SUPERCHARGE!

SUPERBASIC COMPILER

1985

SUPERCHARGE

Published by:
Freddy Vachha BSc
Digital Precision Ltd

CONTENTS

PLEASE READ THIS FIRST!

We understand that you've just bought a fascinating program and you're eager to try it out. You probably don't want to read 40,000 words of wisdom before you can use SUPERCHARGE.

If you read the short summary on this and the next three pages you will be able to get stuck in almost straight away, without false starts or an irritating delay. If you get stuck, come back to the manual and read Chapters 2 and 3 for a more leisurely introduction. SUPERCHARGE is a sophisticated and versatile product - you will find the rest of this manual very useful when you want to squeeze the best possible performance out of your QL.

(ALMOST) INSTANT *MACHINE CODE*

(1) Reset the QL with the SUPERCHARGE cartridge in Microdrive 1. The 'BOOT' program loads the BASIC extensions used by SUPERCHARGE. Leave the cartridge in the drive.

(2) LOAD a short BASIC program (a few hundred lines). The demonstration program we supply is called MDV1_DEMO_BAS.

(3) RUN the SuperBASIC to satisfy yourself that it works normally. Type CLEAR to release as much memory as possible.

(4) Start SUPERCHARGE with this command:

```
MERGE mdv1_supercharge
```

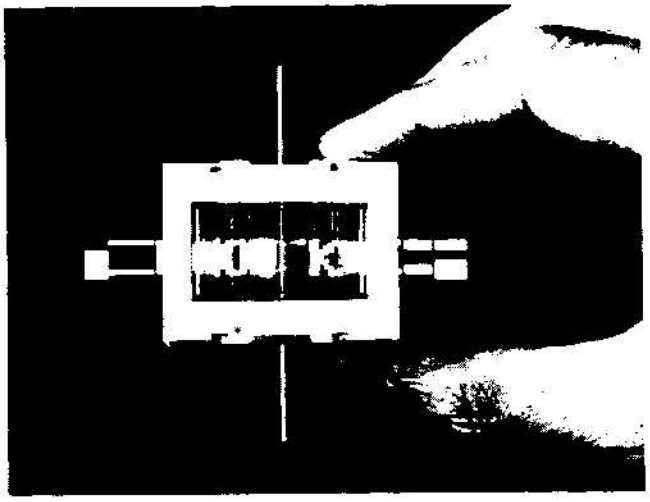
(If you see an 'out of memory' error when this command is entered, you should try a smaller program, or refer to the detailed advice in Chapter 3).

SUPERCHARGE loads and displays a pattern composed of three vertical lines and one horizontal. This pattern is used to align 'LENSLOK' - the black plastic device with a transparent lens at its centre, used to protect SUPERCHARGE.

Adjust the width of the pattern on the screen to match that of the lens-holder. Press the left-arrow key to make the outer lines move apart, and press the right-arrow key to move them closer together. Take the plastic LENSLOK device and hold it lengthways across the screen, without folding it. It should be about 100mm long. Adjust the outer lines so that they line up with the outer edges of the LENSLOK. Tap the SPACE key when the lines are correctly positioned.

A pattern of oblong blocks now appears on the screen. Fold the lens holder into a 'U' shape and press its legs firmly against the display. The holder is marked with various helpful pieces of information, such as 'TOP', 'LEFT' and and 'THIS SIDE OUT' (i.e. towards the viewer).

Close one eye and align the centre line of the lens with the vertical line in the middle of the screen. Look through the transparent lens set within the holder. you should be able to see the letters 'OK', as shown in the picture below. Press the SPACE key when you can read the letters.



Another pattern of blocks then appears. When viewed through the lens, this pattern corresponds to two randomly-chosen characters. You have ten seconds in which to type the characters. SUPERCHARGE starts when you have identified both characters correctly. If you fail to type both characters within ten seconds another pair will appear. You are allowed three tries before it becomes necessary to re-load the program.

(5) If all is well SUPERCHARGE asks you for the name which you want it to give to the compiled program. Try another name if the compiler rejects your entry with an error report. Remember to include the device name (e.g. MDV1).

(6) You are asked whether or not you want a listing of the program as it is compiled. Type Y for Yes, or N for No.

(7) Say where you want to send the 'report' file produced by the compiler. This report contains any error messages or warnings which may be generated, plus the listing (if any). If you press ENTER, the report is displayed on the screen.

SUPERCHARGE then analyses your SuperBASIC program, checking for errors and translating it into intermediate code. Large coloured areas will appear on the screen as the compiler works. This does not indicate a fault - it just shows that SUPERCHARGE is making optimum use of the available memory.

After a short while the total number of errors is reported. If there were any errors the compiler will stop. The error messages are listed and explained in Chapter 3. If no errors were found, the code-generator is automatically loaded, to produce a task file with the name you supplied.

SUPERCHARGE clears the screen when it has finished.

RUNNING A COMPILED PROGRAM

When SUPERCHARGE has finished you are back where you were before you loaded it, except that a compiled task has been saved (if there were no errors). If you specified the name FLP1_GAME, for example, you can run the task by typing:

```
EXEC_W FLP1_GAME
```

The compiled program will load and run, using copies of the windows defined in SuperBASIC for channels 0 and 1.

You can run several programs at once if you use the EXEC command rather than EXEC_W. When a task is loaded with EXEC it runs independently of SuperBASIC - you can type other commands (perhaps including further EXEC commands) while it is executed.

ALLOCATING EXTRA MEMORY

When a compiled program is loaded, 2K of memory is normally reserved for the 'data' it will generate. This space is used to hold variable values, return-points, and channel details. If a compiled program stops with an 'out of memory' report you will need to increase the amount of data space allocated to the task. You can do this with a program called DATASPACE. Load it with this command:

```
EXEC_W MDV1_DATASPACE_TASK
```

Enter the name of the task you want to modify. The program size and data allocation will be shown. Type the new amount of data space, in kilobytes. Press ENTER on its own to stop the task. There are full instructions in Chapter 4.

SUPERCHARGE vs SUPERBASIC

This list is a very concise summary of the differences between interpreted and compiled SuperBASIC. You can find more detailed information, and examples, in Chapter 6. The main differences are that compiled programs are usually much faster, and run as independent tasks.

SUPERCHARGE is closely compatible with SuperBASIC, but there are some discrepancies, since compiled programs are necessarily executed quite differently from normal BASIC.

(1) Editing and debugging commands such as LIST, MERGE and CONTINUE are not supported by SUPERCHARGE, since there is no 'program text' once a program has been compiled.

(2) Floating-point values are computed and displayed to nine decimal places of accuracy; integer (whole number) arithmetic is performed extremely fast. SuperBASIC only displays seven places and handles integers very slowly.

(3) DIM statements should be used to declare strings of more than 256 characters. Array subscripts must be integers up to 32767. Only strings and string arrays may be sliced.

(4) SUPERCHARGE has to analyse ALL of the program before it can generate code, whereas the interpreter only analyses lines as they are executed. Consequently the compiler is a little more rigorous about the syntax of programs:

> (4a) You may not use the same name for more than one purpose in a compiled program (so arrays, procedures and functions may not have the same names). Dollar and per cent signs must be used to distinguish string and integer names, and you may use names which differ only in their last character, e.g. FRED% and FRED$.
>
> (4b) Loops, tests and definitions in compiled programs must have matching ENDs. Of course, you may omit these when you use 'short' (single-line) loops and tests, and loops may contain as many NEXTs and EXITs as you wish.

(5) Only functions return values. Global or less-local variables must be used to pass values out of procedures.

(6) Line numbers and DATA values must be fixed. Calculated values, such as GOTO A*20, are not allowed. Such code should be replaced with ON..GOTO, SELECT or assignments.

A LEISURELY INTRODUCTION

This user's manual describes the SUPERCHARGE compiler - a powerful software tool which 'translates' SuperBASIC programs into machine-code. The manual has been printed on single sheets of A4-sized paper and punched with holes, so that you can keep it in the binder supplied with your QL.

SUPERCHARGE is aimed at programmers with some experience of SuperBASIC. It gives them access to much of the speed and flexibility of machine code, while still allowing them to develop and test their programs in SuperBASIC. Compiled programs may be saved and run without the compiler loaded. You may even sell programs compiled by SUPERCHARGE, so long as the publisher has a site licence and you do not copy any part of the compiler or its manual.

WHAT IS A COMPILER?

A compiler converts programs written in a language designed for humans into machine language. SUPERCHARGE converts SuperBASIC into machine code. The machine code performs in almost exactly the same way as the SuperBASIC - but it is much faster. If you've had a QL for long you must have noticed that all of the fastest, flashiest programs are written in 'machine code' rather than SuperBASIC.

The advantages of a compiler become clear once you understand the difference between BASIC and machine code. That difference stems from the way a home computer works.

At the heart of any micro is the processor. In essence this can only do three things - it can move small values in memory, do simple arithmetic upon them, and select subsequent instructions depending upon the results of the arithmetic. The processor's saving grace is that it works very fast - at a rate approaching a million steps a second. The processor can't directly read files, make sounds or print messages (plus hundreds of other things), but it can perform those operations by combining the simple steps which it CAN handle. For instance, sounds can be generated by sending a message to the second processor, printing can be done by moving patterns to the display memory and so on.

When you turn on your QL, the thousands of instructions in the 48K ROM read commands from the key switches and perform appropriate actions - step by tiny step. The ROM contains a machine code program called the SuperBASIC 'interpreter'.

