SINCLAIR QL WORLD

THE NEW USER GUIDE

To an outsider, the QL has a surprisingly large and loyal set of admirers. Those associated with the computer more closely are less surprised by its continuing appeal. Even with the prices of comparable computers tumbling, the QL still represents remarkable value for money, and even though the competence of personal computers has improved immeasurably since the QL's launch in 1984, it can still claim to be superior to industry standard machines in several important respects.

The QL's construction and the ethos behind the decisions to equip it with a cheap keyboard and low-capacity microdrives are now dated, but its operating system is still superior to those supporting many pcs. Its programming language is a delight compared with the vast majority of Basic dialects available today.

Many QL users take to the computer like ducks to the village stream, but a great many other enthusiastic QL supporters have not found it so easy to come to grips with their computer's facilities. They make full use of the Psion programs, they purchase commercial programs, but there is the constant feeling that this is a second-hand way of using a computer. Their goal is to communicate directly with the QL, to have it obey directly their own commands.

The major hurdle for most unfulfilled computer users is often the 'user guides', which build up expectations and then fail to deliver. The *QL User Guide* is by no means the worst in this respect, especially in its most recent reprint, but nonetheless it is disappointingly incomplete in parts and, on occasion, downright inaccurate.

The QL User Guide was written by Roy Atherton in a concise and academic style which befitted the QL's image as a serious home computer. His task was presumably made more difficult by having to write against the clock and revise his work to accommodate the frequent revisions of language and operating system specifications which occurred before the QL's premature launch date. Certainly, his book on programming the QL is much better organised and easier to follow, although much the same in style as the User Guide.

Here at *Sinclair QL World* we have been aware for a long time, and are continually reminded by letters from readers, that a new User Guide would be extremely welcome. The aims of the new guide would not only be to correct the misleading parts of the original User Guide but also to present the subject from another perspective. If you failed to grasp Roy Atherton's insights you may be more enlightened by a different angle.

The prospects of any new book on the QL being a commercial success appear to be slight: there is no longer a large enough market to sustain a print run of, say, 25,000 copies which would be necessary if the book was to be available at a reasonable price. A low-cost manual printed on A4 sheets and distributed by mail order looked a more promising prospect, but print and advertising costs made the venture a risk. Consequently, it has been decided to include the New User Guide as part of Sinclair QL World, to reach the largest number of QL readers at the most economical cost.

The New User Guide starts with a Beginner's Guide covering very much the same ground as that of the original User Guide. This will allow readers to cross-refer between the two to get the best of both approaches to the subject matter. Additionally, it allows us to point out in detail any corrections which need to be made to the original guide.

Computing is becoming a crucial skill in modern society, but it is one which does not come naturally to the majority of people.

The situation is analogous to driving skills – only the most talented become racing drivers, but almost every adult has the capacity to drive a car competently. Computing should be the same; do not conclude that just because the extraordinary achievements of programmers like Simon Goodwin, Johnathon Oakley, Tony Tebby, Steve Sutton and Chas Dillon seem impossible to emulate that you cannot possibly be a good programmer yourself. After all, the abilities of Nigel Mansell and Louise Aitken-Walker have not dissuaded anyone from taking up car driving.

SECTION ONE

This section of the *New User Guide* covers the subject-matter of the Introduction and Chapters 1 and 2 of the *QL User Guide*.

QL users need little introduction to the physical aspects of their computer systems. The essential parts of the QL system are a keyboard through which information is passed to the computer, a display screen through which information is passed back to the computer user, a box of electronics and silicon chips which process information and storage devices such as microdrive cartridges which can retain information.

In order to use your computer you must know how to connect the computer to a monitor or tv set, how to type at the keyboard, how to insert microdrives and so on. Unless you are completely new to the QL these aspects of computing will already be familiar to you and so need not be covered here in any detail. Newcomers to the QL will find that the introductory section of the User Guide supplied with the computer is straightforward and instructive on these aspects.

This beginner's guide to *SuperBasic* begins with the assumption that you have a working QL in front of you, freshly reset and with no microdrive cartridges installed. The screen will have displayed its random dots of colour which are a side-effect of the computer's self-test and the copyright screen will have appeared.

At this point the QL awaits your instruction. Computers work at a constant speed, no matter what they are required to do, so there is no concept of a computer working hard or having a rest. Either it is working or it is not. At the moment, your QL is ticking over at 8.5 million instructions a second and its main concerns are to refresh the picture 50 times a second and to watch out for any keyboard activity. No matter what else is required of the computer, it must always take time out to perform these housekeeping duties. The instructions which require this to happen form part of the QL's operating system, which is called QDOS, the acronym for 'QL Disk Operating System'. Qdos is also responsible, among other things, for shaping and locating the characters printed on the screen.

Screen windows

The QL is expecting you to press the F1 or F2 key to indicate whether you want to use a high-resolution screen layout suitable for monitors, or a low-resolution screen layout more suited to tvs. Make your choice, and the screen changes to show the chosen layout of screen *windows* into which information can be placed. At first, the only area of the screen which accepts text is at the bottom of the display. This is the *command window,* where any commands you type are printed.

The QL's keyboard is immediately familiar to anyone with typing experience, although it has some additional keys which are found only on computers, and some keys which do more than do their typewriter equivalents. The *return key,* for instance, works exactly like the return key of an electric typewriter in that it forces the character next entered to be displayed at the beginning of the next line. On a computer it also has the role of indicating that a command has been typed in full and should now be actioned. This is known as 'entering' a command, so the return key of many computers, including the QL, is marked ENTER.

Shift keys

Keys unfamiliar to typists include the cursor keys, marked with arrows, which move the flashing cursor around the screen, and a number of additional shift keys which change the effects achieved by pressing ordinary keys. Typewriters usually have only one shift key, which changes the case of characters and gives access to the symbols on the number keys. The QL has such a SHIFT key, but it also has a 'control' key and an 'alternative' key, respectively marked CTRL and ALT. To use them, hold down any shift key (SHIFT, CTRL or ALT) and then press one of the other keys. The shift keys can be used in combination with each other, for instance CTRL-SHIFT-5, but such dexterity is not essential in order to program the computer.

The keyboard also has an 'escape' key, marked ESC, which can be pressed to stop something undesirable happening. The QL's ESC key is said to be 'soft' because, unless a software program includes specific instructions on how to proceed when the ESC key is pressed, it is ignored. Some other computers have 'hard' ESC keys which stop programmed activity whenever they are pressed.

Other 'soft' keys on the QL include the Function keys, labelled from F1 to F5 and situated on the left-hand-side of the keyboard. Without a program to give them some purpose the function keys are invariably ignored.

Delete

The QL has a 'delete' key combination which removes characters from the screen. CTRL-LEFT (hold down the CTRL key and press the left cursor key) deletes the character immediately to the left of the flashing cursor. CTRL-RIGHT deletes the character on which the cursor is flashing. The cursor's only purpose is to indicate where text can be entered or deleted. There is a way of 'turning off' the cursor so that it cannot be seen, but a few seconds of trying to cope without it is usually enough to demonstrate its value as a permanent part of any screen display.

The last key combination to be covered here is 'break', formed by holding down the CTRL key and pressing the SPACE bar. CTRL-SPACE has the effect of aborting some activity, be it the entering of a command or the execution of a program.

This guide to SuperBasic is devoted to the artificial and ephemeral world created within the computer rather than to the physical reality of the computer system's components. The boundaries of this world are imposed by the physical constraints imposed by the computer's design and construction, the logical constraints imposed by the QDOS operating system and the SuperBasic programming language, and the limits of the user's imagination. Sadly, the extent of the latter is too often exhausted long before the other, absolute, limits are encountered.

The first thing to know about the computer's internal world is that it is reactive: it does nothing until instructed to do so by the programmer. You, the programmer, decide when things should happen, in what order actions should take place, how many times activities are to be repeated and when things stop again. The computer translates your commands into instructions it can understand, and then carries them out, often using the QDOS housekeeping routines mentioned earlier.

The second thing to know about the computer's world is that everything within it is irritatingly precise. Computers do not tolerate spelling mistakes, or forgetful punctuation, or imperfect logic. By implication, precision is maintained over time. This means that if the computer produces a particular result once, it will produce exactly the same result if the exact circumstances re-occur. It is this exactness which makes computer programming useful and appealing: computing is one of the few pursuits which has a sense of absolute perfection.

The third thing to know is derived from the previous two: if the computer fouls up, it is always your fault.

Statements

At the heart of instructing the computer is the idea of the 'statement'. Just as sentences are the basic units of English expression, so statements are single SuperBasic entities. SuperBasic is much simpler than English, and so the variety associated with English sentences is not present. Most SuperBasic statements are commands and all SuperBasic statements follow rigid rules. This actually makes programming simpler because in English we can debate whether to say 'This is different from that' or 'This differs from that', or even 'This and that are different' while in SuperBasic we can only say 'This <> That'.

Turn on your QL and try a few experiments to deduce what we can about SuperBasic. Let us try to make the computer say 'Hello'. Type in:

```
SAY HELLO
```

and press Enter. The message 'bad name' appears. The QL is communicating with us, but only to express its lack of understanding. Error message like this are not to be worried about. They are there for our benefit, to highlight a mistake on our part which the computer cannot deal with. If one appears on your screen, respond to it positively by correcting the fault command the computer has objected to.

Print to screen

A quick check of the manual reveals that the word SAY means nothing to the computer, but that the alternative PRINT will be understood, so try typing:

```
PRINT HELLO
```

When the Enter key is pressed the computer does not produce an error message, but the world HELLO has no meaning to it and so it has printed an asterisk. From now on it is assumed that you will automatically press Enter at the end of every command.

A further check of the manual indicates that text to be printed should be enclosed in quotes, like this:

```
PRINT "HELLO"
```

This time the computer carries out the command perfectly. Now that one set of characters has been printed it is possible to replace HELLO with any other set of characters contained within quotes and it is certain that the computer will print them as requested. The only possible exception is if the characters include additional quotation marks, because the computer cannot then determine exactly where the string of characters is supposed to end.

Words with special meaning to the computer are called *keywords,* so PRINT is a keyword and SAY is not. SuperBasic has many keywords, although it is not necessary to know about all of them in order to write programs. Every SuperBasic statement begins with a keyword.

Numbers

Turning to numbers rather than text, typing:

```
PRINT "72"
```

has a predictable outcome. However, typing:

```
PRINT 72
```

has *exactly the same* effect. This is interesting, because it means that numbers are treated differently from text. Try testing the computer's mathematical skills with:

```
PRINT 72+16-5
```

The result, 83, is printed on the screen. The response to the similar command:

```
PRINT "72+16-5"
```

is quite different, though. Find out for yourself what it is.

Values and variables

During my Basic tutorial classes, the next programming experiment takes place away from the computer and involves pieces of paper and shoe boxes. Imagine a set of empty shoe boxes being used to store numbers of text, each box holding one number or one piece of text. To store a number, it is written on a piece of paper and stored in a box. If the number is to be used in an equation, it is examined in the box, the equation is carried out and he result can be stored in another box, or it can replace the first number in the first box. In order to indicate which box holds which number it is convenient to name them. The names of the boxes holding text have a special feature to distinguish them from the boxes holding numbers.

This primitive set-up is a close analogy of the way the QL stores numbers and text in its memory. Parts of the QL's memory are reserved to hold numerous numbers and pieces of text, with each element of that memory being given a name by the programmer. The proper term for one of these elements is 'variable', because what a variable contains can be varied by the programmer.

To place a value in a variable, a new SuperBasic command is necessary:

```
LET salary = 1024
```

Salary, of course, means nothing specific to the computer; it is just a name for a variable. Similarly, the computer does not know or care if 1024 refers to pounds, dollars or carrots. 1024 is simply a value held in the variable. When you type in this command nothing particular happens on the screen, but somewhere in the QL's memory a shoebox called 'salary' has been created to store the value '1024'.

Now the experiments with LET and PRINT can begin in earnest, but before doing so, a few more maths symbols will add some variety. Addition and subtraction are represented by + and -, as you would expect, but multiplication and division are represented by * and / in all computer languages. So here is how to award yourself a pay rise, calculate your tax and print your net income:

```
PRINT salary

LET salary = 1024 + 350

PRINT salary

LET tax = salary * 0.24

PRINT tax

PRINT salary - tax
```

It is important that the chosen variable names do not begin with a number, otherwise the QL would confuse the variable name with a value an print and error message. LET 5X = 25 might be interesting algebra but is not valid SuperBasic. LET X5 = 25 is, however, acceptable to the computer because that initial letter in the variable name removes all ambiguity.

If the screen is now getting cluttered, a useful command is:

```
CLS
```

Clear screen

CLS stands for CLear Screen, but on the QL it actually clears just one window, the one in which the results of your commands have been appearing. Notice that CLS forms a statement by itself, with no need for any number or text to follow it. PRINT can also form a statement by itself – it prints a blank line. LET, however, makes no sense to the computer or to the programmer unless it is followed by some sort of assignment. The QL objects if it finds keywords in unexpected places, such as in:

```
PRINT CLS
```

Keywords are therefore 'reserved' so that they cannot be used as variable names.

Variables, you will recall, can hold text as well as numbers provided that the variable name identifies that it belongs to a text-holding memory box. In SuperBasic, names of text variables must end with a dollar sign, such as NAME$ or CITY$. Examine this statement:

```
LET Text$ = "Made in Britain"
```

The LET keyword indicates that a value is going to be placed in a variable. 'Text$' is a valid name for a text-holding variable. The equals sign separates the variable name from its value, which is a string of characters contained in quotes. The whole command makes perfect sense to both programmer and computer. To prove that the text has been stored, type:

```
PRINT Text$
```

Now we can return to our first attempts to type in valid commands and understand more clearly what was going wrong. SAY HELLO was clearly nonsense because SAY is not a SuperBasic keyword. PRINT HELLO was a correct SuperBasic command, but the response was an asterisk, not the word "HELLO". It is now clear that the computer understood HELLO to be a variable name, but there was no variable so-named in its memory, so it printed an asterisk instead of a value.

This month most of the rules about variable names have been covered, together with some useful keywords and some thoughts about valid and invalid SuperBasic statements. Below are a handful of statements, some of which are valid and some not. If you cannot work out which are which, your QL will help you.

Test yourself

```
LET BEER = 5X
LET NAME$ - "AMANDA"
PRINT AMANDA
PRINT 5+23 * 7/12
LET CLS = 104
PRINT NAME$
```

So far, of course, the computer has only carried out one command at a time. These are known as 'direct commands'. Next month, the SuperBasic tutorial series forms programs from lists of separate commands and introduces more keywords to the SuperBasic vocabulary.

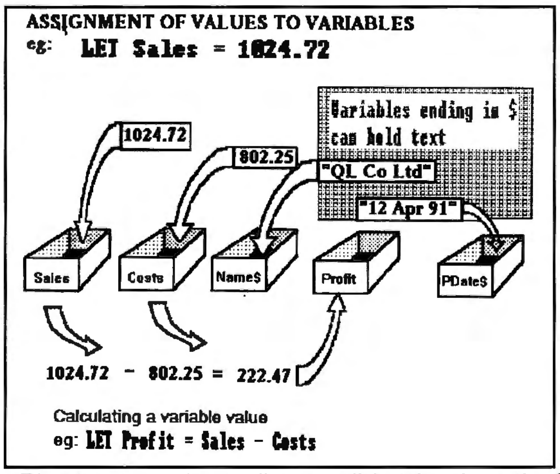
SINCLAIR QL WORLD

THE NEW USER GUIDE

SECTION TWO

The second instalment of our New User Guide for the Sinclair QL. This month, Mike Lloyd parallels the contents of Chapter 2 of the original User Guide, but expands the scope to include what he describes as 'absolutely everything you need to know about computer programming'.

Readers of last month's introductory part of the New User Guide to the QL will now know how direct commands are issued to the QL – and what happens when the command is not recognised for any reason. They will also know about variables – the named memory locations which are used to store single pieces of numeric and textual information in the computer's memory. **Figure one** is a reminder of some of the features of variables, based on the analogy used last month of shoe boxes holding pieces of information.


ASSIGNMENT OF VALUES TO VARIABLES
eg: LET Sales = 1024.72
Variables ending in $ can hold text
1024.72
802.25
"QL Co Ltd"
"12 Apr 91"
Sales
Costs
Name$
Profit
PDate$
1024.72 - 802.25 = 222.47
Calculating a variable value
eg: LET Profit = Sales - Costs

What is a program?

Direct commands are all very well, but they force the QL to act as little more than a clever calculator. The value of a computer arises from its ability to store sequences of commands and then to carry them out automatically. A sequence of commands is called a program. Incredibly, there are only three fundamental things to learn about programs: all the rest is window-dressing.

Direct commands and commands in a program are very closely related. A direct command becomes a part of a program if it is given a *line number.* Line numbers can be any whole, positive number less than 32676. The upper limit is not restricting in any way as even the largest programs rarely exceed 3,000 lines. Commands in programs are often known as *statements.*

Type in the following short program of two lines:

```
100 LET SALES = 240
110 PRINT SALES * 0.15
```

Instead of carrying out the commands the instant that the Enter key is pressed, the QL stores them in its working memory. For the programmer's benefit they are also listed on the screen, using Window #2. The program is executed only when a direct command is given telling the computer to proceed. This command is RUN, so type it in now, remembering not to give this direct command a line number.

Three things may happen. The most likely is that the program instantaneously prints the value of 16% VAT on sales of £240. Alternatively, there may be a typing error in your program which produces an error message. Thirdly, if you have accidentally prefaced the RUN command with a line number it will be tagged onto the end of the program.

Having committed our program to the QL's memory several things can be done with it – and indeed should be done with it – in order to make the most of the computer's programmability. It would be nice, for instance, to be able to edit program lines, not only to correct errors but also to amend the way the program works. The work required to produce a 200-line program makes it uneconomic to type it in each time it is needed. Therefore, programs must be capable of being saved permanently and recalled when needed.

Editing

The command to edit a program line is EDIT. The keyword must be followed by the line number of the line you wish to edit. For example, to edit the second line of the program listed above, type:

```
EDIT 110
```

On pressing the Enter key the line will be copied to the command window where it can be edited. When Enter is pressed again the amended line is replaced in the program listing.

It is of course possible to change the line number while editing the line. You may wish to save typing, for example, by taking an existing line, editing it slightly, and re-inserting it into the program with a new line number. Line numbers in a program must be unique: if you enter a second line numbered 110 it will overwrite the existing program line with that number.

Line numbers

Line numbers are used to sort lines into sequential order, so giving a program line a new number can also have the effect of changing its position in the program. Line numbers do not need to be sequential. Traditionally, programmers begin programs with line numbers incremented by 10. Then, when they are revising the program or adding to it, new program lines using the intervening numbers can be added. The program which will be developed during this article will make use of this approach.

The important thing to grasp is that programs are never written like letters – starting at the beginning and working through to the end. Programs tend to develop like a painting, starting with perhaps the bare outlines and then adding more and more detail until the work is complete.

Save and Load

When a great deal of care and time has been expended on a program, it is important to be able to save and recall it. This is done using the commands SAVE and LOAD. To save a program, type in the direct command SAVE followed by a valid filename. Filenames comprise a drive designation and an identifying name, and the typical 'mdv1_program' or 'flp1_test' constructions will be familiar to all QL users. Having saved a program, it can be reloaded into the Q L's working memory by typing in the direct command LOAD followed by the program's filename.

You are saving a snapshot of your program as it was at the time it was saved. If it is amended after being saved, the amendments will not be reflected in the program file. Always save a program under construction at regular intervals – every 15 minutes is advisable – during development and immediately before switching off the computer. You can estimate the desired frequency of saves by the amount of work you would lose, in time and effort – if you were to crash or switch off without saving.

What is a programmer

It is time now to change your perception of who you are. When you enter a direct command to the computer or type in information you are a user, but when you are constructing a program you are a programmer. A programmer's perceptions of a program is very different from a user's. To a programmer, a program is a series of text-based commands which form a logical structure. To a user, a program is a screen display, a menu of options, a series of requests for information, and a set of facilities. The user sees only the facade, while the programmer sees the backstage props, ropes and beams which keep the facade in place.

Fundamentally, programmers provide solutions to users' problems. Assuming that a user requires assistance with calculating VAT on sales and keeping a tally of how much VAT is owed, let us write a program which meets this need.

The two program lines listed earlier in the article can be put to good use in the solution to the problem of calculating VAT, so let us begin with them:

```
100 LET SALES = 240
110 PRINT SALES * 0.15
```

INPUT

A fundamental drawback is that the program is only of any use when articles valued at £240 (or perhaps £2.40) are sold. The answer is to replace the LET statement with one which obtains input from the user. Having already discovered that the keywords for editing, saving and loading are EDIT, SAVE and LOAD, it is easy to guess that the keyword to obtain input is INPUT.

To form a valid command, the keyword INPUT must be followed by a variable name so that the computer knows exactly where to store whatever is being typed in. The variable can be for numeric information or for text. If the variable is for numbers only, typing text will cause an error message to appear. For the moment, assume that users can be trusted to type what the QL is expecting to read (although programmers will quickly learn that trusting users to do anything predictable is foolhardy). Use the EDIT command to alter Line 100 so that the program reads:

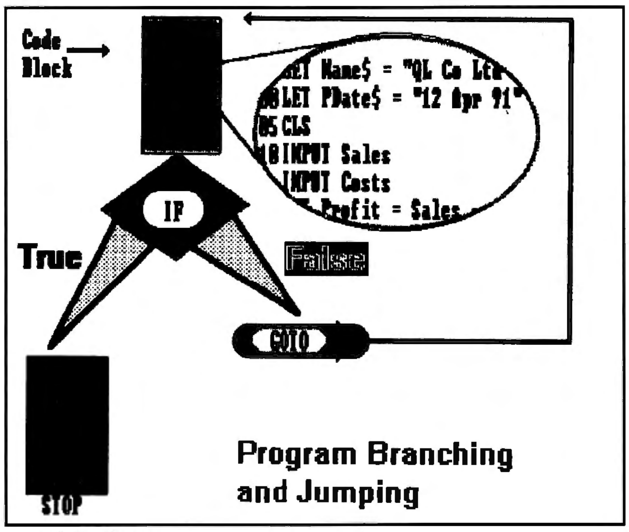
```
100 INPUT SALES
110 PRINT SALES * 0.15
```

Now when the program is run (by typing the direct command RUN) a flashing cursor appears in Window #1. Type an amount and press Enter and the computer should write beneath it the

appropriate VAT value. If not, examine your program to work out why it has failed.
The program is of limited use because it only copes with a single calculation. The user will be selling things all day and will not want to keep typing in RUN to make the program work. After each calculation, the user will wish the computer to be ready for the next.

Repeating commands

The simplest way of achieving this aim is to type in more INPUT and PRINT commands, but this is wasteful and impractical. The best answer is to find some way of directing the computer to carry out the original pair of commands repeatedly. This can be done by telling the computer to go to the first line in the program. Once again the Basic language has a very obvious keyword and statement syntax which provides this facility. Add this command to the end of your program:



GOTO

```
120 GOTO 100
```

Run the program again and as soon as the first calculation is complete the cursor in Window #1 is flashing again for another input. As many inputs as you like can be entered. Having made the program loop eternally, making it stop has new become an important issue. Simply hold down the CTRL key and press the SPACE bar – on the QL the CTRL-SPACE combination equates to the BREAK key on many other computers. Its effect is to bring to a halt any SuperBasic program.
Looking again at the program from the user's point of view (a habit which comes hard to many programmers) the screen is devoid of information to explain what is going on. If the program cannot be seen, type in the direct command LIST to list it on the screen. Now we are going to insert program lines in between existing lines by carefully choosing line numbers. Add the following lines:

```
90 PRINT "Type in a sales value and press Enter"
105 PRINT "The VAT on the above amount is. . ."
```

So that the new Line 90 is included in the program loop, line 120 should be edited so that it reads:

```
120 GOTO 90
```

The revised program demonstrates how important it is to keep the user informed about exactly what the screen display means. The program obtains sales values and outputs the VAT, and the screen display makes this quite clear to the user, but the program has yet to meet the original requirements in full. It does not keep a tally of the total of VAT owed to Customs and Excise.

Adding variables

Some logical thought is needed here to work out how the computer is going to cope with this task. At the beginning of the day, the amount owed in VAT is nothing, because there have been no sales. With each sale, the total amount of VAT can be described as the previous VAT total plus the VAT associated with the most recent sale. These two values can be represented by the variables TOTALVAT and VAT respectively. At the beginning of the day, TOTALVAT is zero, so add this to the program:

```
80 LET TOTALVAT = 0
```

With each sale, VAT must now be calculated and stored, so add the line:

```
108 LET VAT = SALES * 0.15
```

It is worthwhile simplifying the next line in the program to read:

```
110 PRINT VAT
```

This makes the program easier to understand and it saves having to do a calculation twice.
The value of VAT must now be added to TOTALVAT, hence the addition of the line:

```
115 LET TOTALVAT = TOTALVAT = VAT
```

Many novice programmers have a little difficulty understanding how this sort of command can make sense. What it means is: fetch the values held in the variable locations TOTALVAT and VAT, add them together, and store the result in the variable location called TOTALVAT. As only one value can be stored in a variable, the new TOTALVAT value overwrites the old one.

Run the program again to ensure that it is working. When you have entered a few sales values, press the CTRL-SPACE combination to complete the program and find out what TOTALVAT is worth by typing in the direct command:

```
PRINT TOTALVAT
```

Users could do this for themselves, but it is a little messy and means that they need to know a little about programming. It would be preferable to allow them to signal the computer that no more sales are going to be entered, at which point the computer can print the total VAT for the day and stop processing.

Firstly, how can the users signal that no more sales will be entered? A simple solution is to ask them to enter a zero value instead of a sales figure. The second part of the problem is to get the computer to make a decision based on the input value, for which another keyword and another command are required.

IF and THEN

The keyword is IF and the basic structure of the command in which it appears is 'IF something is true THEN do something'. We want to tell the program to skip the remaining lines of the program if a sales value of 0 is encountered; in other words, if SALES equals zero then go to Line 30. Type in:

```
102 IF SALES = 0 THEN GOTO 130
```

At Line 130 we can now include commands to print the total VAT owed for the day's sales:

```
130 PRINT "The total VAT for the day is:"
140 PRINT TOTALVAT
```

Run the program, enter a few sales values and then enter a sales value of zero. If the program does not behave exactly as expected, review the lines you have entered and the lines given in the text above to see where it is going wrong. To help you, **Figure two** is a listing of the complete program using the rather odd line numbers which have been used as the program has developed. For the fastidious, properly spaced program line numbers can be restored by typing in the direct command RENUM, short for 'renumber'. The computer is clever enough to update the GOTO statements as the line numbers change. Save the program with a direct command such as:

```
SAVE mdv1_vatcalc
```

LISTING 1

```
 80 LET TOTALVAT = 0
 90 PRINT "Type in a sales value and press Enter" <
100 INPUT SALES
102 IF SALES = 0 THEN GOTO 130      ==>
105 PRINT "The VAT on the above amount is..."
108 LET VAT = SALES * 0.15
110 PRINT VAT
115 LET TOTALVAT = TOTALVAT + VAT
120 GOTO 90                         ==>
130 PRINT "The total VAT for today is:"  <
140 PRINT TOTALVAT
```

The arrowed lines indicate the way the computer is forced to move through the program lines - they are not part of the program itself.

Branching

When computers are told to go to a specific line number rather than to continue with the next line in the program, the process is called branching. Branching can be forwards or backwards, as the two GOTO commands used in the program demonstrate. When branching occurs only if a condition is found to be true, as in the IF command, then it is called conditional branching. When the branching occurs no matter what the circumstances, as is the case in Line 120, then it is described as unconditional branching.

There is nothing more to program logic than this. Computers perform a series of statements in a pre-determined sequence until they are required to branch, either conditionally or unconditionally. Of course there are many more commands to learn in order to do different things, and there are other branching commands than IF and GOTO, but these extra commands simply add to the richness of the SuperBasic language: they do not provide any alternative method of executing programs.

It is an absolute truth that all SuperBasic programs, and indeed all conventional computer programs, can be re-written so that they consist only of straightforward commands, such as PRINT and LET, and the two branching statements IF and GOTO. Of course, programs are faster, more efficient and more elegant if use is made of the other commands and structures available in the language, so the New User's Guide to SuperBasic therefore continues next month.

THE NEW USER GUIDE

THE SCREEN

SECTION THREE

In the third of our New User Guide series, Mike Lloyd covers the subject matter of Chapter 3 of the *QL User Guide.*

The screen is an essential part of any computer system because it is where the computer communicates with the user. A great part of the operating system of the QL is devoted to managing the screen display. Similarly, a large percentage of Superbasic commands are used to produce images on the screen. This section of the *New User Guide* introduces everything you need to know in order to understand the way the QL handles the screen.

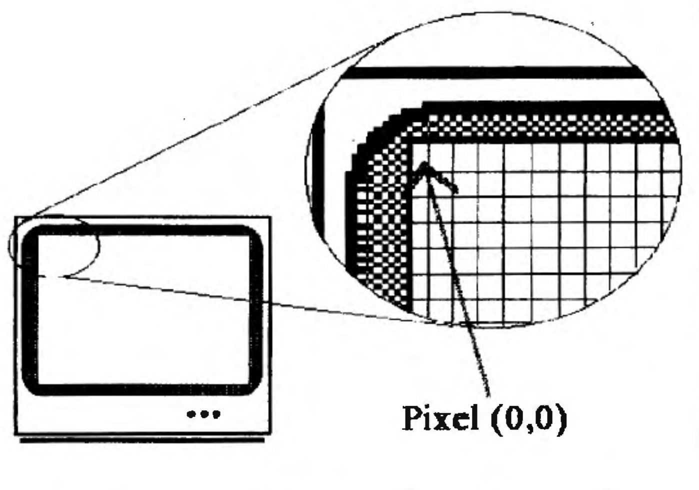
Pixels

Television pictures are made up of lines of dots, each dot called a 'picture element', or pixel. Modern television broadcast pictures produce high-quality pictures with 625 lines. Computers tend to have lower numbers of lines, typically around the 200-300 mark. The QL has 256 lines. Depending on the display mode each line has either 256 or 512 pixels.

The QL's designers allocated 32 kilobytes (32Kb of the computer's 128Kb memory to managing the screen. In high-resolution mode, or monitor mode, this memory limit means that each pixel can be one of four colours. The low-resolution, tv, mode has half as many pixels, allowing more information to be held for each pixel. Not only does the tv mode have eight colours, but the pixels can be individually set to flash.

The disadvantage of using tv mode is that vertical lines are twice as thick as they would appear in monitor mode. For television users the extra width of the characters can make text easier to read, but monitor users generally prefer to use the high-resolution screen mode in order to place more text and graphics on the screen.

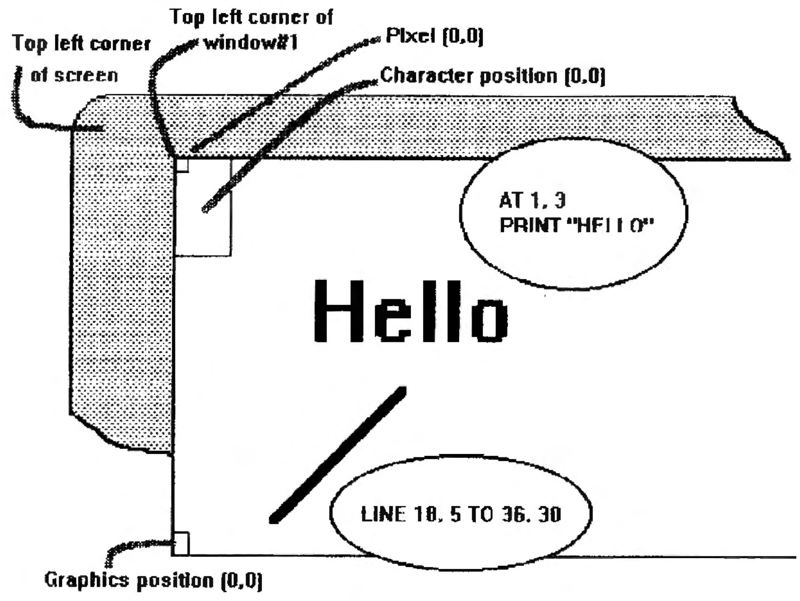
Figure 1: The screen-wide pixel co-ordinate system.



The grid

The computer screen is best imagined as a piece of electronic graph paper where each tiny screen can be painted with a given colour. The location of each square in the grid is described by counting how many squares across and down from the top left square it lies. Moving clockwise around the screen, the four corners are at pixel locations (0,0), (511,0), (511,255) and (0,225). The horizontal co-ordinate is given first.

Figure 2: the window co-ordinate systems



When a programmer issues a command such as PRINT "HELLO" the QL calculates where on the screen the letters must go. It then looks up in its operating system the details of what pattern of pixels forms each of the letters. The letter patterns are transferred to the correct place in the 32Kb memory area reserved for the 'screen map'. Fifty times a second this memory map is accessed by the QL in order to send the current screen picture to the tv set or monitor.

It is quite feasible, if a little unfriendly, for computer programmers to make changes to the contents of the screen map directly. It is much easier to use SuperBasic commands to specify where lines, curves and characters are to be drawn. With SuperBasic, the programmer tells the computer what is wanted and the QL then gets on with the task of fulfilling that request without the user needing to know how many pixels of what colours will be printed where.

It is very useful to be able to tell the QL exactly where on the screen text should appear – for example, to centre a heading or program name. Similarly, if we are going to use the QL to draw graphics there must be a way of specifying where lines start and finish. It would also be nice to be able to move windows around, and change their sizes, to meet the needs of particular programs.

In order to do any of these things, the computer's method of locating points on the screen must be used. The QL is more advanced than most other computers because the screen can be broken down into separate areas, called windows, each acting as if it was a screen by itself.

Windows

When the QL is turned on it has three screens windows opened automatically. These are called the default windows and are numbered 0, 1 and 2. Each has a specific purpose: Window 0 is the command window where your typing normally appears and where error messages are displayed: Window 1 is the default window where printed text output and graphics appear if no other window is specified, and Window 2 is the listing window for SuperBasic programs. Other windows can be opened using different numbers to identify them, but this will be covered later in the guide.

The size, position and colour of windows can be changed using simple SuperBasic commands. Window size and location are specified in pixels, and a typical command to specify a window's attributes is:

```
WINDOW 100, 100, 206, 78
```

WINDOW is a SuperBasic keyword. It is followed by four parameters separated by commas. The first pair describe the dimensions of the window and the next pair describe the location of the top-left corner of the window in relation to the top-left corner of the screen. The command can therefore be translated into English to read "Move the main window so that it is 100 pixels square with its top left pixel located 206 pixels across and 78 pixels down from the pixel in the top left corner of the screen".

You are already aware that the QL can increase the number of colours it can display by halving the number of pixels it draws. The QL's operating system is clever enough to ensure that a 100x100 pixel window stays the same size no matter what mode the screen is in. Always think of the screen as being a grid of 512x256 pixels, no matter what mode the computer is in.

A WINDOW command makes no immediate difference to what you can see on the screen. The computer has merely changed its understanding of where the window lies ready for the next window-related command. To see where the window has moved to, change its colour so that it is different from the background and clear it with the CLS (clear screen) command introduced last month.

Ink and Paper

SuperBasic has two very easy commands to change screen colours. One changes the background colour and the other changes the foreground colour. Text and graphics are drawn in the foreground colour on pixels the colour of the background. The keyword must be followed by a parameter to indicate the required colour. The computer does not understand words such as blue, red and green, but it recognises colours by a number system, as follows:

TV MODE

Colour	Number
Black	0
Blue	1
Red	2
Magenta	3
Green	4
Cyan	5
Yellow	6
White	7

MONITOR MODE

Colour	Number
Black	0 or 1
Red	2 or 3
Green	4 or 5
White	6 or 7

The PAPER command is identical to the INK command except that it controls the background colour of the window. The choice of colours and the numbers by which Superbasic identifies them are the same as for INK.

Colours can be described by any number between 0 and 255, even though the colour chart above suggests that the maximum colour value is 7. Numbers higher than 7 produce speckled effects which will be described in detail later in the Guide. Feel free to experiment, but be aware that the results can make text very difficult to read.

Here are a set of three commands which change the window location and colour it green:

```
WINDOW 80, 80, 100, 48
PAPER 4
CLS
```

Should you enter the above commands on your computer, you will notice that the window is not square, even though there are the height and width of the window are equal in terms of numbers of pixels. This is because pixels are not square but rectangular, being taller than they are wide. True squares can be made by specifying more horizontal pixels in the ratio of around 1.6:1, such as:

```
WINDOW 162, 100, 0, 0
```

Experiment with the WINDOW command to move the default window around the screen. If the QL detects an impossible window location it will print an explanatory error message.

All of the WINDOW, PRINT, PAPER and CLS commands used up to now have affected the main window, leaving the listing window and the command window unchanged. In order to direct such commands at other windows an identifying number is essential.

Channels

Communication between the computer and each window is by a 'channel'. Channels can be connected to many things, including windows, printers and files, and each channel is numbered. With the main window the identifier can be included or omitted as desired. As explained earlier, the command window at the bottom of the screen is WINDOW#0 and the listing window is WINDOW #2. The hash before the number indicates to the QL that it is a channel identifier and not a parameter.

All commands related to windows can be followed by a channel number. Thus, the PAPER command can be followed by a hashed number, such as PAPER 2, 4. PRINT is another window-related command which can be treated in the same way. Note though that if these commands take a hashed number there must be a comma between the window number and the first parameter. To demonstrate this principle, let us relocate and recolour the listing window and print some text in it:

```
WINDOW #2, 200, 200, 50, 10
```

```
PAPER #2, 5
PRINT #2, "This is the listing window"
```

The PAPER command will turn the window light blue, or cyan, if the QL is in tv mode, or the window will be green in Monitor mode.

MODE4 and 8

When the QL is first switched on the user must choose between tv mode and monitor mode. However, there is no reason why that mode must be used throughout the computing session. There is a SuperBasic command called MODE which takes a single number as its parameter. MODE 4 switches the QL into its high-resolution, four-colour mode most suited to monitors and MODE 8 provides the low-resolution, eight-colour mode which TV users most frequently use.

You should be beginning to see by now that SuperBasic is logical and simple way of expressing things. Commands are made up of keywords and, optionally, parameters. Most of the parameters used so far have been numbers, but the PRINT command can be followed by a text parameter enclosed in inverted commas. Although it might be difficult for programmers to remember, it is convenient for the computer to recognise things such as colours and channels by numbers rather than by names.

At the beginning of this section of the User Guide we learnt about the screen-wide pixel co-ordinate system used to locate windows. Each window has its own co-ordinate system with its pixels described as offsets from the top left corner of the particular window they belong to. This is very rarely used by SuperBasic commands.

There is a more useful co-ordinate system linked to each window which determines where text is placed. Like the pixel co-orindates the origin – the point described as (0,0) – is at the top left of the window. However, each location on the grid is exactly large enough to hold a single character. This is a very convenient arrangement because the QL has six different text sizes: the character co-ordinate system adjusts itself automatically to suit the current character size chosen.

The AT command

Character co-ordinates are described by the AT command. AT is always followed by two parameters to represent the horizontal and vertical offset from the top left character in the window. Thus:

```
AT 5, 6
PRINT "HELLO"
```

will print the word HELLO beginning at a character position five lines down and six places in from the top left corner of the window. Because AT is a window-related command you would expect it to be able to take a channel number, and so it does. AT 2, 4, 2 will affect the print location of the next text printed in the listing window with the PRINT 2 command.

There is yet another co-ordinate system linked to windows to control the drawing of graphics such as lines, curves and circles. This is known as the graphics co-ordinate system. It is particularly clever because it changes its scale so that no matter what the size of the window its height always equals 100 graphic units. Another invaluable feature is that a circle drawn in monitor mode can be re-drawn exactly the same size in tv mode, even though there are a different number of pixels and each pixel is a different shape. The origin of the graphics co-ordinate system is at the *bottom* left of the window.

Line and Circle

To experiment with the graphics capabilities of the QL there are two commands which will be fully explained in the next section of the Guide.

Lines are drawn using the LINE command. Four parameters are needed: two to describe the location of the start of the line and two to describe where it will end. A new keyword, TO, separates them. Two examples are:

```
LINE 20, 40 TO 20, 80

LINE 10, 90 to 50, 30
```

Circles are drawn by the CIRCLE command. The CIRCLE keyword takes three parameters: two to describe the location of the centre of the circle and the third identifying its radius in graphics units. Remembering that no matter what shape and size a window is it is always 100 graphics units high, a circle touching the top and bottom of the window can be drawn by the command.

```
CIRCLE 50, 50, 50
```

SINCLAIR QL WORLD

THE NEW USER GUIDE

In the fourth instalment of our New User Guide for the Sinclair QL, Mike Lloyd runs alongside Chapter 4 of the Sinclair QL User Guide and looks at characters and strings, touching on binary digits and Ascii code as he goes.

SECTION FOUR

Text Handling

Devoting an entire article to a straightforward subject like text characters might sound slightly dull. After all, you type a letter at the keyboard and it appears on the screen. However, the way the QL treats characters reveals many of the secrets of the way computers operate. Furthermore, unlike some other languages *SuperBasic* has a rich range of commands devoted to character manipulation. Like other Sinclair computers the QL has by far the neatest text management functions in the world of computer languages. It will take much more than a single article to explore the subject fully, so this is just a start.

Although it might sound surprising for such a capable computer, the QL has no natural understanding of what an alphabet character or a piece of text might be. Absolutely everything in the computer's memory is formed from whole numbers between zero and 255 inclusive. Before looking at characters we must spend a little time seeing what the QL uses to represent them.

Computer memories are composed of millions of tiny transistors etched onto a silicon chip. When the computer is working, each transistor can be holding an electrical charge or not; in other words, transistors can be on or off. With just two possible states to be in, a transistor is not by itself capable of holding a great deal of information. However, if two transistors are grouped together the possible combinations of states is four:

T1 = OFF	T2 = OFF
T1 = OFF	T2 = ON
T1 = ON	T2 = OFF
T1 = ON	T2 =ON

If the 'ons' are used to represent 1 and the 'offs' represent 0 then the states shown above can be written as 00, 01, 10 and 11 – the first four numbers in the binary counting system.

A group of three transistors can represent eight different states and a group of five transistors can assume 32 states, and so on. By adding a transistor to a group the number of possible combinations of ons and offs doubles.

Each transistor holds a 'binary digit', a value of 0 or 1. In the early days of computers groups of five binary digits, or 'bits', were considered sufficient for many tasks because it was the smallest group of bits which could still represent every one of the 26 letters in the alphabet.

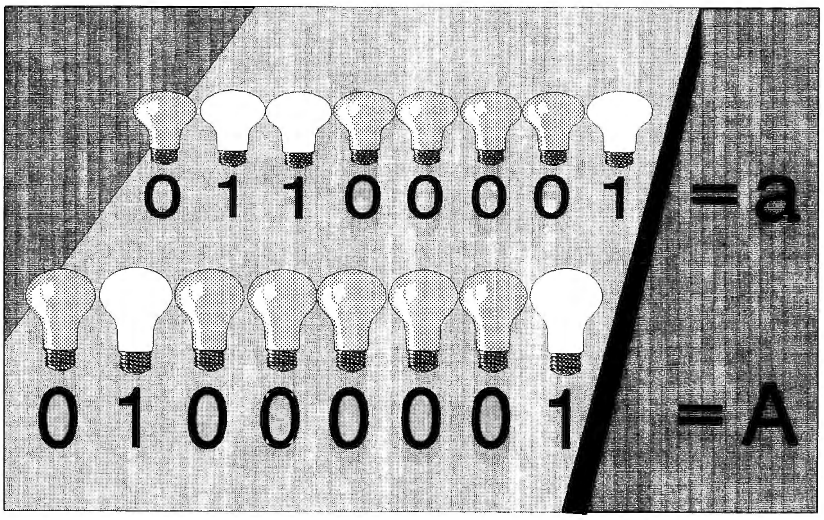
These days almost every computer groups its transistors into eights, thus allowing 256 combinations to be formed by each group. A group of eight bits is a 'byte', and the fact that each byte can have one of 256 states explains why the QL character set has 256 characters in it: it is because each character is represented by a single byte.

Ascii

Many years ago, the Americans introduced a standard arrangement for all computer character sets called the American Standard Code for Information Interchange, more familiarly known as Ascii. It means that a sentence of text stored in one computer can be transferred to another and still make sense because both computers use the same number sequence to represent characters.

It is too logical, of course, for the letter 'A' to be Ascii code 1. The first 32 codes are 'non-printing' codes such as the 'newline' character, the tab character, the backspace character and so on. The first 'printable' character in the Ascii code is the space. Next comes the group of symbols found, in order, above the number keys on an American typewriter. These almost match those found on the QL, but not quite.

The digits between zero and nine inclusive are the next group, beginning at Ascii code 48 and ending at code 57. Some more symbols fill the gap between the digits and the start of the alphabet to force 'A' to have the Ascii code of 65. Sixty-five is not particularly noteworthy as a decimal number, but in binary it is 01000001 (see **Figure one**).



The upper case alphabet follows in its usual order before giving way to yet more symbols. The lower case alphabet starts at Ascii code 97, which is more significant as a binary number, 01100001. The final number allocated a character in the Ascii code is 127. This leaves exactly half the possible codes unspecified so that individual computer and printer manufacturers can add their own characters.

The upper reaches of the character set on the QL contains all sorts of interesting symbols, including arrows, foreign letters, mathematical symbols and a whole set of values reserved to represent the QL function key combinations. If you have a printer with the IBM character set, the higher values contain a completely different but equally interesting set of characters, including box characters.

Because of the way the QL manages keyboard input, it is possible to obtain the non-printing Ascii values and still use the lower part of the Ascii table for other characters, although only the Minerva cpu takes advantage of this quirk. The exception is the newline character, Ascii Code 10 or CTRL-J, which is always a newline.

Strings and dollars

The title of this section of the *Guide* is *'Characters and Strings'*, so it is time to explain what a string is. Quite simply, a string is a group of one or more characters, although there is also a special string which contains no characters at all – the so-called 'null string'. Strings can be stored and manipulated in the same way that the QL treats number values, provided that some rules to distinguish between text and variable names are observed.

You will recall that you can assign a number value to a variable name with the LET command, such as:

```
LET hello = 12
```

If the command PRINT hello was typed in, the computer would print 12 because that is the value which hello represents. In order to print the word "hello" it must be typed in quotes:

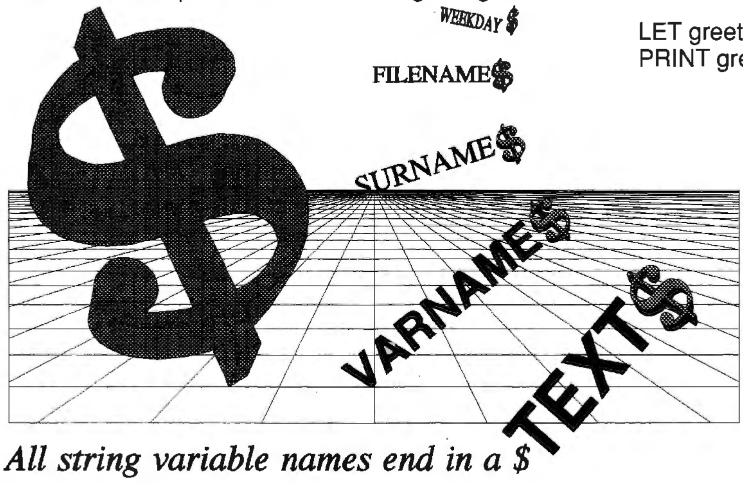
```
PRINT "hello"
```

Strings are stored in variables exactly as numbers are, but the variable name must end in a dollar sign in order to distinguish a string variable from a number variable. Looking at it from the computer's point of view there is an obvious reason for this imposition. Everything in the computer's memory is a sequence of numbers, so how is the computer supposed to know if a group of bytes are supposed to be a number or a piece of text?

One answer is to use different labelling rules, as SuperBasic does. Other computer languages have other means of achieving the same effect, but all Basics use the dollar method (**figure two**). The main advantage of the Basic method is that both the computer and the programmer are immediately aware of a variable's status simply by examining its name.

Here are a couple of commands using string variables:

```
LET greeting$ = "Hello everybody."
PRINT greeting$
```



All string variable names end in a $

& and spaces

SuperBasic includes some extremely neat ways of adding strings together (known as 'concatenation' in textbooks) and splitting them into smaller strings. SuperBasic also has a facility called 'coercion', which means that a string of digits can be treated in the same way as a normal Basic number.

The ampersand is used to add strings together, such as:

```
LET greeting$ = "Hello " & "world"
```

Note that there is a **space** at the end of the first string, otherwise the result would be "Helloworld" and not "Hello world".

Let us now suppose that we want to print the first five characters of the string we have just created. What could be simpler than this:

```
PRINT greeting$ (1 TO 5)
```

The tenth letter of the string can be printed by the command:

```
PRINT greeting$ (10)
```

The same idea is used to assign values to other variables, so to create two variables from the greeting$ string, type:

```
LET first$ = greeting$ (1 TO 5)
LET second$ = greeting$ (7 TO 11)
```

Quotes

A couple of technical problems relating to strings need to be covered here. If the double quotes are used to signify the beginning and end of a string, how on earth can a string include double quotes? The following statement is clearly going to cause the computer problems:

```
LET speech$ = "He said "Goodbye" and left."
```

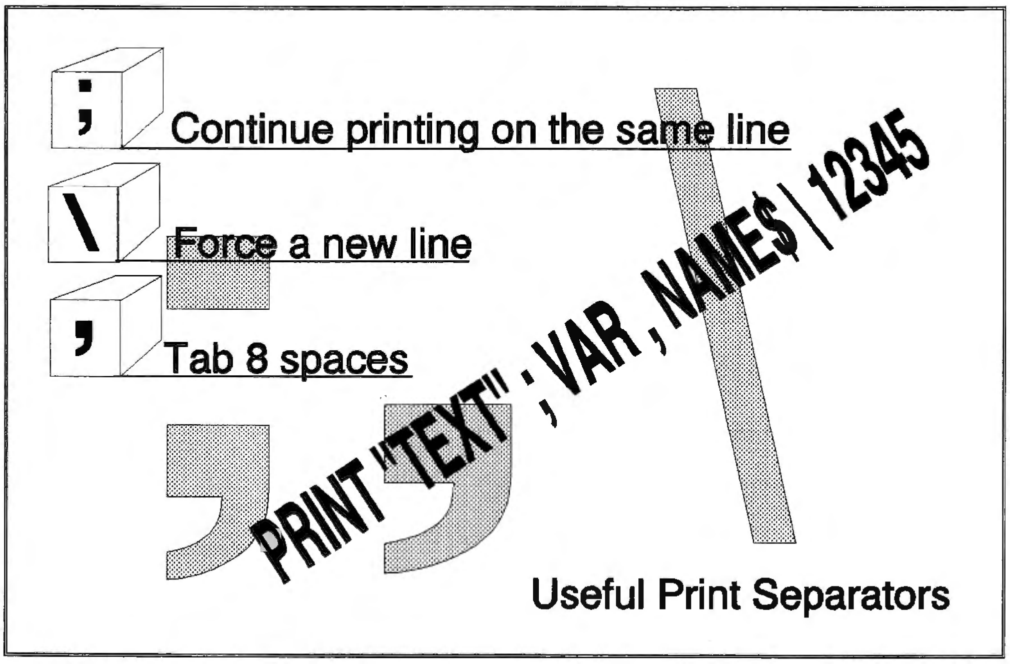
Sinclair's solution is typically simple: where text includes double quotes, use single quotes (apostrophes) to begin and end it. Where text includes an apostrophe, limit it with double quotes. The only rule is that whatever you use to start a string must also appear at the end. The correct way to write the above command is:

```
LET speech$ = 'He said "Goodbye" and left.'
```

A second problem centres on the length of a string. It would be extremely useful to be able to measure the length of a string. SuperBasic includes a valuable little keyword called LEN which provides this service:

```
PRINT LEN("Hello")
PRINT LEN(speech$)
```

The first example is not terribly useful. If you have typed in a string you can also measure it for yourself. The second example shows the true value of LEN because the length of the variable speech$ might be unpredictable for any number of reasons.



; Continue printing on the same line
\ Force a new line
, Tab 8 spaces
PRINT "TEXT" ; VAR , NAMES$ \ 12345
Useful Print Separators

Up to now we have been using PRINT and LET pretty much interchangeably in the examples. PRINT statements are useful in tutorials because the results are immediately visible on the screen. However, the PRINT statement has some interesting variations not shared by the LET command.

So far, every PRINT statement has resulted in text or numbers being printed at the beginning of a fresh line. The AT statement introduced last month can influence where the next print will appear, but even this can be an awkward way of managing the screen layout.

The PRINT statement comes to the rescue by allowing several items to be printed with just one PRINT command. The items are separated by punctuation, rather like the use of commas to separate parameters in WINDOW and AT commands. In PRINT statements, however, the punctuation marks take on special meanings and are described as 'print separators'.

Semi colons

A semi-colon forces the next PRINT item to appear immediately following its predecessor rather than beginning a new line. A comma tabs to the start of the next eight-character-wide column. The backslash forces the next print item to appear on the next line (**figure three**). Print separators can appear at the very end of a PRINT statement so that they affect the next PRINT command. You are allowed to put two or more print separators together, for example to tab by 16 spaces with two commas.

To put all of this into some practical application, let us write a program which will produce multiplication tables on the screen. All programs, even apparently simple ones, need to be planned out so that the end product meets the original requirements efficiently. The first step in the planning process is to understand precisely what is required. Our plan is to:

```
1. Declare which multiplication table to print.
2. Clear the screen.
3. Print "1 x ? = ??" to "12 x ? = ??"
4. Stop.
```

The first two steps of the plan are easy:

```
100 LET number = 5
110 CLS
```

The third step involves a whole set of PRINT statements, each one of which looks quite complicated. As is frequently the case with program designs, it is necessary to break down the overall step into smaller sub-steps. What is needed is a single PRINT statement in the middle of a loop which counts from 1 to 12 before stopping.

```
3.1 Set a counter equal to 1.
3.2 Print "counter x number = ??"
3.3 Add 1 to the counter.
3.4 If the counter = 13, go to the finish.
3.5 Go to Step 3.2.
```

Apart from the actual PRINT statement, which is still a little vague, the overall plan of the program is clear enough to begin writing the commands. The PRINT statement is going to be composed of some text and some numbers with the appropriate print separators. Once again, we need to break down one of the planned steps to reveal a little more detail. This time the SuperBasic language will be used directly:

```
PRINT counter; " x "; number; " = "; counter * number
```

An examination of the listing and the plans printed above show how easy it is to translate a clear design into a working SuperBasic program. Using the same technique, design and program the following exercise:

> Type in a single string containing all of the three-letter abbreviations for the months of the year ("Jan Feb Mar...").
> Now produce a program which slices out each of the months and prints it on a separate line.
> Hint: create a loop surrounding a single PRINT statement.

LISTING 1

```
100 LET number = 8
110 CLS
120 LET counter = 1
130 PRINT counter; " x "; number; " = ";
     counter * number
140 LET counter = counter + 1
150 IF counter = 13 THEN GOTO 170
160 GOTO 130
170 STOP
```

Note: change the value of "number" in Line 100 to obtain different multiplication tables.

THE NEW USER GUIDE

In the fifth part of our New User Guide, Mike Lloyd looks at data statements in Superbasic, and the usual data input sources; keyboard, microdrive and floppy disks.

SECTION FIVE

So far in this series all the information given to the computer has come from SuperBasic commands such as 'LET x = 4'. In real life, programs have to cope with data entered by users.

SuperBasic is extremely rich in keywords associated with obtaining data from a variety of sources. There are three main sources of data input available to Basic programmers: the keyboard, microdrive and floppy disk files, and from special data statements within SuperBasic programs themselves. The latter will be dealt with on another occasion.

The keyboard provides the most direct contact with the user, and often provides the most problems. Users are notoriously prone to typing in the wrong thing, so the only way to trust them is not to trust them at all.

Data files on microdrives or disks and data statements within SuperBasic programs can be relied on to contain valid information provided that the programmer has full control over what is placed in them. If they contain information written by users then it is usually best to treat their contents with caution.

INPUT

Input to a SuperBasic program can be obtained using the keyword INPUT. INPUT is very closely related to the PRINT statement with which you are now familiar. At its simplest, INPUT is followed by a single variable name:

```
INPUT number
```

If the above line is typed in as a direct command, the cursor will flash on the next free line of the default window. Anything typed at the keyboard will appear in that window until the Enter key is pressed, at which point the computer will attempt to make the variable 'number' equal to whatever has been typed in. Try it out, and prove that the input has been accepted by typing the following command:

```
PRINT number
```

With luck, the output from the PRINT command will be identical to the input from the INPUT command. However, if what you typed was not a valid number the computer will respond with an error message, and if you typed a very large or a very small number the computer will have displayed the result in scientific notation. Sticking to numbers with less than six digits will avoid the distraction of scientific notation for the moment.

