXIT, IF, ELSE, SElect, ON or DEFine cannot find its ending clause. That is taken as a messy way of ending the program.

"in use"

A file or device cannot be assigned because it is already open on another channel, or is otherwise occupied. You will have to close it first, or use something different.

"invalid job"

QDOS message when accessing independent jobs.

"end of file"

This is caused by a number of things: the end of a file has been read instead of the expected data; there are no more DATA items in the current program; there are no executable lines after a DEFine statement.

"drive full"

You have managed to fill up a Microdrive medium. Close the channel to put an end-of-file marker on the file. Cure? Put another cassette in or buy a floppy disc.

"bad name"

Covers a multitude of sins. A "bad name" error is generated if you give a command which SuperBASIC takes to be a procedure call, but which is not defined in the nametable as being a procedure. This is also the message if a loop index is not a simple floating point variable. Causes are likely to be a mis-spelling, forgetfulness or, on rare occasions, if you have somehow managed to overwrite the nametable. This last might occur if you have loaded machine code into the wrong place, for example. The cure is to reset the QL.

"format failed"

QDOS has been unable to format a Microdrive medium. Assuming that this isn't because you pulled it out half-way through, then the tape itself

might be created in some way. Try formatting it again a couple of times, or use the other Microdrive slot. The cartridges are fairly delicate and ought to be treated carefully. If you are sure that it isn't your fault, then take the tape back and complain.

"Xmit error"

This stands for 'transmission error' and occurs when the parity set on an SER device doesn't match the parity required. Check the switches on the other (non-QL) end of the RS-232 cable.

"bad or changed medium"

This is a Microdrive access error. It occurs when you have, for instance, got a bad tape; removed the medium while a file on it was open; had a medium in the slot, but now have nothing and are accessing it.

This sort of error often generates two messages. One from the Microdrive handling software also giving the name of the medium and one from SuperBASIC.

"error in expression"

Well, this means exactly what it says, somewhere on the line there is an error in an expression. Unfortunately expressions make up about 90 per cent of any SuperBASIC line (in indices, as parameters, in FOR ranges, etc.), so it might not be apparent to you which one is wrong. Some common causes are :

- an unset variable in an expression (other than a single unset variable parameter which is allowed);
- a term which cannot be converted to the required type for an operand (especially remembering that a blank string does not convert to zero!);
- the whole expression is unable to be converted to the type required for an assignment (checking assignments and inputs to procedure parameters very carefully since the parameter might not be the same type as the argument name);
- using an array as a simple variable;
- mis-spelling a name;
- referring to a function that doesn't exist.

"overflow"

This occurs normally when you have divided a quantity by zero or when you are trying to cram a large (>32767) number into an integer variable or

"read only"

Sufficiently obvious, I think. You are trying to write to a device which you are only allowed to read from. You have either got the OPEN command wrong or you are using the wrong channel number or your device name needs changing or you are just doing the wrong thing.

"bad parameter"

Too many or too few parameters to a built-in procedure or function.

"bad line"

Normally, this error is given when a line of SuperBASIC fails to **parse**; when what you have typed does not correspond to any of the syntax graphs, in short, when a line is invalid SuperBASIC. When this happens, the line will be echoed for you to correct and re-send.

If an invalid line has been read in from a Microdrive file during LOAD or MERGE, the MISTake keyword is inserted in front of it and loading continues. When executed, a MISTake statement produces "bad line". The other case when "bad line" is given during execution is if a LOCAL line is found out of place. LOCALs must come before the first executable statement in a definition structure.

A very rare occurrence is when, whatever you type at the keyboard, you get "bad line" with no echo. The cause of this is very specific. You have re-opened #0 against all advice and RUN something. The only cure is to reset the QL.

OTHER ERRORS

LIST doesn't work - no current program or #2 closed.

AUTO doesn't work - re-opened #0. Reset QL.

RUN doesn't work - re-opened #0 or there is no current program or the program consists entirely of procedures.

Nothing being printed - ink is the same colour as the paper or the strip; window defined incorrectly. If in #0, type invisibly to get a different ink colour.

Rubbish being printed across network or serial lines - baud rate not set

Rubbish being printed across network or serial lines - baud rate not set correctly, cables not wired up properly, framing error (not enough stop bits being transmitted).

If the message "At line" is printed with no error message, it means that a non-existent error number has been generated from a machine code routine.

Appendix C – THE SUPERBASIC TOKENS

A line of SuperBASIC is completely tokenized. I have used the dollar symbol, \$, to indicate hexadecimal quantities. The tokens are as follows :

spaces	two bytes	\$80.number of spaces	1-\$7f
keywords	two bytes	\$81.keyword identifier	1-\$1f
		\$01 END	
		\$02 FOR	
		\$03 IF	
		\$04 REPeat	
		\$05 SELEct	
		\$06 WHEN	
		\$07 DEFine	
		\$08 PROCedure	
		\$09 FuNction	
		\$0a GO	
		\$0b TO	
		\$0c SUB	
		\$0e ERRor	
		\$11 RESTORE	
		\$12 NEXT	
		\$13 EXIT	
		\$14 ELSE	
		\$15 ON	
		\$16 RETurn	
		\$17 REMAINDER	
		\$18 DATA	
		\$19 DIM	
		\$1a LOCaI	
		\$1b LET	
		\$1c THEN	
		\$1d STEP	
		\$1e REMark	
		\$1f MISTake	