Just like a human translator, the interpreter converts words in one language - SuperBASIC - into another language - machine code. This is a two-step process. First the instruction must be recognised, then it must be acted upon.

Interpreters are slow because they perform the first task over and over again. The interpreter spends more time trying to recognise SuperBASIC commands than it does performing the corresponding action. Interpreters do not 'learn by their mistakes', so the same delays crop up over and over again.

We can get a feel for the way an interpreter works by examining the way SuperBASIC handles a small program.

```
10 FOR I=1 TO 1000
20 LET X=2+2
30 NEXT I
```

If you run the above program, the value of X will be worked out as slowly the thousandth time as it was the first. Each time, SuperBASIC looks through line 20 to make sure it isn't anything nonsensical (like LET 7="SAM"). Then it finds out where it keeps the value of 'X', and locates the binary form of the number '2'. Computers use binary arithmetic, unlike the 'decimal' which caught on among humans a couple of millennia ago. Values have to be converted before they can be accepted or displayed.

Next, SuperBASIC makes a note that it will need to do some addition once it has two numbers to play with. It finds another '2' and adds the two values using complicated instructions which are designed to handle all cases; the same code would perform 0.0007 + -99999. Finally it puts the result away under the name X, and looks for the next line.

The QL has used hundreds of simple operations, where four (FETCH, FETCH, ADD and STORE) would have sufficed.

The SuperBASIC compiler looks at the listing of a program - in the verbose form which humans and other QL owners can understand - and converts it into simple steps in the order favoured by the computer. The juggling about, testing and searching are almost eliminated. You end up with a machine code program which works just like SuperBASIC, but faster.

Overleaf you will find a list of the main advantages of SUPERCHARGE over interpreted BASIC and compilers for other languages. The points are discussed in detail later.

TEN REASONS TO BE CHEERFUL

(1) SUPERCHARGED programs run typically five to twenty times faster than interpreted BASIC; small changes to a program may give speed-up factors of one hundred times, or even more. Furthermore the speed-up factor increases with the size of the program to be compiled, as the interpreter wastes more and more time searching for program lines.

A compiler obviously cannot speed up external devices (such as disks or microdrives) but it can reduce the time needed to process data so that devices are used more efficiently.

(2) Compiled BASIC programs may multi-task, whereas the interpreter only allows one BASIC program to run at a time.

(3) Compiled BASIC programs may be interactively tested 'in slow motion' using the BASIC interpreter. Variables may be printed and altered, and the program may be changed at a moment's notice. This is not possible when testing programs in other compiled languages such as C, Fortran or Pascal.

(4) Add-on commands and functions, such as those supplied with disk systems or the Sinclair QL Toolkit, may be used in compiled programs. You can use most 'add-on' commands in compiled programs, so long as they were written using the standard format for user-defined procedures and functions.

(5) Compiled BASIC programs offer fast floating-point mathematics, displayed to nine decimal places of accuracy.

(6) The SuperBASIC compiler implements true integer (whole-number) arithmetic. This is typically performed over thirty times faster than under the interpreter.

(7) Compiled BASIC programs load much more quickly than their interpreted counterparts, since they do not need to be 'tokenised' as they are read.

(8) Compiled programs are protected against unauthorised modification, as they cannot be LISTed.

(9) A number of bugs and restrictions imposed by the BASIC interpreter are corrected or lifted by the compiler.

(10) SUPERCHARGE issues clear, plain-English reports, showing the exact position of errors in programs which are being compiled, rather than the vague messages issued by the interpreter and some other compilers.

THE COMPILATION PROCESS

SUPERCHARGE works in two steps, so as to leave the maximum amount of memory free for your program. The steps are linked automatically, so that one command can be used to invoke both steps. Nevertheless it is important to realise that a two-step approach is used.

In the first step, a program called the 'parser' converts your BASIC program into an 'intermediate code' which is stored in memory (if space permits) or on microdrive. The parser also checks your program to make sure that it does not contain errors, issuing appropriate reports if need be.

In the second step, the intermediate code is translated into machine code for the 68008 microprocessor. This work is done by the code-generator, which loads 'on top of' the parser. If all goes well, the code-generator produces a task which can be loaded and run with the EXEC commands.

Getting underway

Before you can compile a SuperBASIC program you must load it, in the usual way. Run the program if you wish to ensure that it works properly. Start SUPERCHARGE with the command:

```
MERGE mdv1_supercharge
```

(In all of these examples we will assume that the SUPERCHARGE cartridge is kept in microdrive 1).

This loads a file containing a sequence of commands which cause your program to be compiled.

In the unlikely event that you see an 'out of memory' error as soon as this command is entered, there is not enough memory available for the parser and your BASIC. You will have to reduce the size of your program or free more memory. If this cannot be done, consider breaking your program into two or more independent parts, and compiling the parts separately. Some memory is used by add-ons such as disk systems and utility software. You may be able to free memory by using the QL without the add-ons.

You must have the SUPERCHARGE extensions (extensions code) loaded when you use the compiler, or this message appears:

```
DEVICE_STATUS is not loaded.
```

You can correct the error by loading the extensions, which occupy only 640 bytes of memory. Type:

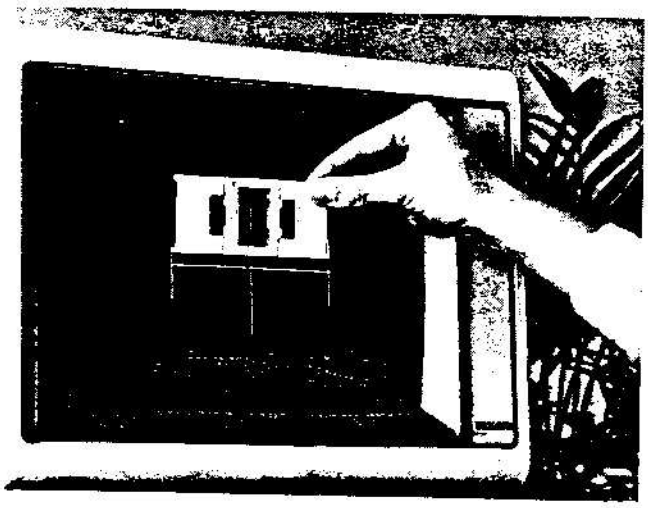
MERGE mdv1_extensions_bas

If all is well SUPERCHARGE loads and displays a pattern composed of three vertical lines and one horizontal. This pattern is used to align 'LENSLOK' - the black plastic device with a transparent lens at its centre, supplied with every copy of SUPERCHARGE.

LENSLOK is used to ensure that your copy of SUPERCHARGE is a legitimate one. It has the advantage over a cartridge or 'dongle' that it does not restrict the use of any of the interfaces on your QL. Its advantage over 'copy protection' is that it does not restrict your ability to make back-up copies of SUPERCHARGE for your own use.

The files on the SUPERCHARGE cartridge or disk can be copied in the usual way. You are strongly advised to make copies at once, just in case you accidentally drill a hole in the 'master' disk, tie a knot in the microdrive cartridge, or do something similarly silly. Don't say it won't happen to you. It will.

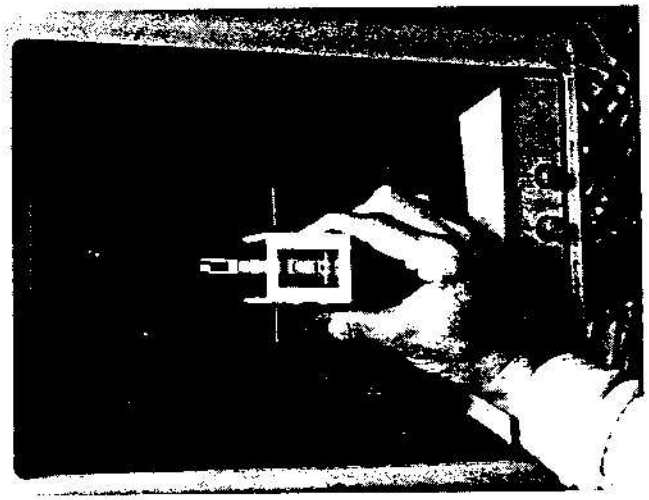
Meanwhile, back at the keyboard, you should adjust the width of the pattern on the screen to match that of the lens. This step is necessary because SUPERCHARGE is designed to work with any size of display (Sinclair pocket animated postage stamps perhaps excepted). Press the LEFT arrow key to make the outer lines move apart, and press the RIGHT arrow key to move them closer together.



Take the plastic LENSLOK device and press it lengthways across the screen, without folding it. It should be about 100mm long (four inches, for the un-metricated). The picture on the previous page shows the display and the way that you should hold the lens-holder.

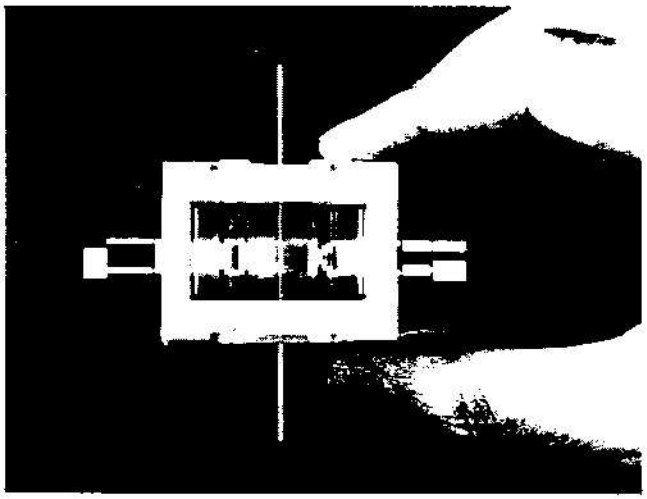
Use the left and right arrow keys (or a joystick or mouse which uses the CTRL1 protocol) to adjust the outer lines so that they line up with the outer edges of the lens holder. The keys auto-repeat if you hold them down for more than a fraction of a second. Press the SPACE key when the lines are correctly positioned. SUPERCHARGE now knows the size of your display.