INPUT's similarity to PRINT shows in the way that it can be preceded by an AT statement to determine where on the screen the input line will appear; it is affected by PAPER and INK statements; a channel number can be included in INPUT statements; and any number of inputs can be obtained provided that each variable name is separated by a valid print separator. It is therefore possible to write code such as:

```
AT#2; 4, 7
PAPER#2, 5
INPUT#2; A, B$, C
```

Basic's fondness of halting processing with error messages is one of its biggest weaknesses, and nowhere is this more irritating than in the way in which Basic objects to invalid data entered in response to INPUT commands. Fortunately, INPUT statements can also accept text data simply by using string variables, such as:

```
INPUT text$
```

No matter what is typed into a text variable the QL will never complain that the input is invalid, so it is a good rule always to use string variables with INPUT statements. If necessary, numbers can be converted, or more properly 'coerced', from text format to the computer's internal numeric format. This is demonstrated by the following program fragment:

```
INPUT text$
LET square = text$ * 2
```

COERCION

For coercion to work properly the string must begin with a valid number, for example it might rear "123 High Street", otherwise we are back to an error message.

The simplest way to obtain trouble-free numeric input has been published in *Sinclair QL World* previously. The trick is to add a leading zero to all strings before coercing them. **Listing one** shows exactly how this can be done by rewriting last month's multiplication table program so that it accepts input from the user before printing a table.

LISTING 1 - MODIFIED MULTIPLICATION TABLE

```
100 PRINT "MULTIPLICATION TABLE"
110 INPUT "Enter number "; INFO$
120 LET NUMBER = "0" & INFO$
130 IF NUMBER = 0 THEN STOP
140 CLS
150 LET COUNT = 1
160 PRINT NUMBER; " x "; COUNT; " = ";
170 PRINT NUMBER * COUNT
180 IF COUNT = 12 THEN GOTO 100
190 LET COUNT = COUNT+1
200 GOTO 160
```

As you begin to develop programs requiring user input, the following sequence of statements will feature regularly in your programs:

```
PRINT "Enter your name"
INPUT name$
```

Computer language developers have added an extra facility to the INPUT command to assist in this circumstance. You can put text strings into an INPUT statement which will be printed out prior to the cursor appearing for input. The two lines above can be encapsulated in a single INPUT command such as:

```
INPUT "Enter your name" \ name$
```

You should by now be well acquainted with the associate between the keyboard and input on the one hand and screen windows and output on the other. Communication between the computer and these devices is by means of 'channels', which can be imagined as conduits for Ascii characters. The concept of channels in central to the way the Qdos and SuperBasic make communications easy within any QL system.

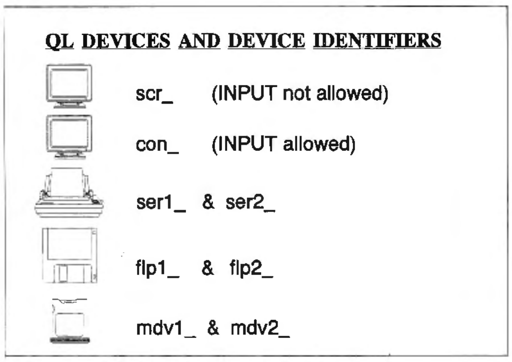
When the QL is switched on three channels are opened by default, one for each of the screen windows. As a special arrangement for typing commands the keyboard is initially associated with the #0 window, also known as the command window.

When an INPUT statement is used, the keyboard is temporarily attached to the #1 window, aka the default window. If the INPUT statement includes the #2 channel identifier, the input appears in the #2, or listing, window. As we will soon learn, channels can also be opened to a printer and to files on microdrives and diskettes.

OPEN

To open other windows new channels are designated with the OPEN command. OPEN must be followed by a channel number and by a device identifier. The identifier can be extended if necessary to include all sorts of extra information which will where possible be avoided in order to keep things simple.

The five device identifiers commonly used on QLs are listed in **Figure one,** although QDOS recognises a sixth, NET, which is used to connect QLs together in a network. Third party manufacturers have extended QDOS to recognise devices not originally included in the basic QL. The best examples are the now ubiquitous FLP1_ and FLP2_ device names for floppy disks.



QL DEVICES AND DEVICE IDENTIFIERS

scr_	(INPUT not allowed)
con_	(INPUT allowed)
ser1_ & ser2_	
flp1_ & flp2_	
mdv1_ & mdv2_	

There are two types of screen window. Windows opened as consoles allow the INPUT command to be used in association with them, but windows opened as screens do not permit INPUT to be used. All three default windows are consoles.

A printer can be attached to the QL via one of the two serial port sockets at the back of the computer. Serial ports are so called because each of the bits of a byte of information are passed one after the other down a single wire, in serial fashion. The alternative arrangement is to have the bits of each byte travel simultaneously down eight wires, ie in parallel.

All QLs have two microdrives units fitted to their superstructure to allow files to be saves and accessed. When a channel is opened to a microdrive it is connected to a named file on a microdrive cartridge loaded onto the specified microdrive.

These days, many QL systems include one or two floppy disk units. Floppy disk drives are much faster and more reliable than microdrives, and floppy disks hold about seven times more information than a microdrive cartridge. Floppy disks are also extremely cheap, working out at around 75p per megabyte (roughly 1,000 kilobytes or a million bytes) whereas £20 worth of microdrives are needed to hold the same amount. Of course, before switching to floppy disks the cost of the disk drive must be considered, but the QL becomes a much more useful, responsive, reliable and fast machine with a disk drive attached.

If a new screen window was required, one which needed to accept INPUT commands, it could be opened with the following command:

```
OPEN#3, con_
```

The window will have default values for its size and colour which can be changed with appropriate commands, such as:

```
WINDOW#3, 140, 20, 10, 10
PAPER#3, 2
INK#3, 0
```

There is no requirement to open channels with consecutive numbers, although to do so is usually a good idea. It is also advisable to re-use channels, simply by opening them as a new device. Novices often fall into the trap of unnecessarily opening large numbers of windows at various screen locations. A more economical approach is to have just one window channel which can be relocated to suit what is being printed at the time.

In order to use a printer in SuperBasic a command such as:

```
OPEN#5, ser1_
```

must be used.

More often than not the device name needs extending to include essential parameters so that the computer and the printer are synchronised. Additionally, the speed of data transmission might need to be changed using the BAUD command, but these refinements will be discussed in a later part of the *New User Guide.*

Once a printer channel is open, the PRINT command can be used to send data to the printer, for instance:

```
INPUT “Type your name.”, name$
PRINT#5, “Your name is ”; name$
```

TO

Some commands associated with screen channels, such as PAPER and INK, have no effect on the printer AT and CLS and all of the graphics commands are also prevented from reaching the printer. A useful substitute for AT which can be used with printers and screen windows is TO, which moves the print position to a specified character position on the current line. In SuperBasic TO is embedded in a PRINT statement, as in:

```
PRINT#5, “This is the”; TO 20; “20th tab stop”
```

TO cannot be used to move the print position to the left of its current location or to the right of the window or page print area.

Programs can be listed to the printer by adding the appropriate channel number, such as:

```
LIST#5
```

CLOSE

Sometimes the last line of text sent to the printer will not be printed because it is held in a memory location called a buffer until some more text comes along to shunt it to the printer. It is possible to force the last few characters out of the buffer by closing the channel to the printer. Unsurprisingly, the keyword is CLOSE and the format for the command is:

```
CLOSE#5
```

By the way, it is a good idea to get in the habit of always using the same channel number for the printer.

Information is stored on microdrive cartridges and floppy disks in files. A file can contain a SuperBasic program, commercial software such as Psion Quill, data such as a word-processed file or some spreadsheet figures, or a representation of a screen display.

To open a new file using SuperBasic, a command from the OPEN family is followed by the name of the device and the file itself, eg:

```
OPEN_NEW#4, mdv 1_demofile
```

Once the file has been opened it can be written to using the PRINT command in exactly the same way as PRINT has been used for screen windows and the printer.

```
PRINT#4, “Here is some stored text”
PRINT#4, “ - and a second line of text”
```

As was the case with the printer, such incidentals as colour information and the AT command cannot be used with files.

Once you have finished writing to the file it can be closed with the CLOSE command:

```
CLOSE#4
```

It is extremely important to remember to close all channels linked to files before switching off the QL or removing the microdrive cartridge or floppy disk from its drive. If a file is open when the QL is turned off its contents will almost certainly be destroyed, and other files on the same medium may no longer be accessible. If you accidentally remove a file medium without closing channels connected to you can usually avoid disaster by re-inserting the correct medium. However, there is no substitute for rigorously following this advice:

CAVEAT!

ALWAYS CLOSE FILE CHANNELS

File contents are accessed by connecting them to a channel with one of two keywords from the OPEN family: OPEN or OPEN_IN. OPEN allows both PRINT and INPUT commands to work, but OPEN_IN only allows INPUT commands to be used, thus protecting the contents of your file from being overwritten by the accidental use of the PRINT command.

It is good practice, at least for the time being, to use OPEN_IN rather than OPEN to read back information from files. You would only use OPEN to write to files which already exist and which contain data you no longer want to keep. To retrieve the two lines of text sent to the file above and print them on the screen, the following commands are required:

```
OPEN_IN#6, mdv 1_demofile
INPUT#6, text$
PRINT text$
INPUT#6, text$
PRINT text$
CLOSE#6
```

Points to note are the choice of OPEN_IN as the safest way of accessing a file for read-only operations, the use of INPUT to read back each line from the file and the CLOSE statement at the end of the sequence. the program as it stands is fine for a two-line file, but for longer files some sort of loop needs to be developed containing the INPUT and PRINT statements.

While it is possible to save a program listing like this:

```
OPEN_NEW#3, flp 1_program
LIST#3
CLOSE#3
```

it is rarely a good idea.

SAVE and LOAD

SuperBasic includes a single command called SAVE which does all of this. The equivalent SAVE command to the sequence above is simply:

```
SAVE flp_1 program
```

To retrieve the program into memory again, type:

```
LOAD flp1_program
```

Once the program has been loaded it can be LISTed, EDITed and RUN. Alternatively, if all you want to do is run the program, type:

LRUN

```
LRUN flp 1_program
```

LRUN is shorthand for ‘LOAD and RUN’. SuperBasic programs can be ‘daisychained’ so that each program loads its successor by including a LRUN command in the program. However, it is a rule that only one SuperBasic program can be in the QL’s memory at one time, so both LRUN and LOAD have the effect of wiping out any program already in memory.

SINCLAIR QL WORLD

THE NEW USER GUIDE

SECTION SIX

Part 6 of the New User Guide builds up FOR and REPEAT loops which allow processes to be repeated automatically — and embedded within other repeated processes — under the programmer's control.

A long time ago I was shown how to prepare a car for display on a garage forecourt. 'Here is a typical car. Wash here, polish here, steam clean the engine compartment, brush out the boot.'

'Now,' instructed the forecourt owner, with a grand sweep of his arm, 'do the same for all of those.'

So far in this tutorial there has not been a concise method of giving the computer similar instructions to repeat a sequence of commands. The nearest we have come to it is the multiplication table program where a GOTO statement has redirected the computer back to a previous line:

```
100 LET N = 2
110 PRINT N * 4
120 LET N = N + 1
130 IF N = 13 THEN STOP
140 GOTO 110
```

This routine undoubtedly works, but it is more than a little cumbersome. It takes up five lines, it does not show at a glance what is going on and the line which should be most prominent — the instruction to multiply N by 4 at Line 110 — is lost among the other statements.

The essence of the routine can be summed up using one word for each statement: initialise; calculate; increment; test; loop. SuperBasic very helpfully has a single statement which combines all but the calculate stage, shortening and simplifying programs.

FOR N

The key to reducing the above five program lines to just two is to recognise that we are here dealing with a range of values for the variable N. The routine could be written in English as 'For each value of N from two to twelve, print N multiplied by four'. SuperBasic would be criticised mercilessly if it were so verbose, so the programming equivalent of the first part of the sentence is:

```
100 FOR N = 2 TO 12
```

In Basic this forms a complete statement, and so it can stand on a line by itself. The second part of the sentence, the action which must be repeatedly carried out, follows on the next line:

```
100  FOR N = 2 TO 12
110 PRINT N * 4
```

In English we can use a full stop to indicate that the sentence is finished. SuperBasic has to allow for the fact that the actions being carried out might occupy a number of lines, and so a special statement is needed to complete the FOR loop:

```
100  FOR N = 2 TO 12
110    PRINT N * 4
120  END FOR N
```

The variable in the FOR statement, N, is no different from other numeric variables, except that it is automatically incremented on each cycle of the FOR loop. Nevertheless, variables used in the control of loop structures have been given special names, such as 'control variable' and 'loop identifier'.

Note that the lines between a FOR statement and its END FOR successor are usually indented so that the structure of the program is clear from a glance at the listing.

Some program editors automatically increment lines in this way (*Archive* is a good case in point), but thanks to the QL's ultra-simple line editor programmers have to remember to do the job for themselves.

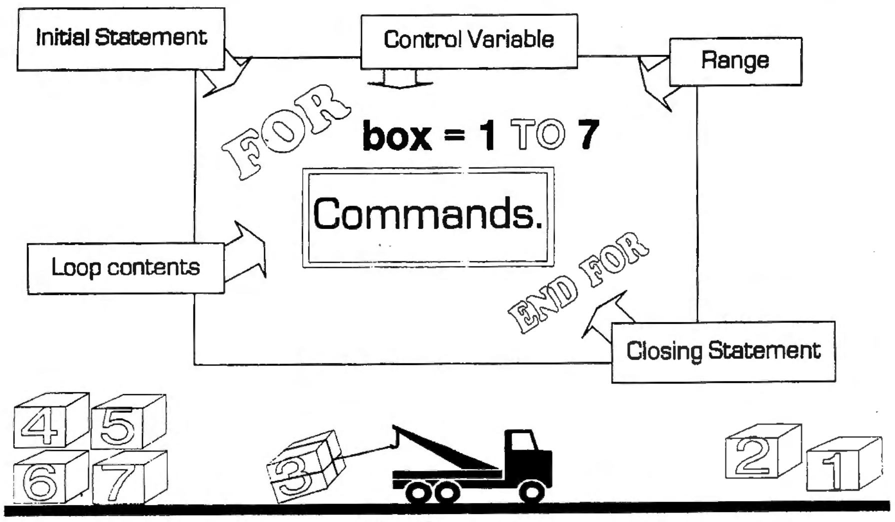


Figure 1: The FOR loop structure.

FOR . . . END FOR

The use of FOR...END FOR has radically improved the clarity of the program, but the earlier promise to reduce five lines to one has yet to be achieved. Alone among all Basic dialects, SuperBasic employs the concept of a 'short form' of its structures. This means that, provided that a structure is wholly contained on a single line, the final statement can be omitted.

The example programs in this series have so far contained one statement per line. To place many statements on a line it is necessary to separate them using a colon — the more English-like fullstop cannot be used to end statements because of its role as a decimal point. The FOR loop can therefore be shortened to:

```
100 FOR N = 2 TO 12: PRINT N * 4
```

SuperBasic makes the same assumption that we do when obeying the instruction 'For values of N between two and twelve...", ie that only whole numbers are intended. This is not always going to be true, so SuperBasic allows different increments to be specified. The numbers between four and 100 which are divisible exactly by four can be quickly listed using the line:

```
100 FOR N = 4 TO 100 STEP 4: PRINT N
```

STEP

The English equivalent would be 'Print every fourth number between four and 100'. The STEP number does not have to be a whole number, nor does it have to be positive. The following two examples count down from ten to one and count up from zero to one in small increments:

```
100 FOR N = 10 TO 1 STEP –1: PRINT N
200 FOR N = 0 TO 1 STEP 0.2: PRINT N
```

FOR loops can be embedded within other FOR loops. You might, for example, want to print every one of the multiplication tables from two to twelve using one program:

```
100 FOR number = 2 TO 12
110   FOR multiplier = 1 TO 12
120   PRINT number;" x "; multiplier;
130   PRINT " = "; number * multiplier
140   END FOR multiplier
150 END FOR number
```

Note the way that incrementing the line indents highlights the structure of the program and makes it easy to see where each loop begins and ends. The important thing to note here is that the 'multiplier' loop is wholly contained within the 'number' loop. It should be obvious that if the *END FOR multiplier* line occurred after the *END FOR number* line then the program would produce nonsense. The concept of containment extends to the use of GOTOs inside FOR loops: they should not direct the interpreter to a point outside the bounds of the FOR and END FOR statements.

An interesting feature of this routine is that it prints 132 lines of information using just one more program line than our initial example. It is a simple demonstration of the way in which computers can be programmed to undertake repetitive tasks with very little effort. The program can be expanded to provide the first 100 multiplication tables, each one with multiples between 2 and 1000 if necessary, simply by changing the values in the FOR loops.

FOR loops are fine provided that the programmer knows in advance the range of values which will be used. There are many occasions when this will not be the case. The first resort is to change the absolute range values for variables.

Take a routine to print out all of the dominos from double blank to double six: the obvious loop structure will print out more dominos than there are in a set:

```
100 FOR left = 0 TO 6
110   FOR right = 0 TO 6
120   PRINT left; " : "; right; " "
130   END FOR right
140   PRINT
150 END FOR left
```

A quick check of the printout from this program reveals that dominos such as 3:5 are also included as 5:3. The only dominos not to be printed twice are the doubles. The listing needs to be altered to remove the duplicates by changing the inner loop:

```
100 FOR left = 0 TO 6
110   FOR right = 0 TO left
120   PRINT left; " : "; right; " ";
130   END FOR right
140   PRINT
150 END FOR left
```

The best way of understanding the exact effect of this seemingly minor change is to type in the listing and run it. Watch carefully for the punctuation in the PRINT statement on Line 120.

Programmers can also feel trapped by the regularity of the STEP feature of FOR loops. What happens if the values taken by the loop control variable are not equally spaced? For instance, what if the first six prime numbers had to be printed? SuperBasic, uniquely, permits a series of comma-separated values to be used instead of a range, which allows it to meet our needs perfectly:

```
100 FOR prime = 1, 2, 3, 5, 7, 11
110 PRINT prime
120 END FOR
```

REPeat loops

The FOR...END FOR structure has many sophistications, some of which have yet to be covered here, designed to cope with specific circumstances. However, enough of this powerful structure's abilities have been covered here to cope with most needs.

There will be circumstances where a loop will cycle an unknown number of times, perhaps even permanently until the program is exited. This could be simulated using a FOR loop with some very large parameters, but SuperBasic provides a better way with another structure called the REPEAT loop.

Like the FOR loop, REPEAT loops have special statements at their beginnings and ends between which appear the program lines which are to be repeated. Here is a typical example of a REPEAT loop:

```
100 REPeat loop
110   PRINT "Hello ";
120 END REPeat loop
```

The characteristic capitalisation of the word 'REPeat' is forced by SuperBasic. It indicates that lazy programmers can get away with typing only REP and leaving the Basic editor to do the rest. This feature only applies to the longer structure-related keywords, which makes one wonder why it was adopted at all.

Returning to the code in hand, the effect of putting the PRINT command between the REPEAT statements is that it is repeated forever, or at least until either the CTRL-SPACE key combination is pressed or the computer is reset. A short form of the REPEAT structure is available governed by the same rules used in short FOR loops. The above example could be rewritten as:

```
100 REPeat loop: PRINT "Hello ";
```

This is the exact equivalent of the old chestnut typed countless times into shop display computers:

```
10 PRINT "Hello ";
20 GOTO 10
```

A more substantial example of a REPeat loop is contained in the original *QL User Guide* (Dec 1984 reprint), but it is not a good example of programming. The routine plays a guessing game: a number is randomly thought of by the computer and a succession of inputs are requested until the player guesses the number. A neater version of the program is:

```
100 LET target = RND(9)
110 REPeat gameloop
120   INPUT "Guess the digit.."; guess;
130   IF guess = target: EXIT gameloop
140   PRINT " is wrong"
150 END REPeat gameloop
160 PRINT "Well done!"
```

EXIT

The vital point about the structure of this game is the inclusion of the EXIT command, otherwise the REPEAT loop would repeat forever. It is usual for EXIT commands to be contained within IF structures, otherwise they would be acted upon during the first cycle of the REPEAT loop, which rather takes away the point of it all. Note that, as in END REPEAT statement the loop's name must be included in EXIT statements.

Loops can contain as many EXIT statements as might be needed, although it is usually a good idea to stick to just one or two to avoid confusing yourself. EXIT statements also have a role to play in some FOR loops where, again, they must be followed by the loop identifier. The value of the EXIT command is that it tells the QL that the loop is finished with and can be forgotten. If a GOTO command was used instead, the QL would spend the rest of the program thinking it was still somewhere in a big loop.

The short form of the FOR and REPeat loops can be borrowed by the IF structure. This provides consistency, meaning that programmers have less to remember when phrasing their commands, and it can ease readability by removing the THEN keyword. Here is a short IF clause:

```
IF x = 5: PRINT x
```

The compatibility between short forms should alert you to the possibility that there is a long form of the IF structure. Only a few Basic dialects allow multi-line IF structures, but SuperBasic is one of them:

```
100 IF account < 0
100   PRINT "Account in red!"
120   LET overdraft = 1
130   INK 2: PAPER 7
140 END IF
```

IF. . . ELSE

There is, of course, no point in having an EXIT clause in an IF statement, nor is there a structure control variable with which to label the EXIT, and so compatibility does not quite stretch this far.

A very useful addition to the IF clause is the ELSE keyword. Very often, two courses of action suggest themselves according to whether or not something is found to be true. In English we might say 'If it is dry I shall go shopping, otherwise I will read a book'. The computer's equivalent is very similar:

```
100 IF account > 0
110   INK 0
120 ELSE
130   INK 2
140 END IF
150 PRINT account
```

There are times when even two courses of action are not enough to cope with circumstances. It is possible to nest IF statements so that the program becomes a succession of IF..ELSE..IF..ELSE lines, but a much neater structure is available using the SELECT keyword. Using SELECT, a number of ranges can be tested for. Each range can have a set of actions associated with it.

NOTES TO LISTING 1

```
LINE 110: A random number is selected.
LINE 150: Each guess is counted...
LINE 160: The gap between the guess and the
          mystery number is calculated.
LINE 180: Every loop must have its EXIT!
LINE 200: Be careful with negative number
          ranges.
LINE 260: The result is printed.
```

LISTING 1

```
100 CLS
110 LET X = RND(100):
120 LET TRIES = 0
130 REPeat GAME
140   INPUT "Your guess ... "; GUESS
150   LET TRIES = TRIES + 1
160   LET DIFFERENCE = GUESS - X
170   SELect ON DIFFERENCE
180     = 0         : EXIT GAME
190     = 1 TO 6    : PRINT "Just too high"
200     = -6 TO -1  : PRINT "Just too low"
210     = 7 TO 20   : PRINT "Too high"
220     = -20 TO -7: PRINT "Too low"
230     = REMAINDER: PRINT "Miles out!"
240   END SELect
250 END REPeat GAME
260 PRINT "You won in "; TRIES; " guesses."
```

To demonstrate SELECT's value, let us expand the guessing game a little so that the users are given hints according to how close they get. See **Listing one** for the program code. The difference between each guess and the target number is calculated and then tested to see in what range it fits. An appropriate message is then printed. Notice the REMAINDER keyword, which must be the last range to be specified if it is used. Readers comparing the syntax with that appearing in the old User Guide will see that the program example is much more concise and easier to read.

The use of ranges in the SELECT structure allies it more with the FOR loop syntax than with that of IF statements. In fact, any ranges and lists which are permitted in FOR commands are also permitted in SELECT commands. This is only sensible because it means that both QLs and programmers need only remember one set of syntax rules.

The intelligent application of structures provides the skeletons for programs: the other commands are merely the flesh. With this section of the New User Guide your programming horizons have been greatly expanded. It may be surprising, therefore, that we have yet to reach the commands which make SuperBasic super...

SINCLAIR QL WORLD

THE NEW USER GUIDE

SECTION SEVEN

In the seventh part of our New User Guide, Mike Lloyd examines functions, parameters, the relationship between whole systems and sub systems, and their application in building large and complex programs.

It has probably already struck several readers that the sort of programming encountered so far in this series has been suitable for small routines such as printing out multiplication tables, but somewhat impractical for projects the size of, say, Digital Precision's *Professional Publisher* (which was nevertheless written largely in SuperBasic). Life becomes complicated with just a few loops and conditional branches, so how can programmers cope with the complexities of scores of menu options, hundreds of conditional operations and dozens of embedded loops without losing their place, their understanding of the program design, and their sanity?

SUBSYSTEMS

Computer programmers found the answer by examining the practices of more conventional engineering. Designers of complex products, such as cars or machine tools, do not design a single entity. Instead, the overall structure is sketched out and its major components are fitted into the sketch. This is done without much attention to the workings of sub components. When this stage is completed the design team concentrates upon the fine detail of each subsystem without having to remember much about the other subsystems.

As the subsystems are joined together fine-tuning takes place to ensure that the finished product operates successfully as a whole. This final step should not be underestimated: the Hubble telescope is much less effective than originally planned because two vital components were not tested together prior to the launch of the satellite.

This process of decomposition and recompostion can be emulated in computer programs. The first step is to decide upon the overall aim of the program and to identify the main components. There may be subsystems, for example, to cope with microdrive activity, or to manage menus. Each subsystem is then designed in isolation before being joined with its fellows and fine-tuned into a completed application.

The first considerations when a computer program subsystem is being designed are the conditions existing when it is first called and the conditions which exist when control passes to the next subsystem. These are known as the boundaries of the subsystem. What happens inside the subsystem should have no direct impact on the other parts of the program, therefore it does not matter how it actually works. Programmers sometimes call this the 'black box' principle of programming. Raw material is fed into the black box and a finished product comes out, but exactly how this is achieved is not visible.

Here is a simple example. It is often very useful to be able to centre a piece of text on the screen. The calculation to find out the character position where the text begins is straightforward: find out the difference between the window width and the length of the string and divide it by two. This then represents the gap between the left border of the window and the beginning of the text. It must follow that the gap at the opposite end will be the same length, although there might be an extra character space if an odd number of spaces have to be accommodated.

Before the routine can do its work the text to be printed must exist and that width of the window must be known. Some thought should also be given to exception conditions, or occasions when the routine will not work properly. For this example we will assume that only one possible error needs to be tested for, and that is if the text string is wider than the window.

A simple program extract, expanded for clarity, which centres a line of text might look like this:

```
100 LET window width = 37
110 LET text$ = "Centred Text"
120 LET text width = LEN (text$)
130 LET start_pos = (window_width - text_width)/2
140 IF start_pos >=0
150 PRINT TO start_pos; text$
160 ENDIF
```

The calculation of the length of text at Line 120 might need some explanation. LEN() is a useful SuperBasic function which calculates the length of any string contained in its brackets, as follows:

LEN

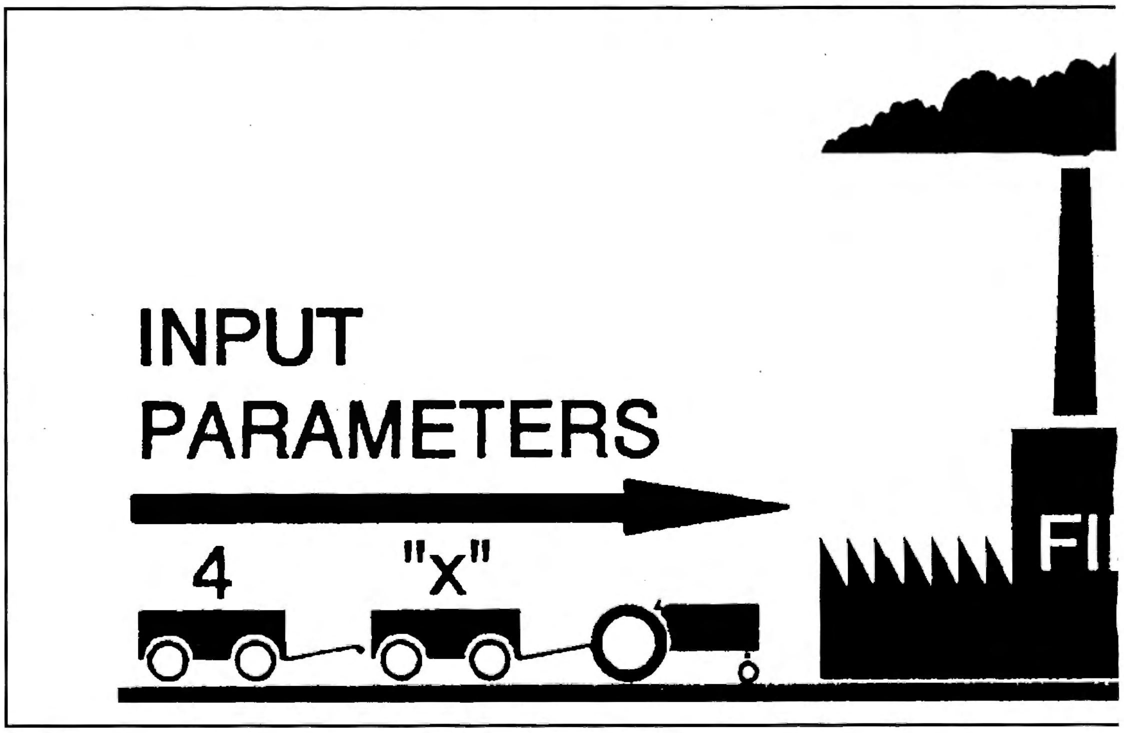
```
PRINT LEN ("Sinclair QL World")
```

The output is the number of characters which make up "Sinclair QL World".
The program snippet is fine for a single instance, but it seems rather a lot of code if centring is going to happen frequently. The early Basic language developers recognised that it would be highly desirable to be able to call up a segment of code such as this from anywhere in a program. Getting the Basic interpreter to move to the segment using the GOTO command was relatively simple. What was a little more complicated was the process of returning the interpreter to where it started from once the segment was completed.
In order to simplify these twin jumps to and from a program segment two new keywords were added to Basic which are also available in SuperBasic. The first is GOSUB, which is used exactly like the familiar GOTO. GOSUB stands for 'Go To Sub-Routine' and it is followed by a line number indicating where the subroutine begins in the program. The end of each subroutine is marked by the keyword RETURN, which directs the interpreter back to where it came from. So, in order to centre a lot of text our earlier example needs to be slightly revised:

GOSUB and RETURN

```
100 LET text$ = "Sinclair QL"
110 GOSUB 500
120 LET text$ = "World"
130 GOSUB 500
140 STOP
.....
500 LET start_pos = (37 - LEN (text$))/2
510 PRINT TO start_pos; text$
520 RETURN
```

Several things have happened here besides the introductions of the GOSUB and RETURN statements. The subroutine has been simplified by assuming that the screen will always be 37 characters wide and that the programmer will never make the mistake of sending an overlong text string to be centred.
Additionally, the calculation of the length of the string has been embedded into a larger expression. Basic is excellent for compressing complicated series of calculations into quite small pieces of code. Unfortunately, it has the effect of making programs more difficult to read, but the saving on space is usually worthwhile.



STOP

Finally, the STOP command at Line 140 ensures that the interpreter does not stumble over the subroutine by mistake after reaching the natural end of the program. When organising programs, it is generally preferable to place all the subroutines together preceded by a STOP command. If subroutines are buried away somewhere in the main program it will be difficult for the programmer to find them and all too easy for the interpreter to find them unexpectedly.

GOSUBs can have an important role to play in ordinary programs, but many SuperBasic programmers are proud of their record of never using GOSUB anywhere in any of their programs. Instead, they use one of the two SuperBasic constructs which truly mark out the Sinclair QL as one of the best programming languages yet produced. The first to be dealt with here is a super version of something which is found in a crude form in many lesser Basics: a user-defined function. The second is very rarely found in Basic dialects: the user-defined procedure.

Functions

If we are going to construct our own function it is useful to know a little more about what goes into a Basic function. Functions are identified by a keyword which is always followed by a pair of brackets. Inside the brackets there may be one or more values or variables called parameters. Parameters are like raw materials entering a factory. The function works on the parameters in order to produce a finished product, or output. The output can be assigned to a variable or printed, but something has to be done with it. In technical terms, functions appear on the right-hand side of expressions. In practice this means that they must be treated as if they were a special form of variable. The first line of those following is incorrect, the subsequent ones are quite valid:

```
100 FILL$ ("x", 100)              :REMark wrong!
150 PRINT FILL$ ("x", 100)        :REMark OK
200 LET x$ = FILL$ ("x", 100)     :REMark OK
```

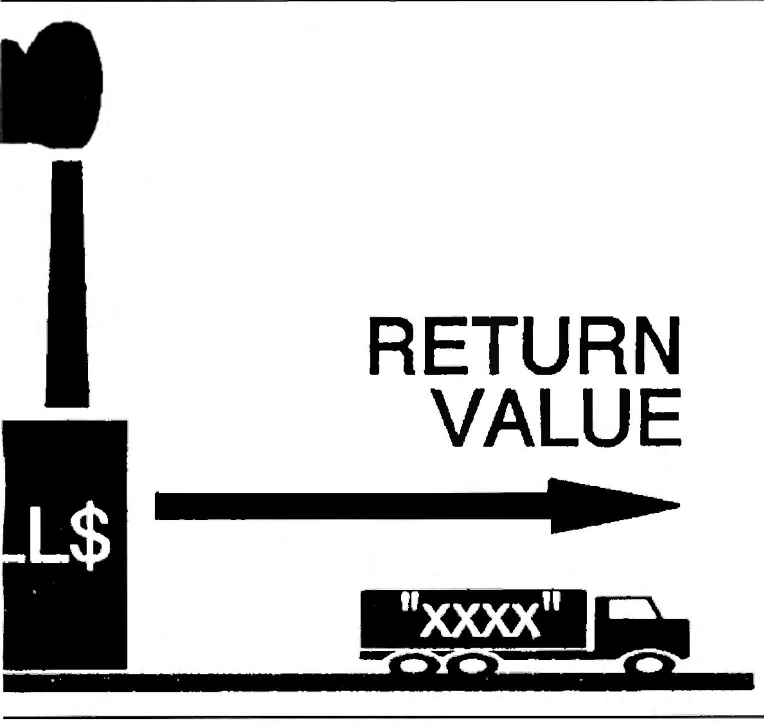
$ again

Another rule is that the function name must end in a dollar sign if its output is going to be a string. The FILL$ function used in the above examples returns a string of Xs 100 characters long, and so its name must end in a dollar. This is exactly the same as is found in the naming rules for variables and so it should be easy to learn. However, if you ever forget to follow this rule the QL produces such a vague error message that the mistake can be horrendously difficult to root out.

The FILL$ function is a SuperBasic keyword, and so how it works is not our concern. We are simply grateful that it does a useful job. When it comes to defining functions of our own we must be able to program its internal workings to produce a final result. There are a number of ways in which the text centring problem might be solved with a user-defined function, only one of which will be examined here. The function is called Centre(). It produces a string with sufficient leading spaces to force a given piece of text into the centre of the window.

Parameters

The first line of function definitions share the same layout. The initial keywords are 'DEFINE FUNCTION' followed by a name which follows normal variable naming conventions. There must also be a pair of brackets which might be empty or, more likely, will contain references to any parameters passed to the function. In the example there are two parameters, the text string and the number of characters to pad it out to. They are given normal variable names but they have a special characteristic: as soon as the function is completed they will be forgotten. Parameters are simply temporary variables created specially for the function and discarded as soon as the interpreter returns to the program proper. Without the concept of parameters it would be impossible to write



The principible behind SuperBasic functions generalised functions which could deal with all manner of variables, rather than just the one whose name happened to appear in the function definition.

Within the function definition the code to produce the final output is written. At some point, usually on the penultimate line of the definition, there must be a statement beginning with the keyword RETURN which is followed by a value or expression. This will be the value returned to the calling expression. Do not confuse this use of RETURN with the use made of it in GOSUB subroutines.

END DEF

The final line, END DEFine, confirms to the interpreter that the function definition is complete. A very short example of a full function definition, with some typical function calls, is provided below:

```
100 PRINT square (25)
120  LET test = square (15)
.....
500 DEFine FuNction square (x)
510     RETurn x * x
520 END DEFine
```

The text centring function is slightly more involved, although it has been expanded to show each stage of the process as a separate command:

```
100 example$ = "Hello World"
110 PRINT Centre (example$, 37)
.....
500 DEFine FuNction Centre (text$, wide)
510     LOCal twide, left, result$
520     twide = LEN (text$)
530     left = (wide - twide)/2
540     result$ = FILL$(" ", left)
550     result$ = result$ & text$
560     RETURN result$
570 END DEFine
```

LOCal

Again this brief listing contains new material. Note first how the parameter variable 'text$' is used to refer to a variable that the main program knows as 'example$'. The second line of the definition lists a number of variables that are only needed for the duration of the function. They are therefore declared as being LOCal, which means that just like the parameter variables they will be forgotten as soon as the interpreter returns to the main program.

To save space and time all of the LET keywords have been omitted. This is a convenience to programmers and produces snappier-looking code. The SuperBasic FILL$ function performs a valuable role in padding out the string. Notice that strings are added together using the ampersand symbol rather than the addition sign, which is reserved for numeric addition only. For those readers who like a challenge, this definition can be reduced to only a DEFine, an END DEFine and a RETURN statement.

Mid Print

There is a much simpler way of approaching the problem. Instead of writing a function we can define our own command. The processor is very similar to that of defining a function, including the use of parameters, but there is no RETURN statement and the first line begins 'DEFine PROCedure' rather than 'DEFine FuNction'. To continue using the text centring problem as an example, what we will be asking the computer to do is not to PRINT CENTRE ("Hello", 37) but simply to MidiPrint "Hello", 37. Here is the code:

```
100 example$ = "Hello World"
110 MidPrint example$, 37


500 DEFine PROCedure MidPrint (text$, wide)
510     LOCal twide, left, pad$
520     twide = LEN (text$)
530     left = (wide - twide)/2
540     pad$ = FILL$ (" ", left),
550     PRINT pad$; text$
560 END DEFine
```

Although there are brackets enclosing the parameters on the first line of the procedure definition there are no brackets in Line 110 when the procedure is actually called. After all, we do not write PRINT ("Hello'), so there is no need to use brackets in procedures we have written ourselves.

It should be absolutely obvious by now that SuperBasic commands such as PRINT, CLS, LINE and OPEN are procedures. The language developers have allowed us to expand two classes of keyword, procedures and functions, to include our own facilities that SuperBasic's creators never dreamed of.

Within each procedure or function definition we are writing what is to all intents and purposes a mini-program. This is exactly how Professional Publisher and other large programs were produced, by splitting the problem down into little chunks and assembling them into the complete solution. User-defined procedures and functions are very easy to understand, but they offer a quantum leap in the power they hold for the programmer prepared to use them properly.

THE NEW USER GUIDE

SECTION EIGHT

Part 8 of the New User Guide turns to handling data in numeric and character arrays, and setting up multi-dimensional arrays. Channels are used to send an array to mdv and retrieve it again.

So far in our exploration of SuperBasic we have been concentrating on the mechanics of getting the computer to do what we want it to. The language's procedures, functions and control structures have been examined and put to some use, but a very important omission must now be put right. The most valuable part of a computer is the data it contains, yet in our efforts to understand SuperBasic, the data it is designed to manipulate has taken a back seat.

Data

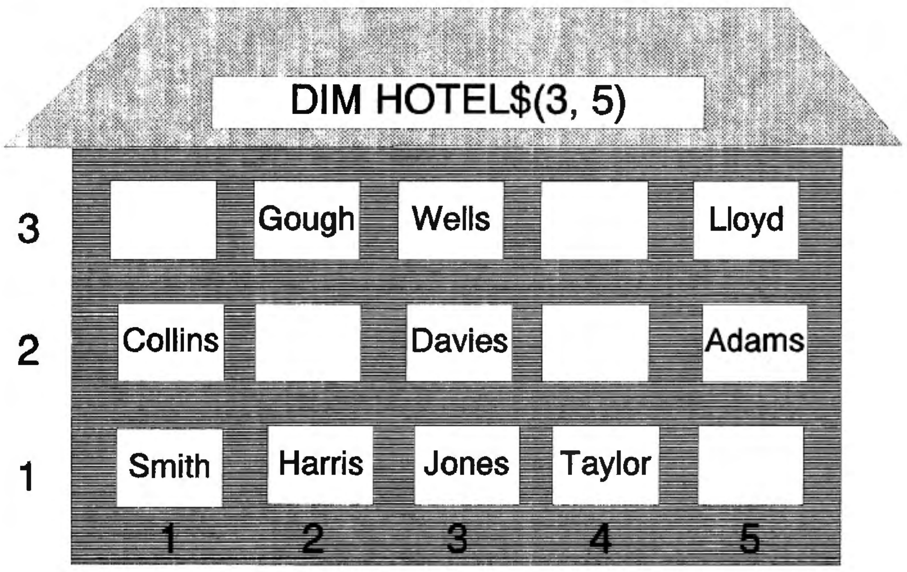
Even when it is first turned on, the QL holds lots of information. It knows SuperBasic and Qdos, microdrives, the alphabet, simple mathematics, trigonometry and much else. These things are knowledge, but they are not data. In computing, 'data' has a narrow meaning, referring to information specific to a computer application. The banking details in an accounts program, the customer details in a marketing database, the possible moves in a chess program and the page objects in a desktop publishing package are all data.

Incidentally, there is some confusion over whether 'data' should be singular or plural. Strictly, a single piece of information is a datum and several pieces are data, but it is normal to treat 'data' as a group noun. 'Datum', which is a Latin word meaning "a thing which is given", is rapidly becoming an archaic word which is little used in English speech. 'Database' is on the other hand a relatively new word used to refer to any organised collection of information, but which is especially used to describe information held on a computer in an organised and accessible form.

What little data has featured in the *New User Guide* up to now has been inconsequential and transient. No sooner was it created than it was printed on the screen and forgotten. Data is not like that in real life. Data tends to be bulky, organised into sets and, usually, valuable enough to keep. SuperBasic needs to be able to deal with such data if it is to be a genuinely useful programming language.

Arrays

In Basic, data is held in variables, but the variables we have used up to now have held only one piece of information. For instance, the computer could hold a single customer's name in a variable called Customer$, but this would be of little value if the details of hundreds of customers had to be stored and manipulated. SuperBasic handles multiple pieces of similar information in data structures called arrays.



At its simplest, an array can be imagined to be like a table in a reference book, for instance:

CUSTOMERS
J Smith
M Jones
S Taylor
A Davies
L Collins

DIMension

To create such an array in SuperBasic a special command is used to define the array and ensure that sufficient memory is set aside to hold the array's contents. This command is DIM, which is short for 'dimension', and it is followed by the name of the array with its size in brackets. Array names follow the usual rules for naming variables, including the rule that requires a $ sign at the end of the names of arrays and variables which hold text data. The command:

```
100 DIM Customer$ (100,30)
```

creates an array called Customer$ which has room for 100 entries, each of which can have up to 30 characters. SuperBasic is an exception to most Basic dialects in that it requires the programmer to state the maximum length of each element in a string array – most Basics delay allocating memory space until a string array element is filled and so have no need for a maximum length.

In this respect, SuperBasic is more modern, like the C programming language and most current database languages. In SuperBasic it is possible to define an array comprising a single string, eg DIM Example$ (40), meaning a string with a maximum of 40 characters. This becomes very useful when a compiler such as Digital Precision's *Turbo* is used, and it will also make C's string handling familiar should your programming requirements ever take you in that direction.

Numeric arrays are even easier than character arrays. If each customer's account balance is to be held alongside their names, a suitable array could be defined with the command:

```
110 DIM Balance (100)
```

'Subscripts'

With numeric arrays it is not necessary to declare how long the longest number will be because SuperBasic holds all numbers from zero upwards in the same amount of memory space.

Information is placed into and read from arrays using a subscript to determine which array element is involved. If the twelfth customer is M Larkin with an account balance of £50, this information could be stored with the following commands:

```
120 Customer$ (12) = "M Larkin"
130 Balance (12) = 50
```

The idea of subscripts is very valuable because of the flexibility it gives to programmers. For example, here is a program fragment which prints out the customer database with the appropriate account balances:

```
200 FOR x = 1 TO 100
210     PRINT Customer$(x),,Balance(x)
220 END FOR x
```

It is perfectly possible to make valuable use of arrays without straying far from the very basic principles so far covered, but the adventurous can take arrays much further. The two arrays used above are 'single dimensioned arrays', in other words they both represent a single column of data.

Multi-dimensional arrays

SuperBasic supports multiple-dimensioned arrays. A two-dimensioned array is like a table with many columns and rows, such as:

J JONES	10 HIGH STREET	WORCESTER
T SMITH	15 THE AVENUE	YORK
A HARRIS	22 ACACIA AVENUE	GLASGOW

A three-dimensioned array might be used to refer to a set of such tables in a book, while a four-dimensioned array might refer to sets of tables in books in a number of libraries. Arrays with more dimensions are catered for quite happily in SuperBasic but programmers tend to have difficulty imagining them, so let us stick with two-dimensioned arrays for now.

Imagine a hotel which has six floors each with twenty bedrooms. The computer can hold the names of each bedroom's occupant in an array defined as:

```
100 DIM Room$ (6, 20, 10)
```

The array represents a matrix of 6 rows and 20 columns and the content of each matrix cell is limited to 10 characters. We can now write procedures which handle several likely occurrences, such as:

LISTING 1

```
100 REMark : Example arrays
110 :
120 REMark : Simple numeric array
130          DIM balance (40)
140 :
150 REMark : Simple string array
160          DIM customer$ (40, 20)
170 :
180 REMark : Multi-dim numeric array
190          DIM demo (10, 10, 20)
```

```
200 REMark : Fill two arrays with dummy data
210 FOR x = 1 TO 40
220   balance(x) = RND(100)
230   customer$(x) = "Mr Smith"
240 END FOR x
```

```
250 REMark : Save both arrays to floppy disk
260 StoreBalance  : REMark see Listing 2
270 StoreCustomer : REMark see Listing 2
```

```
280 REMark : Re-initialise both arrays
290 DIM balance (40)
300 DIM customer$ (40, 20)
```

```
310 REMark : Now restore data from floppy disk
320 FetchBalance  : REMark see Listing 3
330 FetchCustomer : REMark see Listing 3
```

```
340 REMark : Prove that arrays are populated
350 PRINT balance
360 PRINT customer$
```

Listing 2

```
500 DEFine PROCedure StoreBalance
510    DELETE flp1_balance
520    OPEN_NEW#3, flp1_balance
530    PRINT#3, balance
540    CLOSE#3
550 END DEFine
```

```
600 DEFine PROCedure StoreCustomer
610    DELETE flp1_customer
620    OPEN_NEW#3, flp1_customer
630    PRINT#3, customer$
640    CLOSE#3
650 END DEFine
```

Listing 3

```
700 DEFine PROCedure FetchBalance
710    OPEN_IN#4, flp1_balance
720    INPUT#4, junk
730    REMark: removes the 0 element
740    FOR x = 1 TO 40
750      INPUT#4, balance (x)
760    END FOR x
770    CLOSE#4
780 END DEFine
```

```
800 DEFine PROCedure FetchCustomer
810    OPEN_IN#4, flp1_customer
820    FOR x = 1 TO 40
830      INPUT#4, Customer$(x)
840    END FOR x
850    CLOSE#4
860 END DEFine
```

Book Mr Jones into Room 3 on the fifth floor:

```
Room$(3, 5) = "Mr Jones"
```

Find out who is in Room 12 on the second floor:

```
PRINT Room$(2, 12)
```

Record that Mrs Davies has left Room 15 on the first floor:

```
Room$(1, 15) = ""
```

Restrictions on arrays

Unfortunately, SuperBasic places two restrictions on the use of multi-dimensioned arrays which make them less helpful than they might otherwise be. Firstly, the array must contain all strings or all numbers, whereas other languages allow an array to have a mix of strings and numbers just as a reference table might have text and numeric columns. Secondly, SuperBasic insists that the cells in string arrays are all the same size, so you cannot save space by declaring some short elements and some long elements.

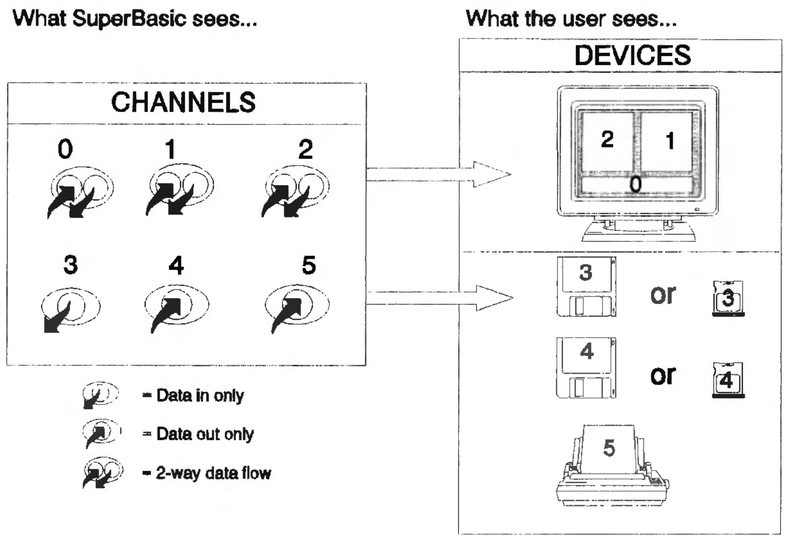
The only way round these problems is to use lots of arrays and make sure that the subscript numbers match as they did in the earlier example of Customer$ and Balance. If you are designing a database, some thought needs to go into the way arrays will be used. To extend the hotel example a little, arrays might be needed to store the dates of arrival and departure of each guest, the morning newspaper they order, their bill total, and so on. The arrays might look like this:

```
100 DIM Room$       (6, 20, 10)
110 DIM Arrival     (6, 20)
120 DIM Departure   (6,20)
130 DIM Newspaper$  (6, 20, 14)
140 DIM Bill        (6, 20)
```

Memory use

It is possible to allocate a great deal of memory space using arrays like this, so a rough calculation is necessary to ensure that the arrays will fit into the QL along with the program necessary to access the data. Numbers occupy five bytes no matter what size the number actually is on screen, so for each of the numeric arrays a total of 6*20*5 bytes will be used, or 600 bytes. The string arrays will occupy 6*20*10 = 1200 bytes and 6*20*14 = 1680 bytes.

The raw data in this small database will eat up no less than 4680 bytes of valuable memory space. Arrays also need additional memory space to hold 'pointers' which allow the computer to find its way around the array matrix. In a string array each element has an overhead of two bytes and each dimension has a further overhead of two bytes. Numeric arrays only need two bytes per dimension, which is usually a negligible amount. Before writing your magnum opus home accounts program it is a very good idea to calculate exactly how much memory space will be needed to store the payee, amount, description and date for every cheque you might write in a year.



Storing an array via a channel

Part 5 of the New User Guide broached the subject of channels. Channels can be used to connect the QL with any device, such as printers, modems, microdrives and disk drives. Once the connection is made data can be sent to the device or received from it via the channel. To conclude this section on data arrays we shall use channels to save an array to a microdrive file and to retrieve it back into memory. These are important techniques because a great deal of time can be expended filling arrays with information which users wish to view again at a later date.

Sending an entire array to a file could not be easier because SuperBasic allows the whole array to be copied with a single command. A file is opened and linked to a channel, the array is copied using the PRINT command, and the file is closed again. **Listing two** shows how this is achieved.

Within the file the array loses much of its structure. Each element is recorded separated by a line feed, so if the file is viewed on the screen using the command:

```
COPY MDV1_ARRAYFILE TO SCR_
```

the display will list all the elements in order. The file will not contain information about the array's name, its total size, the length of its strings or the number of dimensions it has.

Restrictions on arrays

Retrieving the array contents is, accordingly, a little more difficult. Firstly, if the original array no longer exists (because perhaps the program using the array has been re-loaded) the program must declare the array using the correct name and details. The elements can then be inserted from the storage file using a loop for each array dimension. The examples at **Listing three** relate to single-dimensioned arrays and are therefore quite straightforward.

An interesting point to note is that numeric arrays have a 0 element, so an array declared by DIM number(9) will have ten elements with subscripts from 0 to 9. Because string arrays do not have a 0 element it is customary to ignore the existence of the 0 subscript in numeric arrays. In Listing three the 0 element is read from the file and ignored.

With any database program there is a fairly well-defined set of activities which need to be carried out to make the data useful. The program must be capable of adding, altering, displaying and deleting data. It must be able to store data between program sessions and restore the data at the start of a session. The program must usually be able to sort data into order and group items according to a variety of criteria. The program will also need to be able to print the database, or selected parts of it.

All these things are possible within SuperBasic, although the limitations of the Basic dialect can make some of the operations quite slow when dealing with large amounts of information. Most of the activities require loops to be set up. For example, if someone's cheque records were held on the computer it would be possible to construct a loop written over a given period. Sort routines can be highly complex in their use of loops: *Sinclair QL World* has published dozens of SuperBasic sorting routines in its time.

Plan ahead!

Before contemplating a major database program, it is advisable to spend a great deal of time planning before programming commences. Detailed advice on program planning has been published in the *Better Basic* and *Super Basic* series. It is also worthwhile to consider the merits of using a SuperBasic compiler such as *Q-Liberator* or *Turbo.*

SINCLAIR QL WORLD

THE NEW USER GUIDE

SECTION NINE

In this, the final part of the Beginner's section of the Sinclair QL World User Guide, Mike Lloyd examines the all-pervasive influence of computer logic.

Along with all other computers, the Sinclair QL gets its fair, or unfair, share of the blame for 'computer errors'. It is extremely convenient to be able to instruct a machine to carry out a task and then blame it for doing what it was told to do, instead of doing what it was expected to do. The widespread introduction of computers has thrown into sharp relief the human propensity for being vague. My university tutor was always exalting the virtue of being roughly correct over being precisely wrong. Computers do not have that luxury.

Often, computer users first meet this insistence on absolute accuracy when their machine rejects a mis-spelt command. But then, to a computer, a word is either a keyword or it isn't. There are no half-measures, shades of grey or room for doubt.

This yes-no world is underpinned at the very heart of the computer by its binary counting system. A computer circuit is either on or off, which can be an analogue of true or false, yes or no, right or wrong, up or down. If two circuits are compared with each other, they reveal a surprisingly rich diversity of information depending upon how they are compared.

Binary logic

Let us imagine that a circuit is switched on if it is raining and off if it is dry, and another circuit is switched on if you intend to go out and off if you intend to stay in. These two circuits can be used to determine under what circumstances you might put on a coat. If it is dry and you intend to stay in (represented by OFF and ON respectively) you will not need a coat, nor would you if it were dry and you intended to go out (OFF and OFF). If it was raining and you intended to stay in (ON and OFF) a coat is still unnecessary, so only if it was wet and you intended to go out (ON and ON) would you reach for your coat.

The logic of this situation can be reduced to a set of statements along the lines of 'If it is wet and I intend to go out I will wear a coat'. Mathematically, this can be stated by a simpler 'YES and YES equals YES'. The logical statement 'If it is dry and I intend to go out I will not wear a coat' can be restated as 'NO and YES equals NO'. Once the underlying logic of the situation has been prised out it can be applied to no end of similar situations: 'If I have the money and the time I will go to the cinema'; 'If the tide is right and the sails are set the ship will go to sea'; 'If the lights are green and the road is clear, motorists may proceed'.

Taking the last example, the QL can quickly be programmed to give good road safety advice:

```
100   REPeat loop
110       INPUT "Are the lights green? (Y/N)", greenlight$
120       INPUT "Is the road clear? (Y/N)", clearroad$
130       IF greenlight$(1) == "Y" AND clearroad$(1) == "Y"
140           PRINT "You may proceed."
150       ELSE
160           PRINT "Do not proceed."
170       ENDIF
180   END REPeat loop
```

Notice that the double equals signs mean that either an upper case or lower case Y will be interpreted as Yes and that any other character will be assumed to mean No. The use of subscripts in the IF statement allows the user to get carried away and write Yes instead of Y and still obtain the right answer.

Truth tables

Four logical statements can be constructed from the interplay between two conditions to form what is known as a 'truth table'. Each column of the table is headed by the condition, which is often framed as a question (Is it raining? Are the lights green? etc.). To conserve space, the column headings below are simple statements.