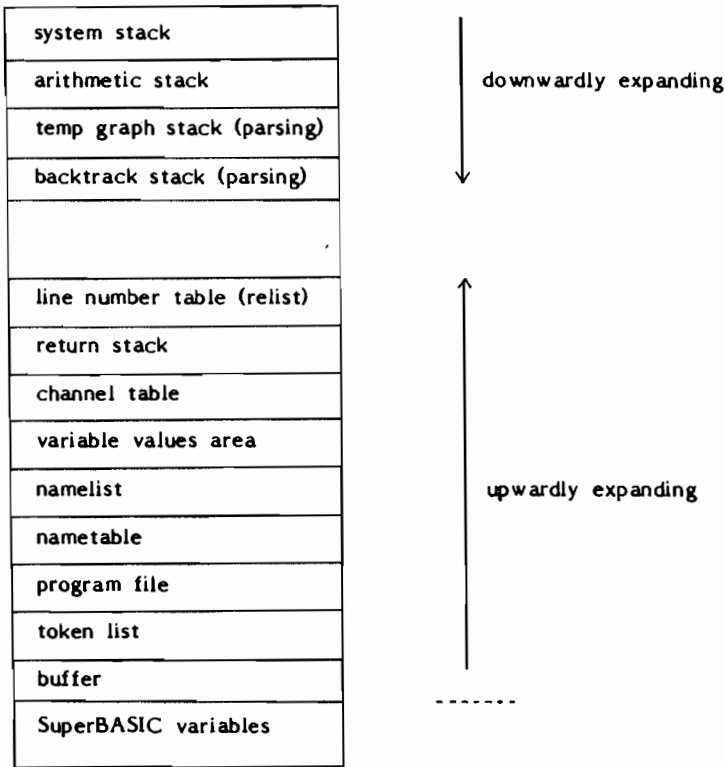
symbols	two bytes	\$84.symbol identifier	1-\$a
		\$01 =	
		\$02 :	
		\$03 #	
		\$04 ,	
		\$05 (
		\$06)	
		\$07 {	
		\$08 }	
		\$09 forced space	
		\$0a line feed (\$a)	
operators	two bytes	\$85.operator identifier	1-\$16
		\$01 +	
		\$02 -	
		\$03 *	
		\$04 /	
		\$05 >=	
		\$06 >	
		\$07 ==	
		\$08 =	
		\$09 <>	
		\$0a <=	
		\$0b <	
		\$0c	
		\$0d &&	
		\$0e ^^	
		\$0f ^	
		\$10 &	
		\$11 OR	
		\$12 AND	
		\$13 XOR	
		\$14 MOD	
		\$15 DIV	
		\$16 INSTR	
monadic ops	two bytes	\$86.monadic operator identifier	1-4
		\$01 +	
		\$02 -	
		\$03 ~~	
		\$04 NOT	
names	four bytes	\$88.00.no of entry in nametable	
strings	four bytes+ 1 byte/char+ spare if nec	\$8b.delimiter code.no of chars. chars in string if >0. unset to make even if necessary	
text	four bytes+ 1 byte/char+ spare if nec	\$8c.00.no of chars. chars of text. unset to make even if necessary	
line number	four bytes	\$8d.00.line number in hex	1-\$7fff

separators	two bytes	\$8e.separator identifier	1-5
		\$01 ;	
		\$02 ;	
		\$03 \	
		\$04 !	
		\$05 TO	
floating point	six bytes	top two bytes: exponent in powers of two offset above and below \$0800 \$f masked into top four bits (e.g., \$f800)	
		next four bytes: mantissa normalized, hexadecimal, left-justified	

In a tokenized program, each line is preceded by two bytes giving the change in length, in bytes, of this line from the last: 0 means that it is the same length; +n means that it is N bytes more; -m means that it is M bytes less.

Appendix D – THE SUPERBASIC STORAGE AREA

The SuperBASIC storage area is arranged thus :



The SuperBASIC variables area is a constant size. All of the other stacks, heaps and lists are constantly expanding and occasionally contracting.

Each area, apart from the SuperBASIC variables, has a base pointer and a running pointer. There is normally a breathing space between the running

pointer and the base pointer of the next area up. As more information is put into an area, this breathing space is eaten up until the running pointer bumps into the bottom of the area above. When this happens, the higher areas are moved up into the central space, leaving more room above the stack concerned.

When an area contracts, the running pointer falls, and the available space above it is increased. That space remains the property of the area in question and is not released except on a CLEAR or NEW (and sometimes not even then!).

If, during the expansion process, the central free area is completely used up, then SuperBASIC has to ask the QDOS system for extra space. When this is allocated, it is inserted into the central area and all of the upwardly expanding areas are correspondingly moved down to make room for it.

Because of the flexibility of this system, absolute addressing cannot be used within SuperBASIC. Every offset is relative to the base of its area, and every base and running pointer is relative to the base of the SuperBASIC area itself.

The SuperBASIC areas

BUFFER

Contains the actual command line typed or read in. Often used for conversions too.

TOKEN LIST

Internal SuperBASIC token list for the line typed or read in, minus any leading, trailing or forced spaces. If it is a program line, it will be transferred to the program file.

PROGRAM FILE

Complete tokenized program. Lines are in the form given in Appendix C.

NAMETABLE

Hub of system, exhaustively discussed in Chapter 4. Each entry is eight bytes long and is in the form

nametype	1 byte
separator type and variabletype	1 byte
pointer to offset of name in namelist/ copy entry	2 bytes
pointer to offset of value of variable in vv area/offset of array descriptor/address of mc procedure or function/ lno of SuperBASIC proc or fn	4 bytes

NAMELIST

List of names in the form

length of name (1 byte) characters in name (1 byte/character)

VARIABLE VALUES AREA

Heap of variable values, array descriptors, array values, temporary tables and loop information. Free space allocated and released by QDOS heap manager. First free space pointed to from SuperBASIC storage area, succeeding free spaces pointing to the next. Particular entries of interest are:

array descriptor - Chapter 4

offset in vv area of values	4 bytes
number of indices	2 bytes

then, for each index

{ maximum index value	2 bytes	}
{ distance between elements	2 bytes	}

REPEAT loop information - Chapter 7

index value	6 bytes
REPEAT line number	2 bytes
END REPEAT line number	2 bytes
statement on REPEAT line	1 byte
statement on END REPEAT line	1 byte

FOR loop information - Chapter 7

index value	6 bytes
FOR line number	2 bytes
END FOR line number	2 bytes
statement on FOR line	1 byte
statement on END FOR line	1 byte
end value of current range	6 bytes
step value of current range	6 bytes
position on line of range	2 bytes

CHANNEL TABLE

Holds information used by graphics and formatting routines

channel id (-1 if not open)	4 bytes
y value of graphics cursor	6 bytes
x value of graphics cursor	6 bytes
turtle angle	6 bytes
turtle pen status	1 byte
spare	9 bytes
character pos on line	2 bytes
width of line in characters	2 bytes
spare	4 bytes

RETURN STACK

Entries are in the form :

offset in nt of base of parameter entries	4 bytes	} SuperBASIC procedure and functions only
offset in nt of base of LOCAL entries	4 bytes	
offset in nt of top of LOCAL entries	4 bytes	
DEFine line number	2 bytes	
function type if applicable	1 byte	
whether args and pars swapped yet or not	1 byte	
routine type: 0-GOSUB,1-proc,2/3 - fn	1 byte	} all the above plus GOSUBs too
statement on calling line	1 byte	
calling line number	2 bytes	
status of calling line	4 bytes	

LINE NUMBER TABLE

Entries for each line of SuperBASIC printed to #2 in the most recent list range. Entries are in the form :

line number	2 bytes
no of window lines taken	2 bytes

BACKTRACK STACK

Keeps track of the progress while parsing a line. Enables retracing of steps to try another path.