A pattern of oblong blocks now appears on the screen. Fold the lens holder into a 'U' shape as shown in the photograph below, and press its feet firmly against the display. The holder is marked with various useful pieces of information, such as 'TOP', 'LEFT' and and 'THIS SIDE OUT' (i.e. towards the viewer).



Close one eye, (or several eyes if you are an octopus) and align the centre line of the lens with the vertical line in the middle of the screen. When you look through the transparent lens in the middle of the holder you should be able to see the letters 'OK', as shown in the picture on the next page. It may help if you adjust your viewpoint or the position of the lens slightly. When you can read the letters press the SPACE key. After about ten seconds SUPERCHARGE will assume that you have positioned the lens correctly, in any case.

Another pattern of blocks then appears. When viewed through the lens, this pattern corresponds to two randomly-chosen characters. You have ten seconds in which to type both the characters. At this point (assuming you are not an octopus) you only have one hand free for typing; the program is arranged in such a way that you do not need to press SHIFT.



As soon as you have identified the characters correctly the compiler will start. If you fail to identify the characters and type them both within ten seconds, another pair will appear. You are allowed three attempts before it becomes necessary to re-load the program.

At first this procedure seems long-winded, but it soon becomes second-nature, even if you are not an octopus. The routine has been carefully set up so that you can 'hurry it along' by pressing SPACE. It only takes a few seconds to 'unlock' SUPERCHARGE, once you have had a little practice.

The LENSLOK device is the 'key' to SUPERCHARGE, just as the 'master' cartridge is the 'key' to many QL programs which are protected against copying. The lens should be kept in a safe place. You should avoid dropping it down drains or feeding it to dogs or small children. They might choke - worse still, they might swallow it!

LENSLOK may appear fragile, but it is certainly more robust than a microdrive cartridge. The 'living hinges' at either side of the lens can withstand being folded indefinitely without breaking, although it is possible to destroy them by cutting or deliberately tearing them.

If you do manage to lose or break LENSLOK, you can obtain a replacement from Digital Precision for a nominal fee. Send either the broken bits or this page from your SUPERCHARGE manual, together with £5, to Digital Precision, at the address given on page 100. Proof of purchase MUST be provided.

When you have 'unlocked' SUPERCHARGE the screen will be cleared and a heading like this will appear:

```
Digital Precision SUPERCHARGE
V 1.0  (C) 1985 Simon N Goodwin

Name for compiled program ?
```

You should type the name which you want SUPERCHARGE to use for the compiled program. When you press ENTER the compiler checks that the name is sensible. It also checks that it can use the name you specify, with 'temp' at the end - a file with that name may be used as a temporary store for intermediate code, if your program is too large for all the intermediate code to fit into memory. If you typed the name 'FLP1_GAME', for example, SUPERCHARGE would use the file 'FLP1_GAME_temp' for intermediate code.

If either file cannot be opened, this message is printed:

```
Output file FLP1_GAME cannot be opened.
```

...and you are asked to specify a different name.

If either file already exists you will be warned and asked whether or not you wish to delete the file:

```
File FLP1_GAME_temp already exists.
Do you want to delete it <Y/N> ?
```

Type Y for Yes or N for No. If you type Y the existing file is deleted to make room for the new one. If you type N you are asked to specify an alternative name. WARNINGS: (1) If you just press ENTER, Yes is assumed. (2) If the file is 'in use' by another task the DELETE fails (sensibly enough) and SUPERCHARGE stops with an appropriate message.

Once you have specified an acceptable name for the Output file you are asked whether or not you want a listing of the program as it is compiled. A listing may be useful since it shows the location of any errors in your program clearly:

```
Compilation listing <Y/N> ?
```

Type Y if you want a listing and N if you do not. If you just press ENTER, Y is assumed.

Finally, you are asked where you want to send the 'report' produced by the compiler. This report contains any error messages or warnings which may be generated, together with the listing (if you asked for one) and the total number of errors found by the parser:

```
Report File <ENTER - SCR> ?
```

Type the name of the device or file to which you want the report sent. If you had a printer connected to serial port 1, and wanted a printed report, you might type 'ser'. To send the report to the microdrive file 'GAME_REP' you should enter 'MDV1_GAME_REP'. If you just press ENTER the report is sent to the QL's screen as the compilation progresses. You can pause the display by holding down the CTRL key and pressing F5. Any key will re-start the report.

SUPERCHARGE checks that you have specified a sensible destination for the report. The display devices (CON and SCR) are not considered sensible, as SUPERCHARGE uses part of the display to store information while compilation takes place. If the destination is not sensible this message appears:

```
Report file FLP1_GAME_REP cannot be opened.
```

You are asked to specify an alternative name.

If the file you specify already exists you are warned and asked whether or not you wish to delete the file:

```
File FLP1_GAME_REP already exists.
Do you want to delete it <Y/N> ?
```

Type Y for Yes or N for No. If you type Y the existing file is deleted to make room for the new one. If you type N you are asked to specify an alternative name. WARNING: as before, if you just press ENTER, Yes is assumed.

The programmer's analyst

The parser then analyses your SuperBASIC program, translating it into intermediate code. Unless it is a long program you will be able to watch the generation of intermediate code - it is stored in screen memory if possible, avoiding the need for slow microdrive access.

Pass the parser

The parser scans your program from beginning to end twice. Each scan is called a 'pass'. SUPERCHARGE displays the number of each program line as it is analysed.

During the first pass through your program SUPERCHARGE works out the way in which each identifier is used, and extracts DATA statements so that they are not muddled up amongst the other program lines. Errors are not detected by the first pass.

During the second pass your program is analysed in detail. If an error is found an appropriate message is produced, together with an the line number in which the error was encountered. Messages produced by SUPERCHARGE are prefixed by four asterisks, to distinguish them from program text.

If you asked for a listing, error messages are inserted just after the point at which each error is discovered. In the case of some errors, such as 'missing ENDs' or 'ambiguous declarations', the true cause of the the error may be at some point earlier in the program; the point at which the error was found is still a useful indication of the exact nature of the error. There is a full list of error messages and warnings later in this chapter.

Unlike the SuperBASIC interpreter, SUPERCHARGE examines every single statement in a program. Consequently it may find errors which are not detected when the interpreter is used.

The compiler skips over the remainder of that statement after an error has been found - perhaps ignoring further errors. Analysis continues from the start of the next statement.

Once an error has been found the compiler stops producing intermediate code. However, SUPERCHARGE continues to analyse the program until it reaches the end, so that any other errors can be detected.

At the end of the second pass SUPERCHARGE reports the total number of errors that it found. If there were any errors the compiler will stop, without trying to convert the incorrect code into an executable task.

Correct the errors by editing your program in the normal way, and re-start SUPERCHARGE from the start.

Code generation

If no errors have occurred when the end of the second pass is reached, the intermediate code must be a complete description of the original SuperBASIC. The code-generator is then loaded automatically. It reads the intermediate code twice: once while selecting 'building-blocks' and once while generating machine-code. The code generator creates the executable task. As previously, the 'line number' being analysed is shown as code is generated. The screen clears when the task is complete.

RUNNING A COMPILED PROGRAM

You can then RUN your original SuperBASIC in the usual way, or run the compiled program with the EXEC command. If you compiled your program into the file FLP1_GAME, this command would run it:

```
EXEC_W FLP1_GAME
```

The compiled program starts up with the same windows for channels 0 and 1 as were defined in SuperBASIC.

Errors in the compiler

SUPERCHARGE has been extensively and carefully tested before release, but it is a very intricate program and it is possible that obscure 'bugs' remain. Possible problems are discussed in this section of the manual.

The compiler may stop with an error message if your BASIC program is too large for analysis, or some circumstance (such as a power-supply fault) causes the compiler to fail.

SUPERCHARGE has to allocate extra memory whenever it 'recovers' after finding an error in the program being analysed. If a number of errors are found when you compile a large program, SUPERCHARGE may run out of memory.

The easiest way to get around this is to fix the errors which were found before the compiler ran out of space, and then run SUPERCHARGE again to find any remaining errors. This will rarely be a problem as the space allocated within the compiler is enough to accommodate quite large programs.

Note for advanced users only:

If your QL has extra memory you may increase the size of the data area assigned to the compiler, so that it can cope with more errors or larger programs. This can be done using the 'Dataspace' utility, as explained in Chapter 4 under the heading 'Tasks and Memory'.

The files which you will need to alter are named PARSER and CODEGEN. These correspond to the two stages of compilation. You should not alter the file CODEGEN unless SUPERCHARGE reports 'Out of Memory' errors during code generation - i.e. after the entire program has been listed in the report. You should only try to change COPIES of the files. DO NOT ALTER THE ORIGINAL FILES - you will need them if you make a mistake.