AND

RAINING		GOING OUT		WEAR A COAT
NO	AND	NO	=	NO
NO	AND	YES	=	NO
YES	AND	NO	=	NO
YES	AND	YES	=	YES

The most important element of the truth table is the character of the link between the two logical states. In the above example the link is an 'AND', which implies that only when both states are true will the result be true. This need not always be the case. Imagine that there is an additional circuit linked to temperature and that we intend to wear a coat if it is raining or if it is cold. Assuming that we will now only test this circuit and the 'raining' circuit when we are going out, the following truth table can be constructed:

OR

RAINING		COLD		WEAR A COAT
NO	OR	NO	=	NO
YES	OR	NO	=	YES
NO	OR	YES	=	YES
YES	OR	YES	=	YES

Now it appears that only if it is dry and warm will we do without a coat. Once again a computer program can be written to simulate the situation, this time with an IF statement along the lines of:

```
130          IF raining OR cold THEN PRINT "Wear a coat"
```

There is a third logical operator which has more use in computing than it does in real life. Should we decide to go out if it's raining or it's cold, but not if it's both, then our decision to wear a coat will go along the following lines:

XOR

RAINING		COLD		WEAR A COAT
NO	XOR	NO	=	NO
YES	XOR	NO	=	YES
NO	XOR	YES	=	YES
YES	XOR	YES	=	NO

XOR is a bit of computerese meaning 'exclusive OR'. In other words, if one thing or the other is true then the result is true, but if both are true or both are untrue then the result is false.

Boolean algebra

The mathematical expression of logic, you might suppose, grew out of the relatively recent study of computing. However, the truth tables and the logic behind them were first developed in the middle of the last century by an English mathematician called George Boole. His fame rests on two publications, *A Mathematical Analysis of Logic* published in 1847 and *An Investigation into the Laws of Thought* published in 1854. His name lives on in the term we use to describe computer logic: Boolean algebra.

Another surprising feature of truth tables is their use in the QL's colour display. In physical terms, the colours on your computer monitor are produced by three 'guns' firing streams of electrons at the reactive surface of the monitor screen. There is a red gun, a green gun and a blue gun, hence the description 'RGB monitor'.

All eight colours in the full QL colour set are formed by different combinations of output from the guns. At the points on the screen where none of the guns strike the colour is black; where all three guns strike is white. Monochrome monitors use only one gun, which is why they are cheaper and smaller. Also, because focusing is more accurate with only one gun, monochrome monitors tend to produce sharper images which are easier on the eye for extended periods of work.

QL colour

Consistent with its obsession with binary digits, the QL likes to represent the use of the colour guns by 1s and the absence of a colour gun by 0s. With three bits to represent green, red and blue (in that order) a total of eight combinations can be made with values ranging from 0 to 7. The following table should come as no surprise to anyone familiar with the PAPER and INK commands in SuperBasic:

GUNS USED	COLOUR	COLOUR VALUES
None	Black	— = 000 = 0
Blue only	Blue	—B = 001 = 1
Red only	Red	-R- = 010 = 2
Red and Blue	Magenta (purple)	-RB = 011 = 3
Green only	Green	G— = 100 = 4
Green and Blue	Cyan (light blue)	G-B = 101 = 5
Green and Red	Yellow	GR- = 110 = 6
Green, Red and Blue	White	GRB = 111 = 7

Bitwise operators, Boolean logic and colour.

Incidentally, when the QL is in four-colour mode the blue gun is used only to produce white. The final binary digit of each colour value is otherwise ignored, producing the well-known combination of black, red, green and white. Check the table above to see how the QL converts the unavailable colours into Mode 4 colours.

Computer designers face a small dilemma when trying to fit this three-bit concept into eight-bit bytes. It is anathema to waste valuable memory space, so the bytes in the computer's screen map would never contain just three used bits preceded by five unused bits. Most computers devote half a byte of memory to each pixel, so the first four bits define one pixel and the next four define its neighbour. The lower three bits of each half-byte (which are affectionately known as 'nibbles') correspond to the colour guns of the monitor. The leading bit is often used to denote brightness or, as is the case with the QL, flashing. Sadly, the QL's designers compromised on functionality in favour of cost-effectiveness in this decision: sixteen colours would have been much more useful than the ability to flash colours, but would have added to the QL's manufacturing costs.

SuperBasic has some interesting operators which allow us to use Boolean algebra directly on binary digits. These are known as the bitwise operators and are listed below:

AND	&&
OR	||
XOR	^^

Let us take the SuperBasic value for magenta (011, or 3) and perform a bitwise AND with the value for cyan (101, or 5). Where the corresponding binary digits are both 1 then the result is 1, otherwise the result is 0. The product of 'magenta AND cyan' is 001, or blue, the only primary colour they have in common. The SuperBasic command PAPER 3 && 5 is therefore a long-winded way of saying PAPER 1.

If the sum were changed so that an OR was performed the result would be white: 011 OR 101 = 111. Neither AND nor OR are particularly useful with screen displays because AND has a 75% chance of removing a primary colour while OR has a 75% chance of adding a primary colour. The results therefore tend towards being black or white.

The XOR operator is different because it produces a similar colour-richness to the colours being mixed. When a colour is XOR'd with white it produces a complimentary colour, so that black becomes white, blue produces yellow and so on. This is demonstrated by the following code snippet:

```
100 MODE 8: CSIZE 3,1: CLS
110 FOR background = 0 TO 7
120       PAPER background
130       INK background ^^7
140       PRINT "The Sinclair QL"
150 END FOR background
```

Using one of the QL's special screen display modes colours can automatically be added together using the exclusive-or (XOR) truth table to produce other colours from the colour set. The command which puts the screen into this mode is OVER -1. If you want OVER to apply to a window other than the default window the command should contain a channel reference (eg OVER#5, -1). To see the effect which OVER -1 has, run the following short routine:

```
100 MODE 8: OVER -1
110 PAPER 0: CLS
120 FOR X = 0 TO 7
130     INK X: FILL 1
140     CIRCLE 50, X*20, 30
150 END FOR X
160 OVER 0: FILL 0
```

Fill bug

Note the careful resetting of the screen mode back to the default of OVER 0: the effects of changing this setting can sometimes be confusing and lead you to believe there is something wrong with the computer. The FILL 1 command is deliberately set into the FOR...NEXT loop precisely because there is something wrong with the computer: a bug in most QL Roms means that subsequent circles are not properly filled unless the FILL command is re-issued.

A more practical use of logical values is to use INK colours as a quick way of identifying negative values in an accounts program. It relies on the way in which the QL can assign a true/false value to any expression it evaluates. A 'true' expression, such as '7 > 3', is given a value of 1 whereas a 'false' expression, such as '7 = 4', is given a value of 0. If negative account balances are to be written in red ink the expression to be tested will be something like 'balance <= 0'. The following program fragment explains how it might be used:

```
3100 REMark Print account balance
3110 INK 7-5* (balance <=0)
3120 PRINT balance
```

KEYROW

Another way to make use of logical values is to test for keypresses. INPUT and INKEY$ are familiar keywords, but the *New User Guide* has not yet covered KEYROW. The main value of KEYROW is that it can detect multiple keypresses. This is important, for example, in arcade-style games where players might want to move right and fire their laser weapon at the same time. KEYROW is completely unconnected with Ascii character values and it divides the keyboard into eight sets of eight keys on a fairly random basis.

Somewhere in the QL's memory there are eight bytes dedicated to representing which keys are being pressed. Each keypress makes its associated bit turn on. As soon as the key is released the bit turns itself off again. KEYROW gives access to these byte values and it is up to the programmer to work out which keys are being pressed. Logical expressions come in very handy.

The first step is to read a KEYROW value. KEYROW(1) reads the byte which reveals whether the solidus, Enter, space, Escape, or cursor keys are being pressed. If the Enter key is pressed it sets the lowest bit in the byte, equating to a value of one. However, if Enter and the left cursor key are pressed simultaneously the KEYROW(1) value is three. In fact, any odd value between 1 and 255 indicates that, amongst other keys, the Enter key is being pressed. Using conventional IF...THEN statements quickly becomes impractical.

What is needed is a single IF statement which will reveal whether or not the Enter key is being pressed regardless of how many other keys are being pressed simultaneously. Only a logical expression, namely a bitwise AND, can do this. Here is a small routine to prove it:

```
100 REPeat loop
110     key = KEYROW(1)
120     AT 2,0
130     IF key && 1: PRINT "Enter ";
140     IF key && 2: PRINT "Left ";
150     IF key && 4: PRINT "Up ";
160     IF key && 8: PRINT "Escape ";
170     CLS4
180 END REPeat loop
```

Each IF statement tests a particular bit and, if it is set to one, prints the key it represents. To save space, only the first four bits are tested here. The CLS 4 command in Line 170 clears the remainder of the cursor line. Try it out by pressing any combination of the four tested keys. Provided that you press the keys simultaneously the screen read-out will correctly identify which keys you are pressing.

Where To Now?

Next month

This article completes the Beginner's Section of the *New User Guide*. Rather like the driving test, it simply prepares you to learn on your own. There are plenty of good books on the subject of SuperBasic programming, the best being by Jan Jones. Next month the New User Guide continues with its detailed and definitive descriptions of the complete SuperBasic vocabulary.

SINCLAIR QL WORLD

THE NEW USER GUIDE

SECTION TEN

KEYBOARD INDEX

This month Mike Lloyd begins a new phase in the New User Guide – the Keyword Index. As well as Qdos keywords, the Index will cover keywords in SuperToolkit 2 and Turbo Toolkit, as well as modifications to standard SuperBasic by Minerva and Toolkit 2.

Welcome to the Keyword Index of Sinclair QL World's *New User Guide*. The Index presents every SuperBasic keyword in alphabetical order with each entry including a full explanation of the syntax. Where it is appropriate, brief program snippets demonstrate how the keyword can be used. SuperBasic is a very rich language, with most commands and functions having a wide variety of mandatory and optional parameters. The *QL User Guide* supplied by Sinclair was incomplete in its descriptions of some commands and inaccurate in its treatment of others, which can be mystifying to programmers eager to learn the fundamentals of the language. The QL World New User Guide aims to put right these shortcomings.

Because the majority of QL programmers no longer restrict themselves to standard SuperBasic the additional commands provided by the two most widely used toolkits, Tony Tebby's *Super Toolkit 2* and Digital Precision's *Turbo Toolkit*, have been included in the Index. Readers without these toolkits can judge for themselves whether these extensions to SuperBasic are likely to be of value to them. Modifications to standard SuperBasic by both *Minerva* and Toolkit 2 are highlighted where they occur. The Keyword Index is therefore an extremely valuable and comprehensive reference work.

The Index groups keywords according to their function, such as 'graphics command', 'text function', 'file command' and so on. However, readers should be aware of three more fundamental categories into which all SuperBasic keywords fall: procedures, functions and structures. Procedures 'do things', such as PRINT, SAVE and BEEP. The things that they print, save or beep are called parameters, such as PRINT *text$*, SAVE *"mdv1_myprogram"* and BEEP *4000, 230*. Every SuperBasic command statement begins with a procedural keyword except for LET statements, where the keyword is optional. Functions take zero, one or more arguments, manipulate them in some way and return a value. An example is LET demo$ = FILL$("X", 12). The FILL$ function takes "X" and 12 as its arguments and produces a string of twelve Xs which are 'returned' to the *demo$* variable. In a well-designed language like SuperBasic, functions are easily distinguished from procedures by the presence of brackets after them. The brackets contain any arguments passed to the function. When no parameters are given, such as in BEEPING, the brackets are optional, but it remains good practice to include them nevertheless.

The third category of keywords provides structures. These keywords are used to control the flow of the program by imposing repitition (FOR... NEXT or REPeat... END REPeat) or by diversion (GOTO, IF... THEN). A special category of structural keywords permit programmers to extend SuperBasic with the inclusion of their own procedures and functions. The only limitation is that the new keywords must be capable of being defined using conventional SuperBasic.

Most programming languages allow statements to be constructed with a confusing range of defaults and optional parameters. User guides must try to represent optionality in the language syntax, but often do so in a way which actually obscures the command structure they are trying to represent.

The Keyword Index provides the full syntax for each keyword and explains the options and defaults in the accompanying text. Keyword variations (LINE and LINE_R) are generally dealt with together in order to avoid repetition. The parameters used to demonstrate the syntax give a clue to what they represent, so 'ELLIPSE xpos, ypos, radius' suggest three numeric values representing the co-ordinates of the centre of an ellipse and a measure of its radius. 'PRINT text$' suggests a text string. Where commands such as PRINT have a huge number of variations several examples of statements are given. Many of these points are demonstrated by the following example:

CIRCLE	**#chan, xpos, ypos, radius, distort, angle [;xpos, ypos, *etc*]**
CIRCLE_R	**#chan, xpos, ypos, radius, distort, angle [;xpos, ypos, *etc*]**
	GRAPHICS COMMAND
#chan	(optional) channel number (Window 1 is default)
xpos, ypos	co-ordinates of circle centre. CIRCLE uses absolute co-ordinates, CIRCLE_R uses relative co-ordinates.
radius	radius of circle in graphics units
distort	(optional) ratio between major and minor axes of an ellipse, between 0 and 1
angle	(optional) orientation of major axis (0 = vertical)

CIRCLE uses absolute co-ordinates; CIRCLE_R uses relative co-ordinates. CIRCLE draws circles and ellipses on the screen in the current INK colour using the graphics co-ordinates system. IF FILL 1 is in effect the circles will be solid. A bug causes "colour leaks" if FILL 1 is not issued between successive CIRCLE commands. By separating sets of parameters by semi-colons a single CIRCLE command can draw many circles. The CIRCLE and ELLIPSE commands are absolutely identical.

```
100 FOR x = 1 TO 100: CIRCLE 50,50, x*2, 0.3, x/20*PI
```

ABS(x)
[SuperBasic]

ABS(x, y, z, etc.)
[Minerva]

	MATHS FUNCTION
x	a numeric expression.
x, y, z	a set of numeric expressions

ABS returns the absolute value of a single numeric parameter, ie ABS(6) retruns 6 and ABS(-3) returns 3. The Minerva ROM extends ABS to allow further parameters; ABS then returns the square root of the sum of the squares of the parameters.

ACOS(x)
ACOT(x)
ASIN(x)
ATAN(x)
ATAN(xpos, ypos)
[Minerva]

	TRIGONOMETRY FUNCTIONS
x	a numeric expression between -1 and 1
xpos, ypos	the graphics co-ordinates of a point

The arc cosine is the inverse of the cosine, or cos(x^-1). The other functions provide the inverse of the cotangent, sine and tangent respectively. The User Guide states wrongly that there is no effective limit to the size of the parameter: values greater than 1 produce an overflow of error. The Minerva rom permits an additional parameter to be added to ATAN which more accurately calculates ATAN(y/x). Minerva's ATAN allows you to pass the two co-ordinates of a point on the circumference of an imaginary circle with its centre at 0,0 (ie it is the relative position which is important, no the absolute location): ATAN will then return the angle from the centre of the given point, with the top of the circle being 0 radians and the bottom being pi radians.

ADATE seconds

	TIME/DATE COMMAND
seconds	a numeric expression

ADATE adjusts the QL's internal clock by a given number of seconds. Negative values move the clock backwards, positive values move the time forward. ADATE 60*60*24*7 advances the QL's clock by exactly one week.

AJOB id, priority
[Super Toolkit II]

AJOB name, priority
[Super Toolkit II]

TASK COMMAND

id	the task's QDOS number
name	the name of the task
priority	a value typically between 1 and 128

AJOB activates a task (or job) with the given priority. The higher the priority, the greater the share of CPU time the task will be given. A task's ID can be found by listing tasks using the JOB command.

ALARM hour, minute
[Super Toolkit II]

TIM/DATE COMMAND

hour	a value between 0 and 24 representing the hour of the day.
minute	a value between 0 and 59 representing the minute of the hour.

ALARM causes the QL's buzzer to sound at a given time.

ALCHP(bytes)
[Super Toolkit II]

ALLOCATION
(bytes, task, tag)
[Turbo Toolkit]

MEMORY FUNCTIONS

bytes	number of bytes to be reserved
task	(optional) task number
tag	(optional) task tag

ALCHP and ALLOCATION perform the same task: they allocate space from the QL's common heap even when several tasks are running. The SuperBasic equivalent, RESPR, only reserves memory when SuperBasic alone is running. The Minerva rom provides the same result by rewriting the RESPR command. All these commands return the base address of the reserved area.

ALTKEY char$,
command1$,
command2$, *etc.*
[Super Toolkit II]

CONTROL COMMAND

char$	a single keyboard character, eg "A" or CHR$(45)
command$	a string containing valid SuperBasic statements

ALTKEY associates one or more statements with a single keypress. The command ALTKEY "L", "CLS#2: LIST","" clears the listing window and lists the current program when the key combination ALT-L is pressed. ALTKEY settings are cancelled when the QL is reset, therefore commonly used ones are often placed in boot files. Where multiple command strings are used a linefeed is inserted between each one. To force a linefeed at the end of a set of commands (as in the example above) a null string is added. ALTKEY settings remain effective when tasks such as Quill are running, although care should be taken to avoid conflicts with keypresses used in the program.

ARC #chan, xpos,
ypos TO x2pos, y2pos,
angle; x3pos,
y3pos TO *etc*

ARC_R #chan, xpos,
ypos TO x2pos, y2pos,
angle; x3pos,
y3pos TO *etc*

GRAPHICS COMMAND

#chan	(optional) screen channel number
xpos, ypos	(optional) start co-ordinates for the line
x2pos, y2pos	end co-ordinates for the line
angle	degree of curvature in radians

ARC draws a curved line using absolute graphics co-ordinates; ARC_R uses co-ordinates relative to the current graphics cursor. When the first pair of co-ordinates are omitted the line starts at the current graphics cursor location. Multiple curves can be drawn using a single ARC command with each set of parameters separated by a semi-colon. Positive values of **angle** cause an anti-clockwise curve; negative values cause a clockwise curve. A semicircle is specified using an angle of pi, therefore the centre of a football field can be obtained by:

```
100 LINE 50,40 TO 50,60
110 ARC 50,40 TO 50,60, PI
120 ARC 50,60 TO 50,40, PI
```

AT #chan, charline,
charcolumn
[Late QL roms]

AT #chan, charcolumn, charline

[Early QL roms]

TEXT COMMAND

chan	(optional) screen channel number
charline	a value representing the text line
charcolumn	a value representing the text column

AT moves the text cursor to the given position, which must be a valid character position within the specified window. The effect of an AT command is seen with the next PRINT statement. Early QL roms (those with dongles attached to them) reversed the order of the parameters. There can be very few such machines still in use.

AUTO startline, increment

CONTROL COMMAND

startline	(optional) an integer in the range 1-32767, default 100
increment	(optional) an integer, default 10

AUTO generates program line numbers beginning at **startline** and incrementing by **increment.** If the line numbers are already taken up by the program their contents will be displayed in the editing window. AUTO is therefore identical to EDIT except for its default settings. If the **startline** value is omitted, a comma must precede the **increment** value to distinguish it from a starting value.

BASIC_B%(offset)

[Turbo Toolkit]

BASIC_W%(offset)

[Turbo Toolkit]

BASIC_L(offset)

[Turbo Toolkit]

BASIC_POINTER (offset)

[Turbo Toolkit]

MEMORY COMMAND

offset	an offset from the start of the SuperBasic system variable area (must be an even number for BASIC_W%, BASIC_POINTER and BASIC_L.

Like all computer operating systems, Qdos needs to store information in ram relating to the position of the cursor, the colour of the screen, the size of characters and so on. The QL stores blocks of standard information, name tables for variables and channel blocks for screens and other devices. Unusually, Qdos is also prone to shunting this information round ram as tasks are added and deleted from memory. The internal layout of the information blocks do not change, so provided that the start of the block can be located a particular value can be traced from its offset from the starting byte. The Turbo Toolkit provides a whole suite of functions which automatically find system information blocks, but programmers must discover for themselves how the contents are arranged.

BASIC_B% returns a one-byte value, BASIC_W% returns a two-byte value and BASIC_L returns a four-byte value. Incidentally, BASIC_L is not an integer (ie it does not end in %) because four-byte numbers potentially exceed the QL's miserly integer range. BASIC_POINTTER is identical to BASIC_L but is included in the Toolkit preseumably to improve program readability: many four-byte values in the system information tables are pointers to other addresses in the QL's memory.

BASIC_INDEX% ("keyword")

[Turbo Toolkit]

BASIC_NAME$(x)

[Turbo Toolkit]

MEMORY COMMAND

"keyword"	a string expression containing a valid SuperBasic keyword or language extension.
x	a valid integer offset.

BASIC_INDEX locates the Qdos name table and searches for the given keyword. The search is case sensitive, so capitals must be used where SuperBasic uses them. Searches for structure keywords such as GOSUB and DEFine will fail as they are stored elsewhere in the QL's memory. If a match is found the function returns the keyword's position in the name table list. BASIC_NAME$ does the opposite by returning the name of the keyword at position X in the name table.

BAUD rate

CHANNEL COMMAND

rate	a valid baud rate, ie: 75, 300, 600, 1200, 2400, 4800, 9600 or 19200

The baud rate describes the speed with which information is passed through the RS232 ports at the back of the QL. The lower baud rates (baud = bits per second) are most often used by modems to send data across telephone lines. The standard rate for printers is 9600 baud, and this is the default setting when the QL is switched on or reset. Before communication can occur between the QL and a device both must have identical baud rate and data transfer protocol settings. Protocol settings can be set using the OPEN command. The QL can only transmit on 19200 baud, not receive. Although the QL has two RS232 serial ports their transmission speeds cannot be set separately.

SINCLAIR QL WORLD

THE NEW USERGUIDE

SECTION ELEVEN

KEYWORD INDEX

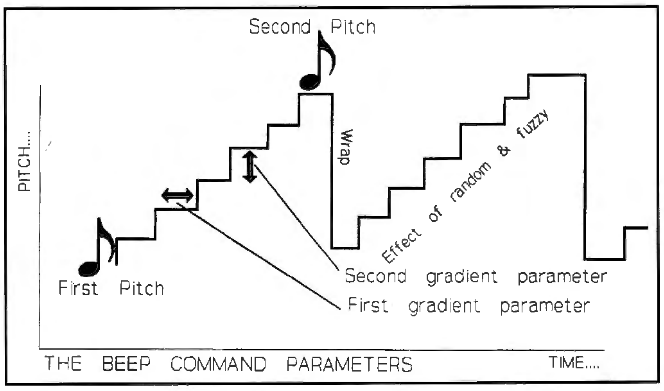
This month in the Keyword Index, Mike Lloyd moves from BEEP to CLOSE in SuperBasic, skirting Super Toolkit 2 and Turbo Toolkit on the way.

BEEP **(duration, pitch1, pitch2, grad1, grad2, wrap, fuzzy, random)**

	SOUND COMMAND
duration (0 to 32767)	the length of the sound. 1 = 72 microseconds
pitch1 (0 to 255)	the first note pitch
pitch2 (0 to 255)	the alternate note pitch
grad1 (-32768 to 32767)	time interval between gradient steps
grad2 (-8 to 7)	pitch difference between steps
wrap (0 to 15)	number of gradient repetitions
fuzzy (0 - 15)	distorts pure tone
random (0 - 15)	random element affecting all parameters

The BEEP command can cause more hair-loss than almost anything else associated with the QL. Bearing in mind that the QL was supposed to be a business machine, its sound capabilities are surprisingly oriented towards arcade games. It is impossible to play a simple tune without a great deal of effort because the pitch values are not related to musical tones and semitones. However, if the sound of a rampaging, homicidal alien is required, the QL can quickly come up with suitable suggestions.

Without any parameters, the BEEP command performs the valuable function of stopping any noise currently in progress. The minimum parameters acceptable are a duration and a pitch. A one second note (of indeterminate pitch) is produced by the command BEEP 720, 8. Note that a permanent note is obtained by giving a duration of 0, which also has the unfortunate effect of changing the pitch of the note. If a second pitch is given you must also give the speed with which the QL will 'bounce' between the two notes and the pitch difference between each tone in the gradient. Wrap, fuzzy and random are further, optional parameters which distort the sound in ways best listened to rather than described.



BEEPING ( )

SOUND FUNCTION

(No parameters – brackets are optional)

The BEEPING function returns True if the QL's sound chip is working and False if it is not. You may not assign these values to a variable (BEEPING will return a 0 if you try) so BEEPING is restricted to IF and SELECT statements. Purists will want to put brackets at the end of the function, but those less fussy can get by without them.

BGET #chan \offset, byte1, byte2 . . .

[SUPER TOOLKIT 2]

LOW LEVEL CHANNEL ACCESS

#chan	a valid channel number, usually a file
offset	an optional value denoting where in the file to begin reading data expressed as a number of bytes from the beginning of the file.
byteX	a value between 0 and 255.

BGET (B stands for Byte) reads bytes from a channel normally associated with a file. BGET assumes that the file is simply one long string of bytes. The following command will read the Ascii values stored between character positions 20 and 22:

```
BGET #3 \20, A, B, C
```

Whether the results are of value depends upon the structure of the file being read.

BIN$ (decval, bits)

[SUPER TOOLKIT 2]

BASE CONVERSION FUNCTION

decval	an integer value
bits	the number of binary digits to produce

BIN$ takes a decimal value and converts it into a string of 1s and 0s which represent its binary value. To see directly the result of a binary AND, use the following commands:

```
PRINT BIN$ (137, 8)
PRINT BINS$ (89, 8)
PRINT BIN$ (137 && 89, 8)
```

BIN (string$)

[SUPER TOOLKIT 2]

BASE CONVERSION FUNCTION

STRING$	any string of characters, but normally made up of 1s and 0s.

BIN converts any string of characters into a decimal number by assuming that characters with even Ascii values represent 0 and those with odd Ascii values represent 1. Conveniently, the Ascii values of 0 and 1 are even and odd respectively.

BLOCK #chan, width,

height, xpos, ypos, ink

PIXEL-BASED GRAPHICS COMMAND

#chan	A screen channel
width	The width of the block
height	The height of the block
xpos	The lateral co-ordinate of the block's top left corner
ypos	The vertical co-ordinate of the block's top left corner
ink	The colour of the block (0 - 255)

BLOCK is unique in that it is the only graphics command to make use of the pixel co-ordinate system starting at the top left corner of each window. Because it does not need to translate between arbitrary floating point graphics co-ordinates and pixel positions it is significantly faster than any other graphics command. Use it to draw vertical and horizontal lines in preference to the conventional LINE command. If OVER is set to -1 a BLOCK of a suitable colour covering the whole window will recolour the window very much faster than RECOL can manage, but with less control over the colours achieved. Unlike conventional graphics commands, BLOCK causes errors if you place any part of the block outside the window.

BORDER #chan, width, paper

GRAPHICS COMMAND

#chan	(optional) A screen channel
width	The width of the border in pixels
paper	(optional) The colour of the border (0 - 255)

Every window can have a border around it, reducing the room available to the active screen. The border's width remains constant whether a screen is displayed in high or low resolution. If no 'paper' value is given the border is transparent, therefore a multi-coloured border can be obtained by specifying the innermost colour first and ending with a wide, transparent border to protect the colours, as in:

```
BORDER 8, 5
BORDER 6, 3
BORDER 4, 1
BORDER 8
```

BPUT #chan, byte, byte, . . .

[SUPER TOOLKIT 2]

LOW LEVEL CHANNEL ACCESS

#chan	a valid channel number, usually a file or the printer
offset	an optional value denoting where in the file to begin reading data expressed as a number of bytes from the beginning of the file.
byte	a value between 0 and 255.

Information is passed to and from QL devices, such as the screen, printer and files, one byte at a time. When you think that you are printing the word HELLO on your screen the computer is actually transmitting 72, 69, 76, 76 and 79 to the channel associated with the default window. These figures are the Ascii equivalent of the characters HELLO. The command BPUT 72, 69, 76, 76, 79 is the low-level equivalent of typing PRINT "HELLO". The command is of more value when storing values in a file or selecting printer settings. Setting a five-character-wide left margin on an Epson printer can be achieved by either of the following commands:

```
PRINT #5, CHR$(27); "I"; CHR$(5): REM 28 significant keypresses
BPUT #5, 27, 108, 5: REM 15 significant keypresses
```

CALL address, data1, data2. . . data 13

MACHINE CODE PROGRAM COMMAND

address	A valid, even-numbered memory location.
dataX	(optional) Four-byte integer values.

CALL is used to initiate a machine-code program previously loaded into a reserved part of the QL's memory with a RESPR command and an LBYTES command. The data parameters are loaded into the cpu addresses D1 to D7 and A0 to A5 prior to the program beginning.

CDEC$ (value, width, decplaces)
[SUPER TOOLKIT 2]

DECIMAL NUMBER FORMATTING COMMAND

value	An integer value
width	The total number of character positions to return
decplaces	The number of decimal places to display

CDEC$ is an extremely useful way of circumventing the QL's habit of placing numbers with relatively few significant digits into exponential format. As a bonus it even provides fixed length strings and separates thousands with commas for attractive, justified columns of figures. Although CDEC$ output can include decimal places, only the integer part of the input value is recognised. This means that you must think of £34.25 as 3,425 pennies. The results can be observed with this snippet:

```
100 REPeat loop
110 pennies = RND (999) * RND (999)
110 PRINT "£"; CDEC$ (pennies, 8, 2)
120 END REPeat loop
```

CHANNEL _ ID (#chan)
[TURBO TOOLKIT]

CHANNEL FUNCTION

chan	An open SuperBasic channel number

SuperBasic and Qdos do not agree over what a particular channel is called. This is so that the QL can support multi-tasking: one program might have Channel #3 linked to a printer at exactly the same time that another program has Channel #3 linked to a screen window. Qdos must be able to service all channel requests and so gives each channel a unique number normally invisible to programmers. *Turbo Toolkit* provides this simple function to return the Qdos channel number for any given SuperBasic channel.

CHARGE (task name)
[TURBO TOOLKIT]

COMPILER DIRECTIVE

filename	(optional) A valid taskname

CHARGE launches a Digital Precision compiler (which one depends upon your default setup). If a taskname is included it is used as the name of the task being compiled.

CHAR_INC #chan, X_inc, Y_inc

[SUPER TOOLKIT 2]

CHARACTER COMMAND

#chan	(optional) A screen channel
X_inc	lateral spacing of characters in pixels
Y_inc	vertical spacing of characters in pixels

SuperBasic offers very limited options for sizing and spacing characters: many of these restrictions are removed by *Super Toolkit 2.* Should you want characters printed at 14 pixel intervals on lines 15 pixels apart, issue the command CHAR_INC 14, 15.

CHAR_USE #chan, font1, font2

[SUPER TOOLKIT 2]

CHARACTER COMMAND

#chan	(optional) A screen channel
font1	the start address of the first font
font2	(optional) the start address of the second font

Qdos divides the full character set into two fonts. Super Toolkit 2 allows you to specify different fonts in each of the screen windows should you wish. Each font is loaded into ram at reserved addresses and the CHAR_USE command is issued to make Qdos aware that the new font is to be used by a specified window. Should you accidentally provide an inaccurate address, all characters will be replaced by garbage. To reset the fonts to their default designs held in the QL's rom, simply issue the command CHAR_USE 0, 0.

CHR$ (ASCII_code)

CHARACTER COMMAND

ASCII_code	An integer between 0 and 255

The Ascii code assigns specific characters to the values 0 to 127, leaving the values 128 to 255 for individual system designers to assign. Thus, every component of a computer's character set can be represented by single byte. The first 32 Ascii codes are non-printable codes representing such functions as backspace, newline and tab. CHR$ is a function which takes a one-byte value (ie an integer between 0 and 255) and produces the character which that number represents. Be careful with the values 0 to 31 because they may produce unexpected results. Minerva owners, however, can obtain printable characters from these values. Super Toolkit 2 owners will prefer to use BPUT to CHR$ wherever possible.

CIRCLE #chan, xpos, ypos, radius, distort, angl, [xpos, ypos, etc]

CIRCLE #chan, xpos, ypos, radius, distort, angl, [xpos, ypos, etc]

GRAPHICS COMMAND

#chan	(optional) channel number
xpos, ypos	co-ordinates of circle centre
radius	radius of circle in graphics units
distort	(optional) ratio between major and minor axes of an ellipse (0 to 1)
angle	(optional) orientation of major axis in radians (0 = vertical)

CIRCLE uses absolute co-ordinates; CIRCLE–R uses relative co-ordinates. CIRCLE draws circles and ellipses on the screen in the current INK colour using the graphics co-ordinates system. IF FILL 1 is in effect the circles will be solid. A bug causes 'colour leaks' if FILL 1 is not issued between successive CIRCLE commands. By separating sets of parameters by semi-colons a single CIRCLE command can draw many circles. The CIRCLE and ELLIPSE commands are absolutely identical. A simple but attractive example of circles is:

```
100 FOR x = 1 TO 100: CIRCLE 50, 50, x*2, 0.3, x/20*PI
```

CL CHIP

MEMORY MANAGEMENT

The QL's memory manager can allocate memory from the 'common heap' for use as reserved memory areas. Qdos also grabs some of this space to record microdrive and disk information (which is why the QL takes seconds to examine a microdrive on the first access and subsequently is content on subsequent accesses to affirm that the drive contains the same cartridge). CLCHP is short for CLear Common HeaP, and it does just that. Use this command with care.

CLEAR

MEMORY MANAGEMENT

When a SuperBasic program runs on the QL it needs memory space to record variable values, etc. CLEAR removes all trace of variables from the QL's memory. Every time a procedure is called, the QL memorises the current state of variables which might have local assignments within the procedure. If a program is halted in the middle of a user-defined procedure or function call the computer still believes it is in the user definition. CLEAR clears this misunderstanding.

SINCLAIR QL WORLD

THE NEW USER GUIDE

SECTION TWELVE

KEYWORD INDEX

This month in the Keyword Index, Mike Lloyd moves from CLOCK to DATA in SuperBasic, with an exposition on the subject of CURSOR.

CLOCK #chan, string

[SUPER TOOLKIT 2]

ON-SCREEN TIME AND DATE DISPLAY

chan	(optional) a valid screen channel
string	(optional) a format for the time display

CLOCK is a simple multi-tasking utility which prints the QL's internal clock time and date to the screen. If no channel is specified a Qdos channel is opened specifically for the clock: it is only suitable for high-resolution mode. When a SuperBasic channel is opened for the clock a string can be declared which determines the output format. The $ and % signs are used to prefix letters to denote the hour, minute, second, day, month and year. An example command and output are shown below:

```
100 OPEN #3, scr_
110 WINDOW #3, 300, 40, 0, 0
120 CLOCK #3, "Time: %h:%m:%s  Date: $d %d $m %y"
```

```
Time: 10:17:22  Date: Wed 03 Jan 86
```

CLOSE #chan

CHANNEL MANAGEMENT

chan	A number representing an open SuperBasic channel

When you have finished with a screen window, a channel to the printer or to a file, the CLOSE command will remove all links between the computer and the device. When windows are closed their contents remain on the screen but the screen area is inactive. If the command window (Channel #0) is closed it can only be reopened as a normal window.

CLS #chan, param

SCREEN COMMAND

#chan (optional)	A valid screen channel (default is 1)
param (optional)	0 Clear all the window (default)
	1 Clear above the cursor line
	2 Clear below the cursor line
	3 Clear whole of cursor line
	4 Clear cursor line to right of cursor

The CLS comand is one of the common factors among all Basic dialects; its influence even extends to MS-DOS. CLS normally has no parameters, but in SuperBasic the command is extended to declare firstly which screen window to clear and secondly what part of the window to clear. This greatly adds to its usefulness. For example, suppose you use the fifth line of a window for user input of indeterminate length. To keep the screen tidy, you could issue a CLS 3 or a CLS 4 command before and after each INPUT command. If you want to preserve the first four lines of a window but clear everything else, position the cursor with an AT 4,0 command and then issue a CLS 2.

CODE (string$)

TEXT HANDLING FUNCTION

string$	A valid string or string variable

CODE is the opposite of CHR$; it takes a string or string variable and returns the Ascii code value of the first character in the string. If the string is empty, CODE returns 0. The Ascii code value of non-printable characters (such as TAB, Newline, Bell, etc.) are also correctly returned.

COMMAND_LINE

TURBO TOOLKIT
CONTROL COMMAND

COMMAND_LINE is a command that takes no parameters. Its purpose is to grasp the attention of the command line interpreter – the part of the QL which responds to your instructions as they are typed in. Because the interpreter is always alert to instructions when SuperBasic is running, COMMAND_LINE is only of use in compiled programs.

COMPILED

TURBO TOOLKIT
COMPILER FUNCTION

Several *Turbo* features, such as IMPLICIT, are only available when the program containing them has been compiled. COMPILED is a function which returns 1 if the program currently running is compiled and 0 if the program is being interpreted. It is best used as follows:

```
200 IF COMPILED
210 REMark Turbo-specific commands follow...
220 .....
500 ENDIF
```

CONNECT **#chan1 TO #chan2**

TURBO TOOLKIT
CHANNEL COMMAND

#chan1	A channel already opened as a pipeline
#chan2	A previously unopened but valid channel number

The CONNECT command is an extremely neat way of pipelining data from one task to another. Anything output from the first channel can be read as input by the second channel command. Before using CONNECT, open a channel as a pipeline with a command such as OPEN #7, pipe_500. The integer following the underscores establishes the length of the pipeline in bytes. Data can now be sent to the pipe using PRINT#7... commands. To read from the pipeline it must be connected to another channel, eg CONNECT #7 TO #5. Now everything PRINTed to channel #7 can be read again as input from channel #5.

If the second channel has a number higher than the highest previously used channel a spurious SuperBasic error is generated. Note also that if the pipe is used to send information from one task to another the CHANNEL_ID function (see separate entry) must be used to identify the correct Qdos channel number.

CONTINUE

CONTROL COMMAND

CONTINUE is a command that has no parameters. Its purpose is to resume the execution of a SuperBasic program on the line following the one being interpreted when processing stopped. Processing can stop due to an error or due to the user pressing the CTRL-SPACE combination (BREAK). SuperBasic also includes a RETRY command which resumes execution by repeating the last command to be executed prior to the interruption.

COPY **device1 TO device2** — SUPERBASIC
COPY_N **device 1 TO device2** — SUPERBASIC
COPY_O **device1 TO device2** — SUPER TOOLKIT II
COPY_H **device1 TO device 2** — SUPER TOOLKIT II

FILE/DATA COMMAND

device1	a filename or other data source
device2	a filename or other data destination (optional when used with Super TK2)

The most frequent use of the COPY command is to relocate or rename a file, eg COPY mdv1_filename TO flp 1_filename. However, Qdos treats all data-sending devices almost identically, so the source of the data could be a file, another QL on the network or another computer hooked up via the RS232 ports. Similarly, the destination can be any data-accepting device, such as the screen, a file or the printer. Files are different from other data entities in that they have a header which provides Qdos with information about their type, length, and so on.

If you are sending data to the screen the header information is automatically stripped off, but if you are copying a file to the printer it must be removed explicitly with the COPY_N version of the command.

Normal Qdos error trapping prevents you from using a destination filename which already exists, but the Super TK2 COPY_0 variant will overwrite an existing file of the same name without warning. In the right circumstances it can save a bit of processing, but if it is used thoughtlessly you might find that you have accidentally overwritten a valuable file.

COS (radians)

TRIGONOMETRY FUNCTION

radians	A variable or value representing an angle measured in radians.

Imagine a circle with a radius drawn to some point on its circumference. The input to the COS (cosine) function represents the distance between the top of the circle and the point where the radius touches the circumference. The output from the COS (cosine) function is the vertical distance between the end of the radius and the horizontal axis flowing through the centre of the circle. The direct distance between the end of the radius and the vertical axis is given by the SIN function, and so it is possible to draw a very accurate circle or ellipse using these two functions and the POINT command (which draws dots).

In the example below, the centre of the circle is located at 50,50 and the radius of the circle is 20. Try varying each of the absolute figures to see the effect they have. (Hint: changing only the SIN radius (eg from 20 to 40) will produce an ellipse.)

```
100 FOR dot = 0 TO 2*P1 STEP .02
110     POINT 50 + SIN(dot) * 20, 50 + COS(dot) * 20
120 ENDFOR dot
```

COT (radians)

TRIGONOMETRY FUNCTION

radians	A variable or value representing an angle measured in radians.

The cotangent function returns the ratio of sine to cosine for a given angle, thus COT(1) is a quick way of saying COS(1)/SIN(1).

CSIZE width, height

CHARACTER DISPLAY COMMAND

width	An integer value from 0 to 3
height	An integer value of 0 or 1

The QL was one of the first computers to offer the simultaneous display of different character sizes on-screen, although the options offered are quite limited. QL characters are based on grids 5 pixels wide and 9 pixels high. Allowing a one-pixel separation distance between each character the smallest permitted characters are thus 6 pixels by 10 pixels, achieved by typing CSIZE 0,0 when in the high-resolution monitor mode. Additional space between characters is added by specifying CSIZE 1,0. The characters can be doubled in width by specifying CSIZE 2,0 or CSIZE 3,0 and doubled in height by changing the second parameter to 1. Much more control over character size and font design is obtained by using *Super TK2* (or screen utilities like *Lightning).*

CURDIS #chan
CURSEN #chan
CURSOR_OFF#chan
CURSOR_ON #chan

SUPER TOOLKIT II
SUPER TOOLKIT II
TURBO TOOLKIT
TURBO TOOLKIT
SCREEN MANAGEMENT COMMAND

#chan	(optional) A valid screen channel – default #1

CURSEN, CURDIS CURSOR_ON and CURSOR_OFF take no parameters other than a valid screen channel; their role is to display and hide cursors. The QL's normal behaviour is to display only one cursor, usually a flashing character-sized block, no matter how many windows are open at the time. However, Qdos actually has a cursor for every open window, although they tend to be invisible when they are inactive. CURSEN and CURSOR_ON allow you to see the inactive cursors as solid, unblinking blocks. CURDIS and CURSOR_OFF hide them and make them inactive.

WARNING: It should be obvious that disabling the command window cursor (eg CURSOR_OFF #0) will disable SuperBasic itself because no further command input can be obtained. If you do this accidentally there is no cure except to reboot the computer, losing anything which has not previously been saved.

CURSOR #chan

x1, y1, x2, y2

#chan	(optional) A valid screen channel – default #1
x1	(optional) Horizontal graphics co-ordinate
y1	(optional) Vertical graphics co-ordinate
x2	Horizontal pixel co-ordinate
y2	Vertical pixel co-ordinate

The CURSOR command's behaviour can seem so mysterious that it is often ignored by programmers, which is a shame because it is extemely useful. It allows you to place text with pixel-point accuracy anywhere within a window. The command's parameters refer to the location of the top left pixel of the next character to be printed. Moreover, by using both the graphics and the pixel co-ordinate systems the CURSOR command lets you mix text and graphics freely in the same window.

In the default window, with no channel parameter, the CURSOR command works correctly on most QLs. If it is followed by one pair of co-ordinates SuperBasic treats them as references to the pixel-based co-ordinate system. If a second pair of co-ordinates are included the first pair are assumed to relate to an absolute position on the graphics grid and the second pair become an offset from the first location using the pixel grid.

If this seems complicated, think of the reasons why SuperBasic author Jan Jones came up with the command. Graphics can be drawn to any scale with an origin anywhere in the window by using the SCALE command. It is often the case that a graphics image such as a pie chart or technical drawing needs to be annotated with text. SuperBasic needed a way of making it easy to add text to graphics easily and flexibly, particularly so that a SCALE command would not dislocate picture and words. CURSOR allows you to select the exact point on a drawing against which some text is to be placed and then allows you to specify, in pixels, the offset from that point where the text should begin.

CURSOR begins to act up when a window reference is made, because a bug in many versions of the QL's rom permits only two or four parameters to be passed to the command. CURSOR #2, 40, 32, 36, 10 should be valid, but it has five parameters and is rejected by the interpreter. The answer is to omit the last parameter – the pixel-based vertical offset – and all is well. Minerva owners can disregard this workaround because the Minerva rom correctly looks for an even number of parameters disregarding any channel number.

As a quick demonstration of CURSOR's power, here is a routine to draw a clock face. Refer to COS() for an explanation of how the co-ordinates were calculated. Note that each number is printed x/12ths of the way around the circumference of a circle which measures 2*pi graphics units.

```
100 CLS
110 FOR X = 1 TO 12
120    LOCN = X/12 * 2*P1
130    CURSOR 50 + 30 * SIN(LOCN), 50 + 40 * COS(LOCN), 0,0
140     PRINT X
150 ENDFOR X
```

DATA a, b, c, d, ...

DATA INDICATOR

a,b,c, etc	An arbitrary list of variables or values

The DATA command is used in conjunction with READ and RESTORE as a means of obtaining input from within a program itself. A full explanation of these commands is provided under the entry for READ. DATA simply flags to the interpreter that what follows is a set of data values rather than an executable statement. There are no restrictions on separating DATA and other statements with colons on a single line, but it is more usual to keep DATA apart from executable program statements.

DATAD$
DATA_USE directory

SUPER TOOLKIT II
SUPER TOOLKIT II
DIRECTORY FUNCTION AND COMMAND

directory	A filename prefix ending in an underscore.

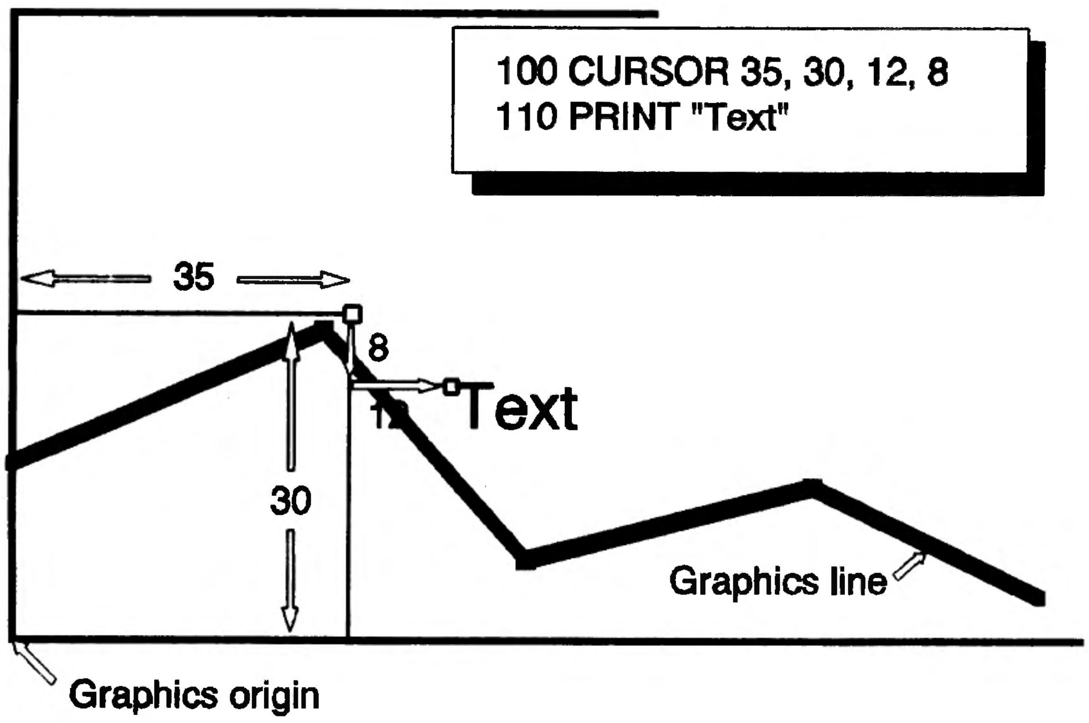
Super Toolkit II belatedly introduced the concept of directories, a means of grouping files together on a disk or microdrive. The idea is that a floppy disk typically contains scores of files which normally exist in one directory. Life would be easier if a bunch of related files could share the same prefix, but nobody likes the thought of having to type long filenames all the time.

Super Toolkit II solves this dilemma by automatically coping with filename prefixes, for instance by prefixing the names of all working files with "flp1_work_". Other data files could be prefixed with "flp2_temp_" or "mdv2_letters_" or whatever seems useful at the time. The prefix is set using the DATA_USE command, which is followed by a text string enclosed in quotes. The string should include the device, eg DATA_USE "flp 1_temp_". If the final underscore is omitted, Super Toolkit II adds it. The data prefix can temporarily be overridden simply by giving the full pathname, eg OPEN#5, "flp1_junk_data". Incidentally, the quotes around filenames are not obligatory but they are a good idea.

With DATA_USE in effect, a file created with the command OPEN_NEW#3, "filename" will actually be called "flp1_temp_filename", assuming that "flp1_temp_" is the current data prefix. The prefix currently in effect is revealed by the command PRINT DATAD$. Note that DATA_USE and DATAD$ are only used by data-related files: files containing SuperBasic and other executable programs have their own prefix.

Note that DATA_USE affects many file-related commands, such as DIR and DELETE.

Further explanations of directories and the Super Toolkit commands which implement them are provided under the individual commands, such as DDOWN, DUP, PROG_USE, and DEST_USE.



```
100 CURSOR 35, 30, 12, 8
110 PRINT "Text"
```

SINCLAIR QL WORLD

THE NEW USER GUIDE

SECTION THIRTEEN

KEYWORD INDEX

This month in the Keyword Index, Mike Lloyd moves from DATASPACE to DEVICE_STATUS. Not photogenic, but fascinating.

DATASPACE (file$)
[TURBO TOOLKIT]
DATA_AREA kbytes

MEMORY FUNCTION

file$	a valid file name
kbytes	an integer constant (<u>not</u> a variable)

When a SuperBasic program is compiled with *Supercharge* or *Turbo,* a set amount of memory is allocated for its personal data. The amount of dataspace can be adjusted from the compiler's control panel and should be made sufficient to hold all variables, arrays and scratch space without wasting any memory. Alternatively, the compiled task can contain a DATA_AREA command which changes the control panel setting. The DATA_AREA parameter represents whole numbers of kilobytes, so DATA_AREA 6 will allocate 6K of memory.

The DATASPACE function returns the number of bytes of memory a task is currently allocated. It returns standard error codes if the filename it uses is not found. Suppose a compiled program which needs 4K for its own variables is about to be loaded to handle a large, user-defined, numeric array. DATASPACE can be used to warn of insufficient dataspace in the following manner:

```
100 DEFine FuNction check dataspace (rows, columns, program$)
110 LOCal array_bytes
120 array_bytes = rows * columns * 7 + columns * 2
130 IF array_bytes + 4000 > DATASPACE (program$)
140   PRINT "Array is too large for dataspace"
150   RETurn 0: REMark "false""
160  ELSE
170   RETurn 1: REMark "true"
180 ENDIF
190 END DEFine
```

DATE
DATE$ (number)
DAY$ (number)
[Superbasic]
DATE (yr,mon,day, hr,min,sec)
[Minerva]

INTERNAL CLOCK FUNCTIONS

number	a large, whole number
yr	an integer year (eg 1961)
mon	an integer month in the range 1-12
day	an integer day in the range 1-31
hr	an integer hour in the range 0-23
min	an integer minute in the range 0-59
sec	an integer second in the range 0-59

For a computer unable to remember which day of the week it is from one working session to the next, the QL has quite a powerful set of time-related commands. The basic premise is that the QL's clock is counting the seconds between 1 January 1961 and 27 February 2097. This is fine for the average business application, but can be restrictive for historical and astronomical applications. Because numbers such as 9632736 are not immediately recognisable as '7 July 1992' SuperBasic includes a number of functions which translate between the internal representation of time and more normal conventions.

In SuperBasic, DATE is a function without parameters (although it may optionally take brackets to please the purists) which returns the QL's date in its internal, numeric representation, so that PRINT DATE might return 9.632736E8. QView, who developed Minerva, have added an additional facility so that DATE can take six parameters representing year, month, date, hour, minutes and seconds respectively, converting a known date and time into the QL's internal value.

To see the same date in a more understandable format the DATE$ function can be used. DATE$ will convert a numeric parameter into a date and time string such as '1961 Jan 01 09:00:05'. If no parameter is provided, the QL's internal date/time counter is converted into a string.

Programmers sometimes attempt, incorrectly, to slice the date/time string, forgetting that DATE$ is a function which expects a numeric parameter and is therefore thoroughly confused by instructions such as PRINT DATE$(6 TO 8). To extract a part of the string it must first be assigned to a variable. The following function shows how the month can be extracted from a date. The function takes a numeric parameter and returns a string.

```
100 DEFine FuNction GetMonth$ (dateval)
110 LOCal datetext$
120 datetext$ = DATE$ (dateval)
130 RETurn datetext$ (6 TO 8)
140 END DEFine
```

SuperBasic helps with the day of the week, which would otherwise be difficult to calculate, by providing a DAY$ function. DAY$ returns the day of the week represented by the QL's internal clock setting (if no parameter is given) or the day represented by a numeric parameter

DDOWN dirname

[Super Toolkit II]

	Directory Navigation
dirname	a valid filename extension

Super Toolkit II implements a multiple-level directory system which was dormant in the original release of Qdos. DDOWN is a command to move 'down' a subdirectory by appending dirname to an existing directory default. It only affects the prefix set by DATA_USE unless PROG_USE has been set to exactly the same prefix, thus 'binding' both the program and data directory pointers. The following commands and their associated remarks should clarify DDOWN's effect:

```
100 DEST_USE scr: DATA_USE flp 1_
110 COPY filename: REMark flp1_filename copied to screen
120 DDOWN work
130 COPY draft: REMark flp1_work_draft copied to screen
```

DEALLOCATE basebyte

[Turbo Toolkit]

	Memory Command
basebyte	A valid, even memory address

Areas of the QL's common heap of unallocated memory can be reserved with functions such as RESPR which return the address of the first byte of memory to be reserved. Hold on to this value, because without it memory cannot be deallocated. Turbo Toolkit includes the command DEALLOCATE, which must be followed by the address of the first byte, to return a reserved memory area back to the common heap. Avoid memory fragmentation, where allocated and free chunks of memory are interspersed with each other, by releasing memory in the opposite order in which it was allocated.

DEFAULT_DEVICE

drive
[Turbo Toolkit]

	Device Command
drive	A drive prefix with final underscore

Turbo Toolkit does not acknowledge the Qdos directory hierarchy but instead allows programmers to declare a default device. This means that a command such as OPEN filename can be issued without error, provided that a command such as DEFAULT_DEVICE flp1_ has been issued earlier.

DEFine FuNction

fname (param1, param2, ...)

DEFine PROCedure

pname (param, param2, ...)
[User-Defined Keywords]

fname/pname	A unique, non-reserved work
param	(optional) Parameter names

When SuperBasic was first introduced the DEFine FuNction and DEFine PROCedure constructs justified to a large degree the 'Super' prefix. Most of SuperBasic's predecessors, and even some of its successors, did not permit users to extend the language in any meaningful way. Where other Basic dialects, such as BBC Basic, allowed procedures to be defined they were not as complete as SuperBasic.

A user-defined procedure or function can do anything which can be defined in terms of existing SuperBasic keywords. Once defined, it can be used by other user-defined structures. If a program has a need for text to be centred in a window, SuperBasic has no command of its own to perform the task, but it does allow programmers to develop one:

```
100 DEFine PROCedure CPRINT (text$, windowwidth)
110 REMark windowwidth gives window width in characters
130 IF LEN(text$) >= windowwidth
140    PRINT text$
150 ELSE
180    PRINT TO (windowwidth-LEN(text$))/2; text$
190 ENDIF
200 END DEFine
```

Functions differ from procedures in that they return a value of some sort and that their names must agree with the type of value returned: the name of a function which returns a string must end in a $.
SuperBasic does not include a function which returns the sign of a value, but a user-defined function can be written which does so:

```
200 DEFine FuNction SIGNUM (value)
210 IF value > 0: RETurn 1
220 IF value < 0: RETurn -1
230 RETurn 0
240 END DEFine
```

Functions and procedures in SuperBasic are fully recursive in that they can call themselves either directly or via a subordinate procedure or function. Although clever and concise code can often be written in this way it is usually preferable to avoid recursion because of the overheads it imposes on the interpreter and the ease with which undesired results can be obtained. However, to show an easy recursive solution to a problem, here is a procedure to print out each word of a long string on a separate line:

```
300 DEFine PROCedure Wordlist (text$)
310 split = " " INSTR text$
320 IF split = 0
330    PRINT text$
340    RETURN
350 ELSE
360    PRINT text$ (1 TO split)
370    Wordlist text$ (split+1 TO)
380 ENDIF
390 END DEFine
```

The basic structure for a user-defined procedure or function is an opening line declaring the name of the structure and any parameters it may have. Definitions do not need to have parameters if they are not wanted. Parameter lists are separated by commas. Note that both procedures and functions require brackets when the definition is called. There then follows a body of lines which define the actions carried out by the user-defined structure. The final line is END DEFine. SuperBasic allows the body of a procedure or function to contain another user-defined segment, but it would be loose programming practice to take advantage of it. Similarly, SuperBasic does not object if a GOTO or GOSUB within a definition jumps to some code outside the definition boundaries, but it would be an incautious programmer who thought that this was a worthwhile activity.
An unusual feature of user-defined parameters is that their names do not have to conform to the normal SuperBasic naming rules. Thus, although the above example uses text$ for its parameter, text would have done equally well. Unless a parameter can represent either a string or a number it is best to stick to the normal naming conventions.
The name given to a definition becomes in effect a new SuperBasic keyword, and can be used as such, as shown in the following examples:

```
500 CPRINT "hello world", 32
550 X = Y * SIGNUM(Z)
600 PRINT userfunction (15, 20)
```

DEG(radians)
[SuperBasic]

	Trigonometry Function
radians	A numeric value

In line with most computers, the QL is happier dealing with radians than with degrees. However, people are usually more familiar with degrees and so conversions between the two are part of the QL's trigonometry suite. DEG converts an angle expressed in radians into its equivalent in degrees.
The ratio between the radius of a circle and its circumference is slightly more than 6:1. In other words, six and a bit lines each the length of a radius would be needed to cover a circle's circumference. This ratio is better expressed as 2 *pi. It should be obvious that DEG (pi) will return 180 and DEG(2 * pi) will produce 360.