TEMPORARY GRAPH STACK

Keeps track of syntax graphs entered during parsing.

ARITHMETIC STACK

Working area for the expression evaluator. Chapter 5.

SYSTEM STACK

Keeps track of machine code procedures and temporarily saved variables used by SuperBASIC. Very important stack.

THE LIST OF SUPERBASIC STORAGE VARIABLES OFFSETS IS AS FOLLOWS:

\$0	buffer base
\$4	buffer running pointer
\$8	token list base
\$c	token list running pointer
\$10	program file base
\$14	program file running pointer
\$18	nametable base
\$1c	nametable running pointer
\$20	namelist base
\$24	namelist running pointer
\$28	variable values base

\$2c variable values running pointer
\$30 channel table base
\$34 channel table running pointer
\$38 return stack base
\$3c return stack running pointer
\$40 line number base
\$44 line number running pointer

\$48 backtrack stack running pointer
\$4c backtrack stack base
\$50 temp graph running pointer
\$54 temp graph base
\$58 arithmetic stack running pointer
\$5c arithmetic stack base
\$60 system stack running pointer
\$64 system stack base

\$68 current line number
\$6a current length
\$6c current statement
\$6d continue or stop
\$6e inline or not
\$6f direct or not
\$70 index variable
\$72 first free vv area space
\$76 out of memory address

\$80 random number

\$84 command channel

\$88 line number to start at
\$8a statement to start at
\$8b command line saved or not
\$8c stop number
\$8e program edited or not
\$8f BREAK or not
\$90 need to unravel return stack or not
\$91 statement to CONTINUE from
\$92 line number to CONTINUE from

\$94 current DATA line number
\$96 statement on DATA line
\$97 item on DATA line

\$98 inline loop index for CONTINUE
\$9a inline loop flag for CONTINUE

\$9b checking listrange or not
\$9c invisible top line
\$9e bottom line in window
\$a0 invisible bottom line
\$a2 length of window line

\$a4 max no of window lines
\$a6 no of window lines so far
\$aa AUTO/EDIT on or off
\$ab print line or leave it in buffer
\$ac line number to edit next
\$ae increment on edit range
\$b0 pos in tklist on entry to procedure
\$b4 temp pointer for GO_PROC
\$b8 undo return stack and redo procedure or not
\$b9 up arrow, down arrow or enter
\$ba fill window when relisting to here
\$100 top of SuperBASIC variables area
CZ L ERNUM
CG W? ERLIN

E.O L TKZ ALCHP

EC L QLIB_RUN

Appendix E – THE QL MEMORY MAP

QDOS memory map (held in RAM) :

resident procedures
transient programs
SuperBASIC area
filing subsystem slave blocks
channels/common heap items
resource management tables/ system variables
display memory

Physical memory map :

ffff	128K add-on ROM
e0000 dffff	8x16K add-on peripheral devices
c0000 bffff	half a megabyte add-on RAM
40000 3ffff	(RAM 1)
30000 2ffff	RAM 0 / SCR 1 (QDOS area)
28000 27fff	SCR 0
20000 1ffff	I/O hardware
10000 0ffff	Plug-in ROM cartridge
0c000 0bfff	System ROM
00000	

INDEX

A

Abbreviations		
	of keywords	4
ABS		
	definition	40
	keyword	166
Accessing memory		
	See Memory access	
ACOS		
	definition	40
	keyword	166
ACOT		
	definition	40
	keyword	166
ADATE		
	definition	143
	keyword	166
Addresses in memory		137
Adjusting the clock		143
Almost equal to, ==		
	action	39
	operator	33
	token	197
ALT		
	code	189
	use with arrow keys	63
Alternative code		
	multiple, SElect	46
	simple, IF	44
Ampersand, &		
	code	184
	concatenation, action	39
	operator	33
	&&, bitwise AND operator	33,40

AND	bitwise, &&, action	40
	operator	33
	token	197
	logical, action	40
	keyword	166
	operator	33
	token	197
Angles	of ellipse to vertical	122
	radians to degrees to radians	40
	subtended in ARC	118
	in ARC_R	126
	trig functions of	40
	turtle angles	130
Animation	graphical	112
	via CURSOR, example	108
	via overwriting, example	122
	using SCROLL and PAN, example	110
Annotating	CURSOR, graphical	124
ARC	definition	118
	keyword	166
ARC_R	definition	126
	keyword	166
Arc-cosine	See ACOS	
Arc-cotangent	See ACOT	
Arc-sine	See ASIN	
Arc-tangent	See ATAN	
Area	codes of windows	103
	filling, with BLOCK	105
	with FILL	127
Arguments	in DEFine PROCedure lines	8
	in DEFine FuNction lines	12
	introduction to	18
	swapping with parameters	18
Arithmetic operators	list of	38
	tokens	197
Arithmetic (RI) stack	during expression evaluation	35
	position in memory	199

Arrays	descriptor	24
	dimensioning	22
	discussion	22
	element assignment	27
	in expressions	33
	order	24
	LOCAl arrays	31
	as parameters	30
	PRINTing of	60
	redimehsioning	25
	size of	22
	storage of values	23
	sub-arrays	25
	type of	23
Arrow keys	codes of	63,187
	used when creating programs	82,83
ASIN	definition	40
	keyword	166
Assignment	to arrays	27
	via INPUT	61
	via LET	16
	via READ	66
	reassignment	17
	to simple variables	16
AT	definition	101
	keyword	166
ATAN	definition	40
	keyword	166
AUTO	definition	81
	keyword	167
	used with LOAD	89
Automatic entry	of line numbers	81
	from file	88
	into program	91

B

Backslash, \	code	186
	separator, general	20
	used with PRINT	58
	syntax graph	164
	token	197
Backup	Microdrive	73
Bad line	in general	194
	on MISTake	88
	while parsing	11,80
BAUD	definition	76
	keyword	167
BEEP	definition	146
	keyword	167
	parameters	146
	exercisē procedure	150
BEEPING	definition	151
	keyword	167
Binary operators	See Logical operators	
Bits	making up bytes	5
	used in colour patterns	98,99
	used in internal representation	37
Bitwise operations	list of	40
	tokens	197
BLOCK	definition	105
	keyword	167
Booting	automatic file entry	91
BORDER	definition	104
	keyword	167
Boundary	word, even boundary	5
Brackets	in array indexing	22
	codes of	185,186
	in definitions	4
	in expressions	36