If you do increase the amount of space used by SUPERCHARGE you should bear in mind that the resultant task will probably be too large to run on a standard QL. You should not REDUCE the amount of space allocated to the parser or the code-generator, under any circumstances. We have given you the freedom to re-configure the compiler; this also means that you have the freedom to make mistakes. Be careful.

If any error other than 'out of memory' occurs, you should reset your computer and try again. If the fault persists, examine the last line which had its number displayed before the fault. Check that your program does not contravene any of the restrictions explained in Chapter 6 or elsewhere in this manual.

Remember that SUPERCHARGE is designed to compile programs that have been tested using the SuperBASIC interpreter. If your program won't work in 'normal' SuperBASIC you should not expect SUPERCHARGE to compile it.

The "MG" ROM

Some problems crop up when graphics commands are used in SuperBASIC programs run on the relatively-rare "MG" version of the QL. These problems stem from a mistake in that version of the QL's ROM, so they also affect programs compiled with SUPERCHARGE. Tony Tebby's 'patch' to correct these problems is effective for both compiled and interpreted programs.

To obtain a free copy of the "MG" ROM correction, write to:

QSOFT, Post Box 56, DK 4000, Roskilde, Denmark.

Enclose a cartridge or disk on which the program can be recorded, and postage stamps to cover the cost of returning the medium. If you are writing from outside Denmark you should send an International Reply Coupon rather than postage stamps.

IMPORTANT

The Last Resort

SUPERCHARGE has been thoroughly tested by many full-time programmers before its launch. It is possible that obscure bugs remain, but it is much more likely that problems stem from incorrect SuperBASIC coding or an incomplete understanding of the contents of this manual.

If the cause of your problem is still unclear after you have checked carefully through your code and this manual, we will do our best to help you. Please send copies of your program before and after compilation, return postage and EXACT details of the fault, to the address on page 100.

We undertake to destroy any copies of your files which are in our possession once we have resolved your problem. Your cartridge will be returned, together with appropriate advice, as soon as the mistake has been diagnosed.

This is not a program advisory service. We can only help you with problems unique to SUPERCHARGE. Furthermore, we cannot help unless you can explain the precise circumstances in which the fault occurs. In particular you should specify the version of your QL and any hardware or software add-ons in use when the error occurred.

We cannot support add-on procedures or functions that turn out to be incompatible with SUPERCHARGE. We have ensured the maximum degree of compatibility concommitant with efficient program execution, but we obviously cannot implement special code for every add-on command that has been written or may be written in the future. If your problem concerns add-on commands, other than the extensions supplied with SUPERCHARGE, you should contact the supplier of the commands for advice.

SUMMARY OF REPORTS AND MESSAGES

Most of the messages produced by SUPERCHARGE are self-explanatory, but we explain them here nonetheless. This section of the manual lists all the reports which may be generated by the compiler; where necessary it explains their significance in more detail than is possible on a single line of the report produced by the compiler.

Some of these messages are only 'warnings'. They indicate that SUPERCHARGE has detected a trivial error, such as a missing END DEF or END FOR, and inserted appropriate code to correct the mistake. SUPERCHARGE can often correct minor errors without the need for human intervention. You should nonetheless check every line for which a 'warning' is given, to ensure that you understand the mistake and the corrective action which SUPERCHARGE has taken.

This is a complete and exhaustive list of compiler messages; some of these are most unlikely to appear in normal use, but we'd hate you to 'miss' a feature!

AMBIGUOUS NAME USED

The line shown contains a name which is of indeterminate type - perhaps it has been declared as an array and also as a function, for example. This message may be generated more than once - this repetition is deliberate, so that you can easily find all the places where an ambiguous name is used.

AMBIGUOUS DECLARATION OF NAME

An name declared on the line indicated has also been declared as an incompatible array, procedure or function. This declaration makes the type of the name unclear.

ARRAY NAME REQUIRED

The first parameter of DIM should be the name of an array used in your program. This message may appear if there is no corresponding DIM for the name specified, or two DIMs with a different number of dimensions have been found for that name. This rule is explained in more detail in Chapter 6, under the heading 'Array and String handling'.

ARRAY OPERATION NOT IMPLEMENTED

The compiler has detected an attempt to 'slice' a numeric array, or otherwise manipulate a number of array elements within a single statement. Array slicing and manipulations are only allowed to act upon the last dimension of strings.

ASSIGNMENT TO FUNCTION ATTEMPTED

A statement starting with a function-name has been found. This indicates one of two logical errors. Either you have tried to call a function without specifying a destination for the value returned, or you have tried to store a value in a variable with the same name as a function. 'Bad name' is the equivalent message produced by the interpreter.

CHANNEL SPECIFICATION NEEDED

You must specify a channel number as the first parameter of this command.

COMMAND MEANINGLESS IF COMPILED

The command indicated is incompatible with the form of a compiled program. When a compiled program is run the text of the program lines is not present. Thus commands which manipulate the listing are useless. Interactive debugging commands are similarly meaningless in a compiled program.

COMPILATION ABORTED

The circumstances described in the previous error report make it impossible for the compiler to continue scanning the program. Any subsequent errors are not detected.

END OF STATEMENT EXPECTED

The compiler reached what it thought should be the end of a statement, and then found extra characters there. This message can appear if you try to pass the wrong number of parameters to a procedure, or close more parentheses than you've opened, in an expression.

END IF EXPECTED

The compiler has found another kind of END when it expected an END IF. This message can also crop up when more than one ELSE statement is associated with a single IF structure. You are allowed to nest IF statements as deeply as you wish, so long as you comply with the simple rule explained in Chapter 6.

END REPEAT EXPECTED

The nesting rules for a REPEAT loop have been broken, so that SUPERCHARGE has found another kind of END when it was expecting an END REPEAT. See Chapter 6 for more advice.

END SELECT EXPECTED

The nesting rules for a SELECT structure have been broken, so that the compiler has found another kind of END when it expected an END SELECT. The rules about nesting are explained in Chapter 6.

ERROR DIAGNOSIS FAILED

An internal problem has occured, causing the compiler to stop the compilation. Examine the listing at the point where the message was generated, to see if the cause is obvious. Otherwise there is a major fault in your computer or your copy of the compiler. Reset the QL and try again, in case the problem was caused by electrical interference.

EXPRESSION NOT ALLOWED IN DATA

Compiled data statements may not contain expressions. They may contain strings (enclosed in single or double quotation marks) and numeric constants, optionally preceded by an unary operator: NOT, "+", "-" or bitwise negate (2 tildes).

EXPRESSION SYNTAX INCORRECT

This message indicates that the compiler did not end up with a single value after evaluating an expression. Likely causes are unmatched or 'extra' brackets, or illegal characters in the expression.

EXPRESSION TOO COMPLEX

This message is extremely unlikely to appear unless you use a phenomenally complex expression - twenty brackets of the same type, a very long sequence of unary operators, or a lot of function-calls one within another. You are too clever for your own good. The solution is to simplify the expression or split it into several stages.

FAULTY LINE NUMBER

The line numbers specified in RESTORE, GO TO, GO SUB and ON..GO statements must be fixed values - not the results of a calculation.

FUNCTIONS MUST RETURN A VALUE

A RETURN statement has been encountered in a function definition, without an associated value to be returned by the function. If the value returned is immaterial in a particular instance, use RETURN 0 to prevent this error.

INCORRECT NUMBER OF PARAMETERS

A call to a SuperBASIC or machine-code procedure has either too few or too many parameters.

INCORRECT SUPERBASIC SYNTAX

A syntax error has been found. Normally these are detected by the SuperBASIC editor, which marks such lines with the keyword MISTAKE.

This message may also appear if the program being compiled is corrupt, or you have used the interpreter while SUPERCHARGE is running. The SuperBASIC data area should not be modified while SUPERCHARGE runs, or the compiler may 'lose its place' and issue this message. The problem cannot occur if you invoke SUPERCHARGE with the MERGE command.

LOCALS MUST FOLLOW DEFINITIONS

Local variables must be declared at the very beginning of a procedure or function - before other statements (apart from comments). This is a rule imposed by SuperBASIC.

LOOP DOES NOT EXIST HERE

An EXIT or NEXT statement has been found after the END of the associated loop, or before the start of the loop.

MISSING ARRAY SUBSCRIPT

The program contains a reference to an array where the required element has not been completely specified.

NAME NOT FOUND

An internal error has occured in the compiler. Check that other jobs are not interfering with memory used by the SuperBASIC interpreter. The SuperBASIC data area should not be modified while SUPERCHARGE runs, or the compiler may 'lose its place' and issue this message. The message may also appear if the program being compiled is corrupt.

NON-EXISTENT LOOP OR SELECTION

A structure has more than one END; a duplicate END IF, END FOR, END SELECT or END REPEAT has been found.

ONLY FUNCTIONS MAY RETURN VALUES

You should not use the RETURN command with a parameter (e.g. RETURN var) inside a procedure definition. Procedures are commands, unlike functions, and commands do not return values.

PREVIOUS DEFINITION INCOMPLETE

The start of the definition of a procedure or function has been found before the END DEFINE of the previous definition, or an 'extra' END DEFINE has been found.

PROCEDURES DO NOT HAVE VALUES

A procedure name (which has no associated value) has been encountered in an arithmetic expression. This generally means the name has been mis-typed, or a procedure has been accidentally defined instead of a function.