DELETE filename
[SuperBasic]

	FILE COMMAND
filename	A valid filename

DELETE is a straightforward command used to remove a file from a directory. Provided that it is followed by a technically valid filename it will issue no warnings or error messages, even if the file does not exist. QL users usually find out to their regret that computers have absolutely no idea about the value of a file, and will blindly delete a large, unique and essential file with as little fuss as a small, inconsequential one.

Fortunately, the DELETE keyword only affects the reference to the file in the disk or microdrive directory, which allows a file repair utility to recover a deleted file provided that its contents have not been overwritten by subsequent saves. A couple of years ago a drug dealer was successfully prosecuted and imprisoned because he believed that DELETE removed all evidence of his database: the police recreated the directory as it was before the delete was carried out and recovered all the incriminating information from the computer.

DEL_DEFB
[Superb Toolkit II]

MEMORY MANAGEMENT

DEL_DEFB takes no parameters. It provides an effective cure for a form of common heap fragmentation. The QL copies directory contents into the common heap area of its memory in order to reduce the access time to reach files. Unfortunately, this habit mixed with user-sponsored memory allocations can fragment the memory so that large chunks of it become temporarily unusable. DEL_DEFB removes all file definition blocks from the common heap. It should not be used when channels are opened to any disk or microdrive.

DESTD$
DEST_USE device
[Super Toolkit II]

DEVICE MANAGEMENT FUNCTION

device	A valid Qdos device or file name

The DESTD$ function reports which device or directory is currently the default destination device. In most cases, the default will be the screen, but it can be changed to the printer, network or a file. The destination device is set with the DEST_USE command.

DEVICE_SPACE (chan)
[Turbo Toolkit]

DEVICE MANAGEMENT FUNCTION

chan	An open channel (leading # is optional)

Before sending data to a file it is convenient to note beforehand whether there is likely to be enough space available to hold it. The size to which a file can grow is determined by the room left on the device which holds it. DEVICE_SPACE assumes that you have opened a channel to a file on a disk or microdrive. It returns the number of bytes left on that device. If the channel refers to a non-storage device such as the network, the screen or the printer DEVICE_STATUS returns a very large number. See DEVICE_STATUS below for a more useful method of obtaining this information.

DEVICE_STATUS
(type,file)
[Turbo Toolkit]

DEVICE MANAGEMENT FUNCTION

type (optional)	an integer representing access type
file	a valid filename

A major limitation to SuperBasic is its lack of error-recovery facilities. All of the QL roms had incomplete WHEN ERROR code which was subsequently completed by Super Toolkit II and by the Minerva roms. However, Digital Precision's compilers needed to have version-independent error-trapping code. Most recoverable errors relate to file access, and so the DEVICE_STATUS function was created. You can provide the filename and, optionally, the sort of access you require to the function and its code will test the device to determine if your intentions can be fulfilled. Any positive return value indicates that you can go ahead, and incidentally reveals the amount of available space on the device. A negative return value can be interpreted as a standard error code.

The access codes are as follows:

0	OPEN
1	OPEN_IN
2	OPEN_NEW
-1	OPEN or OPEN_NEW

The possible error values returned by DEVICE_STATUS are:

-3	Insufficient memory
-6	Insufficient memory
-7	No such device or file
-9	Device busy
-11	Device full
-12	Device is valid, but the filename is illegal
-16	The infamous 'bad or changed medium'
-20	Device is write-protected or being read

SINCLAIR QL WORLD

THE NEW USER GUIDE

SECTION FOURTEEN

KEYWORD INDEX

This month in the Keyword Index, Mike Lloyd starts with DIM array and ends with ED #chan, line – something to channel your thoughts

DIM array(x,y,z...),

array$(x,y,z),...

MEMORY VARIABLE COMMAND

x, y and z	Integers between 0 and 32767 representing the number of elements in a given dimension
(x,y,z)	A list containing at least one array dimension
array	User-defined name of a numeric array
array$	User-defined name of a string array

Arrays are extremely useful because they allow you to associate similar items of data. For instance, in a program for use by a school secretary there might be arrays called PUPIL$, CLASSROOM and TEACHER$. With 12 classes comprising at most 30 pupils, and with 15 teachers and 21 classrooms, the framework of a database could be established with the following command:

```
100             DIM pupil$(12, 30, 20), teacher$(15, 20). classroom(21)
```

Note that the last dimension of string arrays refer to the maximum length of the element. Pupil$(12, 30, 20) indicates that there are 12 lists (one for each school class) with 30 places (one for each classroom place) each of 20 characters (the maximum length of a pupil's name).

SuperBasic arrays have in practice no limit to the number of dimensions they can have, provided that sufficient memory exists to accommodate them. A one-dimensional array can be thought of as a list, and a two-dimensional array is like a table of columns and rows. Arrays with three dimensions are like a series of tables and arrays with four or more dimensions begin to stretch the imagination too far and are thus rarely used.

In line with earlier Sinclair products, the QL has a neat but novel way with string arrays. A string array with one dimension, eg DIM string$(12), is logically identical to an ordinary string variable which happens to be limited to 12 characters. Digital Precision's *Turbo* compiler treats all strings as if they were one-dimensional string arrays with a default maximum length of 100 characters. Longer strings are declared as one-dimensional arrays with the appropriate number of character elements.

The QL initially allocates enough memory space only to hold the pointers needed to navigate through each array. Space for the data itself is only added when an array element is assigned a value. The QL is still quite advanced in its approach as most computer languages seem to prefer allocating huge chunks of memory to arrays whether they contain data or not.

With string arrays the QL remains frugal by allocating memory based on the actual contents of the string, not on the declared maximum length. However, the QL always allocates an even number of bytes to a string, so a declaration of DIM oddstring$(11,15) will actually produce an array which has 11 elements each holding a maximum of 16 characters. The QL's interpreter is prone to a very irritating error message when a program line declaring an array has been written but not yet run. Normally, if you use a variable name which the QL does not recognise it prints a friendly asterisk (or 'twinkle', as my programming friends now insist on calling it). If the undeclared variable name has been used in a DIM statement which hasn't yet been executed you will get an 'Error in expression' message instead. This can lead to lengthy and fruitless agonising over the syntax of a perfectly valid command.

Two problems immediately confront programmers new to arrays: sorting them into order and saving them for later use. Sorting algorithms were honed to a high standard by the mid-1960s with two significant developments. The first was the 'quicksort', an elegant, recursive sorting technique which has been published at least twice in *QL World*. The second was the idea of indexing cumbersome strings so that they never change their physical sequence, but are accessed via a second array of numeric pointers which is easier and faster to re-arrange.

Saving arrays is simple due to the unifying principles of Qdos: an array is just a type of variable and a file is just a type of device. An array can be printed in its entirety with the command PRINT arrayname$. Instead of printing to the screen or printer, the same information can be passed to a file by opening an appropriate channel:

```
100          OPEN_NEW#3, mdv1_arraystore
110          PRINT#3, myarray$
120          CLOSE#3
```

The problem comes, unfortunately, when it is time to reload the array. You have the data in a file and you have the array details in a program. How can they be married together again? Your program should have a sequence like this:

```
500          DEFine PROCedure loadarray
510                DIM myarray$(30,20)
520                OPEN_IN#4, mdv1_arraystore
530                FOR x = 1 TO 30
540                      INPUT#4, myarray(x)
550                END FOR
560                CLOSE#4
570          END DEFine
```

DIMN (arrayname,dimension)

	MEMORY VARIABLE FUNCTION
arrayname	The name of a previously-declared array
dimension	The dimension about which statistics are sought

DIMN returns the number of elements present in an array, or the number of rows in a column of a multi-dimensioned array. In the entry for the DIM keyword an array of PUPIL$(12, 30, 20) was declared. DIMN can be used in a program to reveal the dimensions of the array in the following manner:

```
100          PRINT DIMN(pupil$, 1); " classes with ..."
110          PRINT DIMN(pupil$, 2); " pupils with ..."
120          PRINT DIMN(pupil$, 3); " characters in their names"
```

DIMN comes in particularly handy when arrays are declared at different times with different sizes.

DIR #chan, device

	FILE HANDLING COMMAND
#chan	(Optional) a valid SuperBasic channel
device	A valid media device

DIR is short for 'Directory' and it causes a simple list of files to be printed to the default window or to the given channel. The output begins with the name of the microdrive or floppy disk and an expression of the number of used sectors and the total number of sectors. Each sector holds 512 bytes of information. *Super Toolkit II* slightly modifies the behaviour of DIR so that it can take advantage of Toolkit II's default directories.

The simplest way of trapping directory information so that its presentation can be improved is to divert the output to a file. The following routine is called with a command such as SHOWDOCS "flp1_". It will list only Quill files, assuming that Quill filenames end with the suffix _doc:

```
500    DEFine PROCedure showdocs (drive$)
510      LOCal DIRfile$, loop, filename$
520      DIRfile$ = drive$ & "dirfile"
530      DELETE DIRfile$
540      OPEN_NEW#3, DIRfile$
550      DIR#3: CLOSE#3
560      OPEN_IN#3, DIRfile$
570      REPeat loop
580        IF EOF (#3): EXIT loop
590        INPUT#3, filename$
600        IF "_doc" INSTR filename$
610          PRINT filename$
620        ENDIF
630      END REPeat loop
640      CLOSE#3
650    END DEFine
```

All this effort can be avoided with Super Toolkit II. It contains a WIDR command which allows you to specify any part of a filename which must be matched.

number1 DIV number2

INTEGER OPERATOR

number1, number2 Valid numbers

DIV is not strictly speaking a keyword but an operator. In other words it is from the same family as +, -, * and /, but it happens to be an operator in the form of a set of letters rather than a symbol. Other similar operators are INSTR and MOD. DIV performs an integer divide, throwing any remainder away. None of the variables involved needs to be declared as an integer, so LET X = 8.9 DIV fraction is quite valid. If the result was being placed in X% instead, DIV would perform exactly the same as "/". The main benefit of DIV is that it can be used with floating numbers which can assume much higher values than the QL's integers.

DLINE, line, line TO line, ...

PROGRAM EDITING COMMAND

line A positive integer representing a line number

When SuperBasic reserved the keyword DELETE for file handling, something else had to be found to delete program lines. The clumsy DLINE was therefore coined. DLINE is normally followed by only one parameter, either a single line number or a range of lines, but any number of parameters can be given provided that they are separated by commas. Super Toolkit II includes a full-screen editor which does away with the need for DLINE altogether. Its equivalent is to position the cursor on the offending line and pressing CTRL-ALT-left arrow.

DLIST #chan

[Super Toolkit II]

FILE MANAGEMENT COMMAND

#chan (Optional) A valid SuperBasic channel

Super Toolkit II allows you to declare default directories for program files, data files and destination files, PROGD$, DATAD$ and DESTD$ respectively. Rather than indulging in a set of PRINT PROGD$ commands to remind you of what the settings are, Super Toolkit II includes the handy DLIST command which puts all three settings onto the screen, or to a given channel, automatically. Note that, like SuperBasic's DIR, DLIST is more use at the direct command level than in programs.

DNEXT dir_level

[Super Toolkit II]

FILE MANAGEMENT COMMAND

dir_level	A fragment of a directory name

Super Toolkit II's soft directory structures are implemented with some interesting commands, of which this is one. A directory is defined as a set of files which each begin (or end) with the same sequence of filename fragments. Filename fragments are separated by underscores. In the filename flp1_work_temp_SQLW_index_doc the directory fragments are _work, _temp, and _SQLW. The device name fragment is flp1_, the filename fragment is index and the filename suffix is _doc. Super Toolkit II has commands to move up and down the directory levels and it has DNEXT, which moves sideways. Assume that the default data directory has been set so that we can access the index_doc file directly (DATA_USE flp1_work_temp_SQLW_) and that we now want to use files prefixed with flp1_work_temp_HOME_. Rather than issue another DATA_USE command we can simply say DNEXT home. For readers familiar with the DOS and UNIX directory structures, DNEXT is the equivalent of ..\HOME.

DO filename

[Super Toolkit II]

PROGRAM COMMAND

filename	a valid filename

DO is a lovely addition to SuperBasic because it executes files of commands. Unlike the SuperBasic LRUN command, DO only works with commands which do not have line numbers. This means that the DO file cannot include any multi-line structures such as procedure definitions and the long forms of IFs, REPeats and FOR...NEXTs. However, Tony Tebby believes that including calls to SuperBasic procedures and functions within the DO file causes no problems, but of course it must be assumed that the definitions are part of a SuperBasic program which is already in the QL's memory. DO properly closes the command file when it reaches the end, whereas SuperBasic's MERGE command leaves it open.

If you have a sequence of commands which you frequently call from several programs, it would be worth considering saving them to a text file and calling that file with DO. When writing programs I like to rearrange the default windows so that the listing window occupies the entire left hand side of the screen. A narrow but long command window and a very small default window are relegated to the right hand side of the screen. The commands which lay the screen out in this way are in a file called "edit_screen" so that they can be carried out with the direct command DO edit_screen.

A further advantage of this technique is that once the commands have been carried out they are removed from the QL's memory, thus preserving more space for data and for those program lines which cannot be relegated to files.

DUP

[Super Toolkit II]

FILE MANAGEMENT COMMAND

DUP is another of Super Toolkit II's directory commands. For a brief discussion of the directory system, see the entry for DNEXT above. DUP has no parameters. It removes the last directory segment from the default directory name, thus having the effect of moving up one level of the directory tree. If the default directory is flp1_work_temp_home_, issuing the DUP command changes it to flp1_work_temp_.

ED #chan, line

[Super Toolkit II]

PROGRAM EDITOR

#chan	a valid screen (console) channel
line	an integer representing a line number

Perhaps the biggest disappointment on the QL, even taking into account microdrives, is the crude program editor. Tony Tebby quickly developed a full-screen editor, invoked by the ED command, which is now incorporated into Super Toolkit II. The irony of it is that even the earliest QL ROMs have more than enough space to include a full-screen editor, but the decision to do without one was taken during the fruitless attempts to cram all the ROM code into 32K.

A full description of ED belongs in the Concepts section of the *New User Guide*. Suffice to say that using ED instead of EDIT, DLINE and AUTO is alone worth the cost of buying Super Toolkit II.

THE NEW USER GUIDE

SECTION FIFTEEN

KEYWORD INDEX

This month in the Keyword Index, Mike Lloyd starts with ED line_number and cuts off this instalment with the EXECUTE family.

ED line_number,
increment

SUPERBASIC LINE EDITOR

line_number	(optional) a legitimate line number, default 100
increment	(optional) increment between succeeding line numbers

The QL's SuperBasic line editor was universally mocked when the QL was launched, but it stands comparison with the line editors which afflicted MS-DOS and Unix at the time. All three have since been replaced with more user-friendly offerings.

EDIT only edits SuperBasic program lines: it cannot be used to edit text files. The command assumes you wish to edit line 100 unless you specify otherwise. The program line is brought into the common window for editing an can be navigated using the left and right cursor keys. CTRL-left and CTRL-right delete characters to the left and right of the cursor.

With the Minerva rom fitted, ALT-left and ALT-right move to the beginning and end of the line, TAB and SHIFT-TAB move through the line in eight-character hops, and CTRL-ALT-left/right deletes from the cursor to the beginning or end of the line. Minerva also corrects the annoying bug in Qdos which traps SHIFT-SPACE as an illegal key combination.

Previous and subsequent lines can be edited in turn by pressing the up and down arrow keys. Without a second parameter, EDIT completes its job as soon as ENTER is pressed. However, if an increment of x is specified as the second parameter, the line number x greater than the present line is fetched for editing.

Even with Minerva's improvements. most programmers prefer working with *Super Toolkit's* ED (see last month's Index).

EDIT$ (#chan, default$, chars)
[Turbo Toolkit]
EDIT% (#chan, default, digits)
[Turbo Toolkit]
EDITF (#chan, default, digits)
[Turbo Toolkit]

INPUT EDITING FUNCTIONS

#chan	(optional) console channel number
default($)	default input value
chars/digits	(optional) maximum length of input

All Basic dialects suffer from the inadequate specification of the INPUT command: there is usually no error trapping and no method of limiting the amount of input accepted from the user. *Turbo Toolkit* includes three functions which rectify these weaknesses. As with INPUT, EDITx can be directed

towards any console (such as a screen window opened for input and output). The next parameter must be included, although it can be a null string: it defines the default input value.
The final, optional, value determines the maximum number of characters which can be entered.

In use, the input area is positioned using AT in the normal way. The default value is displayed on the screen either to be accepted unaltered or edited by the user. Editing within the input area is the same as for INPUT, except that any attempt to exceed the maximum input length is met by a warning beep.

Within a program, the EDITx functions return a value to a variable within an expression such as 100 username = EDIT$(1, "Your Name", 10). Each of the EDITx variants ensures that only valid input is returned to the left hand side of the expression, reducing considerably the amount of error trapping which would otherwise be necessary.

ELLIPSE

chan, xpos, ypos, radius, eccentri city, angle

ELLIPSE_R

chan, xpos, ypos, radius, eccentri city, angle

GRAPHICS COMMANDS

The ELLIPSE command is absolutely identical to the SuperBasic CIRCLE command. The general understanding is that the QL's design team could not make a decision over using ELLIPSE or CIRCLE as a keyword and the computer ended up with both.

END DEFine

procname

STRUCTURE DEFINITION

Every user-defined procedure and function must end with an END DEFine line. The keywords may optionally be followed by the structure's name. The requirement for an explicit END statement for structures permits sloppy programmers to insert definitions randomly throughout programs and even to nest them. In orderly programs structure definitions appear consecutively at the end of the main body.

END_CMD

[Turbo Toolkit]

COMMAND FILE TERMINATOR

It has long been known that the QL's concept of associating logical channels with arbitrary physical devices can be widely exploited. In particular, a file of commands without line numbers can be executed directly by LOADing or MERGEing the file. The advantages of doing so are increased execution speed and the immediate removal of the commands from memory once they have been carried out.

However, due to a quirk of Qdos the file then remains open for that session. END_CMD is Turbo Toolkit's way of ensuring that such file numbers are closed correctly. To make use of it, write a file of sequential SuperBasic commands (ie without multi-line structures such as long FOR..NEXT loops) which do not have line numbers. Add END_CMD to the end of the file and close it in the normal way. Use the MERGE command to execute the file. However, be aware that any lines commencing with a line number will be appended to any program in memory and will not be executed directly.

END_WHEN

[Turbo Toolkit]

STRUCTURE TERMINATOR

All QL roms reserve the keyword WHEN, but only the most recent make use of the keyword to implement error-trapping and interrupt-driven facilities. Unfortunately, Sinclair Research never completed the code and it is dangerous to rely on it. Minerva roms and Super Toolkit correct the code, but if you are writing software for other QL users you cannot guarantee that they will have either Minerva or Super Toolkit: in these circumstances, the best bet is to use Turbo Toolkit and the Turbo compiler, which implements error trapping on all QLs. The END_WHEN statement is used to indicate the end of a WHEN structure containing code carried out in the event of an error.

EOF(#chan)

#chan	(optional) a channel capable of inputting data

DATA FUNCTION

The EOF() function determines if the End Of File has been reached. without a channel number, EOF returns 1 (representing True) if there are no further DATA items embedded in the current SuperBasic program. If unread DATA items exist, EOF returns a 0, representing False. More conventionally, EOF with a channel number returns the same values according to whether there is more data to be read in the file associated with the channel.

ERLIN()
ERNUM()
ERLIN%()
[Turbo Toolkit]
ERNUM%()
[Turbo Toolkit]

ERROR REPORTING FUNCTIONS

When the SuperBasic interpreter encounters an error it reports the error message and line number; Minerva owners have the incorrect statement on the line identified as well. The user/programmer can then take corrective action and resume processing. When run-time errors occur, such as the absence of a microdrive file, programs should be able to take appropriate action without halting with an error message and a flashing cursor. For debugging purposes, the line number on which the error occurred gives a useful indication of where corrective action needs to be applied. Late model QLs include the ERLIN() function which returns the line number at fault. More importantly, the type of error can also be identified and returned using the ERNUM() function. Neither of these functions takes a parameter. The obvious place to use these keywords is in a WHEN ERROR clause. The whole error-trapping suite is incomplete on QL roms, but is made workable by the addition of the Minerva rom or Super Toolkit II.

The developers of Turbo Toolkit were able to implement full error-trapping across all QLs by including their own code to replace that which may or may not be in the host machine's rom. Cleverly, they simply added the integer suffix to the existing SuperBasic reserved keywords to identify the Turbo variants. However, this means that programmers have to be alert to the possibility that on JM and JS QLs the accidental omission of the suffix will not cause any problems, but would be fatal on earlier QLs.

All error conditions are identified by a negative integer, listed in the standard *User Guide* in the Concepts section under 'error handling'. Additionally, late QLs contain functions (or, more strictly, system constants) which represent these values so that programmers who cannot automatically associate Error -5 with a buffer overflow might be able to bring ERR BO to mind instead. A full list of the error constants is listed in the accompanying panel.

QDOS ERROR MESSAGE CODES

-1	ERR_NC	Not complete	
-2	ERR_NJ	Invalid Job	
-3	ERR_OM	Out of Memory	
-4	ERR_OR	Out of range	
-5	ERR_BO	Buffer Full	
-6	ERR_NO	Channel not open	💾
-7	ERR_NF	Not found	💾
-8	ERR_EX	Already exists	💾
-9	ERR_IU	In use	💾
-10	ERR_EF	End of file	💾
-11	ERR_DF	Drive full	💾
-12	ERR_BN	Bad name	💾
-13	ERR_TE	Xmit error	
-14	ERR_FF	Format failed	💾
-15	ERR_BP	Bad parameter	
-16	ERR_FE	Bad or changed medium	💾
-17	ERR_XP	Error in expression	
-18	ERR_OV	Overflow	
-19	ERR_NI	Not Implemented	
-20	ERR_RO	Read only	💾
-21	ERR_BL	Bad line	
-22		PROC/FN cleared	

💾 - file-related error message

ET program, pipes; parameter$
[Super Toolkit II]
EX program, pipes; parameter$
[Super Toolkit II]
EW program, pipes; parameter$
[Super Toolkit II]

program	a valid executable file's name
pipes	(optional) a comma-separated list which can comprise channels, files and other executables.
parameter$	(optional) a string passed to the first executable file.

PROGRAM EXECUTION COMMANDS

The first thing to note about the commands listed above is that all the complexity is for the benefit of machine code programmers. They all perform a very similar task, that of launching executable programs. Executable programs include the Psion quartet, the vast majority of other commercial software, and any programs compiled with Turbo. At their simplest they are efficient substitutes for the SuperBasic EXC and EXEC_W commands (see below), in the sense that EX and EW are easier to type than the loner SuperBasic equivalents. EX launches an executable and promptly returns to the calling program, usually SuperBasic, and is thus of most benefit when true multitasking is required. EW launches an executable and suspends all other programs launched in the same way until the executable is exited. ET is a variant which allows a debugger to trace program execution.

In their fuller forms, EX and EW allow programmers to pass a single string parameter to the program being called, perhaps to provide details of a filename or the time and date which can only be determined at run-time. Additionally, SuperBasic takes significant steps towards the Unix concept of 'pipes': sequences of small single-purpose utilities between which a data stream can be routed through temporary channels. An example might be:

EX flpl_strip TO flpl_caps TO flpl_more; "flp2_ quill_doc"

This command is designed to pass the contents of the file flp2_quill_doc to three programs in turn: strip (which removes all Quill control codes from the file), caps (which puts all text into upper case), and more (which puts text onto the screen a pageful at a time). Note that the TO keyword replaces commas to aid readability. The three executable programs would be written in machine code, C or compiled SuperBasic. Interpreted SuperBasic will not do.

EXEC program
EXEC_W program

PROGRAM EXECUTION COMMANDS

program	A valid executable file's name

Qdos is a multitasking operating system which permits more than one executable program to run at a given time. The command EXEC can be followed by the name of an executable file to add another program to those already running. Executable files are by convention distinguished by a _BIN or _EXE suffix. In this context, the SuperBasic interpreter is simply an executable program with one or two special privileges: it does not have to be explicitly launched and it cannot be removed.

Where programs fight for control of resources such as the screen and the printer, uncontrolled multitasking can be a nightmare. QL owners who have accidentally launched Quill with the EXEC command will know exactly what I mean. EXEC_W effectively launches an application in 'single-tasking' mode because the newly-launched program is offered all of the cpu's available time. Control does not return to the calling program (which is usually SuperBasic) until the executable program is finished.

Many boot programs make use of EXEC_W by establishing a loop containing a simple menu, an input statement and an EXEC_W command. The menu displays the executable programs available on the disk or microdrive, the input statement collects the user's choice and the EXEC_ W command launches the appropriate file. When the executable is exited, the loop cycles to the beginning and shows the menu options again.

EXECUTE program
[All Turbo Toolkit]
EXECUTE program;
parameter$! priority
EXECUTE program1
TO program2 TO program3
EXECUTE #chan1 TO
program1 TO filename
EXECUTE_A
(as above)
[Turbo Toolkit]
EXECUTE W
(as above)
[Turbo Toolkit]

program	a valid executable file's name
#chan	a valid SuperBasic channel (interchangeable with a filename)
filename	a destination data file (interchangeable with a channel)
priority	(optional) an integer indicating how great a percentage of CPU time a program is given compared with other multi-tasking programs.
parameter$	(optional) a string passed to the called program

PROGRAM EXECUTION COMMANDS

The EXECUTE family of commands are Turbo Toolkit's equivalents to Super Toolkit's EX and EW commands. Using compiled programs, a SuperBasic programmer can make full use of pipes to link tasks, channels and files. The EXECUTE_A variant scans the keyboard for an Alt-Space keypress, at which point execution is aborted. The EXECUTE_W variant waits for a task to complete before control return to the calling task.

Turbo Toolkit's output pipe is automatically attached to the highest valid channel number, limited to #15 in a compiled program. The input pipe for the receiving program is connected to the next highest channel, normally #14. These channels do not need to be explicitly opened prior to sending and receiving a data stream.

The EXECUTE family allows the programmer to specify how much of the CPU's time should be allocated to an executable program when running simultaneously with other tasks. Values lower than 32 are recommended. Programs may each have a string parameter passed to it which can be retrieved using the OPTION_CMD$() function. The syntax is sufficiently flexible to allow these two options to be specified in any order.

THE NEW USER GUIDE

SECTION EIGHTEEN

KEYWORD INDEX

This month in the Keyword Index, Mike Lloyd starts with EXIT, and concludes with FOR indent. This section should follow section Fifteen (June 1992 issue).

EXIT control

STRUCTURE COMMAND
control a numeric variable controlling a structure

FOR...NEXT loops have a built-in escape route because the syntax insists on a finite number of repetitions before the rest of the program is carried out, but it is always nice to be able to leave a loop prematurely. REPeat loops left to their own devices will last forever and so an explicit exit condition is essential. In both circumstances the EXIT keyword, followed by the variable controlling the loop (called the identifier by Sinclair), will cause the interpreter to skip immediately to the line following the appropriate END structure command. See the entry for FOR...NEXT for an explanation of the distinction between END FOR and NEXT in this context.

The nice thing about EXIT followed by the loop control variable is that it allows you to escape with one bound all of the layers of a multi-nested loop. Programming purists might object to this approach, but the SuperBasic interpreter does not.

EXP(n)

MATHEMATICS FUNCTION
n a number between -500 and 500

EXP, short for exponent, is followed by a value which is taken to be the exponent of the transcendental number e, or 2.718282... One of the many interesting properties of e is that when it is raised to the power of pi multiplied by the square root of -1, the result is -1. Sadly, this is of little use to QL programmers, as the square root of -1 is an unreal number outside the range used by the SQR() function. The value of e is the basis for natural logarithms.

EXTERNAL module%, "group", specification
[Turbo Toolkit]

COMPILER DIRECTIVE
module% an integer constant
"group" a string constant

specification (optional comma-separated repetition) a variable or an array, or a function or procedure call, or the word FUNCTION or PROCEDURE.

The Turbo compiler permits programmers to link together program snippets, so combining frequently-used utility routines into many different programs. This faciilty is not available in interpreted SuperBasic and so is only briefly dealt with here. EXTERNAL and GLOBAL establish links between the program calling a routine and the program containing the routine respectively. Where the specification refers to a procedure or a function the name of the structure is preceded by the keyword PROCEDURE or FUNCTION (spelt out in tin full and followed by a comma),. Arrays are declared with dummy dimensions, such as STRING$(0,0). GLOBAL directives can be split across several lines, each identified by a string constant. If an EXTERNAL call refers to a string constant, it only needs to include the items listed on the given GLOBAL statement.

EXTRAS #chan
[Super Toolkit II]

MEMORY COMMAND
#chan (Optional) A valid output channel.

EXTRAS is a valuable means of finding out what extensions to the SuperBasic language have been added. The list includes, in the order in which they were declared, all additional keywords provided by Super Toolkit II and any other software. There may be a few surprises, such as those which remain when Professional Publisher is exited, and even keywords which you had forgotten about. People are usually astonished simply by how many there are.

FDAT(#chan)
[Super Toolkit II]

PRINT UTILITY FUNCTION
amount an amount to be converted to a string
width the number of characters in the string
decimals the number of decimal places to include

The FDEC$ function replaces yards of code by converting a number to a string of fixed length with a given number of decimal places. Immediately, all the problems of scientific notation are avoided and the difficulties of lining up columns of numbers are resolved. Anyone who has struggled with a way round printing two pence as 0.02 will be delighted!

FEXP$ (amount, width, decimals)
[Super Toolkit II]

PRINT UTILITY FUNCTION
amount an amount to be converted to a string
width the number of characters in the string
decimals the number of decimal places to include

This function prints a fixed-length string giving a value in exponential form (eg " 1.786E=02" or " 3.21E.02"). The string should have at least seven more characters than the specified number of decimal places.

FILL #chan, toggle

SCREEN GRAPHICS COMMAND
#chan (Optional) a screen channel
toggle 1 or 0 representing on and off

The FILL command ensures that any sequence of lines which bounds an area will cause that area to be filled with the current ink colour. The algorithm used is much less sophisticated than that, say, of Digital Precision's Eye-Q drawing and so it is a bit wobbly on re-entrant shapes. A bug can also leave odd lines unfilled in circles. With FILL on and OVER set to -1, a succession of circles causes the leaking ink to form a psychedelic experience all over the screen. To avoid this phenomenon place a FILL 1 command between each shape you draw, or buy the Minerva rom: it corrects the bug and makes life dull again.

FILL$(char$, width)

STRING FUNCTION
char$ a string with one or two characters
width the number of characters returned

FILL$ tackles the problem of declaring long strings of identical characters, such as 80 spaces or even 132 underlines. The first argument is a one- or two-character string which is repeated to form a string with as many characters as that for repetitions of less than around a dozen characters it is more efficient to type them out longhand than use the FILL$ function. FILL$ is unusual because it both takes and returns a string.

FLASH #chan, toggle

TEXT DISPLAY COMMAND
#chan (optional) a screen channel
toggle 1 or 0 representing on and off

FLASH only works in the QL's low resolution eight-colour mode and only applies to text. After issuing a FLASH 1 command all text is displayed in a flashing state. Unlike the earlier Spectrum, the QL's flash does not reverse foreground and background colours, but prints the text in alternating foreground and background colours. FLASH 0 provides a steady state for subsequent text. If a graphic overwrites flashing text the results are almost always unattractive and subsequent graphic drawing becomes unpredictable. Even Minerva does not solve this problem.

FLEN(#chan)

[Super Toolkit II]

FILE FUNCTION
#chan A valid file channel

FLEN belongs to the Super Toolkit family of functions which interrogate the Qdos file header for useful information, in this case the length of the file attached to the given channel.

FLOAT$(number)
[Turbo Toolkit]

NUMERIC CONVERSION FUNCTION
number Any valid number within the QL's range for floating point numbers.

The QL does not store floating point numbers as "123.456" because it wastes space and slows down mathematical operations. Instead it uses a format in which every floating point number is represented in 6 bytes. This is patently wasteful for simple integers, but more efficient for numbers with many significant digits. Sadly, the QL's nonsensical storage and manipulation of integers does not improve on a function which returns the internal storage format of a number as a strong. The string can then be sent to a data file, for example.

FLUSH #chan
[SuperToolkit II]

FILE COMMAND
#chan a file channel

The FLUSH command coaxes information out of temporary Qdos buffers and onto the file storage medium. This operation normally happens automatically when a file is closed, but you may want to flush the buffers to avoid catastrophe should a power cut strike while a file is open.

FNAMES(#chan)
[Super Toolkit II]

FILE FUNCTION
#chan a file channel

Should you forget the name of a file to which a channel is attached, or should you wish to display its name for your user's benefit, this function reads the required information from the file header.

FOPEN(#chan, name)
FOP_IN(#chan, name)
FOR_NEW(#chan, name)
FOR_OVER(#chan, name)
[All Super Toolkit II]

FILE FUNCTIONS
#chan (optional) a channel linked to a file
name a valid filename

SuperBasic is at its most unreliable when files are being opened. Should the file not be there, or the medium full, or the device name invalid there is only one outcome: an error message. If you are developing software for other people, or even if you want intelligent software for your own use, this zeal for bringing proceedings to a premature halt is unacceptable. Super Toolkit's FOP_ family of functions provides a way of intercepting all errors related to opening files. In addition, the functions can select unused channels automatically.

For example, FOPEN(#3,flp_junk) attempts to link channel number three to an existing file flp1_junk for both reading and writing. If the operation fails for any reason, the function will return a negative number representing an error code (see the entry for ERNUM for details). A successful operation returns the value 0. If the channel number is omitted, the function will return either an error code or a positive integer representing the channel number allocated to the file, as in:

```
150 handle = FOP_NEW(flp1_filename)
```

The rest of the FOP_ family works in the same way, but FOP_IN provides read-only access, FOR_NEW creates a new file (assuming that the filename is presently unused), and FOP_OVER overwrites any existing file of the same name.

FOR ident = range1 STEP increment, range 2 STEP increment
FOR ident = num1, num2
EXIT ident
NEXT ident
END FOR ident

LOOP STRUCTURE

ident	A floating-point numeric variable
range1, range2	One or more ranges of the form 5 TO 20 separated by commas
num1, num2	One or more numeric values separated by commas
increment	(Optional) A numeric variable. If increment is not used, the STEP keyword is omitted

and an increment of one is assumed.

In SuperBasic the FOR...NEXT loop updoubtedly reached the heights of sophistication. Given that all computer programs are ultimately stretches of commands structured with loops and branches, the importance of the FOR...NEXT loop is difficult to overstate. The structure begins with a declaration which determines the name of the controlling variable, the range or ranges of values it will adopt, and the value by which it will be increased or decreased with each cycle of the loop. The end of the loop is indicated by an END FOR statement. Any commands between these two lines will be carried out each time the loop is cycled. A trivial example will clarify the above if it is at all unclear:

```
100 FOR x = 1 TO 12
110 PRINT x
120 END FOR x
```

To increase its usefulness, the STEP keyword can be added to alter the default increment of one. To count down from twelve to one in the above example, change line 100 to read FOR x = 12 to 1 STEP -1. The STEP value need be neither whole nor positive. A single FOR loop can have a number of ranges, each with its own STEP value if required.

Ranges and STEP values can be represented by variables, as in the statement FOR a = b TO c STEP d. The interpreter relies on the programmer to ensure that the values held by the variables make sense. A loop from 34 TO 12 STEP 1 will never end. Clever programmers are often tempted to change the values of variables in the FOR statement during the loop itself, but this is usually a dangerous game. Changes to a STEP variable within a loop are ignored.

To print out the numerals, capital letters and lower-case letters from the Ascii set, the following FOR...NEXT loop will suffice:

```
FOR char = 49 TO 57, 65 TO 90, 97 TO 122: PRINT CHR$(char)
```

The example also shows another of SuperBasic's improvements on the ordinary FOR...NEXT loops found in inferior Basics. If the FOR command is followed on the same line by the statement or statements which form the main body of the loop, the END FOR statement can be dispensed with.

SuperBasic also makes it easy to specify an irregular group of values in a FOR...NEXT loop. Simply replace ranges with single values, such as:

```
FOR x = 3, 6, 2, 9, 11, 4: PRINT x
```

There may also be the need to leave a loop prematurely, perhaps because some condition has been met. A typical scenario might be "search this array until either an element equal to 'QL WORLD' is found or until the end of the array is reached". Assuming that the number of array elements is represented by the variable max, a loop can be set up to cycle through the values from 1 to max. Not only is the FOR...NEXT structure able to cope with the algorithm, but it encourages an extension along the lines of: "if a matching element is not found print a warning message."

The functionality to interpret the second part of the algorithm lies in the keyword EXIT and its interaction with the keywords NEXT and END FOR. Here is a suitable program snippet:

```
100 REMark : Assume array of MAX elements exists
110 FOR x = 1 TO max
120             IF array$(x) = "QL WORLD": EXIT x
130 NEXT x
140             PRINT "The search was unsuccessful"
150 END FOR
160 REMark : The program continues ...
```

The NEXT line causes the interpreter to return to the start of the FOR...NEXT loop. If the loop continues uninterrupted to its maximum value the interpreter will drop out at line 130 and print the error message. Such lines between NEXT and END FOR are called the "loop epilogue". The END FOR is simply ignored. If, however, a match is found, the EXIT command will be executed and the interpreter will go to the line immediately following the END FOR, by-passing the loop epilogue. While this can be very neat, it does not suit all circumstances: what to you do when you want to take one action if the loop continues to the end, and another action if the loop is left prematurely?

Traditional Basic programmers are conditioned to ending FOR loops with NEXT, but this does not help the SuperBasic interpreter because it thinks that the rest of the program could well be a loop epilogue. Only when an END FOR is encountered will it forget about the existence of the loop. The short advice is always end FOR...NEXT loops with END FOR.

SINCLAIR QL WORLD

THE NEW USER GUIDE

SECTION SIXTEEN

KEYWORD INDEX

This month in the Keyword Index, Mike Lloyd discusses the point of IDEC$ and ends up adding some INPUT.

IDEC$(value, width, decplaces)
[Super Toolkit 2]

DECIMAL NUMBER FORMATTING COMMAND

value	A decimal value
width	The total number of character positions to return
decplaces	The number of decimal places to display

IDEC$ is one of a group of extremely useful functions which circumvent the QL's habit of converting numbers with relatively few significant digits into exponential format. The function accepts numeric input and produces a fixed-length string of the required number of digits (which may include decimal places).

IF <expr> THEN <statements> (short form)
IF <expr>: <statements>(short form)
IF <expr> THEN (long form)
<statements>
ELSE (optional)
<statements> (optional)
END IF

CONDITIONAL BRANCHING STRUCTURE

<expr>	any expression which can be assessed as being true or false.
<statements>	one or more SuperBasic statements.
THEN	This keyword is wholly optional in SuperBasic. If a SuperBasic statement shares the same logical line as an IF clause they must be separated by a colon.

All computer languages share two fundamental abilities: the ability to repeat segments of code and the ability to skip segments of code. The latter ability has two types, called conditional and unconditional branching. Unconditional branching is performed by GOTO, GOSUB and the user-defined procedures and functions. Conditional branching, where the direction the program takes depends upon a logical decision, is normally performed by the IF...ENDIF structure (although SuperBasic also has an advanced SELECT structure for multiple decisions).

The IF...ENDIF structure can be broken down into three clauses separated by bodies of SuperBasic statements. The first clause begins with the IF keyword and contains a logical expression which must be evaluated to true or false.

In SuperBasic, logical expressions can include an equals sign, in which case their truth is explicit, or they may have a numerical value, in which case truth is implied if the result is not zero. Expressions can be linked with logical operators such as AND and OR, and they may be reversed by being preceded by the NOT operator. Examples of logical expressions are:

IF x = 12	True if x = 12
IF NAME$ = "JIM"	True if NAME$ equals "JIM"
IF y	True if y is not zero
IF 4 + 2 - 5	True always
IF x = 12 AND y = 4	True if both elements are true
IF x = 12 OR y = 4	True if either element is true
IF NOT x = 12	True if x is not 12

The final part of the IF clause is the optional keyword THEN. THEN can improve the reading of an IF structure because it helps to form an English-like sentence, such as IF x = 12 THEN LET result$ = "True!". The alternative is to omit it, replacing it with a colon if there is another statement follows on the same program line. In effect, THEN is simply a fancy statement separator, but it can lead people astray. Common assumptions are that only the statement linked directly with the IF clause by THEN is carried out when the IF expression is true, and that ELSE can manage, like THEN, without statement separators.

Having decided upon how program progress will be determined, the next stage is to define those statements which will be carried out only if the IF clause's expression is true. SuperBasic has a long (multiple line) and a short (single line) format for the IF structure. In the short format, statements follow either the THEN keyword or a colon on the same program line as the IF clause. In the long format, the statement following the IF clause must appear on the next program line, otherwise SuperBasic will assume that it is dealing with the short form. With either format, any number of statements can be included.

For many occasions, it is enough only to define those commands which will be carried out when an IF expression is true. However, there will be times when two distinct alternatives present themselves. This can be summarised as "Do this body of code if the expression is true, but do this body of code if the expression is false." In SuperBasic the ELSE keyword permits this sort of logical structure to be developed.

In the short form of the IF statement it is often forgotten that ELSE should be separated from preceding and succeeding statements by colons. It is too easy to follow the false example offered by THEN and omit one or both colons and thus generate an error. In the long form, it is usual to put the ELSE statement on a line to itself. This is entirely optional, but it does make it easier to see the logic of a program. After the ELSE statement any number of SuperBasic statements can be added, all of which will only be carried out should the IF expression equate to false.

Finally, whether or not there is an ELSE clause, the long form of the IF structure must be concluded by an END IF clause. Again, it is usual to place the END IF clause on a line of its own, but SuperBasic will accept one that appears anywhere on a line. In the short form, the end of the logical line serves as an END IF, but one can still be added if desired.

IF structures can be nested, in order to cope with a complex set of related decisions such as "IF the user's ID and password are correct THEN IF it is Friday remind them to back up the floppy disks, ELSE end the program." Even in English, this algorithm is slightly ambiguous, although most people would infer that the ELSE clause would only be carried out if the first IF clause were untrue, otherwise nobody could enter the system except on a Friday. The QL cannot infer any such thing and relies on the proper use of the ENDIF and ELSE clauses to determine exactly what is meant. In a program, the above might be rendered as:

```
500 IF ID$ == "John" OR ID$ == "Jill" AND PASSWORD$ = "letmein"
510 IF DAY$ == "Friday"
520 PRINT "Remember your backups"
530 ENDIF
540 ELSE
550 PRINT "You are not allowed in"
560 STOP
570 ENDIF
```

IMPLICIT$ var1, var2, var3...

[Turbo Toolkit]

IMPLICIT% var1, var2, var3...

[Turbo Toolkit]

	COMPILER DIRECTIVE
varXa	variable name

The SELECT structure is a useful way of offering many courses of action for the computer to select, but it only works with decimal numbers. It is therefore not possible to write the following code:

```
100 SELect ON name$
110 = "John": PRINT "OK to proceed"
120 = "Jill": PRINT "OK to proceed"
130 = REMAINDER: PRINT "Go away."
140 END SELect
```

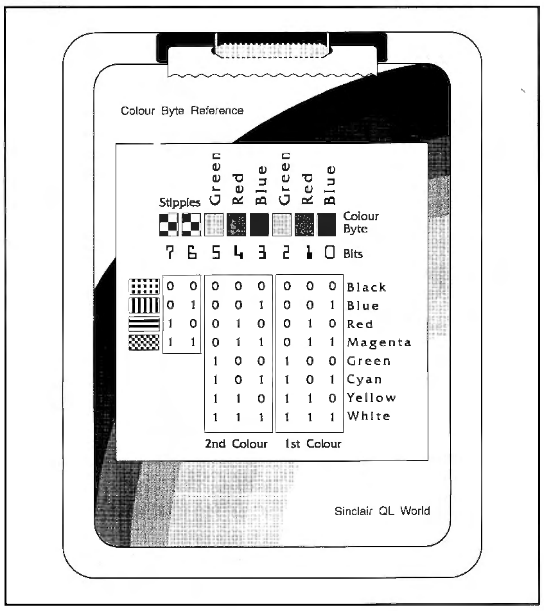
This restriction was not good enough for the developers of the Turbo compiler, but the absolutes of the SuperBasic interpreter's syntax seemed to offer no room for a solution. That is, until the IMPLICIT directive was thought up. This keyword is ignored in interepreted SuperBasic programs, but in a compiled program all of the variable names contained in IMPLICIT statements take on the attributes of variables ending with the character which ends the IMPLICIT keyword. IMPLICIT$ variables become strings and IMPLICIT% variables become integers even though they do not end with the appropriate suffix. In this way the Turbo team managed to allow SELect to work with strings. The above example needs to be altered only slightly, as follows:

```
90  IMPLICIT$ name
100 SELect ON name
```

Programmers with experience only of Basic dialects will at first experience some insecurity when confronted by variables whose class cannot be determined by their names, but others used to languages such as C will probably prefer firstly using suffix-less variable names and secondly enduring the discipline of declaring every variable they use.

INK #chan, main, sub, pattern

	CHARACTER AND GRAPHICS COMMAND
#chan	(optional) channel number
main	an integer between 0 and 255
sub	(optional) an integer between 0 and 7
pattern	(optional) an integer between 0 and 3


Colour Byte Reference

Stipples			Green	Red	Blue	Green	Red	Blue	Colour Byte
Bits	7	6	5	4	3	2	1	0	
▦	0	0	0	0	0	0	0	0	Black
▥	0	1	0	0	1	0	0	1	Blue
▤	1	0	0	1	0	0	1	0	Red
▩	1	1	0	1	1	0	1	1	Magenta
			1	0	0	1	0	0	Green
			1	0	1	1	0	1	Cyan
			1	1	0	1	1	0	Yellow
			1	1	1	1	1	1	White
			2nd Colour			1st Colour			

Sinclair QL World

All character and graphics output to the screen can be coloured, even though on a monochrome monitor the result might only be variations on a theme of grey. Two commands which control screen colours are PAPER and INK, affecting the background and foreground colours respectively. Note that although graphics output responds to INK changes the BLOCK command, which is more closely related to the WINDOW command, has its own colour parameter. The INK command's affect on foreground colours is limited by the current screen mode. Mode4 allows only four colours (black, white, red and green) while mode8 allows four more (cyan, blue, yellow and magenta). However, references to the extended colour set when the screen is in Mode4 does not generate an error message. The QL simply picks the nearest valid colour instead.

The QL has an unusual way of handling colour in its screen map which is reflected in the way in which the INK parameters work. (The screen map structure is dealt with later in the *New User Guide*). Foreground colours can be solid or one of four stipple patterns, and can be represented by a single integer value between 0 and 255. See the accompanying panel for a bit-by-bit explanation of how Qdos interprets colour values.

SuperBasic allows a single one-byte value to represent all of the characteristics of a foreground colour, but programmers find it much easier to declare stipples with three parameters. The first parameter decides the primary colour. If the parameter lies within the range 0 to 7 and there are no other parameters, a solid ink of the required colour is provided. The second parameter determines the alternate colour. With two parameters in the INK statement the QL assumes that a stipple is required and defaults to a checkerboard pattern. Alternatively, a third parameter between 0 and 2 causes other stipple patterns to be adopted (a value of 3 causes the checkerboard pattern to be selected).

On monitor displays, stipples can produce pleasing colour combinations such as pale green (white and green) and vivid pink (magenta and red). The default pattern is best at disguising the fact that the colour is not solid. Interesting horizontal and vertical stripes can be achieved with stipples 1 and 2, while more subtle shadings, or a contrasting polka dot effect, can be achieved with stipple 0. With a TV display the results are more than likely to cause unpleasant strobing.

INK stipples tend to make characters difficult to read, being more suited to blocks of colour such as a FILLed circle. The grainy effect of stippled characters can be exploited when asking for a password to be entered: the user can see that something has been printed on the screen, but a casual observer would find it difficult to read exactly what had been typed.

INKEY$(#chan, frames)

	CHANNEL INPUT FUNCTION
#chan	(optional) a valid channel number.
frames	(optional) a timeout measured in screen refreshes (50 per second).

The INKEY$ function obtains input character by character from the given channel, which by default is the special Channel #0, the user's keyboard, but which could also be a file. This is in contrast with INPUT, which obtains chunks of data before a linefeed (or the use of the ENTER key) is encountered, and READ, which obtains information only from DATA statements, and KEYROW, which is a low-level keyboard reading function. INKEY$ always returns a character value.

With no parameters whatever, INKEY$ returns the keystroke character pressed at the precise instant that the interpreter met the function call. Sometimes this is useful, particularly in games, but it is often more convenient to wait either for a set period of time or forever until a key is pressed. With a single, positive integer parameter INKEY$ will wait that many screen refreshes before giving up hope of a keypress arriving. In the UK, screen refreshes are carried out 50 times a second (where electricity is supplied at 60 Hz the refresh time increases to 60 times a second), so a value of 200 implies a two-second wait. If -1 is specified the program will wait forever for the next keypress.

INKEY$ improves on INPUT where only a single character is required. INPUT insists on a minimum of two keypresses, because ENTER must be pressed, but INKEY$ can respond to just one, as in the following example:

```
200 PRINT "Do you want to do this? (Y/N)"
210 IF INKEY$(-1) == "Y"
220 REMark Do it...
230 END IF
```

INKEY$ will also take input character by character from a given channel, provided that a valid channel identifier is provided. Again, the timeout can come in useful where the channel might be a pipe from another program or be attached to the network. For disk or microdrive access, no timeout is required.

INPUT #chan, prompt1$, var1, prompt2$, var2, ...
INPUT #chan, var1, var2, ...

	CHANNEL INPUT COMMAND
#chan	(optional) a valid channel number
promptX$	(optional) a text string or text variable. Many prompts can be included in an input string.
var1, var2,..	(optional) string or numeric variables. Note: the INPUT command permits commas to be replaced by other text separators such as ! \ TO and ;

The INPUT command is at once one of the most useful Basic facilities and one of its crudest. One of the easiest ways of ensuring that a program grinds to an unexpected halt is to include an INPUT command which expects a numeric input: the operator only has to press a character by mistake and an error is generated. That apart, SuperBasic's implementation of INPUT has several nice features.

INPUT assigns a value to a variable, rather like a LET command. The value does not come from within the program but from an external source such as the keyboard or a file opened on a microdrive cartridge. All input is accepted until the ENTER key is pressed or until a linefeed is detected in a file. A single INPUT statement can assign values to several variables.

The INPUT command is most frequently used to read keyboard input. Without a channel number, the input is displayed in the default window at the flashing cursor. INPUT can be directed to any other window provided it has been defined as a console rather than as a screen (use OPEN#3, con_, not OPEN#3, scr_). All of the windows opened when the QL is booted are consoles, not screens.

Input can be preceded by a prompt to advise the user on what is expected, along the lines of INPUT "Enter your name", name$. The separator has the same value as in a PRINT statement, so a comma forces a tab between the end of the prompt and the point at which input is accepted. The interpreter does not care how many prompts there are or where they appear in the INPUT statement. Neither does INPUT care how many variables the programmer provides; without any at all it resembles a simple PRINT statement.

When INPUT is used in conjunction with a file opened for input its syntax becomes tighter. Prompt strings are meaningless and are ignored. Variables must be separated by commas. If there is no input to be obtained from the file an End of File error message is generated.

With either use, the accidental assignment of a string value to a numeric variable causes the program to halt. For this reason, the Turbo Toolkit EDIT family of functions is usually preferred for keyboard input.

THE NEW USER GUIDE

SECTION SEVENTEEN

KEYWORD INDEX

This month in the Keyword Index, Mike Lloyd starts with INPUT$(#chan, length) and finishes with the command without which none of this would be possible: SuperBasic LRUN.

INPUT$(#chan, length)
[Turbo Toolkit]

INPUT FUNCTION

#chan(Optional)	A valid channel linked to an input device
length	The number of characters to be returned

INPUT$ is an unusual little function that waits for a precise number of input characters before allowing processing to continue. If the input channel is a screen console, no cursor is displayed to prompt for input. INPUT$ is of most value extracting set lengths of characters from files and network channels. The default channel is #1, but keyboard input is obtained by specifying #0.

INSTR

TEXT OPERATOR
Example: IF "z" INSTR surname$ THEN...

Most operators are represented by symbols, but a suitable symbol was presumably unavailable when INSTR was devised. Other languages have implemented INSTR as a function with the format: x = INSTR("w", "hello world") but SuperBasic obviously had to be different. The results, however, are the same. Both operands are string variables or constants. If the first is found within the second then the location is returned. If no match is found then a zero is returned. "W" INSTR "Hello World" returns 7 because "W" is the seventh character of the string being searched. The search is not case-sensitive.

INT(number)

MATHEMATICAL FUNCTION

number	A floating point value

The INT function truncates a floating point value at the decimal point, converting it to an integer.

INTEGER$(number)
[Turbo Toolkit]

BYTE CONVERSION FUNCTION

number	An integer within the range -32768 and 32767

The INTEGER$ function returns a string of exactly two Ascii characters that represent a 16-bit SuperBasic integer. This is usually superior to allowing SuperBasic to convert integers to floating point format. The Turbo Toolkit STRING% function converts the string back to an integer.

JOBS #chan
[Super Toolkit II]
JOBS(id or name)
[Super Toolkit II]

TASK MANAGEMENT

chan(Optional)	A valid channel

The JOBS command simply lists all running jobs to the appropriate channel. The default is #1. When the QL is turned on the only task running is SuperBasic. The JOBS function is used with related functions to point to and operate on jobs in the Qdos job list; operations that lie outside the scope of this SuperBasic guide.

KEYROW(row)

LOW-LEVEL KEYBOARD READER

row An integer between 0 and 7

There are 65 keys on the standard QL keyboard; each is identified to Qdos by a reference number. The two Shift keys share one reference number, which means that all of the keys can be represented on an eight-by-eight grid. The KEYROW function interrogates the grid to discover which, if any, keys are being pressed.
There are three essential differences between KEYROW and the apparently similar INKEY$. KEYROW returns its own code reference, not an Ascii character; it can detect many keys pressed at once; and it reads the keyboard directly, not through a channel. These differences are advantageous in games, compiled programs that do not have an input channel, and when writing intelligent input control routines.
KEYROW works by accepting a row number (from 0 to 7) as its parameter. It instantly returns a byte value that indicates which, if any, keys on the row were being pressed. Eight calls would be necessary to poll the entire keyboard, but in many instances a single row contains all the likely keys. The four cursor keys, Space, Enter and Escape, for example, are all on KEYROW(1).

KEYROW'S KEYBOARD MATRIX

Each row in the matrix is read by a single KEYROW call using the parameter identified in the shaded column on the left. The value returned will be zero if no key in the row is being pressed, or the sum of the column headings of keys being pressed. For instance, if the Q and the T keys are pressed simultaneously, KEYROW(6) will return 72 (that is, 64+8). Note that at least some editions of the Sinclair User Guide contain a number of misprints in the KEYROW matrix.

	1	2	4	8	16	32	64	128
7	SHIFT	CTRL	ALT	X	V	? /	N	< ,
6	* 8	@ 2	^ 6	Q	E	) 0	T	U
5	(9	W	I	TAB	R	_ -	Y	O
4	L	# 3	H	! 1	A	P	D	J
3	{ [	CAPS LOCK	K	S	F	+ =	G	: ;
2	}]	Z	> .	C	B	~ £	M	" '
1	ENTER	LEFT	UP	ESC	RIGHT	| \	SPACE	DOWN
0	F4	F1	% 5	F2	F3	F5	$ 4	& 7

The following routine will be of value when looking up a particular KEYROW value for a single printable keypress:

```
100 DEFine PROCedure KEYTEST
110     REPeat loop
120         char$ = INKEY$(-1)
130             FOR row = 0 TO 7
140              col = KEYROW(row)
150               IF col > 0
160                   PRINT char$;" = ROW "; row; ", COL "; col
170               ENDIF
180           END FOR row
190   END REPeat
```

Each column value in the KEYROW matrix represents a single set bit, so the best way of interpreting the result of a KEYROW call is to use the bitwise operator &&. Something like the following routine will be found at the heart of a great many arcade games. It finds out which of the cursor keys is being pressed and moves a graphics block accordingly:

```
100 WINDOW 512, 256, 0, 0: PAPER 0: CLS
110 Ypos = 150: Xpos = 256
120 REPeat Loop
130 BLOCK 4,4, Xpos,Ypos, 4
140     shift = KEYROW(1)
150     IF shift && 2: xpos = xpos - 2
160     IF shift && 4: ypos = ypos - 2
170     IF shift && 16: xpos = xpos + 2
180     IF shift && 128: ypos = ypos + 2
190     IF shift && 8: STOP
200 END REPeat loop
```

Note: Trying to move the block beyond the edge of the screen will cause an error.