BREAK, CTRL and space together		
	breaking out of AUTO/EDIT	82
	breaking out of a bad line	80
	breaking out of BEEP	151
	breaking out of program execution	92
	breaking out of transmission	74
Broadcast		
	over the NETwork	79
	sound	146
Buffer		
	Command input	80
	CONsole keyboard	77
	peripheral	74
Bytes		
	composition	5
	description	5
	number of in values	37

C

Calendar and clock	reading and setting	142
CALL	definition	138
	keyword	167
Calling	a function	12
	independent machine code routines	139
	a machine code program	138
	a procedure	8,11
Case	comparison in strings	40
	construction - see SElect	
	conversion between	41
	equivalent in names	15
	lower, in examples	5
	in definitions	4
	upper, in examples	5
	in definitions	4
Changing input buffer		139
Changing program lines		80
Channels	brief introduction	58
	changing input channels	139
	closing	69
	end of file on	70
	opening	68
	range of	68
	reserved numbers	58
	table	202
Characters	colour of	101
	flashing	110
	positioning of, via AT	101
	via CURSOR, absolute	107
	via CURSOR, graphics	124
	positioning of, via PRINT	58
	in a name	15
	set	184
	size of	108
	underlining of	111
Checkerboard	as a colour pattern	99

Choice	multiple, SElect	46
	simple, IF	44
CHR\$	definition	41
	keyword	167
CIRCLE	definition	120
	keyword	167
CIRCLE_R	definition	126
	keyword	168
Clauses	discussion	43
CLEAR	definition	90
	keyword	168
CLOSE	definition	69
	keyword	168
Closing files on channels		69
CLS	definition, full	103
	simple	97
	keyword	168
CODE	definition	41
	with INKEY\$	62
	keyword	168
Coercion	forced conversion	37
Colon, :	code	185
	statement separator	14
	token	197
Colour	bit patterns	98
	of blocks	105
	of border	104
	components	98
	composite	99
	of ink	101,102
	in different modes	94,98
	of paper	101
	recolouring via OVER	106
	via RECOL	107
	of strip	101,102
Column	with AT	101
	number on a screen	95
	tabulation	58

Combination logical operators		
	list of tokens	40 197
Comma, ,		
	code separator, general	185
	with PRINT	20
	token	58 198
Commenting	REMark	13
Comparisons		
	arithmetic string	38 39
Complex clauses	discussion	43
Composite colour	formation of	99
Concatenation	of strings	33,39
CONsole		
	device names	76
	keyword	168
	window #0	95
Constants	legal	34
CONTINUE		
	definition	92
	keyword	168
Continuous looping	REPeat	51
Contrast colour		98
Control structures		
	general	43
	IF	44
	SELEct	46
CONTROL Z	to produce	74
Controlled looping	FOR	54
Controlled transfer		
	multiple, SELEct	46
	simple, IF	44
Controlling sound		151
Conversion		
	forced	37
	upper to lower case	42

Coordinates		
	graphics	114
	pixel	105
COPY		
	definition	75
	keyword	168
COPY_N		
	definition	75
	keyword	168
Copying		
	devices	75
	files	72
	with no header	75
Copyrights		3
COS		
	definition	40
	keyword	168
Cosine		
	See COS	
COT		
	definition	40
	keyword	168
Cotangent		
	See COT	
Creating a program		80
Credits		3
C_SIZE		
	definition	108
	keyword	169
CTRL key		
	with C	139
	with F5	61
	with space	80
	using	63
CTS line		
	to use	74
CURSOR		
	definition, full	124
	simple	107
	keyword	169
Cursor		
	codes	63,187
	keys	80,82
	list position	84,97
	graphics cursor	115
	text cursor, positioning with AT	101
	with CURSOR, absolute	107
	with CURSOR, relative	113
	with PRINT	58

Curves

drawing arcs	118
circles	120
ellipses	121
relative curves	125

D

DATA	definition	66
	keyword	169
Data	assigning, READ	66
	constant	65
	data pointer	66
	defining, DATA	66
	holding separately	67
	repeating, RESTORE	67
DATE	definition	143
	keyword	169
Date and time		142
DATE\$	definition	143,144
	keyword	169
DAY\$	definition	144
	keyword	169
Decimal	legal constant	34
	internal representation	37
	syntax graph	162
	token	198
DEFine	FuNction, definition	12
	keyword	169
	PROCedure, definition	8
	token	196
Definitions	clauses, discussion	43
	on RENUM	87
	skipping through	8
	of syntax	4
DEG	definition	40
	keyword	169
DELETE	definition	72
	keyword	169
Deleting	characters on a line	80
	files, DELETE	72
	lines	80
Descriptor	array	24

Depth		
	of border	104
DIM		
	definition	22
	keyword	170
	syntax graph	159
	token	196
Dimension		
	-ing an array	22
	max size of indices, DIMN	29
DIMN		
	definition	29
	keyword	170
DIR		
	definition	71
	keyword	170
Direct commands		
	from keyboard	11
	from Microdrive file	88
Direction		
	of graphics turtle	130
Directory		
	of files on a medium	71
DIV		
	action	38
	keyword	170
	operator	33
	token	197
Divide, /		
	action	38
	code	185
	integer, See. DIV	
	operator	33
	token	197
DLINE		
	definition	81
	keyword	170
Dots		
	as a colour pattern	99
	drawing, POINT	117
	POINT_R	126
DTR line		
	to use	74
Duration		
	of BEEP	146
	of time steps in BEEP	148

E

EDIT

definition 83
keyword 170

Editing

changing lines 80
deleting lines 78
EDIT 83
inserting lines 80,81
LISTing 83
renumbering, RENUM 85

ELLIPSE

definition 121
keyword 170

ELLIPSE_R

definition 126
keyword 170

ELSE

definition 44
keyword 171
syntax graph 154
token 196

END

DEFine 8,12
end clauses 43
FOR 54
IF 44
keyword 171
REPeat 51
SElect 46
syntax graph 158
token 196
WHEN 171