STATEMENT IS NOT YET SUPPORTED

The statement is in the QL ROM but has not yet been documented as a formal part of SuperBASIC. This message may appear if INPUT or EOF are used in an invalid context.

TOO MANY STRUCTURES

SUPERCHARGE has not got enough space to keep details of all the line-references, declarations and nested control constructs in your program. The simplest corrective action possible is to reduce the size of the program. This error is not likely to crop up on a 128K QL. The procedure to expand the area used to store this information is explained earlier in this chapter.

UNEXPECTED SYMBOL IN SUPERBASIC

The SuperBASIC program in memory has become corrupt, or there is a fault in your QL or in your copy of the compiler.

This message may also appear if you have used the interpreter while SUPERCHARGE is running. The SuperBASIC data area should not be modified while SUPERCHARGE runs, or the compiler may 'lose its place'. This never happens if you invoke SUPERCHARGE with the standard MERGE command.

VARIABLE ASSIGNMENT EXPECTED

A reference to a variable was found at the start of statement, yet it was not followed by an equals sign. This suggests that you may have confused a procedure name with a variable name.

WARNING MESSAGES

The next **6** messages are not error reports and do not prevent the generation of an executable task. The messages indicate that a possible error has been diagnosed at the position shown. SUPERCHARGE is an 'intelligent' compiler capable of taking appropriate action to allow the compilation to continue; you should check the program to make sure that you understand what SUPERCHARGE has done and why it has done it.

```
WARNING: DIM STRINGNAME$(256) ASSUMED
WARNING: LOCAL STRINGNAME$(256) ASSUMED
The string variable or local string variable shown is not dimensioned
in your program - a default maximum length of 256 characters is assumed.
If you need more than this, or to save memory, position an explicit DIM
where it will, at run-time, be encountered before the current location.
```

WARNING: END DEFINE sub ASSUMED

SUPERCHARGE has encountered a DEFINE statement while it was still reading the definition of another procedure or function. The name of the definition being read appears in the message ('sub' in the above example). This warning means that an END DEFINE is missing or in the wrong place.

You may 'nest' procedure and function-calls as much as you wish when a program runs, but it is not correct to nest definitions. Definitions should appear one after another, but not within one another. As Jan Jones, the designer of SuperBASIC, points out, "there are absolutely no advantages in nesting procedure definitions".

You may have missed out the END DEFINE because your routine always ends with a RETURN. SUPERCHARGE inserts an END DEFINE before the next definition - this action corrects the error, so long as there was not meant to be any code between the two definitions. If you try to nest one definition within another SUPERCHARGE issues an error message when the end of the 'outer' definition is found. The warning message shows where the END DEFINE should have appeared, so it is easy to correct the error.

WARNING: END FOR var ASSUMED

SUPERCHARGE has encountered the END of a structure enclosing a FOR loop without finding the END FOR for that loop. In the above example, 'var' corresponds to the name of the variable controlling the loop.

In a correct SuperBASIC program every FOR loop should end with an END FOR, just as every IF ends with an END IF and every SELECT with an END SELECT. Short forms have an implied END at the end of the line concerned. 'Traditional' BASIC, however, requires that a FOR loop is terminated by a NEXT statement, rather than an END FOR. The END FOR is often missed out by lazy programmers and those accustomed to different implementations of BASIC. One of the clever but dangerous features of SuperBASIC is the way that you can get away with NEXT instead of END FOR... in most cases!

SUPERCHARGE inserts END FOR statements in a sensible, documented place if they are missing. This warning does not indicate an incompatibility between SUPERCHARGE and interpreted SuperBASIC; it does indicate a potential source of error in the original SuperBASIC, and the exact way in which SUPERCHARGE will cope with the error should it occur.

WARNING: PARAMETERS ARE NOT RETURNED

This indicates that a parameter has appeared in a context where SuperBASIC and SUPERCHARGE might pass it in different ways. SUPERCHARGE always passes parameters in a fixed way - from the caller to the called code - whereas changes to parameter values within interpreted procedures and functions can lead to changes outside the subroutine, depending upon the exact syntax of the calling statement. If this message appears you should check that your program does not expect the value of the parameter indicated in the report to change as a result of the procedure or function call.

The code used to check for this case is rather simple and works on a 'better safe than sorry' basis, so the warning can also crop up when a parameter expression starts with a function-call or a reference to an array element. This doesn't disturb the compilation, since this message is only a warning. You can make things clear to SUPERCHARGE, and to the interpreter, by enclosing each parameter in brackets; thus SETPOS X,Y would become SETPOS (X),(Y).

WARNING: VARIABLE NAME ASSUMED

SUPERCHARGE cannot tell whether the name indicated in the report is that of a variable or a file or device. It has assumed that the name is that of a floating-point variable.

The format of floating-point variable names and file names is identical in SuperBASIC, unless the name is put in inverted commas. If this message appears next to a file name you should put the name in inverted commas to avoid ambiguity.

This message only applies to 'add-on' commands which are not standard to the QL. SUPERCHARGE is clever enough to recognise all the standard commands that expect names (rather than floating-point values) as specific parameters, so you use these commands without being warned:

```
COPY_N flp1_dump TO ser
SEXEC hdk1_temp,X,Y,Z%
```

In the second example SUPERCHARGE treats the file name as a string (rather than as a variable) - yet the other three parameters are treated, correctly, as numeric variables.

RUN-TIME ERRORS

Errors which occur while a compiled program is running are reported using the standard QDOS messages. These are documented in the 'Concepts' section of the QL User Guide.

All errors cause the task to stop; any channels it is using are closed and the memory in which the task was running is released for use by other programs.

It is not possible for the user to 'trap' all errors which may occur within a compiled program, but most errors can be detected or avoided by appropriate programming. The trapping of common errors is discussed in Chapter 5, which introduces a new SuperBASIC function for this purpose.

The SUPERCHARGE error message will give you an indication of the line which was being processed when the error occured. For example, you might see:

**** Supercharged BASIC halted after line 200 Overflow

If the message suggests that the program stopped at line '0' this shows that it stopped while it was setting itself up - before any program lines were executed. This usually means that memory is short - either the QL does not have enough space to run the task, or the task's data area is too small. The procedure to change the amount of data space used by a compiled task is discussed in the next chapter.

Under rare circumstances the line number shown may not be the exact location of the error; this is because SUPER-CHARGE sometimes merges or interleaves lines in order to obtain extra efficiency. In any case the error will be very close (in terms of program flow) to the line shown.

The final part of the report is a standard QDOS error message. In general this corresponds exactly to the message which SuperBASIC would give in similar circumstances.

'Channel not open' will appear if you try to use a channel number greater than 15 (see Chapter 6). 'Bad parameter' has its usual meaning - SUPERCHARGE performs simple parameter checking when a program is compiled, but some mistakes can only be detected when the program is executed.

The 'Overflow' message is generated if values go beyond the accepted range. This is especially relevant to string and integer handling - you will find more details in Chapter 6.

Invalid colours, array subscripts and so forth provoke the usual 'Out of range' message. Remember that all strings must be DIMensioned after a compiled CLEAR.

Unlike some other compilers, SUPERCHARGE always checks parameter values and array subscripts as programs run, so that obscure circumstances do not produce mystifying effects. The compiler's checking mechanism is much more efficient than that of the SuperBASIC interpreter.

You should still test your programs thoroughly with the SuperBASIC interpreter before you compile them, since the interpreter is specifically designed to make interactive testing and debugging easy. Of course, this means that interpreted programs run slowly, but this is rarely a snag when you are testing new code.

You may use most SuperBASIC 'extensions' in compiled programs (see Chapter 5) but these must be loaded when you run the program as well as when you compile it. If any such procedures or functions are absent when a task is started SUPERCHARGE prints a message for each one, of the form:

```
DEVICE_STATUS is not loaded.
LIST_TASKS is not loaded.
SET_PRIORITY is not loaded.
```

The compiled program will not run until you load the procedures or functions concerned.

If you receive a 'Bad parameter' error as soon as a compiled program is loaded, it is likely that the file has somehow become corrupt. You can create a good copy by re-compiling your original SuperBASIC program.

SUPERCHARGE reports and messages are generally printed to window 0, the command window; this will be at the bottom of the screen unless you have re-positioned it. However, messages are re-routed to window 1 (the default window for PRINT and graphics commands) if window 0 is 'in use'. This is most commonly the case when the system is displaying a cursor in window 0 and waiting for you to type a command. Sometimes messages are split between the two windows - this is irritating but inevitable, on a multi-tasking system.

If you start a compiled program with the EXEC_W command, reports and messages appear in window 0. This is because EXEC_W does not allow you to type further commands until the task has finished. If you find the messages annoying you can conceal them by setting INK and PAPER to the same colour in the appropriate windows.

* The SUPERCHARGE cartridge contains a file called UPDATES_DOC. If you look at this file using QUILL, you will find listed all amendments to the user manual

SPACE, TIME AND MULTI-TASKING

A 'task' is any program loaded with the EXEC or EXEC_W commands. Such programs differ from SuperBASIC or machine-code loaded with the LBYTES command, in that more than one task may run at a time. This concurrent execution is called 'multi-tasking'.

All programs compiled by SUPERCHARGE are capable of multi-tasking - in other words, you can run lots of compiled programs 'at once'. Tasks run in rotation, with each one receiving a small 'slice' of the available processing time before another gets a turn; the effect is just as if the tasks were executed concurrently.