KEYROW can return some strange values if the keys forming three corners of an imaginary rectangle on the grid are pressed. The result will suggest that the key at the fourth corner is also being pressed. Shift, Ctrl and Alt are exceptions to this rule.

LBYTES filename, base_address

MEMORY MANAGEMENT COMMAND

filename	The name of a file containing binary data, normally a program
base_address	The lowest location in RAM where the data will be stored

LBYTES is short for 'load bytes'. The bytes in question might be a screen image, some data or, more usually, a program. The command needs to know where to look for the bytes and where in ram to begin loading them. As Qdos rarely misses any opportunity to rearrange the contents of ram, care must be taken to ensure that LBYTES does not overwrite critical areas of memory. Given an amount of space to reserve, the RESPR function will return the start address of a reserved area of memory sufficient for the task. Thus, a program might be activated like this:

```
100 base = RESPR(6000)
110 LBYTES flp1_arcade, base
120 CALL base
```

About the only reliable absolute address on the QL (and even this is open to some change) is the start of the screen map at location 131072. Its size is always 32K, no matter what mode the screen is in.
Files that contain binary data have to be saved using the SBYTES command with the appropriate parameters. Some database designers prefer to store data 'in the raw' rather than use SuperBasic arrays. Sorting data becomes a very fast and relatively simple matter of moving blocks of memory around. Text data can be compressed (to two characters per byte, for instance) and searched for at high speed with the appropriate procedures. Turbo Toolkit is an essential purchase for SuperBasic programmers attempting this sort of data manipulation.

LEN(string$)
LEN(number)

STRING FUNCTION

string$	A string variable
number	A floating point or integer value

This function answers the eternal question 'How long is a piece of string?'. A string in SuperBasic is zero, one or more Ascii characters. The LEN function takes the string as its argument and returns the number of characters it contains. Null, or empty, strings will return the value 0 (or False if used in logical expressions). Although elements of string arrays occupy a fixed number of bytes, LEN returns the exact number of characters contained in a given element. SuperBasic's powerful coercion facility means that LEN works equally well on numeric values, which is of particular value when formatting columns of numbers. When coercing numbers into strings, LEN reflects the length of the scientific format of very large and very small numbers, not the number of digits.

LET <expression>

VARIABLE ASSIGNMENT

<expression>	A variable assignment of the form X = Y + 7 or Y$ = "Hello"

LET is a wholly redundant keyword that was presumably included in SuperBasic for compatibility with earlier dialects. Its job is to precede variable assignments to make absolutely clear what is happening within an assignment statement. However, the following lines are identical in effect:

```
LET x = 25
x = 25
```

LINE #chan, x1, y1
LINE #chan, x1, y1 TO x2, y2 TO ...
LINE #chan, TO x3, y3
LINE_R #chan, x1, y1
LINE_R #chan, x1, y1 TO x2, y2
LINE_R #chan, TO x3, y3

GRAPHICS COMMAND

#chan	(Optional) A screen channel
x1, y1, etc.	Graphics co-ordinates

The LINE and LINE_R commands move the graphics cursor. The _R suffix indicates that the movement is to be relative to the current location of the graphics cursor rather than relative to the graphics origin. The parameters taken by LINE and LINE_R are pairs of comma-separated values representing horizontal and vertical co-ordinates in the QL's graphics co-ordinate system. The TO keyword between co-ordinate pairs inks the path between the current and new cursor positions. If the co-ordinate pairs are separated by commas the cursor simply moves, without leaving an ink trail, to the location of the final pair of co-ordinates. Odd numbers of co-ordinates cause a 'bad parameter' message.

LIST #chan
LIST #chan, 100 TO 200

LIST #chan, 100 TO 200, 300 TO 350
LIST #chan, 10, 50, 30
LIST #chan, x TO y

PROGRAM COMMAND

#chan	(Optional) An output channel
x TO y	Integer values representing line numbers

The LIST command without parameters lists the currently loaded SuperBasic program to Window #2. The listing can be directed to a printer, another window, the network or a file by including a channel reference. Listings can be limited to one or more groups of lines by including ranges in the form 100 TO 200. A number of single lines can be listed by including a comma-separated list. Note that the Super Toolkit II ED command both lists program lines and allows you to edit them within the chosen window.

LIST_TASKS #chan
[Turbo Toolkit]

MEMORY COMMAND

#chan	(Optional) An output channel

This command lists all tasks currently loaded into the QL's memory. SuperBasic is represented as task 0,0. It is the equivalent of Super Toolkit II's JOBS command but has the added advantages of no header, the inclusion of each task's name and the use of a comma-separated format.

LN(pos_value)

MATHEMATICAL FUNCTION

pos_value	A value greater than zero

The LN function returns the natural logarithm of any positive value passed to it. The irrational number e has a LN value of one.

LOAD filename

PROGRAM COMMAND

filename	The drive and name of a file containing a SuperBasic program

The LOAD command fetches a SuperBasic program from the given filename and installs it in the QL's memory, removing any existing program that might already be there. Although LOAD can be used within programs, LRUN is usually preferable because the new program will be executed automatically. The QL is particularly slow at loading SuperBasic programs because they are stored as plain text on file and need to be tokenised when moved into memory. There is at least one utility on the market that makes program loading very much faster.

LOCAL p1, p2, p3 ...

INTERPRETER DIRECTIVE

p1, p2, p3	A comma-separated list of one or more variable names

The addition of user-defined procedures and functions to SuperBasic led to the introduction of the LOCAL keyword. Its place is as the first executable statement after a DEFine command. The variable names included as parameters then have values that only exist within the definition. When the definition is left the local value is lost. If the variable name represented a value before the definition was called then that value is restored.

```
100  X = 12
110  PRINT X: REMark result is 12
120  PRINT Demo_Function     : REMark result is 3
130  PRINT X : REMark result is 12
140  :
150  DEFine FuNction Demo_Function
160  LOCal X
170  X = 3
180  RETurn X
190  END DEFine
```

LOG10(pos_value)

MATHEMATICAL FUNCTION

pos_value	A value greater than zero

The LOG10 function returns the natural logarithm of any positive value passed to it. Ten has a LOG10 value of one and one has a LOG10 value of zero.

LRUN filename

PROGRAM COMMAND

filename	The drive and name of a file containing a SuperBasic program

The LRUN command fetches a SuperBasic program from the given filename, installs it in the QL's memory and runs it. Any existing program that might already be there is removed. LRUN can be used within programs as a way of chaining a number of programs together. Be aware that the QL is particularly slow at loading SuperBasic programs because they are stored as plain text on file and need to be tokenised when moved into memory.

SINCLAIR QL WORLD

THE NEW USER GUIDE

SECTION NINETEEN

KEYWORD INDEX

This month in the Keyword Index, Mike Lloyd starts (as we all do) with FORMAT #chan and works up to HEX$(decvalue, bits). This section follows last month's (Section eighteen), which was itself a little out of sequence. Next month we should be picking up where we left off after LRUN in August 1992.

FORMAT #chan devX_volume

FILE COMMAND

#chan	(optional) valid screen channel
devX_volume	device name (eg mdv1_) and (optionally) a volume name (eg mywork)

New microdrive cartridges and floppy diskettes are usually unformatted and cannot in this condition save data sent to them by the computer. Before using them they must be formatted, a process which identifies sectors into which 512-byte chunks of information can be saved. If a previously-used medium is reformatted all its current contents are deleted irretrievably. Usefully, the Minerva rom amends FORMAT to issue a warning if there are any open files on a medium about to be formatted, but the main onus is on the user to make sure that valuable media are not formatted accidentally.

Each sector is tested to see if it is OK; if it fails it is marked as being "bad". At the end of the formatting process a screen message displays the total number of sectors and the number of available sectors. Microdrive cartridges always report at least two fewer good sectors than total sectors: these are allocated to the sector map which permits the computer to find where on the cartridge various files are stored. For convenience, many users give each cartridge or disk an identifying name of up to 20 characters, but this can be omitted if preferred.

With microdrive cartridges the total number of sectors can vary slightly according to the length of video-standard magnetic tape it contains. It is usually worthwhile to format a new cartridge at least three times in order to reach its normal operating length. Not only will it be more reliable but it might even gain a few more sectors. When old age strikes reformatting can temporarily restore a cartridge's usefulness, but at the loss of all the information currently stored on it.

Diskettes are formatted using the same Qdos command, but the formatting process itself is different. Diskettes are formatted with concentric tracks each broken down into sectors under the control of the disk controller supplied with the disk drives. It is unusual for any diskette to report bad sectors. Although diskettes now come in three densities (double, high and extra-high, or DD, HD and ED) early disk controllers and disk drive mechanisms for the QL probably only recognise the DD standard. Miracle's Gold Card and accompanying disk drive recognises all three formats.

If a microdrive or diskette fails to work, valuable data can often be retrieved using one of the "disk doctor" software packages advertised in *QL World.*

FPOS (#chan) [Super Toolkit II]

FILE HANDLING FUNCTION

#chan	A valid channel number open to a file

A file can be imagined as a stream of binary values. When a file is opened the file pointer is set to its first byte ready for an INPUT command, or maybe a BGET or GET command from *Super Toolkit II*. As successive pieces of information are read from the file the pointer moves to the next unread byte. Should the end of the file be reached then the EOF() function will return a true value. When the file is closed the pointer information is lost. FPOS() indicates how many bytes from the beginning of the file its pointer currently is.

FREE_MEM()

[Super Toolkit II]

FREE_MEMORY()

[Turbo Toolkit]

MEMORY FUNCTION

An essential part of a computer is random access memory, or ram. It must be fast, capacious and cheap: requirements which at present can only be met by memory chips requiring frequent refreshment with electrical energy. This is why the QL's memory fades as soon as it is unplugged from the mains. The QL's ram can be expanded to various sizes by the addition of expansion cards. Its memory can also be shared simultaneously by a number of multi-tasking programs. In such circumstances it can be foolish for a program to make assumptions about how much free memory is available. The FREE_MEM function returns the amount of unallocated memory in the QL. Used from interpreted SuperBasic, FREE_MEMORY performs exactly the same function, but within a Turbocharged program it is only aware of the dataspace allocated to the program.

FSERVE

NETWORK COMMAND

Right from the very beginning it was intended that QLs should be able to communicate with each other across a network. The network standards chosen were very crude and in all of the roms issued by Sinclair Research it was incompletely implemented. Super Toolkit II completed the missing network code, so that any QLs running Super Toolkit II can happily share a network. Network communication is achieved by opening device channels similar to those which establish links with files, printers and screen windows. In addition, any QL on a network can undertake the role of file server. File servers can share up to ten of their devices with other workstations on the network provided that a special job called SERVER has been invoked by issuing the command FSERVE.

FTEST(filename)

[Super Toolkit II]

FILE HANDLING FUNCTION

filename	a target filename (which may include a device identifier, eg mdv1_filename)

Programmers quickly become aware that SuperBasic is extremely fond of generating error messages whenever files are manipulated. FTEST is a handy function which can test for the condition of a file before anything is done to it, allowing the program to make some intelligent choices rather than coming to an unceremonious halt. FTEST tries to open the file identified by the parameter for reading and immediately shuts it again. If the file exists and is available the result is 0, otherwise it is a negative number corresponding to the appropriate error code. For example, -7 would be returned if the file was not found. A full list of error codes was published in the *New User Guide* under the keyword ERNUM. Even if FTEST reports a favourable result, error trapping should still be included on any command which tries to create, delete or communicate with the tested file.

FTYP(#chan)

[Super Toolkit II]

FILE HANDLING FUNCTION

chan	A channel number opened to a file

Qdos categorises its files into three types: ordinary, executable and relocatable machine code, which it refers to as 0, 1 and 2 respectively. Executable files are those launched with an EXEC command (or its near-clones), relocatable machine code files are typically launched with a LRESPR command and all other files are classed as being ordinary. The file type is held in the file header and is read using the FTYP function.

FUNCTION funcname(parameters)

[Turbo Toolkit]

COMPILER DIRECTIVE

funcname	a function definition name
parameters	(optional) a standard, comma-separated list of parameters

The FUNCTION keyword is a clever piece of cheating to allow function definitions to be identified in the EXTERNAL command. EXTERNAL is a compiler directive which looks like a procedure call to Qdos and SuperBasic; its parameters must therefore be a comma-separated list. The list identifies all externally-declared variables, arrays and user definitions mentioned in the program. In the SuperBasic naming conventions there is no difference between a variable name and a function definition name and so the developers of the *Turbo* compiler needed to find a way of distinguishing between them. This was achieved by declaring a new keyword, FUNCTION, which could be placed in the comma-separated list immediately prior to a function name. Unfortunately, there must be a comma between FUNCTION and the function name but apart from that it is a good solution to the problem.

	FILE HANDLING FUNCTION
chan	A channel number opened to a file

Whenever a file is saved, Qdos stamps its header with the current system date. FUPDT is a function to retrieve this information. Date-stamping is only fully achieved by Super Toolkit II, although the Minerva rom dates microdrive files. Some disk controllers may also date files saved to diskettes.

GET #chan, pos, items
[Super Toolkit II]
GETF(#chan)
[Turbo Toolkit]
GET%(#chan)
[Turbo Toolkit]
GET$(#chan)
[Turbo Toolkit]

	FILE HANDLING PROCEDURE/FUNCTIONS
chan	A channel number opened to a file
pos	The location of the first byte
items	A comma-separated list of variables

GET is a means of extracting data which has been filed in the QL's internal format, as opposed to the QL's normal preference for textual format. If a file were opened on channel #3 and the command PRINT#3, 999 was issued, the file would contain three Ascii characters, "999". If, however, the command PUT#3, 999 was issued, the file would hold the QL's internal six-byte representation of the value 999. The reverse of a PRINT#3 command is a INPUT#4 command, the reverse of a PUT#3 command is a GET#4 command. It is up to the programmer to ensure that the file pointer is in the right place when a GET is issued.

Turbo Toolkit implements a similar facility as a set of functions, one for each variable type. GETF is for floating point variables, GET% for integers and GET$ for strings. Whereas the Super Toolkit II command can move the file pointer and cope with a list of variables, the Turbo Toolkit functions only return one value and relies on other commands to move the file pointer to the required location. Both Turbo Toolkit and Super Toolkit II leave the file pointer pointing to the next unread byte.

GLOBAL module
group_id$, name_list

	COMPILER DIRECTIVE
module	An integer between 1 and 252
group_id$	A string constant
name_list	A comma-separated list which can contain the names of variables, arrays, user-defined procedures and user-defined functions

The Turbo compiler permits programmers to link together program snippets, combining frequently-used utility routines into many different programs. This facility is not available in interpreted SuperBasic and so is only briefly dealt with here. EXTERNAL and GLOBAL establish links between the program calling a routine and the program containing the routine respectively. Where the specification refers to a procedure or a function the name of the structure is preceded by the keyword PROCEDURE or FUNCTION (spelt out in full and followed by a comma). Arrays are declared with dummy dimensions, such as STRING$(0,0). GLOBAL directives can be split across several lines, each identified by a string constant. If an EXTERNAL call refers to a string constant it only needs to include the items listed on the given GLOBAL statement.

GOSUB line_no
GOTO line_no

	PROGRAM CONTROL COMMANDS
line_no	A valid line number reference

When SuperBasic was launched much was made of its structured approach to programming made possible by the DEFine PROCedure and DEFine FuNction constructs. These facilities made it possible to do without the much-reviled GOTO and GOSUB commands, but Jan Jones recognised the need to maintain some compatibility with previous Basic dialects and so these keywords are included in SuperBasic and work in the standard way.

GOSUB diverts the interpreter to the stated line, where program execution continues until a RETURN statement is encountered, at which point the interpreter returns to the statement following the GOSUB to continue with its work. If desired, the program line can be given as a variable so that the location of the subroutine is determined at some earlier point in the program.

GOTO operates in a similar manner to GOSUB but the interpreter cannot return to the command following the GOTO (except by means of another GOTO).
The exact program line need not be given; SuperBasic simply finds the line with the next highest number if the given line number has not been used in the program. The Turbo compiler is more precise because it demands that exact line numbers are given in all GOSUB and GOTO commands in compiled programs.

In most Basic dialects GOSUB and GOTO are quickest if they jump to a line early in the program. Interestingly, SuperBasic's GOSUB and GOTO commands are fastest if they jump only a few program lines, which encourages programmers to place subroutines close to where they are called, rather than close to the start of the program.

Good programmers will always discourage the use of GOSUB and GOTO because of their associations with messy, bug-ridden and unmaintainable code. If they are used within procedure definitions, function definitions or WHEN ERROR definitions they can cause even more program integrity problems than usual. However, in their right place and under careful control a few subroutines and a the occasional jump around listings will not make programs any less applicable.

HEX(hexvalue$)
HEX$(decvalue, bits)

	BASE CONVERSION FUNCTIONS
hexvalue$	A string containing a valid hexidecimal value
decvalue	An integer
bits	The number of binary digits represented

Because computers are overwhelmingly obsessed with binary numbers and programmers are only slightly less obsessed with hexadecimal (base 16) numbers, Super Toolkit II contains a number of functions which allow values to be converted between binary, decimal and hex. HEX$ takes a decimal value and converts it firstly into a binary number of a given number of bits (which should be sufficient to represent the decimal value) and then returns a string of hex digits representing that number. HEX is a function to perform the opposite conversion, from a string of hexadecimal digits to a decimal value. Hex digits include the normal numbers 0 to 9 and then represent the remaining six digits with the letters A to F. Each hex digit represents four binary digits, so byte values span the range 00 to FF in hex or 00000000 to 11111111 in binary or 0 to 255 in decimal.

THE NEW USER GUIDE

SECTION TWENTY

KEYWORD INDEX

This month in the Keyword Index, Mike Lloyd finally loops back to follow part 17 (INPUT$ - LRUN, August 1992) with MERGE filename to OJOB (job_id).

MERGE filename

	PROGRAM FILE COMMAND
filename	A valid filename, e.g. "mdv1_myprogram"

The SuperBasic LOAD command automatically wipes out any existing program lines from the QL's memory before loading a new program. It can be useful to load extra program lines from a file without deleting what is already there, and this facility is provided by the MERGE command. It shares an identical syntax with LOAD. Care needs to be taken with line numbering to avoid existing lines being overwritten by new lines with the same line number (unless, of course, this is what is intended).
If the interpreter reads any non-SuperBasic lines while merging a file it marks them with the word "Mistake" and generates an error if the line is reached during program execution. If you are in the habit of writing program scripts using a word processor it is a good idea to MERGE them rather than LOAD them so that all errors of syntax are immediately marked.
MERGE can be quite useful to amend programs prior to running them. For instance, a program might have a number of user-defined procedures, only some of which are applicable to a particular configuration. Each procedure could be saved to its own file and merged with the main program only if it is required.

x MOD y

	MATHEMATICAL OPERATOR
x, y	Numeric values

MOD is not, strictly, a keyword: it is an operator belonging to the same family as the plus, minus and multiplication symbols. MOD is a form of division that gives the modulus, or remainder, when a value is divided by another. MOD answers the question "How many of these 37 oranges if I share out the largest equal number to my four friends?". Mathematically, the question can be phrased "PRINT 37 MOD 4". The QL guide indicates that MOD works only on integers, but it is more correct to say that it will work with any number, integer or real, and always returns a rounded integer value.

MODE x

	SCREEN RESOLUTION COMMAND
x	Numeric value

The MODE command establishes the screen display resolution, setting it to high or low with a corresponding change in the size of the colour palette. Whichever screen resolution is selected the QL allocates exactly 32K of memory to map the screen contents. To increase the resolution (the number of pixels on the screen) it has to reduce the amount of colour information held about each pixel, which is why the high resolution mode has only four colours and no FLASH capability. Perhaps the most useful way of defining the mode is to use Mode4 to obtain the four-colour high resolution mode and Mode8 to obtain the eight-colour low resolution mode. However, any value between 0 and 7 indicates high resolution and any value between 8 and 15 indicates low resolution. To work out what display will result from any MODE value, use the formula "X MOD 16 - 8". If the result is negative the screen will be in high resolution mode, otherwise it will use low resolution.

Note that MODE does not set the QL's windows to their boot-up locations: this could be potentially disastrous while a program was running. To obtain the Mode4 or Mode8 window settings the WMON and WTV commands from *Super Toolkit 2* can be used.

MOVE #chan

distance

	GRAPHICS COMMAND
#chan	A valid screen channel number
distance	A number of graphics units

When the QL was being developed schools in particular were interested in the concept of turtle graphics pioneered by Samuel Papert. The idea was that children learnt basic programming skills from writing programs to control an imaginary "turtle" on the screen (which was, in reality, the graphics cursor). SuperBasic incorporates some turtle commands that can be used interchangeably with the more conventional drawing and trigonometrical keywords. Turtle commands refer to the graphics scale when determining distance: the default graphics scale divides the height of a window by 100. See the SCALE command for further information.

A turtle's orientation is established using the TURN command, its ability to leave a trail of ink pixels by the PENDOWN command. With a positive parameter, MOVE makes the turtle proceed forwards; a negative parameter makes it proceed backwards.

The difference between TURN and the analogous conventional command LINE TO is that it requires only one parameter to indicate the distance to travel whereas LINE TO needs two parameters to indicate a location. Which is better depends on the circumstances.

MOVE_MEMORY source

TO target, size

MOVE_MEMORY size

source TO target

[Turbo Toolkit]

	MEMORY MANAGEMENT COMMAND
source, target	Memory locations
size	Number of bytes

It can be extremely advantageous on occasions to move chunks of memory around, although the operation has obvious dangers if the target location for a move is already occupied by something important to Qdos. Generally, *Turbo Toolkit*'s MOVE_MEMORY is used within reserved memory areas and the screen map. The command has a very wide range of uses from generating special effects on the screen to sorting database indexes at high speed.

The following program is a simple illustration of how the screen map can be manipulated directly to produce a special effect, in this case magnifying the top half of the screen display to twice its height. To understand the routine better you need to be aware that the QL begins its screen map at memory location 131072 and that each line on the screen occupies 128 bytes, no matter what screen resolution is in effect.

```
100 DEFine PROCedure expand
110 FOR x = 127 TO 1 STEP -1
120  source = 131072 * x * 128
130  target = 131072 * x * 256
140  MOVE_MEMORY 128, source TO target
150  MOVE_MEMORY 128, source TO target - 128
160 END FOR x
170 END DEFine expand
```

MRUN filename

	PROGRAM FILE COMMAND
filename	A valid filename, e.g. "mdv1_overlay1"

The SuperBasic LRUN command automatically wipes out any existing program lines from the QL's memory before loading and running a new program. It can be useful to load extra program lines from a file without deleting what is already there and without halting program execution, and this facility is provided by the MRUN command. It shares an identical syntax with LRUN. Care needs to be taken with line numbering to avoid existing lines being overwritten by new lines with the same line number (unless, of course, this is what is intended). When MRUN is used at the command line the QL begins execution from the first SuperBasic line, even if this was not one of those loaded by MRUN. If MRUN is used within a program, program execution continues from the command following the MRUN command.

MRUN could be used in a main program that calls a succession of merged files, all of which overwrite each other, in order to produce a sort of "overlay" effect. The practical difficulties are daunting, though, particularly with ensuring that each overlay had exactly the same number of lines with exactly the same range of line numbers. The effort may have been worthwhile before the widespread use of memory expansion cards and before *SuperCharge* and *Turbo* were available, but is now only applicable to very special circumstances.

NET station

NETWORK COMMAND

station	An integer between 1 and 63

The QL was quite advanced in having a network protocol built into the standard model. (Contrast this with the completely different case of MS-DOS computers that have to have extra hardware added and system memory appropriated in order to communicate with each other.) Sadly, the Qdos network routines built into the QL's roms were crude and incomplete. In order to make any use whatever of networks it is a good idea to purchase Super Toolkit 2 for each computer on the net.

To give a QL a network presence, the NET command is used followed by a parameter representing a unique station identifier. If the network only connects two QLs they can share the same number, usually the default of 1.

Once a network is running it appears to the system as just another device. Channels can be attached to the device using a simple protocol. The following lines allow a document to be printed using a printer attached to a networked QL designated as a file server (see the FSERVE command). Note that the syntax is only available with *Super Toolkit 2* running on both computers.

```
100 NET 5: REMark use a unique identifier
110 OPEN #3, n2_ser1: REMark link channel to printer on file server (NET 2)
120 PRINT#3, flp1_mydoc
```

A more complete understanding of running networks is printed in the concepts section of the *New User Guide.*

NEW

PROGRAM COMMAND

The NEW command simply wipes the QL's program memory area clean and shuts all channels other than 0, 1 and 2, the default screen channels. It does not, however, change their settings back to what they were on boot-up.

NEXT ident

STRUCTURE CONTROL COMMAND

ident	A loop identifier, e.g. "X" in "FOR X = 1 TO 5" or "REPeat X"

NEXT has been embroiled in a contest with the END FOR command throughout the life of the QL. Programmers coming to SuperBasic from one of the other Basic dialects have traditionally used NEXT to end FOR...NEXT loops, but this habit is not encouraged on the QL even though the interpreter allows it. In SuperBasic, NEXT has special features that can be of great value. Firstly, it can indicate a premature end of an iteration of a loop, and secondly it can identify what has been called a loop epilogue.

Whenever it is not necessary to complete a full cycle of a loop, NEXT can be used along with an IF or SELECT ON structure to force a new cycle to begin. For instance, were we to examine each element of an array and only act on elements greater than 100 the following loop structure could be used:

```
100 FOR element = 1 TO DIMN(numbers)
110 IF numbers(element) <= 100 : NEXT element
```

```
120 REMark : The rest of the code.......
...
250 END FOR element
```

The same principle can be used with a REPEAT loop. Loop epilogues, however, are of value only with FOR..NEXT structures. The idea is that there might be some code that is only carried out if a FOR...NEXT sequence is completely exhausted; if the loop ends prematurely with an EXIT the code would not apply. SuperBasic uses an unconditional NEXT and an END FOR to allow this to happen. Using the example array above, we might want to exit the loop as soon as we find a value greater than 1000 and print a warning should there be no such values in the array. Here is the code:

```
100 FOR element = 1 TO DIMN(numbers)
110  REMark : Loop contents go here...
...
200  IF numbers(element) > 1000 : PRINT "1000+ value found" : EXIT element
210 NEXT element
220  REMark : The epilogue begins here.
230  PRINT "No 1000+ values found"
240 END FOR element
```

The inference from this variant of the FOR...NEXT structure is that END FOR is the correct way to terminate all FOR...NEXT loops. Purists might have a point: whenever NEXT ends a loop the interpreter will remain on the look out for a matching END FOR, occupying memory and slowing down interpretation for no good reason.

NFS_USE device_id prefix, prefix prefix, prefix...

NETWORK DEVICE HANDLING COMMAND

device_id	A valid device identifier such as "mdv" or "flp"
prefix	A valid prefix to a networked filename, such as "n2_flp1_" (Up to eight different prefixes can be assigned with one command)

NFS_USE is closely related to FLP_USE. It allows device drivers to be redirected to point to other devices made available to a network. Just as "FLP_USE mdv" indicates that references to "mdv1_" and "mdv2_" should be taken to mean "flp1_" and "flp2_", the "NFS_USE mdv" command indicates which device represents mdv1_, mdv2_ and so on up to mdv8_. On a network of 2 QLs, the remote one being a file server, four microdrives and a distant disk drive could be referenced with the command:

```
100 NFS_USE mdv, n1_mdv1_, n1_mdv2_, n2_mdv1_, n2_mdv2_, n2_flp1_
```

This allows the local QL to refer to its own microdrives as mdv1_ and mdv2_ and the distant floppy disk as mdv5_.

NXJOB(job_id top_job_id)

TASK CONTROL FUNCTION

job_id	The name or identification number of a task
top_job_id	The identification number of the owning task

All tasks are owned by other tasks except for SuperBasic, which always has an identification value of 0. NXJOB returns the identification value of the next job in the job tree headed by the given "top job". This is a very specialist function of value only to machine code programmers (and advanced *Turbo* users) and thus lies outside this Guide.

OJOB(job_id)

TASK CONTROL FUNCTION

job_id	The name or identification number of a task

OJOB is a function to return the owner of a particular task. Thus NXJOB(this_job, OJOB(this_job)) will find the next job in the list to the one currently pointed to. It is unlikely that SuperBasic programmers will use this function.

THE NEW USER GUIDE

SECTION TWENTY ONE

KEYWORD INDEX

This month in the Keyword Index, Mike Lloyd starts with ON and finishes, appropriately for the festive season, with a PARTYP.

ON x GOSUB line1, line2, line3, line4 ...
ON x GOTO line1, line2, line3, line4 ...

LOW-LEVEL PROGRAM STRUCTURES

x	A numeric variable
line1, line2, etc.	Valid line numbers

ON...GOSUB has honourable antecedents back through early Basic dialects to second generation languages such as Fortran. The idea is that the interpreter could be sent to various parts of a program depending upon the value of a variable. The programmer identifies the first line number of each subroutine in a comma-separated list. Prior to reaching the ON... GOTO or ON... GOSUB line a numeric variable is manipulated to point to one of the numbers in the list (ie if X equals 4, the fourth line number is selected from the list). Whether the interpreter returns from the jump depends upon whether GOTO or GOSUB is used (refer to these keywords in an earlier *New User Guide* for guidance). Even though SuperBasic avoids the obvious trap by updating the list automatically when the program is renumbered, these commands remain awkward ways of branching a program and are best avoided. It is generally preferable to divide programs into procedures and functions and forget about GOTOs and GOSUBs in all their variations.

OPEN #chan, name
OPEN_IN #chan, name
OPEN_NEW #chan, name
OPEN_OVER #chan, name
[Super Toolkit 2]

DEVICE HANDLING COMMANDS

#chan	A valid channel number
name	A filename, console definition, screen definition, printer definition or network definition.

Most Basics are interested only in opening files, which would make the OPEN family simple file handling commands. Qdos brings together printers, file storage devices, networks and screen windows with the unifying concepts of "devices" and "channels". Imagine a channel as being one of a line of tubes. The far end of each tube can be attached to a screen, printing, file storage or network

device. Once a tube is connected the size, shape and type of the device at the other end are of little interest; the attributes of the tube itself are much more important. Some tubes allow one-way traffic (typically when attached to printers or files), while others permit two-way communication (which applies to consoles but not to screens).
The OPEN command tries to implement two-way communication between the QL and the device, although the attributes of the device may limit this. OPEN_IN is used only for opening a file in "read only" mode. This is a safety measure to prevent accidental file corruption by incorrectly trying to PRINT to it. The OPEN_NEW variant is also used only with file storage devices. It opens a new file with the given name before channeling output to it. OPEN_OVER will attempt to open a new file with the given name, but if a file with that name already exists it will be replaced and overwritten by the new one.
There are many errors which can be generated when assigning channels to devices, but SuperBasic's answer is simply to halt processing with an error message. *Super Toolkit 2* is equipped with a set of functions such as FTEST() and FOPEN() which test to see if OPEN is possible without bringing a program to a halt. *Turbo Toolkit* provides the same facilities with DEVICE_STATUS().
Once a channel has been connected to a file output is passed along it using PRINT#chan. Input is obtained from the channel using INPUT#chan or the INKEY$#chan() function. In this way information can be passed to a screen, the network, a printer and a file using the same set of commands. A simple procedure to echo everything passed to a screen to the printer (or to a file) is:

```
100 OPEN#5, ser1 : REMark connect channel 5 to the printer
110 ECHO "Here is a line of text"
120 DEFine PROCedure echo (text)
130 PRINT text
140 PRINT #5, text
150 END DEFine echo
```

Note that channels #0, #1 and #2 are already open to screen consoles when the QL is booted up and that channel #0 allows SuperBasic commands to be typed in. If channel #0 is reassigned its special characteristics are lost until the QL is rebooted. In Turbo programs channel #0 is not recognised by the interpreter as a source of direct commands.
The syntax for non-file devices will be dealt with in the Concepts section of the New User Guide.

OPEN_DIR #chan, name
[Super Toolkit 2]

FILE HANDLING COMMAND

#chan	A channel opened to a file storage device (eg a microdrive)
name	The name of a device (eg "flp1_")

One of the earliest gripes about SuperBasic was that it was difficult for a program to read and manipulate the files on a microdrive cartridge. The only method available seemed to be to open a file and write a directory listing to it (eg OPEN_NEW#3, flp1_dirlist: DIR#3: CLOSE #3). Once populated and closed, the file could be re-opened in read-only mode and each line read with an INPUT statement. Super Toolkit 2 came to the rescue with OPEN_DIR, but the documentation made scant reference to it and failed to demonstrate how it could be used.
The OPEN_DIR command actually opens a read-only channel directly to the file map at the start of every microdrive cartridge and floppy disk (if floppy disks can be said to have a start). Carriage return codes only occur by accident in the file map so characters must be read one by one with the INKEY$() function. Fortunately there are a fixed number of bytes per file entry, so a simple loop will gather discrete information about each file. The first 16 bytes of the map refer to the medium itself and can be skipped over. Thereafter each block of 64 bytes refers to a given file. The following routine quickly prints the bytes to the screen. The full decode lies beyond this Guide, but the file name will be obvious at the start of each line.

```
100 CLOSE #6: REMark only if it is already open...
110 OPEN_DIR #6, FLP1_
120 FOR x = 1 TO 16: junk$ = INKEY$(#6)
130 REPeat loop
140 entry$ = ""
150 FOR x = 1 TO 64: entry$ = entry$ & INKEY$(#6)
160 PRINT entry$
170 IF EOF(#6): EXIT loop
180 END REPeat loop
```

OPTION_CMD$()
[Turbo Toolkit]

COMPILED PROGRAM PARAMETER FUNCTION

The Turbo compiler provides many ways in which tasks can pass information to each other. One of the simplest is OPTION_CMD$(). Programmers can call any compiled program using the EXECUTE family of commands and add after the program name a string containing anything which might be useful. This string can be read within the compiled program using the OPTION_CMD$() function. The calling line might be:

```
EXECUTE flp1_task "mdv1_workfile, backup, 7"
```

Within the flp1_task code there might be written a function to strip away the text separated by commas to assign values to three variables. The function could be called with the line:

```
140 filename = DECODE(OPTION_CMD$, 1)
150 process = DECODE(OPTION_CMD$, 2)
160 refnum = DECODE(OPTION_CMD$, 3)
```

Remember, what you place in the string passed to the executable program and what you do with it within the executable is up to you.

OVER #chan, value

SCREEN ATTRIBUTE COMMAND

#chan	(Optional) A channel linked to a screen window or console.
value	Minus one, nought or one.

The OVER command plays clever games with the QL's screen map to produce a limited variety of special effects. OVER 0 is the default and everything is printed in the expected INK and PAPER colours. OVER 1 has no effect on INK colours but makes the PAPER colour "invisible", allowing existing backgrounds to be seen directly beneath the text being printed. This can have the effect of making text difficult or impossible to see. If you have a red background and set INK to red and PAPER to black, text will normally be clearly visible. Following an OVER 1 command, though, the black background will not be printed, so any further PRINT statements will put red ink on the existing red background. OVER 1 has its uses when mixing text and graphics because lines beneath text are not obscured by the strip.
OVER -1 is a different effect altogether. Once OVER -1 has been issued any text or graphics printed in the affected window take their colour from a combination of the existing background colour and the current INK colour. Should you need to know, this is done by a bitwise Exclusive OR. The non-technical need bear in mind only that OVER -1 is a way of ensuring that no matter what background colours are in use that text or graphics placed over them will adjust their colour so that they remain visible. Different degrees of contrast can be achieved by selecting different INK colours, with white (INK 7) providing the most visible contrasts and black (INK 0) being entirely invisible.

PAN #chan, pixels, area

SCREEN MANIPULATION COMMAND

#chan	(Optional) A valid screen channel
pixels	An integer representing the number of pixels to move (can be negative)
area	(Optional) The part of the screen to pan (3 = cursor line, 4 = cursor line to right of cursor)

Panning is what film directors do when they want the camera to sweep across a scene, and the SuperBasic PAN command can be used to give that illusion. What PAN actually does is to move screen pixels horizontally, replacing those which "fall off" the screen on one side with PAPER-coloured pixels on the opposite side. To emulate camera panning the PAN command has to be called in a loop. The panning effect can be limited by an optional third parameter to the cursor line (with a value of 3) or just that part of the cursor line which lies under or to the right of the cursor (with a value of 4).
It would appear that PAN and SCROLL share a single parameter set in that SCROLL's optional screen areas are 1 (above the cursor line) and 2 (below the cursor line). If it was ever intended that PAN and SCROLL would take any of the four values the QL's ROM was never developed in that direction. SCROLLing the cursor line vertically and PANning the top and bottom parts of a screen window seem to be excellent ideas which programmers unfortunately have to implement for themselves.
The quickest way of moving part of a window is to locate another window over the top of it and panning that. The following code gives an example:

```
100 MODE 4: WINDOW 512, 256, 0, 0 : PAPER 0 : CLS
110 FOR x = 1 TO 200: PRINT FILL$(CHR$(32 + RND(100)), 5); "";
120 FOR x = 1 TO 100: minipan 300, 20,80, 40,2
130 DEFine PROCedure minipan (wide, high, xpos, ypos, pixels)
140 OPEN#8, scr_: WINDOW#8, wide, high, xpos, ypos
150 PAN#8, pixels
160 CLOSE#8
```

```
170 END DEFine minipan
```

PAPER #chan, main, sub, stipple

SCREEN HANDLING COMMAND

#chan	(Optional) A screen channel number
main	An integer between 0 and 7 (for solid colour) or between 0 and 255
sub	(Optional) An integer between 0 and 7
stipple	(Optional) An integer between 0 and 3

The PAPER command sets the background colour for the screen and affects the colour underlying printed characters and the screen colour used by CLS. The complete range of colours, including all the stipple combinations, can be represented by a single value between 0 and 255 that PAPER can take as a parameter. Alternatively, colours can be built up from three parameters, the first two determining the colour combination and the third the stipple. Thus a red and cyan checkerboard stipple is obtained by PAPER 2, 5, 3.

PARNAM$ (param) [Super Toolkit 2]

PARAMETER HANDLING FUNCTION

param	The location of a parameter in a list of parameters.

In SuperBasic, parameter names take the place of the true names of values. This means that any change to the value represented by a parameter within a procedure will survive after the procedure call is complete. There are times when it might be useful to know exactly which variable has been passed as a parameter to a procedure. We might have a labelling procedure for fields in a database, for example, which prefixes a title to the field it is printing. PARNAM$ would do this task:

```
100 surname$ = “SMITH”: forename$ = “JANE”
110 LABEL surname$: LABEL forename$
500 DEFine PROCedure LABEL (field$)
510 LOCal result$
520 IF PARNAM$(1) = “surname$”: result$ = “Surname is “
530 IF PARNAM$(1) = “forename$”: result$ = “Forename is “
540 PRINT result$ & field$
550 END DEFine LABEL
```

PARSTR$(varname, paranumber) [Super Toolkit 2]

PARAMETER HANDLING FUNCTION

varname	the “local” name given to a procedure’s parameter
paranumber	the location of the parameter in the parameter list

Even Tony Tebby describes the PARSTR$ function as untidy. It is certainly able to confuse. Its two arguments are a parameter name and its location in the parameter list. It returns either the “true” name of the variable represented by the parameter or, if that variable is a string, the value of the string. It must have been useful for something.

PARTYP(varname) [Super Toolkit 2]

PARAMETER HANDLING FUNCTION

varname	the “local” name given to a procedure’s parameter

Another way of finding out more information about a parameter is provided by the PARTYP() function. When given a parameter name it responds with a value of 0 if the parameter is null, 1 if the parameter represents a string, 2 if it represents a floating point number and 3 if it represents an integer. This is particularly useful because parameter names do not have to follow the same variable naming rules elsewhere in SuperBasic: a parameter without a dollar sign suffix can represent a string. Indeed, because of SuperBasic’s type coercion a parameter could represent a string in one procedure call and an integer in the next.

THE NEW USER GUIDE

SECTION TWENTY TWO

KEYWORD INDEX

This month in the Keyword Index, Mike Lloyd starts with PARUSE(varname) of Super Toolkit 2, and ends with POSITION(#chan) - via PI.

PARUSE(varname)

[Super Toolkit 2]

	PARAMETER HANDLING FUNCTION
varname	the "local" name given to a procedure's parameter

SuperBasic's flexibility with parameters and coercion is extremely welcome, but can occasionally cause a few identification difficulties that can be resolved by PARTYP and PARUSE. The latter function reveals what category of variable is represented by a given user-defined procedure or function parameter. One of three values is returned:

0 = null
1 = variable
2 = array

If the user-defined procedure "CENTRE" is called with a single parameter locally referred to as TEXT the following code within the procedure definition will reveal whether CENTRE has been called with an array or a normal variable or no parameter at all:

```
225 param_usage = PARUSE(text)
```

PAUSE value
PAUSE #chan

[Minerva only]

	KEYBOARD CONTROL COMMAND
value	(Optional) A floating point value
#chan	A valid input channel number

The PAUSE command simply postpones the execution of the next statement by a given amount of time expressed in fiftieths of a second. This assumes the UK's 50 Hz power supply. With a 60 Hz power supply a PAUSE value of 1 will delay the interpreter for one-sixtieth of a second. If a keypress is detected during a PAUSE the delay is abandoned immediately. If no parameter is given to the command then PAUSE will wait forever for a keypress to occur.

PAUSE is perhaps of most value within a loop containing KEYROW functions (the alternative INKEY$ has its own pausing mechanism and INPUT waits for an ENTER regardless). It is important to flush the keyboard buffer before a PAUSE because otherwise any keypresses stacked up in the buffer will terminate the PAUSE unintentionally. Use a dummy KEYROW() function call before the PAUSE command to achieve this.

Be warned that PAUSE will not work in compiled SuperBasic programs because they do not have the special #0 channel through which direct keyboard input is obtained. Minerva owners can provide PAUSE with a channel number that is listening out for a keypress. Turbo users can try using the COMMAND_LINE directive to overcome this problem, or more simply circumvent it by rewriting compiled programs to avoid PAUSE altogether.

PEEK(address)

PEEK_W(even_address)
PEEK_L(even_address)
[Minerva]
PEEK(location)
PEEK_W(location)
PEEK_L(location)

PEEK_F(even_address)
PEEK$(address, bytes)
[Turbo Toolkit]

MEMORY HANDLING FUNCTIONS

address	A valid integer within the range of the memory capacity of your QL.
even_address	A valid even integer within the range of the memory capacity of your QL.
location	EITHER a valid integer
OR	an offset with the format "\\value"
OR	a vector and offset with the format "\vector \format"
bytes	An integer representing the number of bytes to return.

SuperBasic programmers occasionally need directly to examine the contents of memory addresses, a task achieved by the PEEK family. The QL's memory comprises a huge number of addresses each containing an eight-bit (or one byte) value. If you know the address of a byte of memory, PEEK will return its contents. Because the QL's cpu can handle 16-bit and 32-bit values internally SuperBasic contains PEEK_W to return a 16-bit value and PEEK_L to return a 32-bit value. It is an unviolable convention that the first byte of a larger value begins at an even address, but it is up to the programmer to ensure that the argument passed to the function meets this requirement.
When memory contents are discussed a number of terms are used to describe the same type of value. The following table translates between alternatives:

4-bits	half-byte	1 nibble
8-bits	1 byte	1 byte
16-bits	2 bytes	1 word
32-bits	4 bytes	1 long word

Minerva owners are over-blessed with extensions to the functionality of the PEEK functions. Firstly, the restriction on using even addresses for words and long words is removed, which allows PEEK_W and PEEK_L to be used as fast methods of grabbing multi-byte values stored by a program anywhere in reserved memory. For programmers aware of the layout of the QL's system variables the other Minerva enhancements are extremely valuable because they provide a quick, foolproof method of reaching the appropriate offset from the base of the system variables area. The x = PEEK_W(\x \y) expression accesses the offset from the offset! For instance, the location of the Qdos channel table is stored as a long word beginning at the 48th byte from the start of the system table. Each entry in the channel table is 40 bytes long, so PEEK_L(\48 \40*chan) will return the required value. (Please note that Turbo Toolkit provides equivalent functionality with the BASIC() series of functions.)

The Turbo Toolkit team felt that SuperBasic's PEEK family were a little restrictive and added PEEK_F, which assumes that the address is the first byte of the location of a floating point number. All floating point numbers, no matter their decimal size, are held internally as 6-byte sequences: PEEK_F decodes the six bytes into their decimal equivalent (this happens whether or not a genuine floating point number has been stored at that location). Like other multi-byte PEEK functions, PEEK_F must be passed an even address value.

The PEEK$ function, also available in the Turbo Toolkit, can be extremely valuable when moving chunks of memory around. PEEK$ assigns the Ascii values of a declared portion of memory to a string variable. Thus a memory location holding the value 65 will be translated to an "A" in the string it is copied to. PEEK$ can be used to sort, search for and retrieve fixed-length entries in a database managed in reserved memory rather than held in conventional variables. PEEK$ can also be used to manipulate the screen map which begins at memory address 131072 and takes up the first 32K of RAM. Try the following:

```
100 FOR x = 131072 TO 137472 STEP 128
```

```
110 Line_of_Pixel$ = PEEK$(x, 128)
120 POKE$(x + 6400, Line_of_Pixel$)
130 END FOR x
```

PENDOWN #chan
PENUP #chan

TURTLE GRAPHICS DIRECTIVES

#chan	(Optional) A valid screen channel

The QL's graphics cursor simultaneously responds to conventional graphics commands and to turtle graphics commands, which gives the QL programmer the luxury of being able to use whichever is most convenient for the task in hand. Turtle graphics make much of turning and moving specific distances, whereas conventional graphics rely on co-ordinates in the style of map references. PENDOWN ensures that the "turtle" will leave an ink-coloured traces during subsequent manoeuvres. PENDOWN followed by a MOVE command is the equivalent of a LINE x1, y1 TO x2, y2 command. PENUP is used to ensure that the "turtle" leaves no trace during subsequent MOVE commands, which can be replicated in conventional graphics commands by the LINE x1, x2 command (which simply relocates the graphics cursor at the given co-ordinates).

PI

TRIGONOMETRICAL FUNCTION/CONSTANT

PI is a system constant implemented as a function (hence the format PI() will not produce an error). Pi is the ratio between the diameter of a circle and its circumference, roughly 23/7. Pi is an irrational number which can never be precisely expressed in decimal terms, which is why it is convenient to have a function which returns a relatively accurate value for PI rather than have programmers try to remember one.
SuperBasic makes much of the concept of radians, each radian being the length of a circle's radius bent around its circumference. Slightly more than 6 radians are required to make up a full circle; more precisely, 2*PI radians are needed. This should waken distant memories of the 2*pi*r formula of school maths. [A more comprehensive review of Superbasic trigonometry belongs in the Concepts section of the New User Guide.]

PJOB(job)
[Super Toolkit 2]

TASK MANAGEMENT FUNCTION

job	The job identifcation number or name

PJOB is a function that returns the priority of an independent task or job running in Qdos. Qdos splits processor time proportionately between all the genuinely multi-tasking programs (when tasks are run concurrently they may all be loaded into memory simultaneously but only one of them will be active). The priority which can be assigned to a task is represented by a value between 0 and 128, with 32 being the default and 0 representing no time whatsoever. If three tasks are loaded all with a priority of 32 they will receive exactly one third of the processor's attention over a given period of time.

POINT #chan, xpos, ypos, xpos1, ypos1, xpos2, ypos2, ...
POINT_R #chan, xpos, ypos, xpos1, ypos1, xpos2, ypos2, ...

GRAPHICS PROCEDURE

#chan	(Optional) A valid screen channel. Defaults to WINDOW #1.
xpos, ypos	A pair of graphics co-ordinates

At its simplest, with one set of co-ordinates, the POINT command lights up a single pixel in the current INK colour. Multiple sets of co-ordinates are allowed but rarely used; it is usually better to put the POINT command in a loop. POINT always plots pixels relative to the origin of the graphics grid that by default coincides with the bottom left corner of the window. Its location can be changed using the SCALE procedure. POINT_R takes its bearings from the current location of the graphics cursor. A surprisingly effective starry sky can quickly be obtained by repeated, random POINT statements:

```
100 PAPER 0: INK 7: CLS
110 FOR x = 1 TO 60: POINT RND(180), RND(100)
```

POKE address, byte
POKE_W even_address, word
POKE_L even_address, longword

POKE location, byte, byte1, byte2, ...
POKE_W location, word, word1, word2, ...
POKE_L location, longword, longword1, longword2...
[Minerva]

POKE$ address, string
[Turbo Toolkit]

MEMORY HANDLING PROCEDURES

address	Any memory location address, odd or even
even_address	An even-numbered memory location address
byte	An integer between 0 and 255 (unsigned) or between -126 and 127
word	An integer between 0 and 65,535 (unsigned) or between -32,766 and 32,767
long-word	An integer between 0 and 16,777,215 or between -8,388,608 and 8,388,609
string	A string representing a sequence of Ascii values

POKE is the opposite of PEEK in that it sends values to memory addresses rather than extracts them. PEEK is wholly benign and can be used across the entire range of memory addresses without fear. POKE, on the other hand, is restricted to ram addresses only (you cannot POKE a new value into a read-only memory address) and has the potential to bring down your QL. You might, for instance, declare an area of reserved memory with the RESPR command and quite harmlessly start to POKE values into the reserved memory addresses. Qdos will not warn you, however, should you accidentally send a value into a byte outside the reserved area.
The Minerva rom has tinkered with POKE to a much lesser degree than with PEEK, with the effect that you can now update a series of values into successive addresses with a single call of the POKE command. Turbo Toolkit takes this principle one step further and allows you to declare a string of Ascii values which can then be placed into sequential memory addresses. In both cases you merely declare the starting address. One of Minerva's interesting but completely useless side effects is that you can now use the POKE command with an address but no value to put in it.
Should you be tempted to POKE values into the QL's system tables area you should always use one of Turbo's BASIC() or Minerva's PEEK(offset) functions to determine where the values are going to be placed. Qdos has an alarming habit of relocating bits of the system tables without warning.

POSITION(#chan)
[Turbo Toolkit]

FILE INPUT/OUTPUT FUNCTION

#chan	(Optional) A channel opened to a file or pipe

When you are writing at a low level to a re-opened file it can be convenient to be aware of exactly where in the file the output is being placed. Qdos sees files as streams of one-byte-wide Ascii values and it has no understanding of lines and paragraphs in wordprocessed text or records and fields of database files. Rather like its graphics cursors, Qdos keeps track of a pointer which can move through the file. When a file is opened the pointer points to the first byte in the file. As data is read from the file or written to it the pointer automatically increments itself. Programmers who wanted to overwrite a specific portion of a file could do so by moving the file pointer forward with the INPUT(#chan) function and then writing to the file with PRINT #chan. The POSITION and SET_POSITION keywords in the Turbo Toolkit now make this process easier to manage.

Turbo also implements pipes, or communication channels between tasks. A pipe can be likened to a file that only exists in memory and that has both a read pointer and a write pointer open simultaneously. As things are read from the pipe the remaining contents shuffle up towards the front of the file. In all other respects, pipes can be treated like temporary files. In normal use everything written to a pipe should be appended to the end of the file and everything read from the pipe should be taken in sequence from the beginning. In such cases there is little need to know anything about the actual location of the two pointers. However, if you are unsure about whether a pipe is getting full the POSITION() function could provide a warning that nothing further should be written to the pipe until something has been extracted from the other end. Advanced users can use POSITION() and SET_POINTER to shift the pointers up and down the pipe, perhaps to overwrite some previously-sent but unread information or to read data out of sequence.

Finally, although Qdos tries to treat all devices as though they were files there are some attributes of files which cannot apply to every sort of device. Network channels, printers and screens are potentially of infinite capacity and do not have a pointer. If you use POSITION() with any of the non-file devices other than pipes you will generate an error.

SINCLAIR QL WORLD

THE NEW USER GUIDE

SECTION TWENTY THREE

KEYWORD INDEX

This month in the Keyword Index, Mike Lloyd covers PRINT and PROCEDURE. Only two? PRINT is the Big One.

PRINT #chan, exp1, exp2, exp3, ...

	OUTPUT COMMAND
#chan	(Optional) A valid channel number (default is #1)
exp1, exp2	(Optional) String or numeric constants or variables
separators	The commas separating the parameters can be replaced by any of the following:

!(exclamation mark)Intelligent spacing
\(backslash)Force line feed
;(semicolon)No space between print expressions
,(comma)Tab 8 spaces
TO x   Place next print expression at the Xth character position on the line

PRINT is one of those commands that has a superficial simplicity which leads programmers to forget about its greater powers and subtleties. The first Basic command many people learn is PRINT "Hello World" and for many that is as far as it goes. PRINT is the simplest method of putting text onto the screen. To print a column of numbers (left justified) it is necessary only to write a quick loop along the lines of:

```
100 FOR x = 10 TO 99: PRINT x
```

You can print constants such as "HELLO WORLD" and 98.4 as easily as you can variables such as text$ and this_number. PRINT will also force the calculation of numeric expressions such as 25+7 and can concatenate (add together) text using the ampersand symbol (&). PRINT "Hello " & "World" produces exactly the same output as PRINT "Hello "; "World". Note that spaces within inverted commas are significant, but that spaces in other places in a PRINT statement do not form part of the output. Due to SuperBasic's coercion principles, PRINT "2" + "2" is significantly different to PRINT "2" & "2". The former coerces the text into numeric values and adds them together to produce the result 4, whereas the latter treats the expressions as text and so prints "22". PRINT with the null string ("") or with no parameter or separator at all forces a linefeed to be printed. Any number with more than 6 significant digits will be converted to scientific notation before being printed. If you want to format numbers with trailing zeros, comma-separated thousands, leading currency symbols, etc., you need to write your own function or use the PRINT_USING command provided in *SuperToolkit II*.

Expressions can be mixed in the same PRINT statement by separating each expression with a "print separator". Print separators are punctuation marks with special significance for the PRINT command. Commas, for instance, tabulate output into columns each 8 characters wide. The TO separator also tabulates output, but it allows you to declare at which character column the next print item will appear. PRINT TO 30; "Hello World" places the "H" of "Hello" at the thirtieth character position on the current line. PRINT TO will not backtrack to an earlier point in a line and will not print beyond the width of the current device. Fractional parts (such as PRINT TO 20.4) are ignored.

Beware of the natural inclination to place a comma between the TO clause and the next print item: it will force a tabulation to the next 8-character column, which removes the advantage offered by TO.

The exact size of a single character position depends upon the current screen mode and the values used in any CSIZE command. The backslash character forces a linefeed to be printed, so that the command PRINT "Hello " \ "World" places "World" beneath "Hello". The semi-colon places the next print item immediately following the last item, so that PRINT "Hello" ; "World" prints the two words together without any intervening space. The most complicated print separator is the exclamation mark, which produces what Sinclair called the "intelligent space". In the middle of a line, the intelligent space is no different from any other space. PRINT "Hello" ! "World" produces the output "Hello World". At the end of a line, however, where there is insufficient room for the second item a linefeed will be generated. If the exclamation mark would produce a space at the beginning of a line it is suppressed.

Punctuation

Rather against the grain of normal SuperBasic syntax and the standard rules of English, PRINT statements can include an uninterrupted series of punctuation marks, such as PRINT ,,,,, "Hello".
One of the nice things about PRINT statements is that they can end with a print separator which determines what will happen when the next PRINT command is reached. Where there is no separator at the end of a command a linefeed is entered. Adding a separator adjusts the position of the next print item according to the rules just explained. The number-printing loop mentioned earlier could be rewritten to place the values into a number of columns, as follows:

```
100 FOR x = 10 TO 99: PRINT x ;
```

Note that this feature is device-dependant: a PRINT #2 command has no effect on the next PRINT #5 command, for instance. There is a quirk to the way the exclamation mark works when it is used at the beginning or end of a PRINT statement. It will only do its task properly if there are exclamation marks at the end of the previous PRINT statement and at the beginning of the current statement. The terminating exclamation mark can be replaced by a semicolon if required. A single exclamation mark in these circumstances is not enough.