End of file

detection on a text file 74
EOF for DATA 66
EOF for files 70

End of program

action on 92

ENTER

use of 80,82

Entering lines

80

EOF

definition for channels 70
for DATA 66
keyword 171

Equals, =	almost equal to, ==	33,39
	as assignment	16,27
	code	185
	as intermediate SElect clause	46
	= as operator, action	39
	operator	33
	not equal to, <>	33,39
	token, operator	197
	symbol	198
Equivalent, ==	action	39
	operator	33
	token	197
ERRor	keyword	177
Errors	continuing from	93
	messages	190
Even boundary		5
Evened-up length		16
Examples	notes on	5
Exclamation mark, !	code	184
	separator, general	20
	with PRINT	58
	token	198
EXEC	definition	139
	keyword	172
EXEC_W	definition	139
	keyword	172
Execution	freezing screen during	61
	of functions	1264
	of independent machine code	139
	versus parsing	11
	pausing during	65
	of procedures	8,11
	of programs, RUN	90
	stopping	92
Execution window		95

EXIT	definition, FOR	54
	REPEAT	51
	keyword	172
	syntax graph	157
	token	196
Exiting	from FOR	57
	inline	57
	from REPEAT	52
	inline	54
EXP	definition	40
	keyword	172
Exponentiation function	See EXP	
Expressions	components	33
	definition	33
	evaluator	35
	monadic operators	34
	operators	33,38
	process of evaluation	35
	slicing string expressions	29
	sub-expressions	36
	syntax graph	160
	terms	33

F

File system		
	closing files	69
	copying files	72
	deleting files	72
	file names	69
	general	67
	listing files	71
	opening files	68
	reading from file	69
	writing to file	68
	See also Microdrive	
FILL	definition	127
	keyword	172
FILL\$	definition	41
	keyword	172
FLASH	definition	110
	keyword	172
Flashing text		110
Floating point		
	names	16
	syntax graph	163
	internal representation	37
	legal constants	34
	range of	37
	storage in variable values area	17
	syntax graph	162
	token	198
FOR	definition of complex clause	54
	of inline clause	57
	discussion	54
	index variable	55
	keyword	172
	range stepping	55
	syntax graph	156
	token	196
FORMAT	definition	71
	keyword	172
Frame rate	on INKEY\$	62
	on PAUSE	65
Framing error	on transmission	193

Freezing the screen		61
FuNction		
	definition	12
	in expressions	36
	keyword	172
	introduction to	12
	nesting of calls	13
	of definitions	13
	parameters of (See Parameters)	
	returning from	12
	syntax graph	155
	token	196
	type of	12
	using	12
Fuzz		
	as a BEEP parameter	149

G

GOSUB	definition	49
	keyword	173
	ON.....GOSUB	49
	renumbering of	86
	syntax graph	159
	token	196
GOTO	definition	48
	keyword	173
	ON.....GOTO	49
	renumbering of	86
	syntax graph	159
	token	196
Graphics	annotating	124
	ARC	118
	ARC_R	126
	CIRCLE	120
	CIRCLE_R	126
	CURSOR	124
	cursor position	115,116
	ELLIPSE	121
	ELLIPSE_R	126
	FILL	127
	LINE	116
	LINE_R	126
	MOVE	130
	off-window drawing	117
	PENDOWN	129
	PENUP	130
	POINT	117
	POINT_R	126
	relative graphics	125
	SCALE	114
	TURN	130
	TURNT0	130
	turtle graphics	129
	units	114
Greater than, >	action	39
	code	185
	operator	33
	token	197
Greater than or equal to, >=	action	39
	operator	33
	token	197

H

Handshaking	on network lines	78
	on serial lines	74
Hardware map		207
Hash, #	code	184
	storage of	21
	token	197
	using with channels	58
Header	copying without	75
	on files	76
Height	of a block	105
	of a border	104
	of a character	108
	of a graphics window	114
	of a window	96
High resolution	character size in	108
	description	94
Highlighting	characters, See STRIP	
	windows, See BORDER	
Horizontal stripes	as a colour pattern	99

Identifiers	See Names	
	of loops, See Index variable	
IF	definition of complex clause	44
	of inline clause	45
	discussion	44
	keyword	173
	process of choice	44
	syntax graph	154
	token	196
Illegal values	lines	90
	numbers	34
	strings	34
Increments	with AUTO	81
	with EDIT	83
	with FOR ranges	55
	with RENUM	85
Independent machine code routines		139
Index	array indices	22
	maximum size of	29
	string indices	27
	sub-array indices	25
Index variable	of FOR, description	55
	storage	55
	of REPEAT, description	51
	storage	52
INK	definition	101
	keyword	173
	XORed with background	102
INKEY\$	definition	62
	from file	70
	keyword	173
Inline clauses	discussion	43
INPUT	from console windows	61
	definition	61
	from file	69
	keyword	173
	separators	61

Input buffer		
Inputing	for editing	80
	values, See INPUT	
	single characters, See INKEY\$	
	multiple characters, KEYROW	64
	program lines, from keyboard	80
	from Microdrive	88
INSTR		
	action	39
	keyword	173
	operator	33
	token	197
INT		
	definition	40
	keyword	173
Integer		
	arrays	23
	divide, DIV	38
	internal representation	37
	names	16
	operators	38
	range of values	37
	storage	17
Intermediate clauses		
	discussion	43
	syntax graph	157
Internal		
	representation	37
	storage units	5
	tokens	196
Invisible		
	positioning of graphics cursor	
	LINE	116
	LINE_R	126
	PENUP	130
	positioning of print cursor	
	AT	101
	CURSOR, absolute	107
	graphics offset	124
	printing, INK	101

J

Joining files		
	with MERGE	88
Joining strings		
	concatenation, action	39
	operation	33

K

Keyboard matrix	layout	64
Keyboard queue	with console device	77
KEYROW	definition	64
	keyword	173
Keywords	abbreviations in definitions	4
	list of restricted names	165
	tokens	196
	versus procedures and functions	12