The main limitations on multitasking are those of the QL hardware, in particular the available memory.

Every compiled program requires at least 4 kilobytes of memory. This space contains about 2K of code used to set up the task, plus routines to handle and report errors. A further 2K of space for data (channel tables and system variables, etc) is always allocated. Later in this chapter we explain how you can alter the size of this 'data space'.

The total size of a compiled program can be anything from 4K upwards. In general, compiled code is more concise than the original BASIC program. The exact ratio of sizes will vary from one program to the next.

DEVICE-SHARING

When you run several programs concurrently, the QL has to share out other hardware facilities, called 'devices', as well as memory. Some devices, such as the display, can be used by several tasks at once. Others, such as the keyboard, may only be used by one task at a time.

Compiled programs may test any device before they try to use it. This means that they can detect the case when a device or file is already 'in use' and take alternative action, rather than just stop with an error message. A new SuperBASIC function called CHECK_STATUS is used to obtain this information. The function is documented in Chapter 5, under the heading 'Trapping Errors'.

SERIAL DEVICES. It would obviously cause problems if several programs tried to use a printer interface at the same time - you might end up with a printout containing a mixture of several reports. The QL gives the error message 'in use' if a task tries to use a serial device (such as the network or serial port) which is already busy.

FILE HANDLING. Several tasks can share a single disk or microdrive - the QL makes sure that the data for each task is kept separate. Only one task at a time may WRITE data to a specific file. The QL reports 'in use' if a task tries to write to a file which is already being written by another task.

DISPLAY HANDLING. More than one task may write to the display at any time, but in this case it is up to the programmer to make sure that collisions do not occur, by positioning 'windows' appropriately.

THE KEYBOARD. Only one task may read characters from the keyboard at a time - otherwise commands might be 'shared out' between tasks, making the system unusable. The task which is currently receiving key-presses signals this fact with a flashing cursor in one of its windows. Any characters you type appear in that window. If other tasks are also waiting for input from the keyboard, they signal this by displaying a cursor that DOES NOT flash.

You may switch from one window to another with a single key press. When you make the switch the cursor you had been using will stop flashing and one of the others will start. Whatever you type appears in the new window. You can switch between several windows: each cursor is enabled in turn.

When the QL is first turned on it treats the key-press 'Control C' as the signal to swap from one window to another. Type Control C by holding down the CTRL key while pressing and releasing the letter 'C' key. This has no effect unless there is more than one cursor on the screen.

You can change the character code used to switch between windows by POKEing a new code into address 163986. Type:

```
POKE_W 163986,9
```

to make the TABULATE key switch between windows, since 9 is the character code of TABULATE. You should not do this, for obvious reasons, if you are using software which already assigns some purpose to that key. A full list of character-codes appears in the Concepts section of the QL User Guide.

TASKS AND MEMORY

When a program is loaded into the QL's memory with the EXEC or EXEC_W commands, space is reserved for the 'data' which the task will generate as well as for the program code.

SUPERCHARGE uses this 'data space' to store information generated as the program runs - variable (and array) values, temporary results, subroutine linkages and so on. When the data space becomes full, compiled programs stop with the report 'out of memory'.

A fairly small amount of data space is allocated by the compiler when a task is created (so that tasks do not gobble up large tracts of memory unnecessarily), but it is easy to change the amount of data space which will be allocated to a specific task.

The program called DATASPACE, supplied as part of your compiler package, allows you to set the amount of data space associated with any task file. DATASPACE itself is a task, so you load it using the EXEC command:

```
EXEC MDV1_DATASPACE_TASK
```

Once the program has loaded, a window appears near the top of the screen. The prompt 'Task file name?' indicates that the program is waiting for you to specify the file which is to be modified.

Type Control C, or whatever key-press you have assigned to switch between windows, until the cursor in the new window begins to flash. Type the device and file name of the task which you wish to modify. If you are not sure of the name you can switch back to the command cursor and use the DIR command to find it out.

AT ANY TIME you can stop the DATASPACE program by typing a blank line (ENTER on its own). You are returned immediately to the command line. Type Control C (or equivalent) to turn the command cursor back on.

When you have entered a name, DATASPACE tries to find a task file with that name. At this point any of the normal QL error messages might appear, if the name was incorrect. Alternatively, if the file exists but is not a valid QL task, this message appears:

```
That is not a task file.
```

If any error occurs you are returned to the 'Task file name?' prompt so that you can try again. Press ENTER on its own to stop the program, as explained earlier.

Assuming that you typed a valid file name, and all went well, the computer prints out the size of that program: the number of bytes of code and the number of bytes of data space presently assigned.

```
3836 code bytes, 2048 data bytes.
New data size (in Kilobytes) ?
```

The QL asksyou to specify the new data size, in units of 1024 bytes (one kilobyte). Press ENTER on its own to stop the program at this point. Otherwise, type an appropriate figure. You may type a 'K' at the end of the figure if you wish - it is assumed in any case. Any fractional part is ignored.

If the characters which you type do not make up a valid number you are returned to the 'Task file name ?' prompt. You can take advantage of this and examine the sizes of several programs, without changing them, by typing gibberish (such as a question mark) when you are asked to specify the new data size for a file.

A standard, 128K QL has only about 85K free for all tasks. It is not safe to use absolutely all the memory, since this leaves no room for file-buffers.

If you make the data size of a task so large that the code and data cannot fit into memory, an 'out of memory' error will occur when you EXEC the task. You can use DATASPACE to reduce the data space requirement if need be.

DATASPACE does not allow you to specify less than 2K of data space for any task. Compiled programs require this minimum amount of space in which to set up essential tables. If you type a value less than 2K you are asked to try again, specifying a larger value:

```
Too small; the minimum is 2 Kilobytes.
Please try again:
```

You can set or examine any number of tasks once DATASPACE is loaded. After each change you are returned to the 'Task file name ?' prompt. Press ENTER on its own when you have finished.

EVEN MORE SUPERBASIC

Five add-on commands are provided within the SUPERCHARGE package. These give you control over tasks, and allow you to 'trap' errors which would otherwise cause a program to stop. The extra commands may, of course, be used either in BASIC or in compiled programs.

The new commands can be loaded by typing the following instruction whenever you have the compiler cartridge in microdrive 1:

```
MERGE mdv1_extensions_bas
```

This command causes the extensions to be installed into reserved memory from the file EXTENSIONS_CODE, and linked into the SuperBASIC system. You can then use the commands as if they were a normal part of the SuperBASIC language. They will remain available until you reset or turn off the computer. They use only 512 bytes of memory.

As with all other add-on procedures or functions, you should install the commands before you load any program which uses them. This rule applies whether the program using the commands is compiled or ordinary boring BASIC.

Three new commands are provided to give control over tasks. The commands work with any program that is loaded using the EXEC instruction - be it written in compiled BASIC, machine code (e.g. DATASPACE_TASK) or any other language that produces programs that are loaded with EXEC. The other commands, used to find the amount of free memory and to trap errors, are discussed later.

FINDING OUT WHICH TASKS ARE RUNNING

The LIST_TASKS command, as you might expect, produces a list of all the tasks currently running on the QL. The list consists of four columns, separated by commas:

```
Name, Number, Tag, Priority.
```

You can direct the list of tasks to any QL device by following the command with a channel number, just as with PRINT or DIR. The hash character is optional. Thus, to send the list to the command window (channel 0), you type:

```
LIST_TASKS #0
```

If you type LIST TASKS before any tasks have been explicitly loaded, you obtain this response:

```
BASIC, 0, 0, 32.
```

That line indicates that the only task running is the QL's BASIC language, which interprets SuperBASIC programs and allows you to type commands. If there were more tasks running there would be a line for each one. Tasks are listed in the order in which they were loaded.

The first piece of information is the name of the task - BASIC, in this case. Compiled BASIC tasks have the name 'Supercharge'. Other tasks have the name assigned by their programmer, or 'No name' if the programmer ticked the 'no publicity' box and left the code nameless.

After the name come two numbers which identify the task to the QL system. These are called the 'task number' and the 'task tag', or, together, the 'task identifier'. These numbers are needed when you use other task-control commands. It is unfortunate that two numbers are used, rather than one, but - like lots of other unfortunate things - this feature is 'designed into' the QL's operating system. The SuperBASIC interpreter is always task 0, 0.

The last number is the 'priority' of the task. When there is only one task running this figure is not important; otherwise, it determines the proportion of time which the QL spends executing a given task.

Priority numbers range from 0 to 127. If a task has a priority of 0 it never gets any time at all, rather like bug-fixes at Sinclair Research. If a task has any other priority, the proportion of the processing time it receives will depend upon the priority of other tasks.

Later in this section we explain how the priority of any task can be changed - but first we should explain exactly what we mean by 'priority'.

Priority treatment

If three tasks were running, all with a priority of 32 (the standard value given by the EXEC command), they would all receive roughly the same amount of attention and run at roughly the same speed. If the priority of one of the tasks was reduced to 1, that task would receive much less processing time than the others, and appear to run more slowly. In fact, it would be chosen for execution less frequently.

EXEC gives tasks an 'intermediate' priority of 32 by default, since this makes it easy to make tasks faster or slower than the norm. It is a good idea to avoid using high priorities except in rare circumstances, since it can be irritating to have to 'turn down' a number of tasks just to make one relatively faster.