The PRINT command is not limited to placing text onto the screen, but is also capable of printing to files, the network, and to the printer. This is one of the consequences of Qdos's principle of treating all devices as similarly as possible. However, for newcomers to the QL and to SuperBasic, it leads to an unexpected problem: how on earth do you make the printer work? All previous Basic dialects used the LPRINT command which is conspicuously absent in SuperBasic. The answer, of course, is to open a channel to the printer port (eg OPEN #5, ser1) and then to direct PRINT statements to that channel. PRINT is also invaluable when writing information to files, although there are now alternatives offered, such as Super Toolkit's BGET and BPUT, which can be more appropriate to particular circumstances.

The Qdos code that handles all PRINT requests is also used by the INPUT command. Prompt and answer formatting which is typical of INPUT activity can be undertaken using the same separators as are available to the PRINT statement. Incidentally, all of the low-lying Qdos printing code is substantially rewritten by the Minerva rom and by the Lightning and Speedscreen utilities. None of the changes affect the syntax of the language, but there are some side effects other than improved screen handling of which programmers need to be aware.

Despite the apparent flexibility of the PRINT command it has some serious drawbacks and limitations. The first is that because Qdos does nothing to optimise screen handling the PRINT command is extremely slow, even compared with the lowly Spectrum. The cure is to replace the Sinclair rom with Minerva, or to invest in Digital Precision's Lightning, or to purchase a Gold Card from Miracle. The PRINT syntax has an unusual limitation in that exact screen locations cannot be defined within the PRINT statement itself; this is instead the preserve of the AT command. This rather eccentric arrangement is probably a compromise imposed by the principle of treating all devices identically.

Qdos screen handling is always pixel-based, which means that there is no function to read what character has been placed at a particular character position within a window. The advantages that offset this difficulty are that text can be positioned pixel-perfect using the CURSOR command and that various character sizes can be used within a single window. Nevertheless, the loss of the Spectrum's useful SCREEN$ function has been particularly hard on SuperBasic arcade game writers.

The output from PRINT commands is always left-justified, which immediately imposes a problem for programmers anxious to right-justify numbers or to place all decimal points beneath each other in a column of numbers. The user-defined procedures R_JUST and D_JUST listed below go some way towards redressing these weaknesses, but Super Toolkit users can turn to the PRINT_USING procedure described elsewhere.

```
100 DEFine PROCedure R_JUST (text, charpos)
110 PRINT TO charpos - LEN(text); text
120 END DEFine R_JUST
200 DEFine PROCedure D_JUST (text, charpos)
210 PRINT TO charpos - ("." INSTR text); text
220 END DEFine D_JUST
```

PRINT_USING #chan, mask, item1, item2, item3, ...

[Super Toolkit II]

	OUTPUT COMMAND
#chan	(Optional) A valid output channel
mask	A sequence of strings, print separators and field definitions (see below)
item1, item2,..	A list of variables, one for each of the field definitions (see below)

The PRINT_USING command can appear quite daunting at first sight, but it overcomes a number of the weaknesses of the native SuperBasic PRINT command. Its role is to format numeric output according to the standard rules of displaying numeric values, although it can also mix text and numbers together if required. Perhaps its most useful attribute is that it can inhibit the QL's enthusiasm for scientific notation. The easy parts of the PRINT_USING syntax are the conventional channel number and the trailing list of output items. The difficult concept to master is that of the output mask, which is best thought of as providing a template into which the output items are placed. The mask can contain ordinary text strings and should have at least one field definition (although this is technically optional there is no point in using the command unless there is a field definition for it to work on).

At its very simplest, PRINT_USING might be asked to format a number so that it is right justified and has commas separating the thousands:

```
PRINT_USING "###,###,###.##", 1234.5
```

The output will be 6 spaces followed by "1,234.50". The leading spaces ensure that correctly-formatted columns of numbers can easily be printed. The hashes in the format mask each represent a numeral and together form a "field". It is essential to realise that the total width of the field represents the total number of character positions that will be occupied by the associated output item. If the output item formats to more characters than are available in the field an error is produced.
Fields are separated in the mask by at least one space (or the backslash newline character). In order to print two numbers in adjacent columns, the following command might be used:

```
PRINT_USING "#,###.## #,###.##", 3456, 245.89
```

The items following the format mask would, of course, normally be variables.If the values represented currencies the appropriate currency symbols could be added to the front of each value:

```
PRINT_USING "$##,###.## = $$#,###.##", 450.25, 450.25 * dollar_rate
```

The dollar symbol is a special character which indicates that what next follows is the correct currency symbol. For dollars themselves this means that a second $ must be included. European currencies can be represented by "$DM", "$Fr", etc. The difference between a mask of "###,###.##" and "$###,###.##" lies in the location of the pound sign on the screen or page. With the former format the pound sign is always the first character in the column with any spaces placed between it and the first digit of the value. With the second format the pound sign is moved so that it is immediately to the left of the first digit of the value, pushing any leading spaces to the left of the pound sign. Generally, the latter format is to be preferred for neatness.

PRINT_USING format masks contain two more features of particular interest when printing currency values. The first is the asterisk special character which replaces all leading spaces in fields so that the output from PRINT_USING "**,***.**" , 980.40 would be "***980.40". This format is of most use in financial applications that involve the completion of cheques. The second feature provides three methods of showing negative numbers. If your accountant prefers negative amounts in brackets then the PRINT_USING mask should show them, as in "(###.##)". The brackets are automatically replaced by spaces for positive numbers. Otherwise, negative numbers can either be preceded by or followed by a minus sign that, again, has to be shown in the format mask:

"-#,###.##" or "$##,###.##-".

If the minus sign is replaced by a plus sign in the mask then the correct sign will always be displayed, otherwise the minus sign will be replaced by a space for positive numbers.

The QL is always over-anxious to use scientific notation, a weakness that PRINT_USING can overcome. On the other hand, PRINT_USING includes special characters for formatting any number using scientific notation. The rules are quite complicated: the mask must begin with a single hash followed by a decimal point and it must end with four exclamation marks representing the mantissa. The number of decimal places in the exponent is up to you. To reserve a character space for the sign a minus sign can be used for optional signing and a plus sign for compulsory signing, so a typical scientific notation mask looks like "-#.####!!!!", for which the output might be "4.2340E+02".In any format mask it might be necessary to include one of the special characters without intending its special meaning. This is achieved either by prefacing the character with the @ symbol or by placing it inside single or double quotation marks (of the opposite variety to those used at either end of the mask). For example, part numbers on a product database might be prefaced by the hash symbol:

PRINT_USING "@# ######", part_num.

Another special character is the backslash, which can be used within a mask with the same effect as when it is used as a print separator. If the backslash is needed to represent itself it should be preceded by @.

A final advantage offered by PRINT_USING is that the format mask is simply a string like any other and can be replaced by a variable, such as PRINT_USING #5, mask$, 78.25. This allows you to create masks at runtime, or perhaps allow your users to declare an output format or choose one from a menu.

PROCEDURE

[Turbo Toolkit]

COMPILER DIRECTIVE

The PROCEDURE keyword and its companion, FUNCTION, are two of the cleverest workarounds in the Turbo package. They appear only in compiled code and then only in EXTERNAL and GLOBAL declarations. Technically, PROCEDURE is, according to the *Turbo* manual, a numeric function which returns the amount of free memory remaining (the alternative function FREE_MEMORY is recommended for this task, however). Their job is to identify what would otherwise appear to the Turbo compiler to be a variable as the name of a user-defined procedure or function.

Every EXTERNAL statement has a GLOBAL opposite in some other compiled task. The two declarations allow a program in one task to call routines and refer to variables and arrays that belong to the other task. This arrangement permits code and data to be shared between modules. It would be possible, for instance, to create a library of procedures and compile them just once with a GLOBAL declaration to identify that they can be shared by other programs. Client tasks can then include an EXTERNAL declaration that effectively adds the contents of the global task to their own routines. With arrays and standard variables the statement syntax is easy, but procedure and function names posed a problem: how could the compiler distinguish between the variable VARNAME and the simple function FUNNAME, or between an array ARRAY(0, 0) and the procedure PROCNAME(x, y)? The answer was to precede the function and procedure names with the keywords FUNCTION and PROCEDURE, but the SuperBasic interpreter checks some syntax on entry and it rejects more than one item between commas in a list. The workaround is that FUNCTION and PROCEDURE are dummy simple functions that take no arguments. The SuperBasic interpreter thinks nothing unusual of them, but the Turbo compiler recognises them as keys to the true identity of the item that follows them. Ideally, a directive such as EXTERNAL varname, array(12, 4), PROCEDURE procname would have been preferred. Instead, SuperBasic dictates that the format should be:

EXTERNAL varname, array(12, 4), PROCEDURE, procname

What's a comma between friends?

THE NEW USER GUIDE

SECTION TWENTY FOUR

KEYWORD INDEX

This month in the Keyword Index, Mike Lloyd opens with PROGD$() and PROG_USE, and winds up with REFERENCE from Turbo Toolkit.

PROGD$()
PROG_USE
[Super Toolkit 2]

FILE HANDLING FUNCTION

The Sinclair QL's Qdos operating system was always meant to have awareness of file directories, but this was one of the casualties when the team were rushing to finish the product for its premature launch. Directories can be imagined as separate file storage areas on a single device. Normally, it is possible to use a given filename once on a single device such as a microdrive. With directories, filenames have to be unique only within one directory: on a device with four directories a filename can be used four times, once in each directory. Tony Tebby decided to rescue the incomplete Qdos code, finish it, and incorporate it into Super Toolkit 2. Its facilities have been expanded by Miracle Systems for their Winchester drive and for their more recent and more successful Gold Card. The QL's method of declaring directories is very different from that of PCs and, once it has been adjusted to, can even be thought slightly superior.

The key to directories on the QL lies in the use of the underscore to separate parts of the filename. In the simplest case, a complete filename comprises a device identifier ("flp1" or "mdv2"), an underscore and a name of one or more characters. However, further segments separated by more underscores can be added, such as "flp1_work_docs_backup_myletter". With Super Toolkit you are encouraged to give all executable files the same prefix, such as "flp1_exe_fastgame" or "flp1_exe_quill" and then set a system variable using the PROG_USE command to represent that initial portion of the full filename, for instance:

```
200 PROG_USE flp1_exe
```

Having done this, a Toolkit-equipped QL can launch any executable file with such a prefix to its name simply by referring to the final segment, so "flp1_exe_quill" can be executed with the command:

```
EXEC_W quill
```

even though the file itself has been given the longer name. Unfortunately, Psion did not build recognition of the similar system for data files (DATA_USE and DATAD$) into their programs, which cannot recognise directories. The Miracle Gold Card has very neatly worked round this restriction by allowing you to give directories drive names, so the directory "flp1_data_work_letters" can be given the alias "flp5_".

Having established a directory in which all executable files reside, it might be useful to retrieve that directory name to display on the screen or include in a program. This can be done with the PROGD$() function. It takes no parameters (and so the brackets are optional for all except the purists) and returns a string along the lines of "flp1_exe", or whatever prefix has been set with PROG_USE.

Microdrive users would be foolish to consider dividing the small capacity of their media into directories (although "PROG_USE mdv1" and "DATA_USE mdv2" remain useful options), and for many double-density diskette users there will be few opportunities to exploit directories to the full. However, lucky owners of the new extra-density (ED) diskettes capable of holding over 4 megabytes of data will have ample space to experiment with the convenience of files directories.

PUT #chan position, item1, item2, item3...

[Super Toolkit 2]

OUTPUT COMMAND
#chanChannel (normally opened to a file)
position(Optional) The offset from the start of the file at which output will be written
item1, etc.(Optional) A comma-separated list of values to be written.

PUT, BPUT and PRINT perform the task of writing information to a channel, each in a distinctive way. Whereas PRINT writes formatted data, PUT writes data in an unformatted manner; in other words, as the data is stored internally. A string is preceded by a two-byte integer representing its length, an integer is written as a two-byte sequence and a floating point value is written as a two-byte binary exponent and a four-byte signed mantissa. To retrieve data written by PUT, SuperToolkit II contains the keyword GET, described earlier in this series.

The first parameter in a PUT statement can be an offset from the beginning of the file, which provides a pseudo-random-access ability. When data is written with PUT, the file pointer is updated automatically. Note that there is no comma separating the channel number and the position (if there is one). The list of one or more items following the first comma are all written in the QL's internal format to the file. This can cause some problems to the interpreter because of the flexibility in variable type provided by coercion. PUT has some neat methods of ensuring that data is written in the correct internal format.

TO	DO THIS
Force floating point type	Add +0, eg 475+0
Force integer type	Add ||0, eg varvalue||0
Force string type	Add &' ', eg charval & ' '

(Note that string type is forced by adding an ampersand and two quote marks - the null string.)

Without any list of values, PUT is a useful tool for resetting the file pointer to some desired location before using any data-writing command. This is particularly useful in conjunction with PRINT, which otherwise cannot easily control where its output will be placed.

RAD(degrees)

TRIGONOMETRY FUNCTION
degreesA floating point value

The RAD() function converts between degrees and radians. A radian is a section of a circle' circumference which matches the length of the circle's radius. Just over six radians are needed to make a complete circle, as is suggested by the widely-known formula 2*PI*radius that represents the relationship between a radius and a circumference. As there are 360 degrees in a circle, roughly 57 of them are needed to cover an arc of one radian.
Radians are more important than degrees in computer trigonometry because of their relationship with the COS() and SIN() functions. If a circle one unit in radius was drawn with its centre at the graphics origin (use the command SCALE 4, -2, -2 to make this visible on screen) the command LINE SIN(1), COS(1) would draw a radius from the centre to a point one radian away from the 12 o'clock position in a clockwise direction.

RANDOMIZE value

	PSEUDO-RANDOM NUMBER GENERATOR
value	A floating point value

The QL can generate what appears to be a random sequence of values. They are not strictly random because, providing you start with the same value, the sequence can reliably be repeated. This might appear to be something of a disadvantage, but it has a value when creating large sequences of test data that may need to be repeated value for value at a later date. Unless instructed otherwise, the QL "seeds" the random sequence using its clock. The RANDOMIZE command, without any parameter, forces the QL to begin a new random sequence by referring once more to its internal clock. For a more predictable sequence, a value can be added to the RANDOMIZE command that it will use in preference to the current time. Whatever random numbers then appear, they can be reproduced exactly by using the same parameter with RANDOMIZE.

READ var, var, var...

	EMBEDDED DATA COMMAND
var	A variable name

Like all Basics, SuperBasic allows data to be read from DATA lines in programs in an analogous way to reading data from a file or other input channel. In fact, the similarity is so strong that programmers frequently attempt to READ from a channel when what they need to do is INPUT. The following snippet will be used to demonstrate the important features of READ and its associated keywords DATA and RESTORE:

```
100 DATA 100, "One Hundred", 24.3, "Last item"
110 READ value%, text$, fraction
120 PRINT text$
```

The DATA keyword is described earlier in the New User Guide. It identifies a line of comma-separated values that can be assigned to variables by the READ command. Like DATA, READ can take several parameters each separated by a comma. There is no need to match DATA line with READ line, but it is important to ensure that the variable types always match (unless you want to rely on coercion to sort out discrepancies).

If the example snippet were to be run just once there would be no problems: the data values match up with the variables in the READ statement. At the end of the program all of the variables in the READ statement have values assigned to them and the last item in the DATA list remains unread. Should the program be run for a second time, however, an error will result. The interpreter will attempt to push the last data item into the "value%" variable and will generate a type mismatch error. It was decided, for the benefit of the few times when it might be useful, that the RUN command will not perform an implicit RESTORE.

Before you feel tempted to place large quantities of data into DATA statements, remember that they can form a significant part of a program's bulk and therefore occupy valuable memory space. If the DATA items are read into lots of different variables, such as a large array, for instance, the memory loss is doubled: once for the DATA statements and once to store the variables. A colleague of mine thought up the elegant solution of MERGEing DATA statements from files on an as-required basis, but an even better solution is simply to INPUT the data from files and avoid using READ altogether.

RECHP base

[Super Toolkit II]

MEMORY MANAGEMENT COMMAND

base	The base address of an area of memory allocated with ALCHP() function

The QL is able to rope off chunks of memory for whatever purpose its programmer can devise, usually to store raw data or to hold machine code programs. Allocations are made from what is called the "common heap", or what's left once the Qdos system areas have been allocated. Unlike most parts of the QL's memory, common heap allocations stay in the same place throughout the execution of a program. Memory is allocated by a function such as ALCHP() that returns the address of the first byte to be reserved. This should almost always be stored in a variable for future reference.

There will come a time when allocated memory is no longer required and it can be returned to the heap for re-allocation. To do this, RECHP (short for reclaim common heap) needs to know the base address of the memory being returned. Cleverly, it does not need to know how much memory to remove as Qdos can look this up for itself. However, unless common heap memory is contiguous it cannot be re-allocated, so the sequence in which memory is released is of great importance.

RECOL #chan, col0, col1, col2, col3, col4, col5, col6, col7

SCREEN HANDLING COMMAND

#chan	(Optional) A valid screen channel
col0...col7	Integers representing colours according to the following table:

VALUE	COLOUR
0	Black
1	Blue
2	Red
3	Magenta
4	Green
5	Cyan
6	Yellow
7	White

On QLs unmodified by go-faster utilities, Gold Cards and the like, watching RECOL operate on a large area of the screen is marginally more interesting, and takes marginally less time, than watching treacle run down the side of a tin. The object of the exercise is to change any or all of the colours on a screen to some other colours. The eight mandatory parameters of the RECOL command represent the full low-definition colour-set. The position of each parameter indicates the current colour (ie the third parameter affects all red pixels, the eighth all white pixels) and the value represents the colour to which those pixels will be changed. RECOL 0,1,2,3,4,5,6,7 has no effect whatever because the colours are changed to themselves. RECOL 7,6,5,4,3,2,1,0 changes every pixel to its inverse colour, but there is an instant way of achieving this effect:

```
100 OVER -1
120 BLOCK 200, 100, 20, 20, 7
```

Provided there is no rush, interesting effects can be obtained using a succession of RECOL commands. For instance, the following gradually fade out a multi-coloured display to a uniform green:

```
100 RECOL 0,1,2,3,4,5,6,4
110 RECOL 0,1,2,3,4,5,4,4
120 RECOL 0,1,2,3,4,4,4,4
130 RECOL 0,1,2,4,4,4,4,4
140 RECOL 0,1,4,4,4,4,4,4
150 RECOL 0,4,4,4,4,4,4,4
160 RECOL 4,4,4,4,4,4,4,4
```

Sadly, RECOL 0,1,2,3,4,5,6,7 does not restore the display back to its former glory: RECOL's transformation is permanent. In high-resolution mode it is only necessary to change the four colours it is capable of supporting. However, even in Mode 4 all eight parameters are still mandatory.

REFERENCE var1, var2, var3,...

[Turbo Toolkit]

	COMPILER DIRECTIVE
var1, etc	A variable or array name

When a programmer passes a parameter to a user-defined procedure or function, SuperBasic follows simple rules to determine whether any changes to that parameter's value shall remain once the routine has been exited. The string "Hello World" might, for instance, be passed to a parameter called ALTER that turns it into "Goodbye Everyone". The output from the following snippet will determine whether the change survives the return from the parameter definition:

```
100 message$ = "Hello World"
110 ALTER message$
120 PRINT message$
```

If the PRINT statement produces the message "Goodbye Everyone" then the ALTER procedure will have altered the variable permanently. If, however, the variable was changed in the ALTER procedure code but the PRINT statement at Line 120 proved that the message$ variable had reverted to read "Hello World" again, then the changes would not have survived the return to the main program.

These phenomena are described as "passing by reference" and "passing by value". When a variable is passed by reference, the parameter in the procedure that temporarily represents that variable is set up so that it refers to exactly the same part of the variables area in the QL's memory. When a variable is passed by value, the parameter in the procedure is established in another part of the QL's memory and initialised with the value of the variable passed to it.

SuperBasic's simple rules are to pass variables by reference and expressions by value. If you want to pass a variable by value, simply make SuperBasic think it is part of an expression. This is most easily achieved by putting the variable name in brackets in the calling statement. ALTER (message$) might transform the text from "Hello World" to "Goodbye Everyone" within the procedure code, but because the variable was passed in an expression and was therefore passed by value the end result, once the procedure was complete, would be no change whatsoever.

The Turbo Toolkit, however, handles matter slightly differently. By default, Turbo passes variables by value. In order to pass a variable by reference a REFERENCE directive must be placed immediately before the appropriate DEFine PROCedure line to inform the compiler of your intentions, as in:

```
400 REFERENCE name$
410 DEFine PROCedure ALTER (name$)
420...
```

The Turbo Toolkit's conventions apply equally to arrays, but you must remember to use dummy subscripts to indicate the number of dimensions Turbo must cater for:

```
500 DIM text$(15, 12, 20)
510 Fill_up text$
...
600 REFERENCE text$(0,0,0)
610 DEFine PROCedure Fill_up
620 ...
```

It does not matter at all what integers are used as subscripts in the REFERENCE directive: the compiler simply counts how many numbers are there. Incidentally, Turbo Toolkit implements REFERENCE as a new but meaningless keyword in SuperBasic so that you can test your code with the interpreter before compiling it.

THE NEW USER GUIDE

SECTION TWENTY FIVE

KEYWORD INDEX

This month in the Keyword Index, Mike Lloyd opts to RELEASE_TASK, but in the end he must once more RETURN.

RELEASE_TASK tasknum, tasktag [Turbo Toolkit]

TASK MANAGEMENT COMMAND

tasknum	An integer, part of the unique reference for a task
tasktag	An integer, part of the unique reference for a task

The RELEASE_TASK command is very easy to confuse with its REMOVE_TASK cousin, but they have almost opposite effects. "Release" is used in the sense of releasing from bondage, in this case the bonds of sleep imposed by the SUSPEND_TASK command. SUSPEND_TASK places a task into hibernation, usually for a set period of time. Should you need to resurrect the task earlier than expected, a RELEASE_TASK command must be executed in some other running task. If the task has already been awakened, the RELEASE_TASK instruction is ignored. Incidentally, the two parameters uniquely identify the task in question: SuperBasic has the special value of 0,0 but the references for other tasks cannot be predicted. For obvious reasons, a task cannot awake itself. See the LIST_TASKS Turbo Toolkit procedure for details of how to determine which identifiers go with which tasks.

REMark

PROGRAM COMMENTING DIRECTIVE

REMark is not strictly a command but a placeholder for a command. No matter what follows a REMark keyword on a line, it is ignored by the interpreter. This provides an opportunity to add meaningful comments to programs (hence its name) or the means temporarily to prevent some commands from being executed.

REMOVE_TASK tasknum, tasktag
REMOVE_TASK -1

TASK MANAGEMENT COMMAND

tasknum	An integer, part of the unique reference for a task
tasktag	An integer, part of the unique reference for a task

The REMOVE_TASK command kills off any job, whether dormant or active, and removes it from memory, releasing any allocated memory as it does so. The one task impervious to its effects is SuperBasic. Tasks can kill themselves off if a single parameter of -1 is used. This is in contrast to most task-related commands that assume that the current task is meant if no parameters at all are provided. Whether deliberate or not, the requirement to include a parameter here should concentrate the programmer's mind and ensure that no mistakes are made.

RENAME oldname TO newname
RENAME oldname, newname
[Super Toolkit II]

	FILE MANAGEMENT COMMAND
oldname	The name of an existing file
newname	The name the existing file will be given

In SuperBasic the process of renaming a file has two steps: copying the file to provide the new name and deleting its predecessor to remove the old. Super Toolkit II combines these steps into one command. As with most of SuperBasic syntax, the TO keyword and the comma are entirely interchangeable. The SuperToolkit II manual may be slightly misleading in suggesting that all of the COPY options are available with RENAME. In fact, renaming a file with a name that already exists simply produces an "Already exists" message rather than an option to overwrite. RENAME has no equivalents to the COPY_O, COPY_H and COPY_N variants (nor does it need them). Neither does RENAME acknowledge the existence of an implicit destination directory, as set with DEST_USE. Nevertheless, RENAME is a valuable short-cut for the basic task of providing your files with new titles.

RENUM start TO end; newstart, step
RENUM
RENUM start
RENUM TO end
RENUM start TO end
RENUM start TO end, step
[Improved by Minerva]

	PROGRAMMING UTILITY
start	(Optional) A positive integer representing the start of a line sequence
end	(Optional) A positive integer representing the end of a line sequence
step	(Optional) A positive integer representing the amount by which lines are incremented
newstart	(Optional) A positive integer representing a new start for a line sequence

The RENUM command renumbers the lines of SuperBasic programs. By default, line numbers begin at 100 and are incremented in steps of 10, but deletions and insertions soon spoil this orderly sequence. RENUM allows the programmer to restore order to his or her line numbering without explicitly changing each line by hand. Sinclair never seemed to finish debugging the RENUM command and for those that find its idiosyncrasies annoying the Minerva upgrade is thoroughly recommended.
With the Minerva rom fitted, the RENUM command is incredibly flexible about its parameters. Unusually for SuperBasic, the RENUM syntax distinguishes between TO and a comma. In essence, the syntactical rules are as follows:

if they appear at all, parameters must appear in the order shown in the full command syntax above.
a number immediately following the RENUM command is taken to be the start number of a sequence of lines
a number following a TO is taken to be the end number of a line sequence.
a number following a comma indicates the amount by which line numbers are to be incremented.
a number following a semicolon is used as the new start number for the given line sequence.

RENUM with no parameters changes all the line numbers in a program so that they once again follow the rules of beginning at 100 and increasing by 10 at a time. RENUM followed by a single number renumbers all lines beginning with that number in increments of 10. RENUM , 5 renumbers all lines with an increment of 5. RENUM ;500, 20 renumbers an entire program so that it begins at line 500 and each line is numbered 20 higher than its predecessor. The permutations offered are far too many to include here, but many of them will not work with a standard QL rom.
RENUM will successfully renumber all constant line references in RESTORE, GOTO and GOSUB commands. However, RENUM cannot renumber lines in a way that changes the sequence in which they occur. Any attempt to do so will result in an "Out of range" error.
The QL User Guide states flatly that RENUM should not be used in a program. This advice can safely be ignored provided that the RENUM does not renumber the line on which it itself appears: include it in a procedure with very high line numbers and always include a TO value to stop renumbering before the procedure is reached. Alternatively, variables can be assigned to represent start and end values and the RENUM command can be attached to a hotkey (see ALTKEY in this guide for details).

REPeat control
<commands>

EXIT control
<commands>
END REPeat control
REPeat control: <commands>

LOW-LEVEL PROGRAMMING STRUCTURE

control	A numeric variable of indeterminate value used to control and identify the loop
EXIT control	An optional clause

The REPeat structure is perhaps the simplest and the easiest to use of all of the SuperBasic programming structures. It is a valuable alternative to more traditional FOR...NEXT loop because it does not impose a limit to the number of times a loop is executed. Unlike a FOR...NEXT loop the controlling variable of a REPeat loop cannot be accessed from within the program. It does not, for instance, reveal how many times the interpreter has travelled around the loop (although this would be very useful information: Minerva developers please note!).
By default, once the interpreter is caught inside a REPeat loop it is locked in there forever. This can be ideal for controlling games, and so on, but there are many instances where life must go on. The EXIT clause provides a means of escape, either within an IF clause or as part of a SELect structure. Here is a typical, but completely useless, REPeat loop:

```
100 REPeat loop
110 x = RND(40)
120 IF x = 12: EXIT loop
130 END REPeat loop
```

Incidentally, the overwhelming aptness of the variable name "loop" in these circumstances has lead programmers into thinking that "loop" is itself a keyword and perhaps the only permitted name for a REPeat controller.
As with most SuperBasic structures, REPeat supports a short form in which the REPeat statement and the statements to which it pertains are all contained on one line of a program. The short form allows programmers to leave out the END REPeat statement.
REPeat structures can be nested along with other low-level constructs such as IF..THEN..ENDIFs, FOR..NEXTs and SELECTs. From deep within the nesting an EXIT command carrying the name of the outermost structure will quickly extricate the interpreter and send it on its way, but some purists prefer to jump out of the nest one layer at a time. That is entirely up to them, but seems like a waste of time to me.

RESPR(memory_size)
[Modified by Minerva]

MEMORY MANAGEMENT FUNCTION

memory_size	An integer representing a number of bytes of RAM.

RESPR is the function that fences off areas of the QL's memory so that it can be used exclusively by one program. The parameter it requires indicates how much memory should be reserved, but this is rounded to the next highest 16 bytes. The value returned by the function should be carefully retained in a variable as it is the start address of the roped-off memory. This address will always be divisible by 16.
The most frequent use of RESPR is to allocate memory into which can be stored the code relating to an extension of SuperBasic (ie the code that adds a new command to the language). However, reserved memory can be used for all sorts of data storage. For instance, reserved memory can be used instead of arrays, in which case sorting is achieved by shunting blocks of bytes up and down the QL's memory.
On standard QLs RESPR cannot be used if jobs other than SuperBasic are running. This restriction is overcome by Minerva.

RESTORE line

DATA MANAGEMENT COMMAND

line	(Optional) A valid line number (that must exist in Turbocharged code)

The great majority of Basic dialects allow data to be embedded in programs in DATA statements. Whenever a READ statement is encountered then the next available DATA item is allocated to the variable referenced in the READ statement. Sometimes it is desirable to go back over data and use it again. Sometimes it is necessary to go to a block of DATA items while some earlier DATA statements remain unread. Both circumstances are catered for by the RESTORE command.
RESTORE on its own sets the data pointer back to the first item in the first DATA statement in the program. RESTORE followed by a line number puts the data pointer onto the first item in the next DATA statement on or after that line. When preparing programs for compiling with Turbo there is the slight disadvantage that the line number given must contain a DATA statement.
When you RUN a program the data pointer is not disturbed, therefore it is often a good idea to have a RESTORE command somewhere near the start of a program.

RETRY

PROGRAM HANDLING COMMAND

A SuperBasic program can halt because an error has been generated, or because the user pressed Ctrl-Space, or because a STOP statement has been encountered or because the interpreter has executed the last line of a program. In such circumstances the user can enter CONTINUE or RETRY to revive the program. Whereas CONTINUE sends the interpreter on its way from the line following that on which it stopped, RETRY forces the interpreter back for another go at the line it stopped on. This can be an advantage if the problem was, for instance, a printer that was not on line or a variable that was not holding the right value. The mistake can be put right and RETRY used to repeat the line that fell over. The QL User Guide indicates that you should not edit the program during a pause of this nature but you can often get away with it, particularly if any changes affect only the lines higher than the one on which the program stopped.

RETRY_HERE
[Turbo Toolkit]

ERROR HANDLING DIRECTIVE

When SuperBasic program code is compiled by Turbo the concept of line numbers is much less strong, with the result that there is no immediate support for a simple RETRY command within an error-handling procedure (controlled by WHEN_ERROR, detailed later in this manual). Turbo keeps no track of the current line and statement, nor the state of certain key Qdos tables, at the end of each executed statement just in case an error occurs. Because of this, it doesn't know if it is safe to go back to a particular line when directed to do so from within an error-handling code block. Instead, the Turbo Toolkit includes the RETRY_HERE directive. This can be included anywhere deemed sensible in the program and as many times as might be required. When an error is detected and handled by a WHEN_ERROR routine containing a RETRY command the program will return to the last-encountered RETRY_HERE command and continue on its way. It is assumed that the WHEN_ERROR procedure has the intelligence to correct the circumstances of the error, otherwise it is very easy to cycle round small pieces of flawed programs forever.

RETURN
RETURN expression

HIGH-LEVEL STRUCTURE DIRECTIVE

expression	(Only in function definitions) A value of the type indicated by the name of the function

Functions take values called parameters, work on them for a bit and return a value. User-defined functions return the final value using a RETURN statement followed by a value, either a constant or a variable. The value returned must be compatible with the name of the function, so a function name ending in a dollar sign will return a string and one without will return a numeric value. The normal rules of coercion can apply, but only at the expense of time and with the increased prospect of human error.
The following example function is passed a single, numeric parameter. It returns a string indicating whether the parameter is greater than, less than, or equal to zero:

```
500 DEFine FuNction what_value$ (number)
510 SELect ON number
520 ON number < 0
530  answer$ = "Less than Zero"
540 ON number = 0
550  answer$ = "Equal to Zero"
560 ON number > 0
570  answer$ = "Greater than Zero"
580 END SELect
590 RETURN answer$
600 END DEFine what_value$
```

In a user-defined procedure RETURN can also be used, but it must not have a value following it. Its main appeal is that an early escape from the procedure can be engineered from within an IF statement along the lines of IF x < 12 THEN RETURN. When debugging programs early RETURNs can be used in effect to comment out large chunks of user definitions.

THE NEW USER GUIDE

SECTION TWENTY SIX

KEYWORD INDEX

This month in the Keyword Index, Mike Lloyd gets on with the RJOB until he gets to SELECT ON.

RJOB taskname, error_code
RJOB tasknum, tasktag, error_code
[Super Toolkit 2]

TASK MANAGEMENT COMMAND

taskname	The name assigned to a task (eg "Quill") (need not be unique)
tasknum	An integer, part of the unique reference for a task
tasktag	An integer, part of the unique reference for a task
error_code	A negative integer representing an error condition

RJOB can remove any task other than SuperBasic, and so is of more value to machine code programmers (Turbo users would probably prefer the Turbo Toolkit equivalent of REMOVE_TASK). Usefully, the RJOB syntax allows an error code to be generated which might help a co-operating task to understand what has happened to its late colleague. For the benefit of the machine code fraternity, this value is placed in the D0 register.

RND()
RND(value)
RND(value1 TO value2)

RANDOM NUMBER GENERATOR

value...	An integer.

Most computer languages allow random numbers to be generated. They are particularly valuable in games and in program testing. SuperBasic's RND function is, appropriately, superior to the average for Basic language in that it can return integers within a specified range. With no parameters, RND() returns a fraction between 0 and 1. If a single positive integer is passed to it then an integer is returned between 0 and the parameter. Two parameters represent a range within which the integer result will fall. This saves the clumsy X = 5 + RND() * 5 format when random numbers between 5 and 10 are required.

QL random numbers are not at all random, but a predictable and repeatable non-repeating sequence that gives the appearance of being random. Use the RANDOMIZE command to initialise the sequence at a given point to get that feeling of deja vu.

Incidentally, while RND(1, 6) simulates the throw of a single die RND(1, 12) is completely wrong for representing two dice: RND(1, 6) + RND(1, 6) is correct.

RUN line_number

SUPERBASIC PROGRAM COMMAND

line_number	(Optional) A positive integer less than 32,766

RUN is a simple command that starts the interpreter on its way through the program loaded into memory. MRUN and LRUN are also provided in SuperBasic to find and run program lines from a storage device. (Super Toolkit 2 contains the even more useful but often overlooked DO command.) RUN can be followed by an integer indicating the first line number that should be interpreted; if it does not exist, the program starts running from the next higher line. RUN can be used within programs, although its value is limited and it should not replace GOTO, GOSUB or user-defined procedures.

SAVE filename
SAVEfilename, TO linenum
SAVEfilename, linenum TO
SAVEfilename, linenum1 TO linenum2
[SuperBasic]

SAVE_O filename, range
[Super Toolkit 2]

	SUPERBASIC PROGRAM COMMAND
filename	A valid file name such as "flp1_statsprog", or "bankbal" with Super Toolkit 2, or (less usefully) a valid device such as the network or serial port.
linenum..	Positive integers representing line number ranges.
range	The same options as for the SAVE command.

The SAVE command commits a program in the QL's memory to be saved to a storage device. This has the obvious benefit of saving you the effort of typing the whole thing in whenever you want to run it. If it is of value, parts of programs can be saved to separate files provided that appropriate ranges are given. The default is to save the whole program to one file. Programs are stored in memory in an encoded format, but written to files in readable text so that they can be read or printed.

Many programmers stick a short user-defined procedure like the one following at the end of their programs to ease the chore of saving regularly. Just type "S" and return to save the program. The suffix "wip" stands for "work in progress" and distinguishes this version, which might have huge holes in its logic, from the most recent runnable version.

```
25000 DEFine PROCedure S
25005 DELETE flp1_bankbal_wip
25010 SAVE flp1_bankbal_wip
```

Super Toolkit 2 owners can eliminate the second line of this procedure by substituting the SAVE_O command, which automatically overwrites any existing file of the same name. It takes exactly the same parameters as its SuperBasic cousin.

SBYTES filename, base, length
[SuperBasic]

SBYTES_O filename, base, length
[Super Toolkit 2]

	MEMORY HANDLING COMMAND
location	A valid file name such as "flp1_stats", or a valid device such as the network or serial port.
base	An even integer representing a memory address
length	An even integer

The QL is capable of sending large chunks of its memory into storage onto a microdrive or diskette. This most commonly occurs when users want to save a snapshot of the screen display, but SBYTES can be employed to save any memory area, such as a data set created in a RESPR'd memory segment.

All three parameters following the SBYTES keyword are compulsory. The first is the destination, most usually a file. The second indicates the memory byte at which saving is to begin. The final parameter provides the length of the segment being saved. Both should be even integers. To save a screen, use SBYTES filename, 131072, 32768.

The Super Toolkit 2 extension to the basic command will automatically overwrite any existing file of the same name rather than stop with a complaint.

SCALE #chan, units, xpos, ypos
[Modified by Minerva]

	GRAPHICS COMMAND
#chan	(Optional) A valid screen channel number
units	The number of graphics units represented by the height of the window
xpos	The location of the X axis in graphics units
ypos	The location of the Y axis in graphics units

The SCALE command effectively divorces the QL's graphics and its pixels. Pixels, or picture elements, or more recently and unattractively, "pels", are the smallest display units on the screen. It is convenient for the computer to express all graphics in terms of pixels, but for programmers it can be more convenient to deal in other, more arbitrary units. By default, graphics units on the QL are one hundreth of the height of the window in which they are being drawn. SCALE can change this to anything bigger or smaller that the user desires. The remaining two parameters define where on the graphics plane the bottom left of the screen lies. The default is 0,0, but if you are plotting a circle around the origin you might want to shift the origin nearer the centre of the window with a command such as SCALE 50, -25, -25.

Minerva owners have the additional feature of declaring a negative number for the scale units, effectively relocating the window origin to the top left of the window. This brings the graphics location system into line with the pixel-based and character-based systems and, as Stuart McKnight puts it in the Minerva manual, allows you to draw pictures upside-down.

SCROLL #chan, pixels, segment

	SCREEN HANDLING COMMAND
#chan	(Optional) A valid screen channel
pixels	The number of pixels by which to scroll (an integer)
segmen	t(Optional) A part of the screen according to the following code:

0 The whole window (which is also the default)
1 The top of the window down to, but not including, the cursor line
2 The bottom of the window from immediately below the cursor line

The SCROLL command moves segments of the designated window up or down. A positive integer moves the segment downwards and a negative integer moves the segment upwards. The "blank space" left by scrolling is filled with the current PAPER colour. Unlike the scrolling of program lines or directory listings, SCROLL is a brutal affair that delivers the window contents to their new location immediately. For more gentle and attractive scrolling, put the command into a loop like this:

```
100 FOR pixels = 1 TO 50 STEP 2
110 SCROLL 2
120 END FOR pixels
```

SDATE year, month, day, hour, minute, second
[SuperBasic]

SDATE seconds
[Minerva]

TIME COMMAND
All parameters:positive integers

The QL contains a quartz-modulated clock capable of quite accurate time-keeping. It is a shame, then, that for the lack of a permanent battery supply the clock needs to be reset every time the QL is turned on. Some third party devices do exist, however, that retain the clock setting while the QL is off.

When the QL is turned on, its clock begins from midnight on 1 Jan 1961, presumably its understanding of the year dot. If a single parameter follows an SDATE command then a Minerva-equipped QL will assume it to be the number of seconds since the start of 1961. With a more complete set of parameters any QL will set itself to any date between 1961 and February 2097 (perhaps indicating when Clive Sinclair was planning to issue the QL 2).

The validity checker for date input is not particularly rigorous. References to dates such as the 34th of August are taken to mean 3 September, but the results are unpredictable when a month value greater than 12 or less than 1 is entered.

SEARCH_MEMORY (address, length, string$)
[Turbo Toolkit]

	MEMORY HANDLING FUNCTION
address	An even integer representing a valid memory address
length	An even integer representing a number of memory bytes
string$	A valid string

This valuable function whips through large chunks of memory very quickly indeed to track down occurrences of whatever you have placed in its third parameter. If no match is found then a 0 is returned, otherwise the function returns the address of the first byte where the match has been found.

To quickly give a flavour of how to make use of the function, imagine setting aside an area of memory to act as a database, using a sequence such as:

```
500 Base = RESPR(26000)
510 LBYTES flp1_data, Base
```

The user could be asked for input to identify what record to retrieve and SEARCH_MEMORY will have the result almost immediately.

```
520 INPUT "Locate what? " ! string$
530 location = SEARCH_MEMORY(Base, 26000, string$)
540 IF location > 0
550 PRINT "Data item located"
560 ELSE
570 PRINT "Date item was not found"
580 ENDIF
```

If the data was in an array, the search would take much, much longer, although the programming would be simpler.

	CONDITIONAL BRANCHING STRUCTURE
variable	A numeric value
variable	[Minerva only] A numeric value or a string

SELect ON variable = 1 TO 10, 20, 50 TO 90:
[Commands]
SELect ON variable
ON variable = x
[Commands]
ON variable = 50 TO 100, 500
[Commands]
ON variable = REMAINDER
[Commands]

END SELect
SELect ON variable = x
[Commands]
= 50 TO 100, 500
[Commands]
= REMAINDER
[Commands]
END SELect

The SELect structure marks a significant improvement in conditional branching over that offered by the IF..THEN...ELSE structure of traditional Basics. Borrowed shamelessly from Pascal, a typical SELect structure comprises a number of tests for a variable, each followed by a set of actions that will be carried out should the variable meet the conditions. The QL's standard range syntax is available, so ON x = can be followed by a single value, a range of values or any combination of the two.

The ON variable = range syntax is rather long-winded and is best left to the purists. For each test, simply start the condition with the equals sign. Should the conditions be prefaced with the optional ON variable then you must make sure that the same variable name is used each time.

The short version of the SELect structure provides a quick way of finding out if a variable's value lies within one or more ranges. If there is only one value to test for, or an open-ended range is involved, use IF in preference to SELect. In other words, IF x > 1000 is preferable to SELECT ON x = 1000 TO 9999999.

The interpreter will work its way down the list of tests until it finds one that is true. It will then ignore those that follow and carry on with interpreting the line after the END SELect. It makes sense, therefore, to put the conditions most likely to be true near the top of the structure. You can opt to put a REMAINDER statement at the end of the structure to catch those occasions when none of the tests produces a positive result.

It is worth being aware that seemingly foolproof SELect structures can produce slightly unexpected results, simply because they lull programmers into a false logic. Look at the following:

```
100 SELect ON x
110 = 0 TO 9
120 PRINT "This positive value is less than 10"
130 = 10 TO 99
140 PRINT "This positive value is between 10 and 100"
150 = REMAINDER
140 PRINT "This value is greater than 100 or less than 0"
150 END SELect
```

The whole thing works fine apart from the fact that if X is worth 9.5 you are told it is actually greater than 100. The answer is to take more care over the range values. You might, for instance, rearrange the test conditions to look for the higher values first and then search for values from 0 TO 10. 9.999 will correctly come into this category, but a full 10 will have been picked up in the earlier test condition for 10 TO 99.

With the Minerva rom fitted, SELect becomes even more useful by being able to test strings as well as numbers. It is wonderful to be able to construct tests such as:

```
500 INPUT "Enter your name here: " ! name$
510 SELect ON name$
520 = "David"
530 PRINT "Welcome David, you have full access rights"
535 access = 5
540 = "Sarah" TO "Sarah"
550 PRINT "Welcome Sarah, you have read-only rights"
555 access = 1
560 = REMAINDER
570 PRINT "You do not have access to the program."
580 END SELect
```

Where single strings are involved, checking is case-independent, as demonstrated by the test for "David" above. The test for "Sarah", however, involves a range check and will reject any capitalisation other than that shown.

THE NEW USER GUIDE SECTION TWENTY SEVEN

KEYWORD INDEX

This month in the Keyword Index Mike Lloyd SETs his _CHANNEL and keeps on to SUSPEND. TASK.

SET_CHANNEL #chan, chan_id
[Turbo Toolkit]

	QDOS CONTROL DIRECTIVE
#chan	A valid SuperBasic channel number
#chan_id	A Qdos internal reference for a channel

The SET_CHANNEL command links a SuperBasic channel reference number to a Qdos channel reference number. The two are not closely related. For example, the QL's input and command channel is #0 to SuperBasic but 65537 to Qdos. If it is of value to have two SuperBasic channels pointing to the same device, this can be done using SET_CHANNEL Say that SuperBasic channel #5 needs to be connected to the same device as SuperBasic channel #2. The following command achieves this:

```
100 SET_CHANNEL #5, CHANNEL_ID(#2)
```

Due to administrative difficulties, the Turbo Toolkit imposes a slight restriction on referencing SuperBasic channels in several of its commands, including SET_CHANNEL You should not reference the highest SuperBasic channel number yet opened. In the above example, therefore, a channel number higher than 5 should already be in use.

SET_FONT #chan, base, base2
[Turbo Toolkit]

	SCREEN DISPLAY PROCEDURE
#chan(Optional)	An open console channel
base	A valid memory address, indicating the start of the first font section
base2	A valid memory address, indicating the start of the second font section

The QL's fonts come in halves, dividing the Ascii character set at character number 127. Each half can be stored at a different address. Indeed, screen windows can be defined that use different character sets below and including Ascii 127, yet share the same set (perhaps the rom-based one) for lesser-used characters above Ascii 127. The base addresses may be set to 0 to indicate the rom-based character set, which saves you the trouble of paging through the QL's rom to find where the characters are stored.

Before using SET_FONT, load a new character set into a reserved area of memory with a function such as LRESPR, LRESPR returns the address of the first memory location allocated to the font. Use this value in the SET_FONT command to associate the new font with the required screen window channel.

SET_POSITION #chan, position
[Turbo Toolkit]

FILE ACCESS COMMAND

#chan	A channel opened to a storage device such as a microdrive file
position	An integer

The QL stores information in files in sequential bytes, whether or not you consider the data to be blocked in two or more dimensions (such as a screen dump or a row-and-column database). The contents of files are read in sequential order. However, the SET_POSITION command moves the file pointer anywhere in the file. This effect can also be achieved by opening the file and discarding its contents until the right place is reached, but SET_POSITION is faster. Once the pointer is in position, data can be read from or written to the file from that point.

You could use the SET_POSITION command and the POSITION() function to provide quasi-random access to a file, thus saving random access memory. However, do not expect the results to be anywhere near as fast as accessing ram.

Qdos may trip over very large files on microdrives and report falsely that there is a "bad or changed medium".

SET_PRIORITY task, tag, priority
[Turbo Toolkit]

QDOS TASK MANAGEMENT

task	A task's identity integer (use 0 to represent the current task)
tag	A task's tag number
priority	An integer between 0 and 127

Each task, or program, running under Qdos has a two-part identity value called the task number and the tag number. SuperBasic is 0,0. Additionally, tasks have a priority that indicates the proportion of processing time allocated to each of them in turn. SuperBasic's default is 32. When SuperBasic is the only task running it has 100% of the processing time to itself, but if another task was launched with a priority of 32 SuperBasic would have the processor's attention for only 50% of the time. A priority of 0 sends a task to sleep, allocating it no processing time whatever. A priority of 1 is useful for background tasks that watch for some event to occur before springing into life. Task priorities can be changed at any time. Task numbers and tags can be obtained by using the LIST_TASKS command, or its Super Toolkit equivalent.

Note: There is no point in assigning bigger values to all tasks in the hope that the QL will execute them more quickly: if two tasks have priorities of 64 they still share the processor on a fifty-fifty basis.

SEXEC filename, base, length, dataspace
[SuperBasic]
SEXEC_O filename, base, length, dataspace
[Super Toolkit II]

FILE HANDLING COMMAND

filename	A valid file name
base	The base address of an area occupied by an executable task
length	The size of the executable task
dataspace	The size of the data area to be reserved for the program

SEXEC is related to SBYTES, and performs a similar task, but the file header is altered to identify the contents as an executable job that can multi-task under Qdos. The SEXEC_O variant supplied with Super Toolkit II automatically overwrites any previous file of the same name. Only machine code programmers familiar with the requirements for Qdos multi-tasking have need to use these commands.

SIN(radians)

TRIGONOMETRY FUNCTION

radians	A floating point value, normally between 0 and 2*PI

The SIN function computes the sine of an angle measured in radians. A radian is an arc of the same length as the radius of the circle to which it belongs. Every angle from 0 degrees to 360 degrees can be represented by a radian value lying between 0 and 2*PI. An explanation of basic trigonometry is included in the concepts section of the New User Guide.

SNOOZE
[Turbo Toolkit]

TASK HANDLING COMMAND

SNOOZE takes no parameters and only applies to the program in which it occurs. It has the effect of setting the program priority to 0 (see SET_PRIORITY above). When a program snoozes, any procedure and function definitions shared with GLOBAL and EXTERNAL directives can be accessed. Note that a snoozing task can only be removed by deleting it or its owner from the task table (for example with REMOVE_TASK).

SPJOB jobid, priority
SPJOB jobname, priority
[Super Toolkit II]

TASK HANDLING COMMAND

jobid	An integer representing the job number and the job tag multiplied by 65536
jobname	The name allocated to the job, as determined by the JOB$() function
priority	An integer between 0 and 127

The SPJOB performs the same role as Turbo Toolkit's SET_PRIORITY command: it changes the proportion of processing time allocated to the job specified in the command parameters. A more complete explanation of job priorities is given under the SET_PRIORITY topic above.

SPL filename TO device
SPL #chan TO #chan
SPLF filename TO device
SPL_USE device
[Super Toolkit II]

DEVICE HANDLING COMMAND

filename	A valid filename
device	A valid device name, normally a printer or the network
channel	A valid channel number

Normally, when the QL prints a file under SuperBasic no more commands can be entered until the printer has finished. Depending on the size of the printer's buffer, this might take some time. A useful option is to spool the output, "printing" it to a temporary memory location and then relying on a multi-tasking job in the background to feed the printer with the next buffer-load as and when required. SPL performs this function for Super Toolkit II owners. If you were to spool a file called "letter", a command such as the following would be required:

```
SPL flp1_letter TO ser1
```

The SPLF variant adds a form feed to the end of the file to ensure that the next item to be printed begins on a fresh page.

In line with Super Toolkit's use of default destinations, SPL_USE can identify where spooled output is to be directed, for instance:

```
SPL_USE ser1
SPL flp1_letter
```

SPL_USE is in fact identical to DEST_USE except in one particular: if the destination does not end in an underscore, DEST_USE adds one but SPL_USE does not. This distinction ensures that SER is not mistaken for a directory name. You should be aware that issuing a SPL_USE command will overwrite the name specified in a previous DEST_USE command, and vice versa.

SQRT(value)

MATHEMATICAL FUNCTION

value	Any positive floating point number

The SQRT function returns the square root of the value passed to it. Parameters less than zero generate an error.

STAT #chan, name
[Super Toolkit II]

FILE HANDLING COMMAND

#chan(Optional)	A valid output channel
name(Optional)	A valid drive or directory name

The STAT command does exactly what the WDIR command does, but alongside the list of files it adds their size and the date on which they were last updated.

STOP

PROGRAM CONTROL COMMAND

The STOP command halts the SuperBasic interpreter (or the Turbo executor) in its tracks, returning control to the next job up the tasklist. Turbo programs relinquish the memory they occupy as soon as STOP is encountered. SuperBasic programs remain in memory after they have been stopped. With both Turbo and SuperBasic programs there is an implicit STOP at the end of the in-line part of the program, excluding any procedure and function definitions. A stopped program can be restarted at the next line following STOP by issuing a CONTINUE command.

STRINGF(number$)
STRING%(integer$)
STRING$(text$)
[Turbo Toolkit]

FORMAT CONVERSION FUNCTIONS

number$	A sequence of six bytes whose ASCII values are used to represent a floating point number
integer$	A sequence of two bytes whose Ascii values are used to represent an integer number
text$	A string of Ascii characters preceded by a two-byte integer representation of their length

Under Qdos, variables of different classes are stored in different ways. Floating point numbers of any value are converted into a six-byte sequence, integers are stored in two bytes, and text strings are preceded by a two-byte integer representing their length. The STRINGx() functions in the Turbo Toolkit convert values from their internal Qdos representation into a form readable by users.

The STRINGx() functions have their opposites in the FLOAT$() function, INTEGER$() function and STRING$ function.

STRIP #chan, colour
STRIP #chan, main, contrast, pattern

SCREEN HANDLING COMMAND

#chan(Optional)	A valid display channel
colour	A single integer between 0 and 255 representing a colour strip
main	An integer between 0 and 7 representing a colour
contrast	An integer between 0 and 7 representing a colour
pattern	An integer between 0 and 3 representing a pattern

The STRIP colour combination is used as the background for text when OVER 1 is in effect. The colour combination can be represented by a single value between 0 and 255, but more often this is broken down into three separate values. The main and contrast colours can be any value between 0 and 7 to represent black, blue, red, magenta, green, cyan, yellow or white respectively. The stipple pattern can be any value between 0 and 3. A stipple is formed from four pixels arranged in a box. Stipple 0 puts a contrast pixel in the upper right corner of each box. Stipple 1 imposes horizontal lines of contrast colour. Stipple 2 imposes vertical lines of contrast colours. Stipple 3 provides a checkerboard pattern. For best results, view stipples on a monitor rather than a television set.

SUSPEND_TASK task, tag, ticks
[Turbo Toolkit]

TASK CONTROL PROCEDURE

task(Optional)	A valid task identity number
tag(Optional)	A valid task tag
ticks	An integer representing the time a task is to be suspended

When multi-tasking several jobs it might be useful to turn one or more of them off until they are needed. This can be achieved with the SUSPEND_TASK command. Without any task identifiers the command works on the job in which it appears. Alternatively, a task can be suspended from another task. The time for which a task is to be suspended is measured in power supply cycles, 50 hertz in the United Kingdom, 60 hertz in many other countries, including the United States. A suspension of ten seconds equates to 500 UK cycles or 600 USA cycles. A tick value of -1 causes the task to wait until it is removed from the tasklist, released by another task or forever, whichever is the sooner.

THE NEW USER GUIDE

SECTION TWENTY EIGHT

KEYWORD INDEX

This month in the Keyword Index Mike Lloyd is working on his TAN and making multiple copies with WCOPY.

TAN(radians)

TRIGONOMETRY FUNCTION

radians	A floating point value representing an angle in radians.

The tangent of an angle is the ratio of its sine and cosine. The TAN() function returns the tangent of the angle represented by the parameter passed to it. As with all other trigonometric functions, the angle must be measured in radians. A further explanation of tangents will be found in the Concepts section of the New User Guide.

TK2_EXT
[Super Toolkit II]

SUPERBASIC LANGUAGE DIRECTIVE

Some of the keywords contained in Super Toolkit II have been incorporated under licence in disk controllers and memory expansion units from manufacturers such as Miracle Systems. In some instances, the code supporting the keywords was slightly modified, or is now superseded by improved versions from later editions of Super Toolkit II. The TK2_EXT command forces the QL to ignore any keyword definitions it finds except those defined by Super Toolkit II itself. Refer to your hardware documentation for instructions for using the FLP_EXT and EXP_EXT cousins to this command.

TRUNCATE #chan, position
[Super Toolkit II]

FILE HANDLING COMMAND

#chan	A channel opened to a file
position(Optional)	An integer representing a character location within a file

During low-level file-handling it can be useful to reduce the length of a file. Without the TRUNCATE command this could only be done by loading the file's contents into the QL's memory and then saving back only part of it. The TRUNCATE command can take a new position point as a parameter or it can close the file at the file pointer's current location.

TURN #chan, angle
TURNTO #chan, angle

	TURTLE GRAPHICS
#chan(Optional)	A valid screen channel
angle	An angle measured in degrees (not radians)

TURN and TURNTO are basic features of the turtle graphics language set. Turtles are invisible graphics cursors that leave traces behind them when the MOVE command is issued, provided their pen is down. TURN rotates the turtle a given number of degrees. Positive values cause the turtle to twist clockwise, negative values turn the turtle anti-clockwise. A turn of -40 degrees is exactly the same as a turn of 320 degrees. The TURNTO variant of this command points the turtle to an absolute reference with 0 degrees pointing vertically upwards.

TYPEIN string$ **
[Turbo Toolkit]

	COMMAND INPUT UTILITY
string$	A text string or character variable containing valid SuperBasic commands

One of the minor benefits of a multi-tasking computer system is the ability of one task to offer data to another. This is particularly useful when the receiving task is the SuperBasic interpreter, because an application can trick it into thinking that input is coming from a human operator typing at the keyboard. This allows you to build test suites where key sequences are played back to see if modified programs work as expected. You could create demonstration or tutorial utilities which drive an application by simulating keyboard activity. Alternatively, you could write macros to take the drudge out of regularly-used command sequences; for example, all of the commands to restore the default windows to your preferred sizes, locations, colours and character sizes might be executed by pressing just one key.