L

LBYTES	definition	137
	keyword	174
LEN	definition	29
	keyword	174
Length	of floating points, internal representation	37
	storage	17
	of integers, internal representation	37
	storage	17
	of keyboard queue	77
	of strings	38
	of strings in arrays	23
Less than, <	action	39
	code	185
	operator	33
	token	197
Less than or equal to, <=	action	39
	operator	33
	token	197
LET	definition, array assignment	27
	simple assignment	16
	keyword	174
	syntax graph	159
	token	196
Letters	also see Case	
	also see Characters	
	codes	185
	conversion	42
	in a name	16
LINE	definition	116
	keyword	174
Line numbers	used when editing	80
	used with GOSUB	49
	used with GOTO	48
	range of	8
	range with AUTO	81
	with DLINE	81
	with LIST	83
	with RENUM	85
	with RUN	90

	with SAVE	88
	token	196
	when used	5
LINE_R		
	definition	126
	keyword	174
Lines		
	drawing, absolute	116
	relative	125
	turtle	129
	program, editing	80
	listing	83
	syntax graph of	154
Linking		
	machine code	138
	two programs	88
LIST		
	definition	83
	keyword	174
List		
	list range	84
	relisting	84
	list window	95
	effect on	97
Listing		
	Microdrive names	71
	programs	83
LN		
	definition	41
	keyword	175
LOAD		
	definition	88
	keyword	174
Loading		
	from Microdrive	88
	with AUTO	89
	direct commands	89
	across peripherals	91
LOCAl		
	definition, full	32
	simple	21
	keyword	174
	syntax graph	158
	token	196
Local variables		
	arrays	31
	discussion	21
	swapping	21
	using	21
LOGIO		
	definition	41
	keyword	175

Logarithms		
Logical operators	See LOG10, LN	
	bitwise	40
	combinational	39
	relational	40
	tokens	197
Long word		
Loops	as storage unit	5
	continuous	51
	controlled	54
	discussion	51
	loop epilogue	57
	loop identifier	51,55
	storage of information	52,55
Low resolution		
	character size in	108
	description of	94
Lower case		
	checking for	42
	comparisons	39
	converting to	42
	used in definitions	4
	in examples	5
	in names	15
LRUN		
	definition	90
	keyword	175

M

Machine code	discussion	137
	executing independently	139
	procedures and functions, linking in	138
	type of	16
Main	colour	98
	pitch	147
Major radius		121
Mathematical functions	list of	40
Maximum	characters in a name	15
	in a Microdrive name	71
	in a string	38
	index in array	22
	in string	28
	line number	8
	number of jobs	139
	number of network stations	78
	size of values	37
MDVn_	keyword	175
	See Microdrive	
Memory access	via LBYTES	137
	via PEEK	140
	via POKE	140
	via SBYTES	137
MERGE	definition	88
	keyword	175
Microdrive	backup	73
	booting from	91
	copying files on	72
	deleting files on	72
	direct commands from	88
	formatting	71
	listing names on	71
	loading programs from	88
	names of files	69
	of mediums	71
	opening files on	68
	reading from files on	69
	saving programs onto	87
	writing to files on	68

Minor radius		121
Minus, -		
	code	185
	diadic, action	38
	operator	33
	token	197
	monadic, operator	34
	token	197
	part of a number	34
MISTake		
	definition	80
	keyword	175
	syntax graph	159
	token	196
MOD		
	action	38
	keyword	175
	operator	33
	token	197
Mode		
	definition	94
	keyword	175
Modulus		
	See MOD	
Monadic operators		
	list	34
	syntax graph	164
	tokens	197
Monitor screen		
	default display	95
	mode	94
	size of in pixels	94
Motorola chip		5
MOVE		
	definition	130
	keyword	175
Moving contents of windows		109
MRUN		
	definition	90
	keyword	175
Multi-line clauses		43
Multiple choice		46
Multiple entry to procedure		10

Multiply, *

action
code
operator
token

38
185
33
197

N

Namelist	definition of entries	15
	pointers to	15
	position in storage area	199
	table	201
Namepass		
Names	discussion	18
	definition of	16
	of files	69
	of functions	12
	introduction to	15
	of loop identifiers	51
	of mediums	71
	of procedures	7
	when set up	15
	syntax graph of	163
	tokens	197
	assigning values to	16
Nametable	construction	15
	definition of entries	15
	during expression evaluation	35
	pointers to	15
	position in storage area	199
	table	201
Nametypes	of procedures and functions	17
	from parameter swapping	20
	from usage	16
Nesting	calls	9
	clauses	43
	DEFinitions	9
	IFs	44,45
	FORs	57
	REPeats	53,54
	SELeCts	47
NET	definition	78
	keyword	175
NETI	keyword	176
	usage	78
NETO	keyword	176
	usage	78

Network		
	broadcasting	79
	general	78
	station numbers	78
NEXT		
	definition, with FOR	54
	with REPEAT	51
	keyword	176
	syntax graph	157
	token	196
NEW		
	definition	90
	keyword	176
New Microdrive tapes		
	formatting	71
Non-reentrance		
	discussion	127
NOT		
	bitwise, ^^, action	40
	operator	33
	token	197
	logical, action	40
	operator	33
	keyword	176
	token	197
Not complete		
	Also see BREAK	
	general	190
Not equal to, <>		
	action	39
	definition	33
	token	196
Number		
	codes of	185
	legal	34

O

Off-window drawing	117
ON	
definition	46
with GOSUB	49
with GOTO	49
with SElect	46
keyword	176
syntax graph	157
with GOSUB/GOTO	159
with SElect	154
token	196
OPEN	
definition	68
keyword	176
OPEN_IN	
definition	68
keyword	176
OPEN_NEW	
definition	68
keyword	177
Opening channels	
to files	68
over network	78
over peripherals	74
Opening clauses	
discussion	43
syntax graphs	154,156
Operators	
action of arithmetic	38
bitwise	40
integer	38
logical	39
string	39
list of	33
monadic	34
syntax graph of	164
tokens	197
precedence in expression	33
storage during evaluation	35
syntax graph	164
token	197
type of operands	33

OR		
	bitwise, , action	40
	operator	33
	token	197
	exclusive or, See XOR	
	logical, action	40
	keyword	177
	operator	33
	token	197
Other books		2
Out of memory		
	general	190
Out of range		
	general	191
Output		
	to console windows	58
	to file	68
	to peripherals	74
OVER		
	definition	102
	keyword	177
Overlaying windows		97
Overwriting modes		102

P

PAN	definition	109
	keyword	177
PAPER	definition	101
	keyword	177
Parameters	arrays as	30
	brief description	10
	entries on nametable	19
	parameter list	11
	syntax graph	161
	returning	19
	swapping with arguments	19
Parity		
Parsing	over serial lines	74
	definition	11
	errors during	80
	name creation during	15
Pattern		
PAUSE	of colour pixels	99
	definition	65
	keyword	177
Pausing	during execution	65
	by freezing screen	61
PEEK	definition	140
	keyword	177
PEEK_L	definition	140
	keyword	177
PEEK_W	definition	140
	keyword	177
PENDOWN	definition	130
	keyword	178
PENUP	definition	129
	keyword	178
Peripherals	device names	73
	using	74
	with current program	91