The exact ratio of execution times depends upon what each task is doing. In general, high priority tasks receive the largest proportion of processing time, but this is not always the case. If two tasks are both waiting for information (from the keyboard or serial port, perhaps), the QL does not waste time on them - whatever their priority - until they have some data to process; in this case, a third task with a priority of 1 might receive most of the time, simply because it might be the only task which was immediately ready to run.

The QL does not 'forget' about tasks unless they have a priority of zero. Even if a task has a priority of 1 it is executed occasionally - but it may not run for long each time it is awakened, and such awakenings may be infrequent.

Sometimes you can see this process at work. It is common to set the priority of 'clock' or 'calendar' tasks, which display the current date, to a low value, so that they only use a small proportion of the QL's time. If you have such a program you may notice that it shows the exact time, accurate to the second, when the computer is idle, but while you type in commands, or list programs, the display may only be updated every few seconds.

The 'priority' of QL tasks is much like the 'priorities' which you might attach to tasks at home. Fixing the hole in the roof might be a high priority, hoovering the carpet a lower priority and experimenting with your QL the lowest priority of all, only to be done when other tasks are not pressing. In our case, we only attend to the roof when it looks as if the rain is going to get inside the machine!

CHANGING THE PRIORITY OF TASKS

You can change the priority of any task that is loaded. The command to do this is (wait for it!) SET_PRIORITY.

The QL needs two things in order to change the priority of a task - the task identifier (the number and the tag) and the new priority. Priority values may range from 0 to 127, as explained earlier in this section.

Use the LIST_TASKS command to find the names of tasks and the corresponding 'task identifier' numbers. You must use identifier numbers to specify a task, rather than names, since it is quite possible to run several tasks which have the same name.

The format of the SET_PRIORITY command is shown below:

```
SET_PRIORITY 0, 0, 16
```

This command sets the priority of task number 0,0 (built-in BASIC) to 16 - half the value set when you turn your computer on. Such a command might be used to give more time to other tasks once they had been loaded by BASIC. You are not allowed to set the priority of task 0,0 to zero, since that would make the entry of further commands impossible! If you try to do so you receive the 'Bad parameter' report. If the task you specify does not exist, the error report is 'Invalid job' - 'job' is just another term for 'task'.

STOPPING TASKS

You can remove a task from memory with the REMOVE_TASK command. You must identify the task with the two numbers from the list, as with SET_PRIORITY:

```
REMOVE_TASK 1,1
```

If the task identifier you specify does not correspond to a job which is currently loaded, 'Invalid Job' is reported. 'Job' means the same thing as 'task'. 'Not complete' is reported if you try to remove task 0,0. This is not allowed as it would make it impossible to enter further commands.

When a task is removed, all the channels it was using are immediately closed, devices are made free for the use of other tasks, and the memory in which the task was running is released. This happens automatically when STOP or NEW is encountered in a compiled program.

MEMORY

The function FREE_MEMORY returns the amount of space available to SuperBASIC, or the amount of unused space within a task's data area, depending upon whether it is called from SuperBASIC or Supercharged BASIC. For example:

```
PRINT FREE_MEMORY
```

TRAPPING ERRORS

A number of un-documented commands which deal with error trapping are built-in to later QL systems. SUPERCHARGE does not recognise these, for three reasons:

(1) They have not been formally specified or documented, so their operation may well change from one version of the QL to the next. SUPERCHARGED programs are designed to run on all versions of the QL.

(2) There are several serious bugs in the new commands (at least in the "JS" and "MG" ROM implementations, which were the only versions of the commands which had escaped from Sinclair when SUPERCHARGE was written).

(3) The first versions of the QL which featured the error trapping commands did not become available until nine months after work on SUPERCHARGE began, and the launch of the compiler would have had to be delayed to allow the new commands to be accommodated in any form, since they have not been implemented as 'proper' SuperBASIC extensions (add-on procedures or functions).

In view of these problems we have not implemented 'trap' commands based around the WHEN keyword.

However, we have provided a new function, DEVICE_STATUS, which lets you check for possible errors in the most common problem-area - when you need to open a channel to a new device, perhaps using a name supplied by the user. It is difficult to get around the need for such a facility when writing serious programs in SuperBASIC - indeed, we wrote DEVICE_STATUS when it became obvious that we would need it in order to write SUPERCHARGE properly!

DEVICE_STATUS is a function which expects a single, string parameter. The string should be the name of a QL device, followed by parameters (if any) or a file name.

The function analyses the string to find out whether or not it starts with the name of a device on the current QL. Any parameters of the name (for example 'con_448X180A32X16' or 'ser1EHC') are then checked.

Finally the function attempts to open a channel to communicate with the device specified. If successful, the channel is closed and any file which has been generated 'en route' is deleted.

DEVICE_STATUS returns a number which indicates the degree of success it had in performing the above operations. The numbers are tabulated below.

The function automatically adapts to different hardware, so you can use it on a basic QL system, secure in the knowledge that it will also work with floppy disks, modems, 'parallel' printers and so on. For example, the following command indicates that the file 'SUPERCHARGE_SPEC' exists on floppy disk number 1:

```
PRINT DEVICE_STATUS("flp1_supercharge_spec")
-8
```

VALUE RETURNED	MEANING
0 or more	The device exists, and is not busy; a file with the name specified (if any) does not yet exist. The name or other parameters (if any) are valid.
-3 or -6	The device name and parameters are valid, but the QL has insufficient free space to open a new channel to the device.
-7	There is no device with the specified name on this QL.
-8	A file with the name specified exists on the device specified.
-9	EITHER the device specified exists, but it is already in use and no other task may use it until the present one has finished; OR the file specified is in the process of being written.
-12	The device name is valid, but the file name or parameters are not.

Table of values returned by the DEVICE STATUS function.

Input validation — a useful 'trick'

If you normally use WHEN ERROR to trap errors which crop up when users type text instead of numbers, consider this simple 'trick'. Read input into a string variable, rather than directly into a numeric variable. Then, if A$ is the string which you wish to evaluate, the command:

```
X="0" & A$
```

will set X to the value of A$, or zero if A$ is not a number. In either case there will be no error.

COPYING EXTENSION COMMANDS

In view of the fact that DEVICE_STATUS is such a useful command, and in the interests of encouraging compatibility among QL BASIC programs, the command and its code may be copied and used in any of your own programs, including published work. This offer applies whether or not your programs are compiled, and means that you may copy these two files for others, without restriction:

```
extensions_code  &  extensions_bas
```

These files also contain code for the commands LIST_TASKS, REMOVE_TASK, SET_PRIORITY and FREE_MEMORY. Similarly, you may use those in your own programs without restriction. You are NOT allowed to SELL the above files except as a component of your own programs. Of course, you may **distribute/sell** your own compiled programs - so long as they do not compete with SUPERCHARGE itself - **subject only to acquisition of a Site Licence.**

You may NOT copy the compiler or other parts of the SUPERCHARGE package for others; nor may you copy the documentation of the commands, or any other part of this manual, which is protected by International Copyright law.

We have spent over a year developing SUPERCHARGE, and further development work and maintenance is still going on. This work is only viable if those who use SUPERCHARGE pay for their copies. Action under civil or criminal law may be taken against those who ignore this notice.

If you are offered an 'unofficial' or 'pirate' copy of SUPERCHARGE, please note that Digital Precision will pay a reward for information leading to the successful prosecution of those who copy SUPERCHARGE without authorisation.

COMPILING 'NEW' PROCEDURES AND FUNCTIONS

SuperBASIC is designed to be an extensible language - procedures and functions can be added easily to the language, using a standard format. Most of these add-on procedures and functions may be used in compiled programs. We cannot say with certainty that ALL such commands will work, since we cannot allow for every remote possibility.

When the compiler detects the use of a non-standard routine it finds the parameter values in the usual way, 'pretends' that the interpreter is running, and then calls the code of the routine to do the work. Before execution starts the compiled program checks that all of the commands it needs are loaded; if this is not the case an error message is produced:

<name> IS NOT LOADED.

for each missing command. You must load the commands into SuperBASIC before the compiled program will run.

SUPERCHARGE cannot check the 'types' of parameters when a program is compiled, since it does not know what is valid. All it can do is extract the specified values from the compiled task's variable area, and hope for the best. If you type REMOVE_TASK "FRED" instead of REMOVE_TASK 1,2 you only discover the error when you come to run the compiled code, since SUPERCHARGE doesn't know that the command should have two integer parameters.

There again, if you'd typed REMOVE_TROUSERS "fred" the compiler would spot the mistake at once, since REMOVE TROUSERS is not a valid SuperBASIC command (on our systems, at any rate!).

There is one point which may not be obvious. File or device names which are used as parameters of 'add on' commands MUST be typed in inverted commas. Otherwise SUPERCHARGE cannot tell whether you are refering to a floating-point variable or a file. The format of a variable name and a file name is identical, so you have to make the distinction by using inverted commas. For example:

EX flp1_fred

would have to be changed to read:

EX "flp1_fred"

To help you detect this problem, SUPERCHARGE issues a 'warning' message whenever it encounters a name that could be taken as a device name or a variable.

SUPERCHARGE does not insist that add-on code is loaded into any particular area of the QL's memory; as a compiled program begins to execute it searches out appropriate routines by name.

This can cause problems if you have two different commands with the same name, and compile while one is in memory and execute the program with the other loaded.

In such a case SUPERCHARGE passes parameters intended for the first routine to the second, probably causing a 'bad parameter' error. The effect is just as if you had loaded the wrong definition in normal SuperBASIC: you get the wrong result, or - more likely - the program stops with an error message.