To understand what TYPEIN does, you must know a little about how keypresses become input for the SuperBasic interpreter. When a key is pressed, an electrical connection is made in a mat under the keyboard that is transmitted to the QL's motherboard where a byte with the appropriate ASCII value is generated and stored at the end of a queue in the keyboard buffer - an area of RAM. The SuperBasic interpreter regularly inspects the keyboard buffer and extracts characters up to the next carriage return and tries to interpret what it has read as a SuperBasic command. When several programs are running simultaneously one of them will call the keyboard buffer its own and fend off all attempts by other programs to read it. Pressing Ctrl-C switches control from one multi-tasking program to the next, and with it the keyboard buffer.

TYPEIN works by stuffing a string of characters directly into the keyboard buffer. With only three exceptions, Qdos is quite unable to distinguish between such characters and those that arrive conventionally from the keyboard itself. Because of their unique roles, the following combinations cannot be included in a TYPEIN string: Caps Lock, Ctrl-F5 and Ctrl-C.

The string that follows TYPEIN can contain anything that could be typed in at the command line. You can opt simply to stuff a command into the command interpreter's buffer and wait for the user to press the Enter key, or you can force SuperBasic into action by including a CHR$(10) at the end of every command. A TYPEIN string can contain many commands separated by CHR$(10)s. Strings that appear as parameters within TYPEIN's command string should have single inverted commas inside the double inverted commas delimiting the string, or vice versa. The following example shows both these points in action:

```
100 TYPEIN "PRINT#0, 'Hello' " + CHR$(10) + "CLS #2" + CHR$(10)
```

To make full use of TYPEIN in conjunction with the SuperBasic interpreter you will also need to use the COMMAND_LINE procedure in the Turbo Toolkit.

There are two pitfalls to avoid with TYPEIN. If you execute programs with EX_W or EX_A, or any other of their ilk, the SuperBasic interpreter is disabled until that program is completed, so TYPEIN is limited to simulating keyboard activity for that application alone. Secondly, the KEYROW command bypasses the keyboard buffer to read keyboard activity at a lower level: in order to prevent confusion between keypresses read instantly with KEYROW and those patiently waiting in the buffer queue QDOS empties the keyboard buffer before accepting a KEYROW character. (Incidentally, this feature can be turned to your advantage because there are several occasions when the buffer should be emptied of stray input and KEYROW is a good way to do it.)

UNDER #chan, toggle

	TEXT DISPLAY PROCEDURE
#chan (Optional)	A valid display channel (default is 1)
	toggle 1 represents underlining, 0 represents no underlining

The UNDER command toggles between underlining and non-underlining (the default) for all text subsequently printed to the window concerned. Underlining appears in the same ink colour as used by the text. A quick way to achieve underlining of a different colour is to print the text once with underlining enabled and then overtype (with OVER -1) in a different colour without underlining. UNDER does not underline text sent to the printer.

VER$
[SuperBasic]
VER$ (flag)
[Minerva]

	SYSTEM FUNCTION
flag(Optional)	Indicates which information to return from the following list:

2 The base address of the Qdos system variables area
1 The ID value of the currently executing job
0 The revision number of SuperBasic (the default)
1 The revision number of Qdos

In simple SuperBasic using a Sinclair rom, VER$ is a plain and unassuming utility function that simply reports which version of the rom is inside your computer. If the rom is a Minerva from QView then VER$ can take one of four parameters, one of which is actually useful. The base address of the system variables area is an essential piece of information for those who circumvent some of SuperBasic's weaknesses by accessing Qdos tables directly. For the most part the system variables can be assumed to begin at a given address and once found, provided the QL's configuration remains unaltered, the base address can be relied upon for session after session. However, this cannot be relied on if you are writing for several machines, or for machines with different configurations, or for virtual QLs running with Atari or IBM-compatible hardware, or for future revisions of QL roms. VER$(-2) is therefore a genuinely useful function, although Turbo Toolkit's family of BASIC_B%-related functions incorporate the same facility. The other variants in Minerva's VER$ are less likely to be valuable, but are nevertheless welcome.

VIEW #chan, source, target
[Super Toolkit II]

	FILE MANAGEMENT COMMAND
#chan (Optional)	A valid output channel
source	The name of a file to be viewed
target (Optional)	The name of a file receiving the source output

VIEW is a neat alternative to the clumsy COPY command when you want to view a file's contents on screen. Instead of opening a temporary Qdos window with its unattractive defaults, VIEW places its output into an existing window, truncating long lines rather than wrapping them. So that you can view each screenful of output at your leisure, VIEW periodically inserts a Ctrl-F5. If its output is sent to a printer or a file, VIEW chops off each line at the eigtieth character.

WCOPY #chan, name1 TO name2
WDEL #chan, name1
WDIR #chan, name1 TO name2
WREN #chan, name1 TO name2
WSTAT #chan, name1
[All from Super Toolkit II]

FILE MANAGEMENT COMMANDS

#chan (Optional)	A valid output channel, used for screen dialogues
name1	A full or partial filename (see below for details)
name2	A full or partial filename (see below for details)

SuperBasic file management commands operate only on one file at a time. In Super Toolkit II many of them have been given a more powerful equivalent that operates on all files sharing similar filenames. For instance, a single command can copy all word processing files with the suffix "_doc" from one microdrive to another, or delete all files beginning with "flp1_junk_", or renaming all files beginning "flp2_work_" with the prefix "mdv1_".

Super Toolkit wildcards are in fact prefixes and suffixes separated by underscores from the main filenames. These elements of filenames are also the QL's equivalents of the directory structures found on IBM-compatible computers. The first part of a full filename identifies the device, such as "flp1_" or "mdv2_". There can then follow several filename segments separated by underscores as there is room for within the overall maximum filename length. For most users it is enough to provide even ED disks with just one or two subdirectory levels. You might wish to rename all the executable files "flp1_prog_" followed by their usual name and then ensure that all your datafiles are prefixed "flp1_data_" or "flp1_work_". With Gold Card you can even get the Psion applications to play along as subdirectories can be given virtual drive names - in other words all files with names beginning "flp1_work_" can be made to appear as though they were on a drive called "flp3_".

Once you have sorted out your disks and microdrives in this way full use can be made of the wildcard versions of SuperBasic's file handling commands. The wildcard commands have all the same functionality as their normal equivalents but they operate on every file sharing the same prefix or suffix. The command:

```
WCOPY flp1_ TO flp2_
```

will copy every file on the first drive to the second. At the outset of this operation you will see a screen message similar to "Copy flp1_ to flp2_. Proceed?" and offering the options of pressing Y, N, A and Q. These letters have the following meanings:

Y Yes, take action on this file
N No, do not action this file, but continue with the next
A Yes, action all files matching the wildcard
Q No, do not action this file or any others (ie Quit)

The same options are presented no matter which wildcard file command is used and they are consistent with Super Toolkit's enhancements of the standard equivalents. By default the text appears in the command window (#0) but you can change this by including a channel parameter.

Some practical examples of using wildcards follow. See the New User Guide entries for the root keywords for a full explanation of their syntax.

Archive all your old documents from a work-in-progress area by typing:

```
WREN _doc TO archive_
```

Backup a microdrive to a disk subdirectory by typing:

```
WCOPY mdv2_ TO flp1_backup_
```

Delete document files accidentally stored in the program area (note the use of two adjacent underscores, without an intervening space, indicating where in the full filename a match needs to take place):

```
WDEL flp1_prog_ _doc
```

List all the Abacus files on a microdrive:

```
WDIR mdv1_ _aba
```

List the size, age and type of all files on a floppy drive to a file:

```
OPEN #6, flp1_statistics
WSTAT #6, flp1_
```

THE NEW USER GUIDE

SECTION TWENTY NINE

KEYWORD INDEX

In the last part of the Keyword Index, Mike Lloyd goes through WHENs, WIDTHs and WINDOWs

WHEN <expression>
<commands>
CONTINUE
ENDWHEN

WHEN <expression> :
<commands>: CONTINUE

WHEN var
[Minerva]

EVENT-DRIVEN PROGRAMMING STRUCTURE

<expression>	Can be any expression involving a single variable that can evaluate to true
<commands>	Can be one or more SuperBasic statements
var	Can be any variable (its value is ignored)

The WHEN keyword appeared on a quite early version of the QL's rom, but was never adequately implemented until Minerva came along. While late Sinclair roms can just about make sense of a WHEN structure it is foolhardy to use it as it is very unreliable. However, Minerva users can make full use of the structure to add an exciting new level of sophistication to their programs.

A WHEN structure is similar to an IF structure, but instead of evaluating the expression immediately and taking the appropriate action, the SuperBasic interpreter will remember the WHEN expression and as soon as it becomes true it will nip back and execute the commands contained in the WHEN structure. This behaviour can be changed by executing another WHEN command referencing the same variable, or by issuing a "bare" WHEN command comprising WHEN followed by a variable. WHEN statements can appear anywhere in a program but only take effect after the interpreter has read the structure. Use CONTINUE to return to the line following the one that triggered the call to the WHEN code.

You can include several simultaneous WHENs in a program provided each refers to a different variable in the logical expression or the variable used is declared LOCAL

WHEN ERROR
<commands>
ENDWHEN
[Minerva, some late Sinclair roms]

EVENT DRIVEN PROGRAMMING STRUCTURE

<commands>	One or more SuperBasic statements.

WHEN ERROR is very similar to the WHEN structure described above, but the statements in the structure are executed only when an error occurs. WHEN ERROR can be used in conjunction with the SuperBasic keywords RETRY and CONTINUE in an attempt to circumvent the error. If you leave a WHEN ERROR block using GOTO or a procedure/function call the interpreter still imagines that it is inside the error handler, which is very likely not what you want. For a full explanation of error trapping see the New User Guide's forthcoming Concepts section.

WHEN_ERROR flag
<commands>
RETRY
ENDWHEN

WHEN_ERROR flag
CONTINUE
ENDWHEN
[Turbo Toolkit]

EVENT DRIVEN PROGRAMMING STRUCTURE

flag	Either 0 or 1, indicating the error level being defined
<commands>	One or more SuperBasic statements

Turbo Toolkit's WHEN_ERROR (note the significant underscore) has the enormous advantage of working perfectly on every QL rom variant. It also offers two levels of error-trapping. If an error occurs in a WHEN_ERROR 1 block, or if the end of such a block is reached without a RETRY command occuring, then the most recent WHEN_ERROR 0 structure will be executed. Error processing can be cancelled only by writing a WHEN_ERROR construct that contains just one command: CONTINUE.

In compiled code, no line numbers and statement locations can be assumed to exist, so the RETRY command does not work in the conventional way. Instead, program execution resumes at the line following the last RETRY_HERE statement to be encountered. RETRY_HERE should be strategically placed at the start of several procedures so that reprocessing can commence from logical points. Note that the Turbo compiler has no concept of "roll-back": it will not restore matters to exactly how they were when the RETRY_HERE command was first read.

WIDTH #chan, columns

OUTPUT CONFIGURATION COMMAND

#chan (Optional)	A channel opened to a non-console device
columns	The maximum number of columns to print per line

The WIDTH command, on early Sinclair roms at least, is not particularly reliable. It is supposed to limit the line width of output directed to printers to whatever is stated in the WIDTH parameter. In practice, it is much easier to send a printer control code.

WINDOW #chan, width, height, xpos, ypos
[SuperBasic]
WMON mode
[Super Toolkit II]
WTV mode
[Super Toolkit II]

SCREEN HANDLING PROCEDURES

#chan (Optional)	A valid console channel
width	The width of the window in pixels (assuming high resolution)
height	The height of the window in pixels
xpos	The horizontal location of the top left corner of the window
ypos	The vertical location of the top left corner of the window
	mode 0, 4 or 512 for high resolution, 8 or 256 for low resolution

The WINDOW command allows you to reconfigure screen windows without redefining them with the OPEN command. Regardless of the screen mode all co-ordinates and sizes assume that there are 512 pixel columns on the screen. In low resolution mode, of course, only even values are significant.

To ease the task of returning windows to their default configuration, Super Toolkit II contains two handy commands: WMON restores the initial layout of high resolution screens and WTV restores the layout of low resolution screens. They do not, however, change the current resolution. This is managed by the parameter that follows each command.

The NEW USER GUIDE

Concepts Section

Section Thirty

It is possible to use a computer without knowing much about how it works, just as you can drive a car without understanding its engine. However, to get the most from a computer, as with a car, it is highly desirable to know something about its internal workings. Otherwise you may find yourself doing the equivalent of driving slowly uphill in top gear and wondering why your computer is letting you down.

The Concepts section is your guide to the theory of computing with the Sinclair QL This Concepts section reaches further than the original Sinclair guide and tackles topics in greater depth. Technical terms are avoided wherever possible and the unavoidable ones are carefully explained. No prior knowledge is assumed. Whereas the original user guide dealt with its concepts in alphabetical order, the New User Guide's Concepts section is arranged so that hardware issues are covered first, followed by Qdos and SuperBasic topics. The first two topics provide an overview of the entire system from hardware to software.

Towards a unified theory

Why is computing so difficult to understand? Some people have a natural gift for seeing life from a computer's perspective, but others suffer from extreme forms of technophobia. Most of us think we will know enough to cope just as soon as we've mastered a little bit more detail. Computing is difficult for a number of reasons. Firstly, it is a very immature technology: we frequently haven't found the best way of doing things yet. Secondly, development in computing science is the fastest known to human society, with technologies continually leapfrogging each other. Thirdly, computers are a new class of brain-assisting machines; all other machines are there in some capacity to assist human musclepower. Fourthly, computers are chameleons whose purpose and appearance changes with each program that is loaded. Finally, and most confusingly, there is nothing like computing for contradictions, illusions and virtuality. Nothing in computing is what it seems.

It was once observed that any sufficiently advanced technology is indistinguishable from magic. To medieval people, things like railway engines and aircraft would fall into this category. For early Victorians, transistors, calculators and television would have been magic. For us it is fractal geometry, the unified theory and true artificial intelligence. Magic is arbitrary, fathomless and unpredictable. Technology, on the other hand, is predictable, understandable and within the control of those who understand it. The difference between magic and technology is knowledge.

Concepts for knowledge

Sherlock Holmes thought that the brain had a finite capacity for knowledge. When he was told that the Earth revolves around the Sun, he dismissed the information as valueless and tried to forget it in order to leave room in his mind for something more useful. Whether the brain's capacity is fixed or not, it often seems as though it is, so what chance do we have of keeping up with the fastest-growing body of knowledge ever known?

The answer is concepts. Information can be compressed into concepts that can be understood and remembered even though the fine detail might be missing. In this way, Stephen Hawking can explain the mechanics of the universe without resorting to blackboards full of equations, I can understand the concept of fuel injection without being in the least able to design a fuel injection system, and you can understand what makes your computer work without a doctorate in computing science or electronic engineering.

Concepts are particularly valuable in computing because computer systems "exist" at many different levels and mutate alarmingly according to the perspective from which they are viewed. From the electron's point of view, computers are enormous quantities of tiny circuits that are either switched on or switched off. From the hardware designer's point of view, computers tend to be chips linked by wire pathways (known as buses): how precisely the internals of each chip go about their business is not particularly relevant. From the operating system's point of view a computer system is a central processing unit, a working memory area and links to devices, or peripherals, between which information in the form of integer numbers can flow. From the programmer's perspective, a computer is a repository for information, often in the form of text or real numbers, and for the instructions that process the information. From the user's viewpoint, the computer is one minute a games console and the next a word processor. None of these perspectives tells the whole story, but each depends on its predecessor, like the storeys of a house of cards.

Something out of nothing

Like Sherlock Holmes, someone concentrating on one of the levels tends to "hide" much of the levels above and beneath, so when a user views text on a computer screen all trace of the hardware layer has disappeared. There is no text on the screen, simply dots of light. There is no programming language that people can read, but only impenetrable forests of numbers. There are in fact no decimal numbers, but only binary digits. There are, of course, no binary digits, but only pulses of electrical charge.

If you turn off a car's engine, it is still demonstrably a car. You can service the engine, sit in the seats, polish the paintwork. You can release the handbrake and push it around. Switch off a computer and it's a doorstop. The computer's marvellous edifice of abstraction upon abstraction collapses back into its component parts, like a soap bubble, literally as if it had never been. Computers never learn anything, they never get better at anything, and they never profit from experience. Even computer programs that log previous behaviour never get any better at interpreting it.

Computers are real-life equivalents of Doctor Who's Tardis: their external and internal dimensions are entirely contradictory. No other machine ever invented has this capacity to exist at several different levels and take on such different appearances at each level. In the computer, things that appear to be significantly different physically, such as a microdrive cartridge and a random access chip, might at some level become completely indistinguishable. Things that are physically similar, such as floppy diskettes, might be readable on one type of computer and completely unreadable by another. Something made up of many things, such as the QL's random access memory which is spread across several chips, might actually be one (supposedly) seamless whole. It all depends on the level at which you look at it.

Appearance is misleading

Just as with the Tardis, size is almost always misleading. Physically, the largest chip in a QL is the central processing unit, but its internal capacity is dwarfed by the storage ability of the much smaller random access memory chips. Double density, high density and extra density diskettes all share exactly the same dimensions, but each has double the storage capacity of its predecessor. A physically large computer, such as an IBM 286, might have less storage capacity and operate more slowly than the smaller QL. Even when its "true" internal capacity is measured, an IBM PC often uses its memory in such a way that a QL with much less memory might actually have room to store more information.

To cap it all, appearances can be doubly deceptive. A computer might look like a QL on the outside, but it might be runnning a program (called an emulator) that makes it think that it is a Sinclair Spectrum on the inside. You can fool an Atari ST into thinking that it is a QL and even buy an expansion card for an IBM PC that, to all intents and purposes, it really is a QL. To return the compliment, a QL can be convinced that it is an IBM PC.

It is also difficult to determine the extent of a computer system. Does it include peripherals, such as printers and monitor screens, or not? If a floppy disk drive is a peripheral because it is in a separate box and linked by cable, is a microdrive a peripheral even though it is physically inside the computer casing? The QL has a keyboard on top of its casing, but can also have an external keyboard added: are they both peripherals? Most people understand keyboards, disk drives, monitors and so on to be "devices" and thus peripheral to the computer proper even if they are integral to the computer case, but having gone down that route where do we stop? Is random access memory a peripheral or part of the computer? Is the central processing unit "a computer on a chip"? Is there a better answer than "it depends"?

Working through the layers

Even when we tie down the computer so that it is a QL on the outside and a QL on the inside, it can be a word processor, a number-cruncher, a golf game or a chess player - or all these things at once. Computers are electronic chameleons employing layers of abstraction to turn millions of tiny, simple electronic circuits capable only of being "on" or "off" into, for instance, sufficient calculating power to outwit all but the very top human players at chess. And then, as soon as they are turned off, they forget completely that they have ever done it.

To make sense of these contradictions we must take a journey through the layers, beginning with the physical components and ending with the high-level programs, from what the Americans call "close to the iron" to what is becoming increasingly familiar as "virtual reality".

Next month Mike Lloyd starts to analyse the rock bottom concepts of computing.

Concepts Section

Section Thirty One

At least five separate layers exist between the CPU and the user.

Most of the difficulty in understanding computer concepts is that there are so many contradictory ideas to reconcile. Just when you have learned that microdrives are called mdv1_ and mdv2_, someone tells you that they can create the equivalent of a microdrive in the QL's memory, and call it mdv1_ or mdv2_ if they like. Even simple words like "memory" can be confusing: do I mean internal random access memory or external storage capacity? Why am I calling a microdrive an external storage device when it's located inside the QL's casing? A big help in understanding computers is to realise that they experience reality on several different layers, as if they lived in more than the three dimensions that we inhabit. Computers are incredibly stupid machines with a limited alphabet of two characters, a minimalist counting system of two digits and a philosophy of life that extends as far as the concepts of "Yes" and "No". To make computers look intelligent, software designers have to build up a set of codes out of strings of binary numbers. These codes are used as a basis for other codes, upon which yet more codes are based. Something approaching English language and sophisticated intelligence emerges near the top of the hierarchy.

It is convenient to divide the QL's world into five layers. The first is the physical, the things that are brought to life by electricity. The second is the machine code, the region of wall-to-wall binary numbers. The third layer is the operating system, which manages the housekeeping tasks in the background. The fourth layer is the domain of the command line and the SuperBasic programming language. The final layer is whatever program you choose to run on your computer. The differences between each layer and its neighbours might not appear great, but the difference between a circuit board full of electrons and a chess game capable of beating all but the very best players is enormous.

The Physical Layer

The life blood of a computer is electrical power, so it is appropriate that we begin our journey with an electron's-eye-view of the QL. Computer components mostly run at between 5 and 12 volts, so the power supply is stepped down to an acceptable voltage before being distributed across the QL's circuit board, also known as the motherboard. The chips scattered across the motherboard, whatever their function, contain many tiny circuits etched into silicon wafers, each circuit set to be either "on" or "off". These are the equivalent of transistors. Just to confuse, when a circuit holds a voltage it might be deemed to be off, and when it has no voltage it might be deemed to be on. On and off and "current" and "no current" are simply opposite states and the meaning applied to them is arbitrary. People usually find it easier to think of "on" and "off" representing the binary digits 1 and 0. This is the first level of abstraction. The amount of information held in one bit is far too small to be of value, so computers deal with bits in groups of eight, sixteen or thirty-two. The QL shunts around bits in groups of eight, called bytes.

The heart of the computer is the central processing unit, a single, large chip that conducts most of the data processing. It contains several small storage areas called registers, each large enough to hold 16 bits. In some operations, registers are paired to hold a continuous binary number of 32 digits (or 32 bits, or 32 electrical charges). Unfortunately, to transport numbers to other components the CPU must first split them back into 8-bit (one-byte) chunks, leading to the QL sometimes being called an 8/32bit machine.

An important attribute of the CPU is its speed. The shortest operation carried out within the CPU takes one "tick". Unfortunately, even a simple task such as placing one character on the screen takes scores of instructions, and the QL must repaint the whole screen 50 times every second and still find time between each screen refresh to process information. This is possible because the CPU runs at the improbably fast speed of eight and a half million ticks a second, or 8.5 megahertz (MHz). For comparison, the human heart works at just above 1 Hertz.

At 8.5 million instructions a second, the CPU is a voracious consumer of information. If instructions and data were stored on magnetic devices such as microdrives the CPU would spend most of its time waiting for information to arrive and very little time processing it. To speed things along, data and instructions are transferred from microdrives and diskettes into silicon chips designed especially to store data. These work at around the 80-nanosecond mark, which is just about fast enough for the CPU to keep itself occupied between screen refreshes.

Silicon memory chips come in two forms. Read only memory, or rom, has its contents burned permanently into its circuits so they can never be altered. Random access memory, or ram, contains circuits that can be switched on and off. The disadvantage of ram is that the circuits must be refreshed with a new electric charge fifty times every second otherwise all of the ones turn to zeroes and the memory becomes empty. It would be nice to have memory that could be switched at will but which retained its settings when the electricity was turned off, but such chips are slow and very expensive.

A CPU and fast memory are in themselves a complete computer, but channels are needed for information to be pumped into the system and for the results to be fed out again. Devices that perform these duties include the keyboard, microdrives, disk drives, monitors and printers. Another term for a device is "peripheral": the two words are used interchangeably.

THE MACHINE CODE LAYER

Immediately above the physical layer is the haunt of the machine code junkies, although strictly speaking they use a first-generation, low-level language called Assembler. Machine code is based entirely on numbers, with the position of the number in a series denoting whether it is to be interpreted as an instruction or a piece of data or a memory address. A typical (but ficticious) command might be "204 12 45 65", which might mean "load the contents of memory address 3117 into the first register of the CPU". Whole sequences of commands form programs, and there is no other way of getting the CPU to do anything useful. Numbers might be the stuff of life to computers, but they are difficult for programmers to handle as a language, so most programming at this level is done using a relatively simple program called an assembler (hence "assembly language"). Assembly language is based on a set of mnemonics representing the numeric commands, so that the example quoted above might instead read "ld 12 45 65". Not much of an improvement, but sufficient to make it practicable to write machine code programs.

Early computers had very few instructions, but instruction sets have grown dramatically since the development of the silicon chip. The QL's Motorola 68008 CPU has hundreds of instructions in its vocabulary. The latest trend in CPU design is to reduce the number of instructions available while at the same time accelerating the speed at which they are carried out. This is the philosophy behind Reduced Instruction Set CPUs, or RISCs.

Each CPU has its own unique set of instructions, which is why machine code is sometimes known as "native code". QL programs are written with the Motorola 68008 CPU in mind. Computers with other chips, such as the Intel 80x86 family used in IBM-compatibles, cannot make sense of 68008 machine code, which is one good reason why QL programs cannot run on an IBM pc. Programmers have developed programs that run constantly to give the impression to other programs that they are passing instructions to one type of CPU when in fact they are running on another. Such programs are called emulators. It is easier to write emulators for CPUs in the same family, such as the QL emulator available for Ataris. However, the need to have a constantly-running emulator slows down the computer's ability to carry out its instructions.

The simplest machine code programs carry on running until they run out of instructions, always assuming that the flow of instructions continues to make sense. This is known as linear flow. Because some instructions might need to be carried out repeatedly, and other sets of instructions might need to be carried out only in specific circumstances, machine code includes labels, jumps, loops and branches. A label is a named location in a program that the CPU can search for. Instructions can be issued to force the CPU to go to a label, from where linear processing resumes. A jump is an instruction for the CPU to skip a given number of commands forwards or backwards before resuming linear processing. Jumping and label-seeking instructions can be part of a conditional instruction that in English might resemble "If x = y then go to label Start".

If the CPU moves backwards through a program to reach a label, or if it receives an instruction to jump back a few steps, the result is a loop. If the CPU moves forwards through a program to reach a label or as a result of a jump instruction it has created a branch. All machine code programs comprise segments of linear flow broken up by loops and branches. SuperBasic might dress it up a bit, but every structure in its vocabulary is either a loop (for instance, FOR ... NEXT and REPeat ... END REPeat) or a branch (for instance, IF ... THEN and SELECT ON ...).

Machine code programs are very powerful. Tiny routines can be developed to accomplish quite sophisticated tasks, but they are also difficult to design and write. A simple error such as writing "552" when you really meant "522" can make nonsense of an otherwise correct program. The computer might respond by carrying out inappropriate commands or by waiting endlessly for an instruction that never comes. These are examples of "crashes" caused by "bugs". Debugging machine code programs, which simply means correcting the errors in them, often takes programmers longer than the task of writing the programs in the first place.

The very first programs were written directly in machine code, but very quickly assembler programs came along to simplify the job. Assembler is therefore known as a first generation computer language.

THE OPERATING SYSTEM LAYER

Operating systems are suites of utilities written in machine code to handle frequently-used routine tasks. This saves programmers the effort of producing their own utilities for each of their programs. Instead of writing a little program that causes dots in the shape of an "A" to appear on the screen, programmers simply pass an "A" to a pre-written routine that does all the rest of the work for them. These facilities are so time-saving that they are used by almost every machine code programmer, except on rare occasions such as when programmers got so fed up with the QL's screen-handling routines that they write speed-up programs to replace them. One side-effect of this reliance on the operating system is that programs tend to share a "personality" imposed partly by the instruction set recognised by the CPU and partly by the operating system utilities available.

The operating system's primary concern is communication with peripheral devices. The QL's Disk Operating System (or QDOS) handles all communications to and from microdrive cartridges, the keyboard, the printer and the monitor. Despite its name, QDOS does not include the code to handle disks! These are provided by QDOS extensions written by Tony Tebby.

The suite of small routines that make up an operating system works on a simple basis. Some scrap of information is put into a specific location in the computer's memory and then the appropriate utility is called to do something with it. The information might be the number code representing a capital "A", and the called routine might print it onto the screen. QDOS also needs to store banks of ever-changing information regarding the current display mode, the size and colour of each of the windows, the characters typed at the keyboard that it hasn't had chance to operate on yet, and so on. This data is stored in tight-knit areas of random access memory called "system tables". When you issue the command

```
PAPER #1,4
```

QDOS places the value "4" into the right address inside a system table so that next time it needs to refer to the background colour of the default window it remembers that you have asked for green. The system tables occupy a large area of random access memory that QDOS grabs for itself as soon as the QL is turned on.

THE COMMAND LINE LAYER

The command line layer, which I have extended to include normal SuperBasic programs, is the level QL owners perhaps know best, and needs least explanation. Although obvious to most users, it is worthwhile to state that SuperBasic is actually a machine code program that constantly monitors keyboard and file input. At the end of the boot-up sequence, QDOS begins running SuperBasic and allocates it the special program identification of 0,0.

SuperBasic is an interpreted language. In other words, the SuperBasic interpreter analyses directly and executes the program code that you write. In effect, a SuperBasic program forms a special type of data input for the SuperBasic interpreter program to operate on. Other programming languages, including the QL's Turbo, have their source programs compiled into faster machine code routines by a utility called (unsurprisingly) a compiler. Once in machine code they do not need the machine code SuperBasic interpreter to interpret them: they deal directly with the CPU and operate much faster as a result.

Unless you load a specific type of program, the QL's layers stop here with SuperBasic. However, command line interpreters are becoming very rare these days, and for some QL owners SuperBasic is rarely used, hence the inclusion of the final layer in this article.

THE EVENT-DRIVEN LAYER

Many people think it unfriendly for a computer to hang around waiting for them to type something and then refuse to co-operate because the spelling or the punctuation was not precisely up to scratch. Instead of giving a command to re-size a window, they would prefer to drag the window borders around the screen, or call up a menu option called "Load" which lists all the available programs, rather than type "LOAD flp1_quill" (for instance). These facilities are provided by event-driven interfaces, epitomised on the QL by QPAC II by Tony Tebby. QPAC II replaces command-line SuperBasic with menus, pick-and-point lists and dialogue messages. There is less to learn, because the computer prompts you with the available options. It is a safer environment because inappropriate actions are forbidden and potentially dangerous ones (such as file deletion) must be confirmed. Event-driven interfaces often work best with a mouse - a rolling device that controls a pointer that floats across the screen. When the pointer is over the option you want to carry out, press the button on the mouse instead of hitting keyboard Enter, and you have the equivalent of a SuperBasic command without touching the keyboard.

Computer games are almost all event-driven, often controlling many functions with just a few keys. In the early days of microcomputing, games programmers taught their mainframe colleagues valuable lessons in concise coding and rich data structures. Now games lead the way in offering friendly and powerful interfaces, so why not put similar interfaces onto business programs? The QL screen display, unfortunately, lacks detail and only has a few colours, but a range of workable interfaces is nevertheless now available.

The QL's main strength is its operating system, still the best available for any micro on the market. In turn, QDOS supports powerful programs that are very compact compared with their MS-DOS and Unix equivalents. It is a great shame that weak hardware and poor marketing spoiled it for Sinclair.

GLOSSARY

68008 The least powerful member of the Motorola 68000-series of CPUs. The 68008 shares the same instruction set as its siblings but only has an 8-bit data bus (which also provided the "8" in its name).

bug An error in any type of program that leads to the computer doing something wrong or not doing something right. Relatively harmless bugs are sometimes jovially called "features".

bus A means of channelling digital information (in the form of electrical signals) around a circuit board. In a computer, buses are wires or copper lines, usually in clumps of eight, sixteen or thirty-two which operate in parallel. The two most important buses are the data bus (which transports data) and the address bus (which simultaneously carries instructions about where to store the data).

crash When the computer stops responding to input. This might be due to a bug, a power fluctuation or a power failure. Because the computer has to be re-started, any information stored in ram and not saved in a permanent medium elsewhere will be lost.

cpu Central Processing Unit, the large microchip which contains the main processor or "brain" of the computer.

diskette (disk) A flat disc of video-quality magnetic tape in a case. Magnetised heads in disk drives write data onto diskettes by polarising small sections of the disk. Other heads can read the information back again.

file Any digital information stored under a single filename or header on a disk or microdrive cartridge. The location marker contained in the filename is the basic means of storing and retrieving computer information.

microdrive A means of reading microdrive cartridges. Microdrives contain a reading and a writing head, like a disk drive (see above).

microdrive cartridge (microcassette) A means of storing data magnetically: a loop of about 22 inches of video-quality tape inside a small cartridge.

nanosecond A thousand-millionth of a second, the standard unit for measuring short wavelengths in the class including visible light and electrical current.

ram Random access memory, called so because it takes the same amount of time to retrieve one byte of data no matter where in the ram it is stored. This is not true of tapes and disks because the heads or tape must first move to the right location.

rom Read only memory, called so because its contents can be read into the CPU, but any attempt to write data back to rom addresses fails.

*****

WHAT IS BINARY?

The decimal counting system, or base-10, comprises ten digits, 0 to 9. Provided we use enough of them, these can represent all the finite, real numbers that we can imagine. The binary counting system, or base-2, only has two digits, 0 to 1. Again, provided there are enough of them they can represent any finite, real number. The main advantage of binary is that its digits can be represented by the "on" and "off" states of electronic circuits. The main disadvantage is that you need so many of them to represent even quite modest numbers. Any value from 100 to 999 can be represented using three decimal digits, but three binary digits can only represent 100 to 111 (4 to 7 in decimal). A lesser disadvantage is that the "significant" numbers in the decimal and binary counting systems rarely co-incide. The concept of significance is entirely arbitrary and base-related: 1000 is only thought to be more significant than 986 because of its nice pattern of zeroes. Look at the stock market: if prices fall below a value that ends in lots of zeroes it makes the evening news. The binary value of 1000 is 1111101000, an entirely insignificant collection of digits. The decimal value 1024 is of more interest to computers because it is a one with ten zeroes behind it - which explains why a 128K computer has 131,072 bytes rather than 128,000 bytes in its memory.

One does not need to count in binary to use a computer, but significant binary values crop up all over the place: the memory expansion options, the number of colours that can be displayed, the lack of the flashing status in high resolution mode, the total number of characters in the Ascii character set. If you are a programmer you should at least recognise the significant binary values.

Section Thirty One

The NEW USER GUIDE

Concepts Section

Section Thirty Two

This month we begin the Hardware section of the Concepts guide.

The original Sinclair QL User Guide's Concepts section was an alphabetical mix of the trivial, the valuable and the arcane. Topics were not inter-related and the content was somewhat haphazard. The section on QDOS, for instance, is both inadequate for a machine code programmer and inaccessible to a SuperBasic programmer. The topic on peripheral expansion included the necessary disclaimer that the author did not expect QL users to make their own expansion devices based on two pages of text, but justified its presence as a way of gaining a basic understanding about the expansion mechanism. Potentially useful subjects such as arrays are dismissed in a few lines. The original Concepts section is unclear about its audience, arbitrarily assembled, and ultimately failing in its intended purpose.

The **Concepts** section in the New User Guide began last month with an overview describing the relationship between the layers of a computer system. The remaining sections will be dealt with in the following sequence: hardware, operating system (QDOS) and programming (SuperBasic). The New User Guide's reader is the intelligent non-specialist QL user who wants to make SuperBasic work but is not willing to be buried under technical minutiae. Cross references to other QL World articles are included for those who want to go further.

HARDWARE

Communications

The QL communicates with the outside world through a variety of ports, sockets and connectors. These fall into two major hardware categories: simplex devices that permit one-way communication only and duplex devices that allow two-way communications between device and computer, sometimes only in one direction at a time (known as half-duplex mode).

The simplex input connections are the keyboard (technically a peripheral), and for games joysticks (CTL1 and CTL2). The keyboard connector lies in the centre of the motherboard and is only accessible by unscrewing the body and carefully lifting the cover. Two ultra-thin ribbon connectors thread their way from the keyboard mat (for which the only benefit ever claimed was that it survived coffee spills) into two long, thin sockets. If they become detached they are a devil to force back in.

Plenty of alternative keyboards have been, and are still, available. Advanced keyboards replace the QL's original 6502 controller chip and link to any standard, PC-compatible, 102-key keyboard. This approach is a little more technically demanding than replacing one pair of ribbon connectors with another, but has the advantage that signal timing problems affecting the serial ports are avoided. These days, many QL owners use an external keyboard because they are more comfortable and offer numeric keypads and improved key-mapping (for instance, the Pause key emulates Shift-F5 to interrupt screen scrolling).

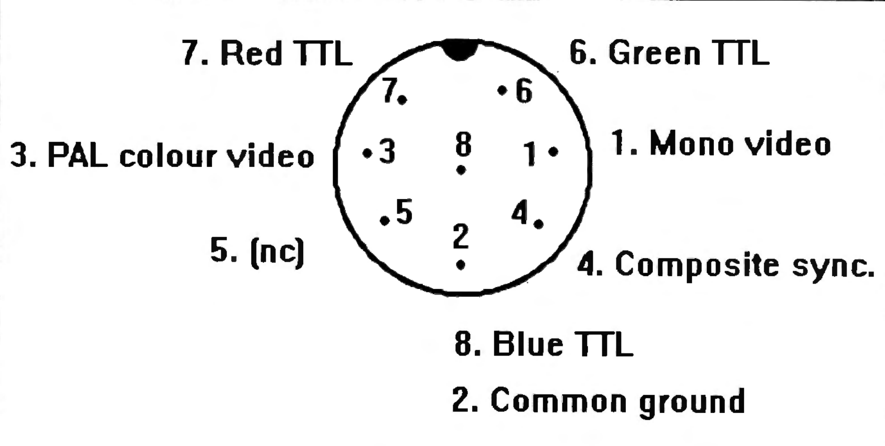
The 9-pin Atari "D" connector games joystick ports, CTL1 and CTL2, have by and large been forgotten by QL users as little use was ever made of them. Games written for two joysticks, such as Psion's *Tennis*, can also be used from a shared keyboard. Signals from the games ports can be read by KEYROW() to emulate the following keyboard presses:

ACTION	CTL1	CTL2
Joystick up	up cursor key	F4
Joystick down	down cursor key	F2
Joystick left	left cursor key	F1
Joystick right	right cursor key	F3
Fire button	spacebar	F5

The only simplex output device is the picture socket. This is a multi-purpose 8-pin DIN socket with output compatible with TTL colour, monochrome video, PAL and composite sync. **Figure one** (over the page) shows the layout of the socket as it is seen looking at the socket in its position at the back of the QL The connector at the other end of the picture-carrying cable depends very much on the type of monitor in use. At least one type of monitor rather dangerously sports an identical 8-pin DIN socket to the QL's but with an entirely different pin-out Connecting the cable the wrong way round can destroy the computer's picture-generating components.

The half-duplex connections are the rom port at the rear left of the QL, the large expansion port on the left of the casing, the two network sockets near the power socket, and the serial ports in the centre of the computer's rear panel.

In the early days, the rom port was occupied by overspill hardware for the operating system and SuperBasic, universally known as the Kludge. More recent and happier times have seen the rom port used by QDOS extensions such as Lightning and SuperToolkit II, language enhancements such as MetaCompco's C compiler, and even device connectors such as a MIDI music port and a hard disk controller. One-to-many units used to be available to allow more than one external rom to be present at a time. Installation is a simple matter of removing the cover plate and inserting the rom into the waiting edge connector, but QLs are well-known to resent frequent changes of occupant in the rom port.



QL multi-format video socket

It is a rare QL that does not have something, usually from Miracle Systems, hanging from the expansion port to the left of the QL body. Some of these expansion devices have a "through port" to allow others to be daisy-chained to them. One of my QLs, for example, has an Expanderam card and a Miracle disk drive controller attached, making the system 66cm wide. The latest favourite for the expansion port is the Gold Card. Installation of any peripheral through this port is easy, although care needs to be taken: it is best to remove the lid of the computer entirely so that you can be absolutely sure that the 64 pins line up precisely with the 64 holes in the QL's socket.

The **serial ports** are, to most users, the printer ports, and are only used for simplex traffic to drive a printer. Serial cables, where files are passed bit by bit down a single piece of wire, are notoriously slow and difficult to set up. The simplest solution to most printing problems is to fit a serial to parallel converter, the device that set Miracle Systems on the road to fame. The QL thinks it is talking across a serial device using its default settings, but the signals are processed inside a small box from which they emerge on a parallel cable with each bit in its own wire. Printers receive the information synchronised byte by byte, which is faster and more secure.

The QL can accept mouse input through one of these serial ports, provided that QDOS has been extended with mouse-aware utilities such as those in the QPTR software. Other uses for the serial ports are to link to modems and to perform file transfers with other computers, notably the IBM PC, the BBC micro and the Sinclair Z88 portable. In these circumstances the port is used in half-duplex mode so that information travels in either direction.

Serial ports are deemed to be connected to DCE or DTE devices (Data Communication Equipment and Data Terminal Equipment respectively). Although in the mainframe world where serial protocols were developed it is clear which device is which, there has always been confusion with microcomputers that can at once be a DCE in relation to their printer and a DTE in relation to a mainframe connection. The QL embodies this confusion by configuring the SER1 port as a DCE and the SER2 port as a DTE, so you can take your pick. For printing, though, SER1 is the obvious choice.

Physically, the serial ports are telephone RJ11 sockets. When the QL came out this was thought by many to be an outrageous cost-cutting exercise, but the socket is very reliable and has since become widely copied by other systems, including the latest PC network structured cabling hubs. The QL's ports can be set to run at speeds of up to 9,600 baud (or transmit on 19,200 baud), but the speed cannot be set independently for each port. See the **BAUD command** in the New User Guide Keywords Section for further detail.

Useful references:

Keyboards: SQLW Feb 91 page 34, Apr 91 page 34, Jan 93 page 20

Monitor connections: SQLW May 87 page 25

Gold Card: SQLW Jun 92 page 22

RS232 serial interface: RS-232 Made Easy by Martin D Seyer published by Prentice-Hall, ISBN 0-13-783472-1

Storage Devices

The working memory of a QL is volatile: when you switch off the computer, you have 0.02 seconds to restore power before everything stored in random access memory disappears. An essential part of the computer is therefore a means of storing information somewhere where constant electrical power is not required to keep the binary information intact.

To begin with, the QL relied on **microdrives**, Sinclair's unique serial-storage format. Later improvements included a variety of floppy-disk formats and even hard disk drive units.

The microdrive cartridge contains a continuous loop of about 56cm (22 inches) of video-quality magnetic tape. The tape spools up through the centre of a drum and follows a path around the periphery of the casing before being wound back onto the drum. At the top of the casing it becomes exposed. Where the heads will make contact it is cushioned by a small felt pad on a springy metal tab. Should the tab fall off it must be replaced if the tape is to be readable. Cartridges can be made read-only by removing a plastic tab from one side of the casing. Should you change your mind, the process can be reversed by applying sticky tape to the gap.

Microdrive cartridges need to be formatted, or divided into data segments, before use. The tape is notorious for stretching over time, which quickly puts the data segments outside the tape drive's limited tolerance. Always format a tape between four and six times and reformat your tapes every year as a precaution against further stretching. Each sector holds 512 bytes and it is perfectly possible to achieve between 210 and 220 sectors on a tape, giving over 100KB of storage capacity.

Stretching is also caused by heat. Unfortunately, the microdrives are placed close to a heat radiator inside the QL's body, so one of the least safe places to leave a tape cartridge for any length of time is in a microdrive slot.

The drives themselves have a mechanically-governed electric motor where contact brushes "lift off" at around 2,400 revs per minute. The motor drives a rubber roller that pulls the tape past a capstan. Crude adjustment to the position of the reading and writing head can be made using a small screw mounted immediately below the mouth of each drive on the base of the QL. Changes should be made cautiously and tested for improvements. As a first trial, try tightening rather than loosening the screws by half a turn. Always remember how many half-turns you have made so that you can return to the original settings if necessary.

Sinclair Research seemed to have a phobia about floppy disk drives, which is why no QL rom contains diskette-aware routines. Although the drives are more expensive than microdrives, floppy disks are much cheaper than microdrive cartridges and far more reliable. Fortunately, QDOS is very flexible when it comes to device types and names and both of the expansion ports can be used to link to disk drives of some sort. The traditional way is to add a disk controller, often with additional ram on board, to the QL's expansion slot. Most QL systems have 3.5-in disk drives of various densities fitted, but early upgraders also chose to use the older 5.25-in disks - the last floppy disk format to be genuinely floppy. Floppy disks are made of the same material as microdrives, but of a stiffer gauge [and flattened into a disk] and magnetically coated on both sides. Formats are Double Density (720K), High Density (1.44MB) and Extra Density (3.2MB). A system will only accept disks formatted to a capacity compatible with both the disk drive and the disk controller. The most capable system on the market today is Miracle System's Gold Card, which copes with all these disk formats (which is not true even of your above-average PC) as well as supporting huge ram expansions and ultra-fast access to the QDOS and SuperBasic routines previously stored in slow roms on the motherboard.

While floppy disk drives arrived early and are still very popular, hard disks for the QL came late. A hard disk drive is permanently attached to the computer and stores, in the QL's case, around 32MB of data. ABC and Rebel once made controllers that attached to the expansion port. Miracle's hard disk unit slotted into the rom port and had a through-port to daisy-chain another rom. More recent hard disk interfaces, like Jurgen Falkenburg's, can be connected in a variety of ways.

Hard disks were expensive on a price per megabyte basis, and noisy, due to the spinning disk and the cooling fan. In the PC world, where 20MB programs are no longer rare, large hard disks are essential, but most QL users are better served by Extra Density diskettes. These offer the same sort of access speeds as hard disks, but are cheaper and more flexible in use. QL programs are very compact compared with PC equivalents, so a single 3.2MB diskette can store a major program and a substantial amount of related data.

Useful references:

Microdrives: SQLW Jan/Feb 86 page 36, Mar 86 page 34, Jan 89 page 33

Floppy disk drives: SQLW Jun 86 page 16, Mar 89 page 34

Hard disk drives: SQLW May 90 page 14, Sep 90 page 36

Gold Card: SQLW Jun 92 page 22

Roms, Eproms and Expanson Boards

The personality of a QL lies in its system roms, mounted on the motherboard. **Sinclair Research roms** came in pairs, but can be replaced by a single Minerva rom. The saga of the roms encapsulates all that went wrong with the QL, but that story belongs to the QDOS part of the Concepts section. The system roms for the QL fall into three categories: the early series up to and including AH, the late series of Sinclair roms from JM to MG, and the Minerva. The early roms had hardware differences which make them difficult to upgrade. All roms from JM onwards, including Minerva, are plug-compatible. In other words, you can remove one rom set and insert any of the others. Minerva looks different from the other roms because it does not fit directly into the motherboard socket but sits on a daughterboard.

Physically, the system roms look like large memory chips: the legs fit into sockets on the QL motherboard. Replacing a rom involves removing any chance of static electricity destroying the components and then levering the old chips from their sockets and firmly and persistently applying pressure to encourage the new chip or chips to take their place.

The NEW USER GUIDE

Concepts Section

Section Thirty Three

This month the Concepts Guide begins QDOS

BINARY, HEX and DECIMAL

The relevance of the **binary counting** system to computers is well known. Ones and noughts can be represented by the presence and absence of an electrical current. Binary digits can be combined in groups of eight to represent values up to 255, groups of sixteen for values up to 65535, or groups of thirty-two for values up to 4,294,967,295. Binary values are also very good at representing **Boolean logic** - on the QL, non-zero values are interpreted as "yes", and zero values are deemed to be "no". See the next section for more details.

Unfortunately, decimal and binary are not easy to translate between. They share very few significant values in common. **Binary numbers** are mind-numbingly long series of ones and noughts. Programmers needed to find a way of representing binary values succinctly, so after flirting for a few years with base eight, or octal, they have almost universally settled for base sixteen, or **hexadecimal** (hex) as their counting base of choice. Hex values need a quarter of the number of digits to represent an equivalent binary number, and translations between hex and binary are very straightforward: every four binary digits become a single hex digit.

Talking of digits, mankind thoughtlessly stopped inventing numerical digits when they reached nine. The hex counting system needs fifteen digits in addition to zero, so the first six letters of the alphabet succeed 9. Some key decimal, binary and hexadecimal values are shown in **Figure one**.

Figure 1: Some key values in decimal, hex and binary

Decimal	Hex	Binary
Eight-bit values		
7	07	00000111
15	0F	00001111
16	10	00010000
32	20	00100000
127	7F	01111111
128	80	10000000
255	FF	11111111
Sixteen-bit values		
32767	7FFF	0111111111111111
32768	8000	1000000000000000
65535	FFFF	1111111111111111

Super Toolkit II contains functions to translate between binary, decimal and hexadecimal values. Alternatively, SuperBasic functions can be created that perform translations. The calculator project published in *QL World* a couple of years ago provided a full set of conversion functions. **Figure two** lists functions that convert between binary values, held as strings, to their decimal equivalents.

Figure 2a: Function to convert a string of binary values to its decimal equivalent

```
100 DEFine FuNction Bin2Dec (binstring$)
110  LOCal result, bit, bitcode
120  result = 0
130  FOR bit = 1 TO 8
140   IF bit > LEN(bitstring$) : EXIT bit
150   IF bitstring$(bit) = "1"
160     result = result + 2^(8-bit)
170   ENDIF
180  END FOR bit
290  RETURN result
200 END DEFine Bin2Dec
```

Figure 2b: Function to convert a decimal value between 0 and 255 into its equivalent binary value

```
300 DEFine FuNction Dec2Bin$ (num)
310  LOCal result$, x
320  result$ = FILL$("0", 8)
330  FOR x = 0 TO 7
340   IF 2^x && num: result$(8-x) = "1"
350  END FOR x
360  RETurn result$
370 END DEFine Dec2Bin$
```

Boolean Algebra

The English mathematician **George Boole** (1815 - 1864) wrote two important treatises: *The Mathematical Analysis of Logic* (1847) and *An Investigation of the Laws of Thought* (1854). He applied his mathematical knowledge to what had previously been the philosophers' preserve of human logic. His name is now synonymous with computer logic, which tends to see things in black and white terms.

His logic vocabulary is not extensive: TRUE, FALSE, AND, OR, XOR and NOT form the bulk of it. True and false are conditions easily represented by one and nought. The other words are operators similar to plus, minus and divide. The essentials of Boole's logical algebra are that statements that are true or false can be combined with Boolean operators to form further expressions that can be evaluated to be true or false. The following examples use **pseudo-code** (a form of note-taking that mixes SuperBasic keywords with standard English) to demonstrate the IF statement's debt to Boolean algebra:

IF it is cold THEN I will wear a coat: REMark what, even indoors?
IF it is cold AND I am outside THEN I will wear a coat: REMark but what if it is mild and raining?
IF (it is cold OR it is raining) AND I am outside THEN I will wear a coat

In short, if two statements that are true are linked with AND then the result is also true. Other combinations would involve at least one false statement and the result is false. If the link is OR, then if either statement is true then the result is also true. If both statements are false the result will be false.

In the final example, brackets are used to show precedence. There are actually two Boolean expression pairs even though there are only three things to test: the first pair is contained in the parentheses and the second pair is the result of evaluating the bracketed expression compared with the third statement.

The pseudo-code contains implicit true/false conditions that are better brought out by rewriting the IF statements along the following lines:

"IF cold = TRUE AND outside = TRUE THEN wear_coat = TRUE".

The only **changes** now needed to form a valid SuperBasic line are to assign values to the variables cold, outside and wear_coat and replace TRUE with 1. In SuperBasic, any zero value can be interpreted as being false and any non-zero value as being true.

Turning to the real world of computer data, most people's linguistic instincts mislead them into using AND when they mean OR and vice versa. Imagine a customer database being searched for records that fulfil two conditions. In English we might say "Show me all customers in Yorkshire and all customers in Lancashire", but Boolean logic insists that the AND is replaced by OR: "Retrieve all records where Location = Yorkshire OR Location = Lancashire".

Computers also have to represent a logical condition that is awkward to express in English: where one thing is true or another thing is true, but not both together. The concept has been given the name XOR, or Exclusive OR. The table at **Figure three** shows truth tables that demonstrate more clearly the workings of the logical operators.

Figure three: Truth tables for AND, XOR and OR

AND	XOR	OR
101	101	101
100 &&	100 ^^	100 ||
100	001	101

Note: SuperBasic bitwise operators are shown: these can be used in programs or direct commands

There is no sense of "carrying" values between columns in bitwise logical expressions.

QDOS makes most obvious use of Boolean logic in its handling of screen colours (See the section on Colour for details) when the OVER -1 command is invoked. This XORs the current screen colours with whatever is on the screen to produce a new combination of colour.

BREAK, SWITCH AND PAUSE

The following key combinations are trapped by QDOS for special purposes.

A SuperBasic program can be interrupted using the Ctrl-Space keys. On some external keyboards this facility is also mapped to a dedicated Break key. Sadly, Sinclair have given no such power to the Esc key, even though it was the most obvious candidate. Their preference was to allow software developers to trap the Esc key and attach whatever significance they wished to it.

The Ctrl-F5 keys can be pressed to interrupt a window that is busily scrolling a column of information off the screen faster than you can read it. On external keyboards, this facility might also be mapped to a Pause key. *Super Toolkit II* implements this feature automatically for commands such as DIR and WLIST, prompting you to press a key to release each screenful.

If several jobs are running simultaneously or concurrently, Ctrl-C can be pressed to move between jobs. Some quite respected programs run simultaneously with SuperBasic. This gives you instant access to the command line and saves the programmers from writing basic file handling routines. The disadvantage is that the screen can get to look very untidy and confusing with SuperBasic commands mixed in with the application's text. The Psion suite, unless doctored to behave differently, only run in a stand-alone mode. Try launching *Quill* with the EXEC command rather than EXEC_W and then experiment with Ctrl-C.

Ctrl-Alt-F7 is the rather strange combination left in by QDOS developers at the end of debugging (if that process was ever formally ended). It has the unwelcome effect of crashing the QL.

DEVICES AND CHANNELS

A device is anything capable of receiving or sending information. A channel is the link between a device and the CPU along which information is sent or received. The revolutionary thing about QDOS was that vastly different devices could be treated more or less identically because as far as the computer was concerned it connected itself to channels: what lay at the other end became less important. This rationality contrasts sharply with the mess PC users have to contend with.

The class of "devices" is large and disparate. Obvious devices seem to include printers, monitors, and disk drives because they are distinct, physical things. However, only the printer meets QDOS's strict definition of a device (and pedants will tell you that it is the serial socket that the printer is connected to that is the true device!): the others are better termed peripherals. QDOS, for instance, has no way of communicating to the monitor screen without first opening a window: the screen might have dozens of windows open and each is a separate QDOS device. The same is true of a disk drive or microdrive: it is not the storage medium that QDOS connects to but to a specific file on the medium.

Devices also include things that cannot be touched, such as network connections to distant computers or ram disks (areas of memory formatted to resemble microdrives). Interestingly, a special case has had to be made for the keyboard. It might be a peripheral but it is not quite a device in QDOS terms. Type in a command and the keyboard can be deemed a part of Window #0. If the instruction is an INPUT statement, then as soon as it is executed keyboard activity is reflected in Window #1. It is not the keyboard from which QDOS receives key presses, it is from the channel attached to Window #1. Should you then press Ctrl-C to switch to another task, the keyboard should move with you, otherwise you cannot communicate with the task. This behaviour has two effects worth knowing about. Firstly, if you close Window #0 in a program you might be able to re-open it as a console device, but until the QL is rebooted you will not be able to issue SuperBasic statements from it. Secondly, QDOS has two sorts of windows: consoles, which have two-way communications, and screens, which do not: if you declare a channel to be linked to a screen, you can send it text and graphics, but the INPUT and INKEY commands will be ignored.

Channels lie at the heart of the QL's flexibility with devices. Like an irrigation system, QDOS channels convey information between the CPU and devices. A channel can be opened to a window on the screen and text sent to it. The same channel can be diverted to a printer, or to the network, or to a file, and the same process repeated: programs shovel character codes into the channel and more or less wave them goodbye. QDOS invisibly and silently ensures that the information is presented to the device at the other end in the manner it best understands. One of its jobs is to filter out incompatible data or instructions, such as sending PAPER and INK commands to a printer or file.

There are one-way (simplex) channels such as those attached to screens and networks and two-way (half-duplex) channels such as those attached to consoles and serial ports. The QL does not permit simultaneous two-way communications (full duplex).

When the QL is first powered-up three **default channels** are created: #0 for command input, #1 for default screen output and #2 for program listings. If other channels are required, for instance to link to a printer or a file, they must be created and configured (or their defaults accepted). The syntax for doing this is complex and sometimes inconsistent. Firstly, we must review the OPEN family of commands:

OPEN #x, dev_name tries to open a two-way channel numbered #x to the device called dev_name. If the channel links to a screen or a network, only one-way communication will be permitted.

OPEN_IN #x, dev_name tries to open a one-way channel to an existing file on a microdrive, disk drive or ram drive, in effect making the file "read-only". This command is only for use with files.

OPEN_NEW #x, dev_name tries to create a new file and then link a two-way channel to it. This command is only for use with files.