PI	definition	41
	keyword	178
	section of circle	119
Pitch	main pitch with BEEP	147
	secondary	147
	pitch step	148
Pixels	number in a border	104
	number in a character	108
	used with colours	98
	number in a CON/SCR device	76
	counted from for blocks	105
	for CURSOR, absolute	107
	for CURSOR, graphics	124
	for windows	96
	number in a graphics unit	114
Plus, +	number on a screen	76,94
	code	185
POINT	diadic, action	38
	operator	33
	token	197
	monadic, operator	34
	token	197
	part of number	34
POINT_R	definition	117
	keyword	178
POINT_R	definition	126
	keyword	178
Pointers	to array values	24
	in nametable entries	15
	to nametable entries	19
POKE	definition	140
	keyword	178
POKE_L	definition	140
	keyword	178
POKE_W	definition	140
	keyword	178
Ports	definition	140
	keyword	178
	network	78
	serial	73

Positioning		
	of blocks	105
	of border	104
	of text by AT	101
	by CURSOR, absolute	107
	by CURSOR, graphics	124
	by PRINT separators	58
	of windows	96
	within windows	105
Power, ^		
	action	38
	code	186
	operator	33
	token	197
PRINT		
	definition	58
	keyword	178
Printing		
	arrays	60
	to console windows	58
	to file	68
	parameters	59
	to peripherals	75
	separators	58
	effect on windows	97
PROCedure		
	definition	8
	keyword	178
	syntax graph	155
	token	196
Procedures		
	calling	8,11
	definition	8
	introduction	7
	nametable entry	18
	names of	7
	nesting of calls	9
	definitions	9
	parameters, See Parameters	
	return stack	12
	returning from	12
	single entry point	9
	skipping through	9
	type of	17
Program		
	creation	80
	brief introduction	11
	running	90
Protocol		
	serial	74

Q

QDOS

filing system	67
memory map	206
property of	3
trademark	3

QL

available from	3
trademark	3

Quotes

codes	184
string delimiters	34

R

RAD	definition	41
	keyword	179
Radians	used in angles	119
Radius	of circle	120
	major, minor of ellipse	121
Random	notes in BEEP	150
	numbers	41
RANDOMISE	definition	41
	keyword	179
Range	of floating point	37
	of indices, array	22
	string	27
	sub-array	26
	of integers	37
	of lines, See Line numbers	
	of loop variables	55
	of an ON	46
Ratio	minor to major radius of an ellipse	122
READ	definition	66
	keyword	179
Reading	the clock	142
	from DATA items	66
	from devices	75
	from file	69
	with INKEY\$	62
	with INPUT	61
	with KEYROW	63
	with PAUSE	65
Receiving	over network lines	78
	over serial lines	74
RECOL	definition	107
	keyword	179
Recolouring	with OVER	106
	with RECOL	107
Recovering	from errors	93

Registers		
	assembler	138
Relational operators		
	action of	39
	list of	33
	tokens of	197
Relative drawing		
	discussion	125
	with turtle	129
REMAINDER		
	action	47
	definition	46
	keyword	179
	syntax graph	157
	token	196
REMark		
	definition	13
	keyword	179
	syntax graph	159
	token	196
RENUM		
	definition	85
	keyword	179
ReNUMBER table		86
REPeat		
	definition, complex	51
	inline	54
	keyword	179
	syntax graph	156
	token	196
Repeating		
	BEEP sweeps	149
	loops, continuous	51
	controlled	54
Resident program area		
	position in memory	206
	using	138
Resolution		
	low, high	94
RESPR		
	definition	138
	keyword	179
RESTORE		
	definition	67
	keyword	179
	renumbering of	87
	syntax graph	159
	token	196

Restoring	DATA items	67
	programs	88
	and running	90
RETRY	definition	93
	keyword	180
RETurn	definition, from function	12
	from procedure	8
	keyword	180
	syntax graph	157
	token	196
Return stack	cleared out	93
	description	12
	function entry	12
	GOSUB entry	49
	after error	93
	position in memory	199
	procedure entry	12
	table	202
Returning	from a function	12
	from a GOSUB	49
	from a procedure	8
RI stack	during expression evaluation	35
	position in memory	199
RND	definition	41
	keyword	180
Rotation	of ellipse	122
Row	of keyboard matrix	63
	of window	95
	with AT	101
RS-232-C	serial device	74
RUN	definition	90
	keyword	180
Running	loading and	90
	machine code	138
	independently	139
	procedures	8,12
	programs	90

S

SAVE	definition	87
	keyword	180
Saving	bytes, See SBYTES	
	executable code, See SEXEC	
	to Microdrive	87
	across peripheral lines	91
SBYTES	definition	137
	keyword	180
SCALE	definition	114
	keyword	180
Scenic procedures	examples	133
SCR	definition of a screen device	77
	keyword	180
Screen	device	77
	modes	94
	windows	94
	colours in	98
	default	95
	redefining	96
SCROLL	definition	109
	keyword	181
SDATE	definition	142
	keyword	181
Secondary pitch	BEEP parameter	147
Seconds	in dates	143
Sectors	formatted on Microdrive	71
	used/left	72
SElect	definition, complex	46
	inline	47
	keyword	181
	ranges	46
	choosing	46
	matching	47
	syntax graph	154
	token	196

Semicolon, ;	code	185
	separator, general	20
	for PRINT	58
	for RENUM	85
	token	198
Separators	between parameters	20
	storage with parameters	20
	syntax graph	164
	token	198
	types	21
SER	device names of serial links	73
	keyword	181
SEXEC	definition	139
	keyword	131
Shift		
Shifting	combination codes	63
Simple	contents of windows	109
	choice	44
	clauses	43
	input	60
	output	58
SIN	definition	41
	keyword	181
Sinclair Research Ltd		
	credits to	3
Sine	See SIN	
Size		
	of a block	105
	of a border	104
	of a character	108
	of an index	22
	of a string	38
	of a window	96
Slicing		
	arrays	25
	strings	27
Sound		
	making it	146
	stopping it	151

Space	code	184
	forced	11
	in syntax graphs	154
	leading, trailing	11
	token	197
	in variable values area	17
Spacing out code		
Square root	ways of	14
SQRT	See SQRT	
	definition	41
	keyword	181
Stacks		
	arithmetic, with evaluator	35
	position in memory	199
	system, with evaluator	35
	position in memory	199
Station number		
	assigning	78
	for broadcast	79
	incorporating into device	78
STEP		
	definition with FOR	54
	keyword	181
	syntax graph	156
	token	196
Stipple		
STOP	pattern of colours	99
	definition	92
	keyword	181
Stopping execution		
	accidentally	192
	by BREAKing	92
	with an error	93
	by freezing the screen	63
	with PAUSE	65
	on purpose	92
Storage		
	of array descriptor	24
	of array values	23
	of loop information, FOR	55
	REPeat	52
	memory map	206
	of SuperBASIC items	199
	units	15
	of variables	16
	in variable values area	

STRIP	definition	101
	keyword	182
	transparent	106
Stripes	as colour patterns	99
String	arrays	23
	expressions	29
	indexing	27
	internal representation	37
	legal	34
	length of	29
	names	16
	operators	39
	tokens	197
	as parameters	28
	substrings	28
	syntax graphs	162
	as term of expression	34
	token	197
	variable storage	17
SUB	definition	49
	keyword	182
	syntax graph	159
	token	196
Sub-arrays	definition	25
	as parameters	30
Sub-expressions	discussion	36
Substrings	of arrays	27
	of strings	27
Subtended angle	of ARC	118
	of ARC_R	126
SuperBASIC	introduction to	1
	property of	3
Swapping	arguments and parameters	19
	LOCAL names	21
Syntax	notes on definition of	4
	the syntax graphs	152
	when used	11

T

Tabulating	via AT	108
	via CURSOR, absolute graphics	107 124
	via PRINT separators	58
TAN	definition	41
	keyword	182
Tangent	See TAN	
Television screen	default display	95
	mode	94
	size	95
	relative to monitor	96
Terms	of expression	33
Text	changing size of	108
	flashing	110
	with MISTake	80
	positioning, See Tabulating	
	printing	58
	with REMark	13
	syntax graph	164
	token	197
	underlining	111
THEN	definition, complex IF	44
	inline IF	45
	keyword	182
	syntax graph	154
	token	196
Time	duration of BEEP	148
	on INKEY\$	62
	on PAUSE	65
	units	62
TO	with GO	48
	syntax graph	159
	in index ranges	25
	syntax graph	161
	keyword	182
	in loop ranges	54
	syntax graph	161
	in ON ranges	46
	syntax graph	161
	as separator, general	20

	with ARC	118
	with ARC R	126
	with COPY	75
	with DLINE	81
	with INPUT	61
	with LINE	116
	with LINE_R	126
	with LIST	83
	with PRINT	58
	with RENUM	85
	with SAVE	88
	token, keyword separator	196
Toolkit		198
	available from	3
Transfer of control		
	via ELSE	44,45
	via ENDFOR	56
	via ENDREP	51
	via EXIT	52,57
	via GOSUB	49
	via GOTO	48
	via IF	44
	via NEXT	52,56
	via ON	47
	via RUN	90
	via SElect	46
Transmitting		
	over network	78
	over serial lines	74
Transparent		
	borders	104
	moves	130
	strips	102
	tabulating, See Tabulation	
Trigonometric functions		
	list of	40
Turtle graphics		129
TURN		
	definition	131
	key word	182
TURNT0		
	definition	131
	key word	182

U

Unconditional transfer	48
Uncontrolled structures	
description	48
GOSUB	49
GOTO	48
ON.....GOTO/SUB	49
UNDER	
definition	111
keyword	182
Underlining text	111
Units	
graphical	114
sound	146
storage	5
timeout	62
Unset	
expression entries for parameters	19
for LOCALs	21
variables	59
PRINTing them	59
Upper case	
checking for	42
codes of	42
converting to	42
used in definitions	4
in examples	5
equivalent to lower case	40
used in names	15
User defined	
functions	12
procedures	8

V

Values	as terms in expressions	34
	internal representation	37
	legal constants	34
Variable types	from letters in name	16
	from parameter swapping	19
	position in nametable entry	15
Variable values (vv) area	description	16
	used with names	15
	position in storage area	199
	storage of array descriptor	24
	array values	23
	loop information, FOR	55
	REPEAT	52
	values	17
Variables	assignment to	15
	dimensioned	22
	loop index, FOR	55
	REPEAT	51
	SELECT variable	46
	simple	15
	as terms in expressions	33
VER\$	unset	16
	definition	42
	keyword	182
Vertical stripes	as colour pattern	99

W

WHEN	keyword	182
	token	196
WIDTH	definition	75
	keyword	183
Width	of blocks	105
	of borders	104
	of characters	108
	of printout	75
	of windows	96
WINDOW	definition	96
	keyword	183
Windows	areas of	103
	for PAN	109
	for SCROLL	109
	borders round	104
	changing character size in	108
	default	95
	flashing text in	110
	ink colour in	101
	introduction to	94
	moving contents of	109
	opening as channels, console	76
	screen	77
	overlaying	97
	paper colour of	101
	redefining	96
	to default	112
	strip colour in	101
	tabulating in, See Tabulating	
	underlining text in	111
Words	word boundary	5
	long words	5
	as a storage unit	5
Writing	to devices	75
	to file	68
	to windows, See Window	

X

XOR

bitwise, ^, action	40
operator	33
token	197
logical, action	39
keyword	183
operator	33
token	197

XORing

of ink with background	102
of block colour	106



.

The Definitive Handbook . . .

Yes, this book was written by the designer and writer of Sinclair's QL SuperBASIC language.

Far more than a detailed description of SuperBASIC commands, this book explains the structure of the language and the reasons for its design. Used properly, SuperBASIC is a uniquely fast, flexible and elegant language, owing something to BASIC, something to Pascal, and much to Jan Jones's innovative design.

Although this book is not intended for the novice programmer, almost anyone who has read the QL User Guide and has begun to write SuperBASIC programs will find it invaluable. There is as little jargon as possible, and as much useful information as possible! For the experienced programmer, there is the information needed to ensure the elegance, speed and efficiency that sets SuperBASIC head and shoulders above older languages.

Every aspect of SuperBASIC is covered, with lots of sample programs.

Start using the real power of the QL!

Published by

QUANTA — The Independent QL User Group
15 Grosvenor Crescent GRIMSBY South Humberside England