The construction of a valid dev_name for the OPEN commands often fills programmers with dread. However, useful defaults are accepted by QDOS in the absence of some of the details. A console can be opened, for example, with the simple command OPEN #5, CON_ and then adjusted for size and location with the easier-to-handle WINDOW command. For those who want their devices properly specified, here are the commoner options:

Consoles:
con_wideXhighAxposXypos_buffer,eg con_200x100a50x50_64, where wide and high determine the size of the window, xpos and ypos determine its screen location and buffer indicates the size of the keyboard type-ahead buffer. The defaults are the equivalent of con_448x180a32x16_128.

Screens:
scr_wideXhighAxposXypos, eg scr_340x140a48x16, where wide and high determine the size of the window and xpos and ypos determine its screen location. The defaults are the equivalent of scr_448x180a32x16. Note that screens are one-way devices that ignore keyboard input and so do not have a type-ahead buffer.

Serial ports:
serNPHZ, eg ser1ehr, where N is 1 or 2 to indicate serial port 1 or 2, P reflects the parity protocol (the first character of Even, Odd, Mark and Space), H reflects the handshaking protocol (H for handshake and I for ignore) and Z reflects the data transfer protocol (R indicates no end-of-file code will be sent, Z indicates that a Ctrl-Z end-of-file marker can be expected, C indicates that QDOS's CHR$(10) newline character will be converted to the more usual CHR$(13) carriage return). All of these settings are optional and they can appear in any order. Note the lack of an underscore and a fixed parameter sequence compared with other device declarations.

Network connections:
netD_S, eg neti_0, where D indicates the direction information flows (I for input and O for output) and S is a station number (on a two-station network, each QL can be 0, but for larger nets each QL needs a unique station id less than 63). Note that this is an entirely different syntax for one-way data traffic to the OPEN_IN equivalent associated with files.

Files on microdrives, ram drives, and disk drives: devX_filename, eg FLP1_MyFile, where dev can be MDV, ram or FLP, X is a unique sequential number for that particular device type, and filename is a valid filename. Similar device names, ie three letters followed by a number and an underscore, have been associated with third-party disk drives and other devices.

CHARACTER SET

The QL's character set is based on the well-known **Ascii** (American Standard Code for Information Interchange) conventions, albeit with its own wrinkles added in for good measure. Incidentally, Ascii is on its way out in the computer world, being replaced with its close cousin the ANSI (American National Standards Institute) character set. Characters and bytes are closely related: there is a one-to-one relationship between them and they can readily represent each other. For instance, the QL's screen map comprises 32,768 bytes, and chunks of the screen can be saved to

The NEW USER GUIDE

Concepts Section

Section Thirty-Three

This month the Concepts Guide continues with QDOS.

FILES

QL users tend to have a clearly-defined understanding of what constitutes a file, but every device on the QL appears to Qdos as a **file** of some sort. This is something the QL has in common with the Unix operating system and makes for a very powerful, consistent and compact **device** operating system. Devices such as screens and printers are, in effect, infinitely large files. Real files, bound by the restrictions of the medium on which they exist, have a finite length.

All files, **real and virtual,** share some common characteristics. They must each have a unique **name**. They each have a Qdos **channel** associated with them (with the possible exception of the sound chip, which should be a device but which does not have a conventional channel accessible to SuperBasic). Each channel has a memory block allocated to it to hold Qdos-related information.

From now on we will ignore virtual files such as screens. Conventional files fall into three types. There are **data files** that can contain data created at the command line or in programs, data created by applications like Quill, or SuperBasic programs. The inclusion of the latter might appear odd, but SuperBasic code is data input to the 0,0 job running under Qdos, the SuperBasic interpreter. Data files can be created with the PRINT command to a file open for writing or the SAVE command for SuperBasic program lines. The file contents are retrieved into memory using the INPUT and INKEY$ commands. Super Toolkit II users can also read and write using the GET, BGET, PUT and BPUT keywords.

The second class of file is the **executables** that run either simultaneously with others (multi-tasking) or which grab the attention of the system until they are closed (single-tasking). This class of file can only be run using the extended family of EXEC commands. About the only way ordinary SuperBasic users are able to create executable files is through the Turbo compiler.

The third class are repositories of **sections of QL memory** saved with the SBYTES command and retrieved with the LBYTES command. The contents of memory thus saved might be the default fonts, the screen map, Qdos channel tables, a SuperBasic program in its tokenised state, or (most likely) a binary program that can be invoked with the CALL command.

Files can be stored on microdrives, floppy disks or on ram disks, which are areas of working memory formatted to act exactly like microdrives but unable to store information without constant power. Storage media are formatted into blocks or sectors of a pre-determined size. Large files will occupy several sectors. One disadvantage of this arrangement is that a whole number of sectors must be allocated to a file: if there is space left over in a file's final sector it is unusable. Storing a few large files is more space-efficient than storing several small files because the number of partly-used sectors is so much lower.

Files are given a header of 64 bytes, broken down as follows:

00 - 03	**An integer representing the file length**
04 - 04	**The file access key**
05 - 05	**The file type byte**
06 - 13	**Eight bytes of file information**
14 - 15	**The length of the filename (up to 36 bytes)**
16 - 51	**Reserved to hold the filename**
52 - 55	**(disks only) Date stamp**
52 - 55	**(microdrives) not used**
56 - 63	**Reserved**

The **file access byte** is normally set to zero. The **file type byte** is set to one for executable files and is zero for both of the other file types. For executable files the first four bytes of the file information block form a long word integer representing the size of dataspace to be allocated to the task as soon as it is launched. Super Toolkit II and Minerva implement date stamping when creating and updating files on microdrives, but neither can alter the inherent dating behaviour of floppy disk controllers.

Microdrives and floppy disks do not just contain conventional files. The first sector of a microdrive, for instance, is reserved to hold a **map** showing which segment of which file is stored in each of the other 512-byte sectors. Microdrives also need directory files, normally invisible to the user, containing copies of the file headers for each of the files on the medium. Qdos also avoids writing to bad sectors by assigning them to a non-existent and unreachable file.

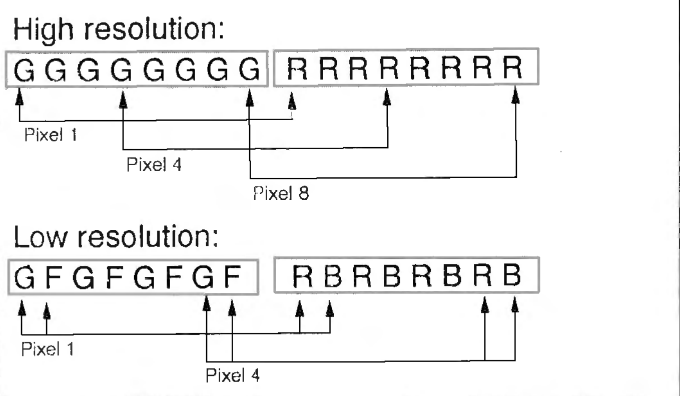
When microdrive tapes are formatted the sectors are created and the **sector map** and directory entry are written. The reported total sector count is usually around 220 sectors. If you are lucky, the available sectors will be exactly two less than the maximum: one sector is taken up by the microdrive map and the other by the directory. The length of microdrive tapes must be kept within a maximum size of 255 sectors otherwise the mapping conventions would break down. No matter how carefully you treat your microdrives, one sector is always going to be bad

- the one where the tape is spliced - and it is ignored in the sector count.

Floppy disk formats are more complicated than microdrives to take account of their several tracks and two sides. Differences include the following: areas of disks are allocated in clumps of three sectors at a time, the file headers still exist but are more or less redundant in favour of the directory map, and after formatting a perfect 1440 sectors is normally achieved, 6 of which are immediately allocated to the disk map and the file directory.

Users may have noticed something slightly spooky when running small programs on QLs with large memories. After spending an age finding the first program on a microdrive, it suddenly seems that only the briefest examination of a drive is

Screen map layout - first 2 bytes



necessary to load the next program or piece of data. This is because Qdos does its level best only ever to read microdrive and disk sectors once. It copies images of all sectors it happens to pass across into any ram that is otherwise unoccupied. If the user requests access to a file stored on sectors paged into memory, Qdos goes to the memory image rather than the original source. As memory access is considerably faster than disk or microdrive access the result is a significant acceleration in loading speeds. The brief twitch of the drive light is Qdos checking that you have not sneakily swapped one microdrive for another while it was not looking. Of course, as you fill up ram with multi-tasking programs or large data structures there is less room for Qdos to squirrel away copies of the microdrive sectors and so it must resort to the slow task of reading directly from the medium.

IDENTIFIERS

Loop identifiers and **variables** can look very similar, but there are significant differences. Every REPeat loop needs to be given a name, using the same rules as those governing numeric variable names. Other languages make do without loop names, but they demand an explicit exit from each level of a nested series of loops. In SuperBasic, if you are in the deepest level of a set of nested loops you can escape with a single bound using just one EXIT command, as long as it mentions the outermost loop's name.

FOR...NEXT loop identifiers have a specific numeric floating point value that can be used within the loop. REPeat loops have no such luxury, so a loop counter must be created and maintained explicitly. The variable used in a SELect structure has this same restriction of being a floating point number. Minerva users and creators of compiled **Turbo** code can ignore these restrictions. In the case of Turbo, it is well worth the effort of fooling SuperBasic into using integer variables for loops: the acceleration is dramatic.

MEMORY MAP

Every location in the QL's chip-based memory has an address, beginning at 0. The first 48K is reserved for **system rom**, the internal layout of which changed significantly to put language-specific data such as fonts and error messages in the higher addresses. The next 16K is allocated to **rom boards** plugged into the back of the QL. There follows an area of 8K reserved for input and output. The screen map, 32K in size, follows that. The rest of addressable ram is then apportioned between the **user, Qdos and microdrive slave blocks.** Things placed within this area have a habit of being shunted around as Qdos re-organises the memory to meet new demands.

Qdos grabs a significant part of the QL's available ram, most of it located immediately after the screen map beginning at location 163840. It begins with a tableaux des tableaux that includes pointers to other tables that may in turn point to yet other tables. **Tables** might contain device channel information, variable names and values, keywords, and so on. SuperBasic programs are stored near the top of available ram and are held in tokenised form. Part of the remainder forms a **"heap"** from which memory may be allocated. Unused memory is sectioned into blocks to mirror microdrive storage sectors (see the earlier entry on Files for more details).

It is not possible to give a comprehensive listing of Qdos memory usage within the format of this guide. Nevertheless, some of the most useful addresses are listed in the first table.

NETWORK

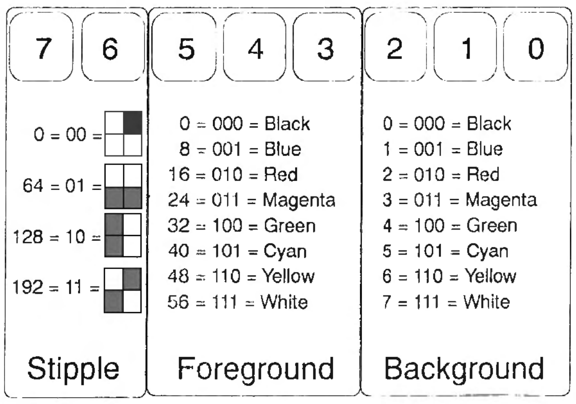
As with so much else connected with early Qdos, networks

Pure colour combinations

Colour	Green	Red	Blue	Value
Black	Off	Off	Off	000 = 0
Blue	Off	Off	On	001 = 1
Red	Off	On	Off	010 = 2
Magenta	Off	On	On	011 = 3
Green	On	Off	Off	100 = 4
Cyan	On	Off	On	101 = 5
Yellow	On	On	Off	110 = 6
White	On	On	On	111 = 7

did not really work until **Super Toolkit II** arrived. Many QLs before those with a D14 batch number were built and sold without working network ports. Since then, there have been several interesting success stories with connecting several QLs - even with some of the notoriously unco-operative pre-D14 production batches. The physical layer of a network is rudimentary in the extreme: bell wire is connected to the network ports behind the microdrive housing of all of the QLs partaking in the exercise to form a circuit. This is usually the easy bit.

Sinclair QL colour byte



Data is transmitted across the network in blocks reminiscent of the way files are blocked onto sectors of a floppy disk or microdrive. Each block has a **header** identifying where it has come from, where it is going, its size and whether or not it is the last block in the set. Sending **network blocks** is exactly like sending Morse code with a torch down a tunnel: everyone looking into the tunnel can see flashing lights, but if two signallers try it at once the result is confusion. The network is managed by a protocol - a set of rules that determines who can send what and when. The stations transmit "scouts" to see if the network is free and there is someone listening. If the scout is successful, blocks get transmitted. Even though the blocks may be destined for one particular QL, every other QL on the network detects the signal, but ignores it.

At the operating system level, QLs are joined to the network by channels. The following commands represent some the options available:

OPEN #6, neti_3_12: REMark input channel, 3 is this QL's identifier, 12K buffer
OPEN #7, neto_7: REMark output channel to QL station 7
OPEN #8, neto_0: REMark general broadcast to everyone else

With Super Toolkit II installed, input channels can have a buffer size allocated to them, otherwise the network will grab almost all the free memory available. If you try networking without Super Toolkit II (surely a recipe for frustration) this refinement is not available. Once the channels are open, files are transmitted with the COPY family of commands.

Each computer (or "**station**") on the network has to have a unique identifier so that it can recognise data blocks intended for itself and ignore blocks destined for other computers. Where there are only two QLs in the net this is not an important factor, provided they only open **one input channel and one output channel each** to the network, because they can both use the default station number of 1. When three or more are gathered together, each QL should be assigned a station number with a command such as OPEN #6, neti_4. Station numbers must be integers within the range 1 - 63.

A much more valuable mode of network operation can be obtained by invoking Super Toolkit's **file server task**, FSERVE. This multi-tasking utility sits in the background connecting remote QLs to up to ten resources (windows, printers, disk drives, etc) on the file server. Filer servers must have a network identifier in the range 1 to 8, and the lower the better. The file server is implemented by issuing the FSERVE command. To all the other computers on the network, the file server appears as a two character prefix placed before any normal channel information, such as:

OPEN #6, n3_con_200x40a0x0: REMark open a window on the file server
INK#6, 4: PAPER#6, 0: INPUT #6, "Can you hear me, Mother?", reply$
OPEN#7, n3_ser1: PRINT #7, "This is output to a remote printer"

Even better, remote devices can be given what appear to be local names using the NFS_USE command. This works exactly like the more familiar FLP_USE by assigning a virtual drive name to a remote actual drive. FLP1_ on a file server can become MDV1_ by issuing the command:

NFS_USE mdv, n3_flp1_, n3_flp2_

In practice, FSERVE is a **device server** of great flexibility and real benefit. About the only weakness (and this applies to all networks no matter what the computer) is the possibility that a remote station is turned off with files open on a remote server. The potential for damage is reduced by the user of the file server explicitly cancelling the SERVER job before turning off the computer using the RJOB SERVER command.

QDOS

The QL's operating system contains device drivers, permits multi-tasking and it was designed to be extendible right from the outset. It was demonstrably far ahead of the version of Microsoft DOS current at the QL's launch and has several advantages over even the latest version of MS-DOS. Given Sinclair Research's initial distaste for such conventional things as floppy disk drives, it seems unlikely that Qdos was intended to stand for QL Disk Operating System. Some prefer QL Device Operating System, but it is not clear that this was thought of when the QL was launched: early publicity material studiously avoided giving any decode of the initials. Perhaps it stood for nothing very much but was designed to recall to people's minds MS-DOS and the general DOS concept.

An unusual but very useful feature of Qdos, almost always overlooked, is its **integration** with SuperBasic, saving on duplicating file handling facilities and encouraging non-programmers to type in at least a few commands at the command prompt.

SCREEN COLOURS

Colour is represented within the QL at two levels: firstly, all foreground/background/stipple colour combinations can be represented by a single byte and, secondly, coloured pixels are represented in a raster scan scheme within the screen map. This topic deals with both representations in turn. Note that foreground and background colours are not the same as INK and PAPER, but refer to the combinations of colours that make up stippled shades.

All possible colours on any monitor or TV screen are created by mixing light from combinations of three potential sources, one red, one green and the other blue. In natural-colour systems, such as televisions, the individual colour sources can be fired at different strengths, but for the QL the options for each source are either on or off, leading to just eight colours.

The **table of colour combinations** shows that the eight colours can be represented using three bits. Within any colour byte the last three bits represent the background colour and the next three bits represent the foreground colour. This occupies six of the eight bits in a byte, leaving the first two bytes free to represent something else. On the QL, the first two bits of a colour byte identify the stipple used to mix the foreground and background colours.

SuperBasic allows **colour statements** to have up to three parameters, representing background, foreground and stipple. PAPER 5, for instance, produces solid cyan and INK 4, 2 produces an even stipple of red and green. There are some instances where instead of three parameters it can be useful to declare the full colour combination with only one value. This can be done by referring to the **Colour Byte table** and adding the values together for the required stipple, foreground and background. PAPER 167 will produce stripes of green and white, for example.

Some useful QDOS addresses

163856	Long	Base of SuperBasic area
163886	Word	Random number
163890	Byte	Holds 1 if in TV mode
163891	Byte	Holds 0 if screen active, 1 if frozen
163895	Byte	Network number
163900	Long	Pointer to polled task list
163904	Long	Pointer to scheduler tasks list
163908	Long	Pointer to device drivers list
163912	Long	Pointer to directory devices list
163916	Long	Pointer to start of keyboard queue
163960	Long	Pointer to start of channel table
163976	Word	Caps lock: 255 = on, 0 = off
163980	Word	Key auto-repeat delay
163982	Word	Key auto-repeat rate
164078	Byte	Currently active microdrive

The way that screen colours are represented in the QL's memory map makes it difficult to control colour directly in the memory map. In low resolution mode, each set of four pixels share 16 bits-worth of data in the memory map. To reach high definition, eight pixels have to share the same amount of data, which reduces the range of available colours by half but doubles the number of pixels that can be addressed. The virtual screen that all of the window-related commands recognise is a constant 512 pixels by 256 pixels, but in low resolution mode the real screen is only 256 pixels wide by 256 pixels deep. Figure four shows the relationship between bits in the screen map and coloured pixels on the screen. Note how Sinclair made use of the "spare" bit in low resolution mode to signify flashing.

There have been moves to add another screen resolution to the QL for some time and it may soon become reality for Miracle Super Gold Card owners. This will be a godsend to users of programs with a lot of detail like Professional Publisher. Keep reading QL World for details.

START UP

When a QL with a Sinclair rom starts up or is reset the screen offers two choices: press F1 to begin a high-resolution session or F2 for a low-resolution session. Although Sinclair recommended using F2 only on televisions, either option can be selected whether you are using a monitor or a television, although windows sizes and locations may need adjusting if they are not to bleed off the picture on a TV set.

When a QL has its rom replaced by a Minerva, the start-up sequence offers slightly different choices. F1 and F2 work as before. F3 and F4 start up a second screen using high and low resolution respectively. If no key is pressed within about 15 seconds of startup, **Minerva QLs** will assume F2 has been pressed and will proceed anyway. This can be useful to ensure kick-starting a system left on its own that has suffered from a power failure. However, the chances of a microdrive or floppy disk being readable after a crash and reboot are distinctly low.

Minerva roms also trap the Ctrl-Alt-Shift-Tab sequence to force a reboot, as an alternative to pressing the little spring-loaded button underneath the right edge of the QL's superstructure.

Super Toolkit II contains two handy keywords to reset windows to the locations and sizes they began a session in. WMON establishes high resolution attributes and WTV does the same for the low resolution option. Either command must be followed by a positive integer less than 512 to represent the display resolution. WTV 4, for instance, sets up the default window locations for a low resolution session but places the screen in high resolution mode. WMON 8 does the opposite. These commands do not change the window colours or border colours: these have to be adjusted using other SuperBasic commands.

WINDOWS

A window is an area of the screen that operates independently. Unfortunately, when windows overlap on the screen there is no sense of one window being "on top" of another window. In Qdos, whichever window is written to last will **overwrite** whatever the other screen had previously displayed. Colour, scrolling, character sizes and so on are controlled separately for each window. Qdos keeps track of these attributes using a special channel table for each window.

There are in fact two distinct types of window recognised by Qdos, those that accept input and those that do not. Windows in which the INPUT command does something are called **consoles**; windows in which the INPUT command is ignored are called **screens.** A window is defined as being either a console or a screen when its channel is opened. All of the default windows opened when the QL is booted are consoles.

Screen referencing uses one of **four co-ordinate systems:** screen, character, block and graphics. The **screen** co-ordinate system is used to define the location of windows and has its origin at the top left corner of the screen. The origin for both the **character** and the **block** co-ordinate system is at the top left corner of a window, while that for graphics is at the bottom left corner of a window unless it is moved with the SCALE command. Block co-ordinates directly correspond to pixels and are used by the BLOCK command. Character co-ordinates are related to character sizes set by the CSIZE command. **Graphics** co-ordinates reflect the current setting of the SCALE command and so unlike the other co-ordinate systems is not based on pixels.

Some screen-related commands, such as PAN, SCROLL and CLS, recognise window parts using the following codes:

0 The whole screen
1 Above the cursor line (but excluding it)
2 Below the cursor line (but excluding it)
3 The cursor line
4 The cursor line to the right of the cursor, including the cursor

The NEW USER GUIDE

Concepts Section

Section Thirty-Five

This month the Concepts Guide continues with Arrays, Basic and the QL Clock.

ARRAYS

If a single variable was a bicycle, capable of transporting only a single piece of information, an **array** would be the equivalent of a double-decker bus, able to arrange masses of information in rows and columns. Arrays are incredibly important because they are the only mass data storage concept available to SuperBasic programmers without recourse to writing their own data structures.

An array is a **set of variables** that share a common name and are distinguished by one or more subscripts. An array designed to hold names of football team members (including substitutes and so on) might have the following structure:

```
Player$(1)
Player$(2)
Player$(3)
....
Player$(22)
```

Whereas ordinary variables can be used without any preparations, the first step towards using an array is to **declare its dimensions using a DIM statement** such as DIM Player$(22, 20). It saves time to dimension many arrays using one DIM statement, such as:

```
DIM Player$(22, 20), Matches(22, 26), Goals(22, 26)
```

In procedures and functions, **local arrays** can be declared using the LOCal statement at the beginning of the code, such as LOCal Temp$(12, 14), Tval(6). Arrays can be passed to procedures and functions as arguments, but only by reference and never by value. Once declared, the dimensions of an array can be discovered using the DIMN function.

A strange result of including a DIM statement in a program is that until the program is run and the line is interpreted, any reference to the array name in a direct command will cause an error. This seems to be because Qdos becomes aware of the variable name and so includes it in the appropriate lookup table as soon as the DIM line is entered, but it does not allocate memory space until the SuperBasic interpreter reaches the DIM command and carries it out.

Array names obey all the rules associated with ordinary variables, including the use of a trailing dollar sign or percentage sign to indicate their type. The dimensions of the array appear in brackets immediately after the variable name. In the example above, the array contains eleven rows each of twenty columns. Each column in a two-dimensional string array can hold one character: in numeric arrays each column can hold a complete value. DIM Manager$(20) is little different from a normal string variable, except that its absolute length has been made explicit.

Sinclair Basic **string array conventions** are unique among Basic dialects in that you must explicitly declare a maximum length for strings when the array is first declared. The convention has both strengths and drawbacks, and having been an ardent supporter of the Sinclair way of handling string arrays in the past I must say that I am now less inclined to argue strongly for it, although it still has some useful strengths.

The main irritation of Sinclair arrays is the need to **rope off memory space** that might never be used. To accommodate a set of strings, the longest of which is 100 bytes, but the average length is nearer 30 bytes, every string in the array must be 100 bytes long. The same goes for rows: unless you have memory to spare it would be foolish to declare an array with 600 rows just on the off-chance that they might all be used.

On the other hand, it is wonderful to be able to extract the fourth character of the ninth string array element with the statement X$ = ELEM$(9, 4) rather than the depressingly clumsy X$ = MID$(ELEM$(9), 4, 1) syntax found in other Basics. Indeed, this simplicity is extended to ordinary string variables not dimensioned as part of an array, which is unique amongst Basic implementations but can be found on more powerful languages such as C. Fortunately, the disadvantages of the Sinclair system can by and large be corrected with a bit of programming, but the syntactical muddles of other Basics are only overcome by adopting another language.

Arrays can have **a large number of dimensions**, but the most common are single dimension arrays, or lists, and two-dimensioned arrays, or tables. Continuing the football team scenario introduced earlier, let us assume that a season comprises 26 games and that a code of 2 indicates that a player was on the pitch for the whole match, 1 indicates a substitution and 0 means he or she did not play that week. The best way of holding this information might be a two-dimensioned numeric array declared with the command DIM MATCH(22, 26). This creates a table of 22 rows, one for each member of the full team, and 26 columns, one for each game of the season. Of course, if the team is also part of a knockout series and plays in Europe, two other arrays are required or a third dimension could be added to the MATCH array.

Array **subscripts** are whole numbers within the range 0 to 32,767, although the final subscript to a string array (ie the length of each string) must not exceed 32,766. No array can contain more than 65,535 elements, a figure that excludes the final dimension of a string array. As the space required to hold an array is allocated in full as soon as it is declared, a further restriction on array size is imposed by the memory available in your QL.

Apart from the unusual declaration rules, string arrays on the QL exhibit one or two **further oddities**. The first is that even when the maximum length of strings within a string array is declared with an odd number the true length of the strings will be increased by one to make the total length an even number. The second oddity is that the **zero element** in the final dimension of a string array is inaccessible. Numeric arrays include the zero element, so that MATCH(0, 0) is a valid array reference. PLAYER$(4, 0), however, is not, because this is where SuperBasic stores the true length of the string. You can read from this location (it returns a numeric value), but you cannot write to it.

A significant **advantage** of arrays is the ability to scan through them using variables as subscripts. For example, the names of the members of the football team can be entered with the following code snippet:

```
FOR X = 1 TO 22
INPUT "Enter a player's name", Player$(X)
ENDFOR
```

Other ways of getting information into arrays are from hard-coded expressions, from DATA statements, and from data files with each line holding a single value. Example snippets for all three follow:

```
FOR x = 1 TO 12: Random(x) = RND(20)
FOR x = 1 TO 12: READ DataLine(x)
REPeat loop
INPUT #5, BigList(x)
x = x + 1
IF EOF(#5): EXIT loop
END REPeat loop
```

All of the Sinclair Basic dialects are good at handling array slicing, in other words extracting small parts of a larger array structure. PRINT Player$(1 TO 22, 1), for instance, prints the initial letter of each footballer in the array.

SuperBasic is less good at handling array slices to the left of an expression. LET Player$(1 TO 22, 1) = "?" produces a plaintive "not implemented yet" message. The beginning of a slice must be explicitly included, not implied with a PRINT Player$(5, TO 12) syntax. The end of a slice, however, can be left to the computer to gauge, as with PRINT Player$(3, 6 TO).

Should you omit all subscripts from an array reference, SuperBasic attempts to give you the whole array. PRINT Player$ is therefore the easiest way of obtaining a printed list of all the array contents. Unfortunately, this option cannot be used to initialise an entire array to something other than zeroes or null strings.

Arrays can be passed to procedures and functions in order to be operated on, but you cannot take advantage of the optional ability of the SuperBasic interpreter to pass arrays by reference instead of by value. In other words, array names cannot be passed to procedures or functions within brackets.

Redimensioning

When SuperBasic arrays are **re-dimensioned** Qdos blanks out all previous values to start again with a clean sheet. With the Turbo compiler, however, arrays can be re-dimensioned "on the fly" without losing all of their contents. This can be exceptionally useful when adding or deleting a row in a **two-dimensional database array**. The drawbacks are that the stretching and shrinking can only take place on the first dimension of an array (although this is usually what is wanted anyway) and that the program must be compiled in order to work. The adjusted lengths of rubber arrays can be determined using the DIMN function.

Arrays are all very well, but they can waste space and they all their elements must be of the same data type. Some alternatives worth considering are linked lists (with variable length entries, of course), queues and stacks. These can be created out of a single very long string, or from reserved memory areas, or directly accessed from data files, and they can be managed using SuperBasic procedures and functions. If you decide to implement data structures in reserved memory, remember to make use of Turbo Toolkit's memory handling functions such as SEARCH_MEMORY, MOVE_MEMORY, PEEK$ and POKE$. The Turbo Tookit package also contains worked examples of running virtual arrays from datafiles. If your Sinclair QL World library goes back that far, check out the long series of *Super Basic* articles on the subject of data structures published in the Autumn of 1988.

The **memory space** occupied by an array becomes important when it is large enough to take up a significant proportion of the available memory and when the array belongs to a compiled program whose dataspace needs to be calculated. Each floating point value in a numeric array occupies six bytes. Each value in an integer array will need two bytes. Each string in a string array will occupy its true length, rounded up to the next even number, plus two more bytes. Remember that all arrays have zero elements, so an array declared as NUM(5, 9) has 60 elements in it, not 45.

The **Turbo compiler** treats all string variables as one-dimensioned string arrays with a default length of 100 characters. Even if you only plan to hold only a single character in a variable, Turbo will rope off 100 valuable bytes of memory for the task. Conversely, should you plan to extend a string variable to include more than 100 bytes, Turbo will truncate the string to the 100 character maximum, thus losing data and potentially causing problems with slicing beyond the 100 character point. Programmers therefore need to get into the habit of explicitly declaring string variables as if they were single-dimensioned arrays.

BASIC

Basic - the **Beginners' All-purpose Symbolic Instruction Code** - began life as an interpreted language when all about it existed more demanding, compiled languages. It was recognised that it was too difficult to ask beginners to write complete chunks of program code and wait ages for it to be compiled, only to discover that there was a fundamental error in the first line. My very first program was written in **Fortran** and it was supposed to calculate fuel consumption figures from distances travelled and fuel used. Even though it was only about twenty lines long, the program took hours to create on punched cards. These were sent by courier to the University computer centre and two days later the output returned: error in line 3. The next revision produced fifty pages of output instead of one because a loop had no way of terminating. In all, it took over a week to debug a very simple routine.

A Valuable Language

The Basic language performed a very valuable service by allowing programmers to run programs without compiling them first and thus immediately finding out whether they worked or not. It was powerful enough to explore data structures, but never intended for "industrial strength" programming. In general, its error handling was appalling in that it preferred to stop processing with a vague error message rather than handle the crisis without crashing, it was slow in execution and it lacked the more advanced program structures and data types available in other languages. These weaknesses were acceptable when the role of Basic was to train programmers to use other, more capable, languages. Over the years, though, the weaknesses of Basic have been addressed and its strengths enhanced so that now it is possible to use it to write professional quality programs and compile the results into a fast-running and error-proofed application.

Apart from its unprepossessing name, Basic has gained a big hold on the programming community so that it stands among the most-used programming languages in the business computing world (the others being **Cobol**, mainly for mainframes, and **C**, mainly for Unix and MS-DOS). Basic's closest rival, also available for the QL, is C. C is a **compiled language** of enormous flexibility that spans the gap between high-level languages and assembler code. Its chief advantage over Basic is speed of execution, but when Basic is compiled with Turbo this distinction by and large disappears.

Compared with other languages, Basic is also very simple. It does not demand that programmers declare all the variables they plan to use. It does not have complicated data structures. It does not require an intimate knowledge of the workings of the central processing unit. As far as the demands made on the computer are concerned, Basic does not require large overheads of memory, nor does it need large storage areas for source, intermediary and object files.

Basic is an **interpreted language**, meaning that each program line is converted into a machine code subprogram that is passed to the central processor to be carried out. This is a slow process as one Basic keyword might equate to several machine code instructions and the English-like syntax of a Basic command needs to be converted into an entirely different syntax more suited to the logic of the CPU. A further disadvantage of interpreters is that they have no sense of the large structure of a program. If they meet a command inside a loop they painstakingly analyse it, convert it to machine code and execute it in isolation even though it is exactly the same command that they were handling a few microseconds ago. This is the computing equivalent of a goldfish excitedly exploring its small goldfish-bowl: every time it does a circuit it thinks "A plastic shipwreck - never seen one of those before".

Compilers get all of the interpreting into machine code over with in one operation, producing a new file of "object code" that is impossible for humans to make sense of but which the CPU understands perfectly. Compiling also gets around the problem of translating and re-translating identical commands inside loops: the process is optimised just once. Turbo does this for SuperBasic, giving programmers the very best of both worlds.

The essentials of Basic are its **reserved keywords**, its syntax, its program structures and its data types. Reserved words are those that have special meaning to the interpreter and thus cannot be used to represent anything else. The syntax of Basic centres on "expressions". One of the simplest expressions is "Hello" - a string expression. "Hello " and " World" is a slightly more complicated expression that includes an operator, in this case one that joins the two strings together. The other types of expression are numeric, such as 5 * 7, and logical, such as (Name$ = "Jones"). The latter expression returns a value of true or false, on the QL represented by one and zero respectively.

Reserved keywords fall into two main groups: procedures and functions. A **procedure** is a "doing" word, such as PRINT, DIM, GOTO and STOP. A **function** is often described as an expression factory: you pour in raw material expressions, known as arguments, and the function manufactures a new expression from them, called the return value. For instance, LEN("Hello") takes a string expression and returns a numeric expression, 5, representing the number of characters in the string. Function names follow the same rules as variables in that they return a string if they end in a dollar sign, otherwise they return a number.

Symbols play a large part in Basic syntax because they convey so much information in a single character. Some are **operators**, such as +, * and ^, that carry out a mathematical process. Others are **separators**, such as ;!, that help the interpreter to make sense of a command. Some operators are represented by words such as NOT, AND, OR and INSTR, and TO is often an acceptable substitute for the comma as a separator.

SuperBasic has two main data types, **numeric** and **string**, and two minor data types, "**identifiers**" and "**names**" that are hybrids between variables and constants. Numbers can be conjoined using a wide variety of operators, whereas there are far fewer available for strings. Strings are character sequences. Variable names can be used to represent numbers or strings, with string variables being distinguished by trailing dollar signs. Variables must have a value assigned to them before they can be referred to. In other words, a variable must appear in the left-hand-side of an assignment expression before it can appear on the right-hand-side. Names are used for devices and files and can be expressed with or without enclosing quotation marks. Identifiers control and identify REPeat loops.

A **command** is the equivalent of an English sentence and it can contain procedure keywords, functions, constant expressions, variables, operators and separators. It always begins with a reserved procedure keyword, except for assignment expressions where the LET keyword is optional. There are some commands that only have a keyword, such as STOP, whereas others have keywords followed by parameters, which can be expressions or functions. Parameters are separated by separators of which TO and the comma are the most frequently used.

A **program** is a numbered list of Basic commands. Basic is alone in numbering its statements, a habit that modern Basics have now dropped. SuperBasic is very close to not needing line numbers provided that commands like GOTOs and GOSUBs are not used, but the final, radical step was not taken. Mind you, SuperBasic was presented as being an evolution of the language, not a revolution. Visual Basic, SuperBasic's most modern sibling, is hardly recognisable as a Basic dialect.

Program structures allow certain commands to be repeated a number of times (**iteration**) or bypassed under certain circumstances (**conditional branching**) or jumped to regardless (**unconditional branching**). A Basic dialect will usually have more than one example of constructing each type of structure. All programming languages, no matter how complex their vocabulary and syntax, base their structures on iterations and branches. SuperBasic has advanced unconditional branching structures that allow programmers to invent their own procedures and functions, providing they can be defined using the keywords already present in the language. SuperBasic also eases some of the pain of typing by allowing structural keywords to be abbreviated: the lower case parts of keywords like DEFine PROCedure can be omitted. It is a shame, though, that simple words like LOCal have an abbreviation while long words like RANDOMISE do not.

A key feature about Basic, and all other computer languages, is that absolute precision is expected in the construction of commands. **Errors** are likely to be generated by mis-spellings, the unexpected appearance of spaces, the lack of an appropriate symbol or the placing of values in the wrong order. Sometimes the interpreter will signal that a mistake has been made and refuse to carry on. Sometimes the mistake will be syntactically correct but the result will not be what was expected. In an age when standards of English expression are so loose, it is interesting to see the popularity of programming languages that demand such extraordinary attention to detail.

THE CLOCK

The Sinclair QL's **internal clock** maintains reasonably accurate time while the computer is switched on, but loses all knowledge of time when the power is turned off. The clock works by tracking the alternations in the UK mains current and assuming that there are exactly 50 cycles per second. Rather than writing a boot sequence that requests the correct date and time every time the computer is turned on, a more accurate quartz-based, battery-backed clock can be purchased and installed to ensure that the time is kept even when mains power is removed. Miracle Gold Cards also include such a facility.

As far as the QL is concerned, prehistory ended with the last second of 1960. Its internal clock represents all times and dates as the number of seconds that have elapsed since the first second of 1961. The value for the current time is obtained by the command PRINT DATE. More usefully, the DATE$ function returns the date in year, month, date, hour, minute and second order, as in the following output obtained by typing PRINT DATE$ in the command window:

```
1994 JUN 14 09:50:00
```

It should have been possible to strip away bits of the date using string slicing, but this is only available on **Minerva**-equipped QLs. The recommended work-around is to assign date output to a string variable and then slice that.

The clock shows more than anything else Sinclair's target market of small businesses, where dates prior to 1961 have little relevance, split-second accuracy is not vital and the need to time routines to the nearest hundredth of a second is limited.

COERCION

Coercion was trumpeted as a key difference between SuperBasic and ordinary Basics, but its advantages have limited appeal and the overheads in terms of error-trapping and conversion coding seem to be wasted. By coercion, Sinclair means that a value of one type can be automatically converted to another type if the circumstances seem to warrant it. Numbers can become strings, integers can become floating point values and names can become strings, all without intervention from the programmer.

The direction in which coercion flows is important. Coercion works **from integer to floating point and from floating point to string**, but it does not necessarily work in the other direction. Floating point values are truncated to the nearest whole number when they are coerced into integers. It would have been nice for SuperBasic to treat strings that do not begin with numerals as if they are zero, but instead it produces an error message. This can be overcome by prefacing strings with a "0" whenever they are to be coerced into a number, such as PRINT 6 + ("0" & Text$).

Coercion is of most value when passing parameters to a function or procedure because one routine might then cope with strings or numeric variables equally well. In practice, this is useful only in trivial cases because different processing might be required for text and numbers, or different error-trapping might be needed. Personally, I would have much preferred to see coercion dropped in favour of a function that identifies whether a parameter is a numeric or string value, the end of the need to follow string variables and functions with a dollar sign, and the ability to assign numeric and string values to the same variable or array element when the need arises, just as the dBase language does.

Coercion has, however, forced the introduction of a genuinely useful new operator, the **ampersand**. The ampersand in SuperBasic is used to join together two strings, whereas it is more common in other dialects to use the + symbol. Through coercion, however, PRINT "2" + "5" results in a figure 7, not the number 25, so another operator needed to be found to ensure that the correct operation was carried out. PRINT "2" & "5" therefore returns the equally-correct "25".

The NEW USER GUIDE

Concepts Section

Section Thirty-Six

This month the Concepts Guide explores QDOS further.

DATA TYPES, IDENTIFIERS AND VARIABLES

SuperBasic data can be stored on the QL in one of a relatively limited number of ways, called **data types**. Most data types can be represented by **identifiers,** which are sequences of letters and numbers that follow strict rules. The most often used class of identifiers comprise **variable names**, but other classes control loops, identify user-defined routines and name devices.

All data can be referred to as an **absolute value** or a **variable value**, ie one represented by a variable name. The statement PRINT "Hello" includes an absolute value of "Hello" whereas PRINT text$ refers to a **variable string** that might represent a different set of characters each time it is executed.

Strings are sequences of zero, one or more characters (up to a theoretical maximum of 32,766). **Absolute strings** are enclosed using matching single or double quotation marks. If a string is bounded by double quotation marks it can include single quotes within it, such as "Don't touch Mike's computer". The reverse is also true, so that 'Mike said "Do not touch" ' is a valid string. A string both bounded by and including the same type of quotation marks will be rejected by the interpreter. Strings can comprise any selection of characters from the full Ascii character set, although the effect that some of the non-printing characters have on the screen might not be all that is desired. A special sort of string, called the **nul string**, has no characters in it at all.

Floating point numbers can comprise the digits 0 to 9, the decimal point and, optionally, a unary plus or minus. **Floating point numbers** have 8 significant digits and range from the infinitesimally small (10 raised to the power of -615) to the impressively large (10 raised to the power of 615), both positive and negative. Floating point values are stored internally in a six-byte sequence. Sadly, numerical output is limited to 5 significant figures before being automatically converted to scientific (**exponential**) notation, such as 6.2345E4. The relatively low point at which numbers are coerced into scientific notation is a major QL weakness, especially in financial applications.

Whole number values, or **integers**, can be represented in a different way to floating point values. Integers are limited to the range of -32,768 to +32767, indicating that they are stored in just two bytes. However, the QL does not make much use of the integer type. All constant numbers are represented internally in floating point format and all arithmetic is carried out using floating point formats. This means that instead of integers and integer arithmetic speeding up time-critical processes, as would be the case on all other computers, the QL is actually **slowed down** by them as it suffers from the overhead of having to translate all integer values into floating point values before it can do anything with them. This is not true of programs compiled with **Turbo**: these benefit a great deal from the judicious use of integers, particularly to control loops. The difference in speed is very marked, even for quite small numbers of iterations, and is well worth the clever trickery needed to make the SuperBasic interpreter think it sees floating point control variables while the Turbo compiler knows differently.

SuperBasic **names** are used to identify devices such as microdrive files, screen windows, serial ports and network addresses. Strings can be coerced into names and carry less of an overhead in the Qdos name table. COPY flp1_myfile TO scr_ and COPY "flp1_myfile" TO "scr_" are therefore equally acceptable, with the latter having the slight edge in conserving memory.

SuperBasic has no **logical data type** such as the TRUE and FALSE in BBC Basic. Instead, logical true and false are represented by non-zero and zero values respectively. Three of the most efficient ways to represent logical values are: to use **integer variables** holding 1 for true and 0 for false; to use a **character variable** holding "1" or "0" that can be coerced to a numeric value; or, most efficiently, to use a **single character** to represent up to eight logical states measured using bitwise operators. The memory overhead for these methods is, respectively, two bytes per logical value, one byte per logical value and, best, one byte for up to eight logical values.

Identifier names, including variable names, must begin with a letter and can then include further letters and digits and can also contain underscores. Names for devices begin with a prefix identifying the type of device, such as FLP for a floppy disk drive. Identifiers are not case sensitive, so THISVAR is logically identical to ThisVar and thisvar. The maximum length of a variable name is 255 characters, all of which are significant. Identifiers cannot be the same as **reserved keywords**, nor can they be the same as the capitalised parts of control keywords (eg def, define, proc and procedure are all illegal because they conflict with the keywords DEFine or PROCedure). This limitation applies to file names used without drive prefixes unless they are enclosed in quotation marks. Interestingly, file names can include all sorts of normally illegal characters, including spaces, provided they are declared in a string. Finally, for obvious reasons, variable names cannot begin with Qdos device identifiers, such as mdv1_.

SuperBasic, like most other Basic dialects, has **typed variables**. This means that the presence (or absence) of a suffix to the variable name determines the sort of information that it can represent. Variable names without a suffix represent floating point numbers. If suffixed by a percentage sign, ie wholenum%, they represent an integer value. Strings are represented by variable names that end in a dollar sign, such as Text$. Names are not represented directly by a variable type, but are easily coerced from strings.

Array names must follow these rules and are restricted to holding data types consistent with the array name suffix. Programmers looking for some of the flexibility of the C 'struct' multi-type data construct can develop two-dimensional string and numeric arrays that match row for row, so that CUSTOMER$(32) and BALANCE(32) refer to the same customer. Alternatively, programmers emulating **dBase multi-type fields** in a database application might experiment with a two-dimensional array of very long strings containing fixed-length fields within it. For instance, the first thirty characters might be the customer name, the next eight might be the current account balance, the next fourteen the telephone number and so on. Functions need to be written to convert the string representation of numbers into numeric format for calculations and to replace the rather tedious string slicing syntax with something a little more user-friendly.

Loop and branch identifiers have a superficial similarity to variable names, but there are some special rules to consider. While the control variable in a FOR...NEXT loop has a value - indeed, without it the construct would be worthless - the **control identifier** (note, not a variable) for a REPeat loop does not. The expression in an IF statement can be based on any data type as all expressions equate to true or false, or non-zero and zero. The control variable in a **SELect statement**, however, must be of the floating point type: another huge weakness for SuperBasic.

Digital Precision gets round these restrictions using the IMPLICIT family of keywords. Should you wish to test a string variable using a SELect structure, simply include a command such as IMPLICIT$ MyString prior to giving MyString a text value and using it in the SELect construct. Life is even easier for **Minerva** owners as Minerva SuperBasic accurately handles integers and strings in its SELect statements.

DIRECT COMMAND

For many QL owners, **direct commands** are as near to programming as they feel they need to get. In effect, a direct command is a one-line program that performs one simple duty, often something mundane like loading a program file, changing the screen resolution or listing the contents of a microdrive. Some computers have a command line language separate from the primary programming language (step forward, MS-DOS), but the QL avoids this unnecessary duplication by **mixing** programming commands and operating system commands into the one language, SuperBasic.

The distinction between a direct command and a program command is simple: **direct commands** have no line numbers preceding them, while **program commands** must begin with an integer line number in the range 1 to 32,766. Program commands are collected to be carried out in line number order (jumps, loops and branches permitting) while direct commands are carried out the instant the Return key is pressed.

Because SuperBasic can **concatenate** several statements on one logical line by separating them with colons, the capabilities of a direct command are actually quite extensive. Short form structures can allow loops and branches, for instance, as long as they are properly nested. The disadvantage of trying to enter lengthy multi-statement lines is that a typing error will force you to re-write the whole thing again. That is, unless you own Super Toolkit II and remember that the Alt-Enter combination retrieves the previous line.

Programmers quickly latched on to another way of concatenating direct commands. They simply opened a file and printed to it a series of commands, each on its own line. By loading the file as if it was a SuperBasic program, the command interpreter is forced to carry out each unnumbered command as if it had been typed directly at the keyboard. The main advantage is that no memory is occupied by a program as each line is discarded as soon as it has been executed. The technique is ideal for **boot files** as these rarely need complex multi-line structures and memory conservation is paramount.

By and large, **statements** are equally at home in programs as they are when used directly. Some commands are of fairly limited value in programs, such as RUN and EDIT and some statements are meaningless as direct commands, such as DEFine PROCedure.

ERROR HANDLING AND PROGRAM DEBUGGING

For most QL programmers, program errors are to be avoided at all costs because they inevitably bring programs to an unexpected halt with nothing more than a gnomic comment and a curse from the user. For errors caused by poor programming, there is only the programmer to blame, but users can accidentally crash otherwise perfectly-mannered applications by, for instance, ignoring a request to place a cartridge in a microdrive.

Other programmers have discovered the freedoms given by a properly implemented **WHEN ERRor** command (or its Turbo equivalent, the wittily titled WHEN_ERROR). Using this event-driven procedure definition, whenever an error occurs the SuperBasic interpreter scuttles across to some code that should be able to recover from the error without halting the program. With this facility, all fatal errors can fairly be blamed on the programmer alone.

Even the earliest QL versions had WHEN and ERRor as reserved keywords, and later variants actually included some code to support them, but the only reliable error trapping is provided by **Turbo Toolkit** (supported by whatever QL rom you care to mention), the **Minerva roms** and QLs with **Super Toolkit II** hanging off them.

Compared with some operating systems and applications that have hundreds of error codes, SuperBasic manages quite adequately with a miserly twenty-two.

Late models of the QL rom included new keywords to handle errors. These have been copied across to Minerva roms and their stability enhanced. ERLIN, for instance, is a function that returns the line number on which an error occurred. ERNUM will report an error code between -1 and -21 that identifies the type of error. ERR_NF is an example of the error-specific functions: it returns true if a device was "not found". The REPORT command translates error codes into the appropriate message. REPORT by itself gives the message appropriate to the last error. Alternatively, if followed by a valid error number it prints that message. REPORT #2, -13 causes "Xmit error" to be printed in the listing window. A full list of **error codes** has already been published in the alphabetical section of the New User Guide (under the keyword **ERLIN**). An informative Super Basic article on the subject of incorporating error-handling code into programs was published in QL World June 1990.

Error handling depends on the presence of a WHEN ERRor clause (or the equivalent WHEN_ERROR variant in Turbo Toolkit). Within this clause it is important that errors cannot possibly occur because recursive calls to the WHEN ERROR block are impossible to untangle and may crash the system. Particularly if the error was related to file access, it might be necessary to ask the users if they wish to try the process again or abandon it. The decision can then be passed back to the routine in which the error was noted and the program flow adjusted to suit.

The interpreter can be guided out of a WHEN ERRor block using the RETRY or CONTINUE commands. RETRY attempts to execute the command that caused the problem to occur, whereas CONTINUE restarts program execution at the line following the faulty one. Neither offers all the answers to trouble-free error-handling, so Turbo users have the luxury of setting RETRY_HERE statements in their code. When an error occurs in a Turbo program the program flow is directed to the WHEN_ERROR block. When a RETRY command is reached the next line to be executed is the one immediately after the most-recently encountered RETRY_HERE command. This arrangement will only trip programmers up if they do not update the RETRY_HERE pointer at appropriate places.

Both Minerva and Turbo include ways of helping programmers **debug code** so that run-time errors are reduced to the mainly file-related problems that programmers can do nothing about. Minerva includes the command TRON to set a **tracing facility** on. As a program is run, the line number and statement number of the command currently being executed is displayed on the screen. The facility is removed with the command TROFF so that programmers can isolate problematical areas of the application to trace through. The speed of execution is often so fast that the true cause of faults is passed before the program stops with an error message. In these circumstances the **Minerva** command **SSTEP** comes into its own. When issued, SSTEP forces the interpreter to wait after each command is executed until the user presses a key.

The **Turbo Toolkit** comes with example routines developed using Turbo SuperBasic extensions. Two of those are relevant to the problems of debugging code. HOW_COME lists all the procedure calls that brought the interpreter to a particular part of the program. This is invaluable for determining where variables were set or changed or under what conditions subroutines are reached. The Toolkit package also comes with an implementation of TRACE that prints the entire program line in the command window. The utility TURBO_TRACE performs a similar function for programs after they have been compiled.

Turbo also includes routines for listing the procedure and function definitions in a program and for "profiling" programs in a way that identifies where the speed-critical parts of the application lie.

EXPRESSIONS

Expressions are the life-blood of programs. They store, manipulate and transport information. At their simplest they reflect a single value, but at their most complex they combine several values and functions linked with operators or nested in parentheses. Larger expressions can always be broken down into constituent expressions.

Expressions take three basic forms. The simplest, **atomic**, expression is formed from a constant, a variable, an array element of a function. "Hello World", MyFileName, Player$(12) and Area(xlen, ylen) are all examples of atomic expressions. Note that expressions can all be typed as string, integer, numeric and name, in line with the rules for identifiers described earlier. Typing is determined by the value that an expression represents, not necessarily the same as the type of the constituents it might contain. LEN(Text$), for instance, returns the length of the string Text$: it is a numeric expression even though it includes a reference to a string.

Atomic expressions can be linked with **operators** to form the sort of expression that a mathematician would recognise, such as Xlen * Ylen + 6. Most operators link numeric atomic expressions, but a few link strings.

An expression might include a **unary operator**, one that is followed by an atomic expression but not preceded by one. The expression -7 is formed from a constant preceded by a unary minus, for instance. The three unary operators are **plus, minus and the Boolean operator NOT**. In expressions that lack a unary operator, plus is assumed. See the "Operators" topic for details of the 26 operators recognised by SuperBasic.

The third class of expressions are those enclosed in parentheses that return **a value of true or false** according to their contents. PRINT (X = 8) will cause a zero to appear if the variable X is not equal to 8, otherwise a 1 will be printed. Several such expressions can be tested to see if they are all true by multiplying them together, or tested to see if any one is true by adding them together.

Expressions can form the parameters of commands or the arguments of functions, or their result might be assigned to a variable and so appear on the right hand side of an equation. Because functions can have expressions as arguments and are themselves atomic expressions, **complex expressions** can be broken down recursively into ever-smaller expressions. The following example begins to show just how complicated an expression can get:

```
PRINT#3, Text$(4 TO 8) &
Forename$(Decode$(UserCode)) & FILL$("..", WinWi
dth(3) + 17)
```

The constituent expressions are:

```
Text$(4 TO 8)              (Despite its appearance, this is
an atomic expression)
Forename$(Decode$(UserCode))
          Decode$(UserCode))
```

```
                    UserCode
FILL$("..", WinWidth(3) + 17)
          ".."
          WinWidth(3) + 17
                    WinWidth(3)
                              3
                    17
```

Expressions are sometimes confused with **equations.** Equations always include a **comparator** (such as the equals sign) separating two expressions. In programming, the expression on the left of an equation is most often a single variable to which the result of the expression on the right of the equation is being assigned. However, equations in brackets and within IF statements are treated as logical expressions that return true or false. This means that:

```
IF Xval + 7 = 19          :is a valid clause
LET Xval + 7 = 19         :is not
```

FUNCTIONS

A function is often described as being a factory that accepts expressions as its raw materials and manufactures another expression as a finished product. The simplest of such functions is SQRT(x) that takes any number, represented by the variable x, and converts it into its square root.

Some functions do not always need an argument passed to them, such as EOF(). One function, PI(), always produces the same result and is thus more akin to a system constant. In SuperBasic it is possible to dispense with the **brackets** following a function with no parameters, but most programmers find code easier to read if the status of every function is identified by the presence of parentheses.

If a function needs one or more arguments, the brackets around the arguments are compulsory. This clearly distinguishes them from procedures, which do not have their parameters in brackets. As explained in the earlier section on expressions, any function argument can itself be a function. This concept is nesting, where functions contain functions that contain functions. The difficulty with **nested functions** is to have the right number of left and right parentheses and have them in the right places.

If a function needs several arguments, such as FILL$(string$, chars), the arguments are separated by commas. In a few special cases readability can be improved by replacing some or all of the commas with the keyword TO. The SuperBasic syntax checker usually treats TOs and commas identically in argument lists.

SuperBasic allows programmers to define their own functions. A function definition begins with a DEFine FuNction line that includes the name of the function and, in brackets, any formal parameters that may be necessary. The name of the function must **match the type** of the value to be returned from it, so that the name of a function returning a string must end with a dollar sign, and so on. **Formal parameters** are local variable names that represent the data values passed to the function. A key difference between formal parameters and normal variables is that they take their type from the value passed to them and do not need to follow the normal rules of appending a dollar sign or percentage sign to indicate a string or integer variable.

When a user-defined function is called, values called **actual parameters (or arguments)** are passed in strict sequence and with a one-to-one match to their equivalent formal parameters in the opening line of the definition. If the parameter is a single variable, any changes to the formal parameter are reflected in the actual parameter. Programmers describe this as passing variables **by reference**. If the actual parameter being passed is an expression then its value alone is passed to the function: any changes to the value of the formal parameter within the function definition have no effect on the outside world. This is called "**passing by value**". Arrays are always passed by reference.

To show the impact of these rules, let us assume that a string is passed to a user-defined function and that during the course of the definition it is necessary to remove all the spaces from the string. Once the function was finished, will the string have spaces in it or not? Well, with a call like X$ = Split$(String$, 5) any damage done to the variable String$ inside the function definition will persist. If the call had been X$ = Split$((String$) , 5) then anything could have happened to the formal parameter representing String$ inside the function and no change would be seen to String$ after the function was complete.

Turbo treats array and expression parameters in the same way as SuperBasic, but turns the default for simple variables on its head. Without special action from the programmer, variables in **Turbo-compiled code** are passed by value. To pass them by reference the DEFine FuNction line must be preceded immediately by a REFERENCE statement listing those parameters in the DEFine line that are to be treated as being passed by reference. This must always be done prior to passing an array because all arrays must be passed by reference. An array name in a REFERENCE statement must identify how many dimensions it has, for instance REFERENCE Player$(0,0,0). The number of elements is disregarded, so zeroes are as good as anything else.

At the outset of a **function definition** programmers can opt to declare one or more local variables for use within the definition. Even if these identifiers occur elsewhere , SuperBasic ignores any previous value associated with the variable. Once the definition is left, the previous value for such variable names is restored.

Function definitions can include the full range of SuperBasic statements and structures, although it is always considered bad practice to define a function or procedure within another user definition, simply because it makes the program code difficult to read. The aim of a function definition is to produce some value that can be returned to the expression from which it was called.

The last active line of a well-written function definition must always be an unconditional RETurn statement. In functions, RETurn always takes a parameter. This might be a variable (often called RESULT, for obvious reasons) or a more complex expression. A minimalist function would therefore be a DEFine statement followed by an unconditional RETurn statement, such as:

```
DEFine FuNction Increment (xval): RETurn xval + 1
```

It is possible to pepper a multi-line function definition with several conditional RETurn statements with IF or SELect structures determining which, if any, of the RETurns to activate.