If a name has been loaded more than once, the FIRST version loaded is used by SUPERCHARGE. The interpreter uses the same rule in versions "AH" and "JM" of the QL, but later versions ("JS", "MG" and thereafter) use the most recent version.

As ever, SuperBASIC is a moving target and SUPERCHARGE can't be exactly compatible with both versions. So long as you resist the temptation to load more than one copy of a given command, all will be well.

The degree with which the compiled program can 'pretend' that it is the interpreter is obviously limited - if the imitation was perfect the compiler would have to work almost exactly like the interpreter, in which case there would not be much point in bothering to compile programs!

SUMMARY

Procedures or functions that modify their parameter values, process arrays (other than single strings), manipulate the stored program text, or rely on other interpreter data structures (such as the name table and name list) will not work when compiled. The majority of add-on commands do not do this, and consequently work perfectly.

SUPERCHARGE is fully compatible with the SUPER SPRITE GENERATOR and other Digital Precision extensions to SuperBASIC.

BASIC EMULATION — SOME TECHNICAL DETAILS

The format of add-on procedures and functions is discussed in detail in a number of books on the QL and its operating system. We do not propose to duplicate that information in this manual, but here are a few technical comments may be useful to those who seek to write new procedures and functions, or to check that existing ones are compatible with SUPERCHARGE. In general, procedures and functions which do not rely on the presence of SuperBASIC interpreter data-structures will work perfectly.

Parameters

SUPERCHARGE emulates the SuperBASIC mechanism for parameter-passing. In particular, it sets up a table of parameter descriptions between (A3) and (A5), like the Name Table entries used by the interpreter. Only parameter-types, addresses and separators appear in the table - other variables, procedures and so forth are NOT represented.

Values passed back to the program via parameters will be lost, but functions work as normal. The result of a function should be stacked on completion, with its type indicated by the value in D4 on completion. The compiled code automatically coerces this value to a type corresponding to the name of the function - string if the name ends with a '$' sign, integer if it ends with '%', and floating-point otherwise.

The Maths stack

The Maths stack, addressed by (A6,A1.L), is maintained in a form identical to that used by the interpreter. Integer, Float and String values are represented in the usual way.

Resident procedures and functions are not guaranteed to find more than 100 bytes free on the Maths stack. The BV.CHRIX routine cannot be relied upon to expand the Maths stack, as tasks have to run within fixed (rather than dynamic) bounds. If the Maths stack overflows SUPERCHARGE can usually diagnose the problem when it recovers control. If your procedure or function requires a lot of Maths stack space, you should avert errors by checking the free space before the procedure or function is called. The FREE_MEMORY function returns the amount of space available below the Maths stack if it is called from within a compiled program.

Some 'free' space is taken up by the parameter-passing mechanism. You should allow 10 bytes for each integer parameter, 14 bytes for each floating-point value, and (length+11) bytes for each string parameter.

A better way of checking the amount of free space is to examine the values of the system variables BV.BFBAS and BV.TKBAS, within your code. These delimit the SuperBASIC area called 'buffer'. When a compiled program is running these A6-relative pointers indicate the bottom and the top of the area into which the stack can expand, respectively.

Of course, this trick only works when code is called by the compiler; you can check whether or not this is the case by examining the value of the system variable BV.TGBAS - $54(A6) - this will always be negative when code is called from within a compiled program, whereas it will always be positive when the SuperBASIC interpreter is in control. The function FREE_MEMORY uses this fact to determine whether it should return the size of the task stack or that of the QDOS 'free' area.

Routines should not use more than 128 bytes on the User (A7) stack. This should not be a restriction, since the same requirement is imposed by the interpreter.

System variables

A set of pseudo-system-variables is provided for the use of procedures and functions; these are addressed by A6, as usual. Some of these system variables are 'dummies' - only those needed to support standard ROM routines are given meaningful values.

You can make use of the values of BV.RIP and BV.VVBAS, although you should not try to store values in the VV area. BV.TGBAS contains a 'dummy' value, so that inadvertent calls to BV.CHRIX do not try to 'expand' the task, with potentially disastrous consequences. As explained above, the 'free space' area is delimited by the values of BV.BFBAS and BV.TGBAS. BV.BRK is set before each call.

A table similar to the SuperBASIC channel table (but limited to sixteen standard 40-byte entries) is maintained within compiled tasks. The pseudo-system-variable BV.CHBAS points to the base of this table. The byte at offset 17 within each channel entry is reserved by SUPERCHARGE - it is used to handle the '!' separator.

SUPERCHARGE AND SUPERBASIC COMPATIBILITY

Insofar as is possible, the SuperBASIC compiler is exactly compatible with the interpreter - programs written to run under the interpreter perform in just the same way when compiled. However, there are a few differences, explained in this section. In each case the reasons for the difference are given, along with suggested actions to be taken if the difference affects your program.

The headings under which the differences are discussed are listed below. Detailed explanations follow the list.

(a) SuperBASIC Identifiers.

(b) Arithmetic range and accuracy.

(c) Array and String handling.

(d) Nesting of structures.

(e) Computed line references.

(f) Program editing.

(g) Calculations in DATA.

(h) Channel numbers.

(i) Parameter locality.

(a) SuperBASIC Identifiers

Identifiers are defined in the 'Concepts' section of the Sinclair QL User Guide. In brief, they are sequences of up to 255 characters, used to identify variables, procedures, functions and program structures. They are often referred to in QL documentation as 'names'.

In interpreted SuperBASIC, you may use the same identifer for different purposes at various points in your program. For instance, you could use the identifier VECTOR to describe a one-dimensional array in one part of a program, and a three-dimensional array elsewhere. This is bad practice, since it might cause confusion and errors, but it is allowed by the interpreter, which always uses the 'most recent' declared meaning of the identifier. The flow of the program determines what the most recent declaration was.

If a program is to be compiled each identifier must have a single, distinct meaning throughout the listing, so that the compiler can generate reliable code to process it. If an identifier is used for a simple (un-subscripted) variable at one point, it may not be used for an array elsewhere in the program. Functions and procedures must also have 'unique' names.

The last character of a name is treated as significant when the compiler checks the uniqueness of names. Thus:

VECTOR VECTOR% VECTOR$

are all unique names, and may be used for different, distinct purposes in a compiled program.

Variable values may have three types in SuperBASIC: Integer, Floating-point or String. The compiler deduces the type of a name from its last character. Thus VECTOR may only be used to refer to float (decimal) values, VECTOR% to integer values (whole numbers between -32768 and 32767) and VECTOR$ to string values (sequences of characters).

This rule differs in one small respect from interpreted SuperBASIC, where the type of a parameter passed to a procedure or function is determined by the type of the corresponding value. In compiled programs, the type of each parameter must be known at the time of compilation - otherwise code to manipulate the parameter cannot be generated. Thus, in a compiled program, the last character of a parameter name dictates its type, just as is the case for other variables.

There are three distinct uses for an identifier:

(i) Simple variables: integer, string or float values. These names may be used to store values or to identify FOR and REPEAT loops. Only simple variable names may be used in the first line of a SELECT statement. These rules correspond to those in SuperBASIC, with the added feature that integer and string identifiers may be the subject of compiled FOR, REPEAT and SELECT statements.

(ii) Array variables: the names used for arrays, declared in DIM or LOCAL statements. An array identifier may be declared more than once in a program, but the number of dimensions (not necessarily their sizes) must be the same in each declaration.

(iii) Definition names: the names used for procedures and functions. Every procedure or function must have an unique name.

If an identifier is used for any one of these three purposes it may not be used for either of the others. In such a case the identifier would be rejected by the compiler as 'ambiguous'. All references to an 'ambiguous' name are clearly marked. The only solution - if you must compile the program - is to go through the code, assigning new names, until the ambiguity is removed.

The SuperBASIC interpreter makes very few checks for the ambiguous use of identifiers. There are some cases in which the interpreter allows ambiguous names to be used in a program, but in general they cause ambiguous results or program failure.

(b) Arithmetic range and accuracy

The SuperBASIC compiler supports the same range of arithmetic values as the interpreter. However the displayed accuracy for floating-point values is greater - nine decimal digits are generally displayed, rather than the seven or eight shown by the interpreter. The internal format is sufficient to allow numbers to be accurately stored to nine digits of precision, so long as a simple rule is followed.

WORK IN WHOLE UNITS when exact precision is needed. In UK business programs this means working in pence. Similarly, in other countries with decimalised currency, work in cents or their equivalent. If your country has only one unit of currency you should obviously only use that unit, and not fractions thereof. The increased precision available through the use of SUPERCHARGE will be especially useful to those working in Yen, Pesetas, Lire or similar currency where one unit represents a very small amount of wealth.

SuperBASIC (whether interpreted or compiled) stores numbers in binary form, which means that fractions are stored as reciprocals of powers of two. The value 3/4 is stored, accurately, as 0.11 in binary - 1/2 plus 1/4 - but many other fractions, such as a tenth or a hundredth, cannot be expressed exactly as a binary fraction, however many digits are used. So long as you work in whole units this is not a problem, since only fractions - up to 1e9 - are inexact.

To make the writing of business programs easy we have listed two routines which allow numbers to be read and printed in decimal form without loss of accuracy.

The routines are as relevant to interpreted programming as they are to compiled code. In both cases errors may occur if monetary quantities are entered as decimal values. You can confirm this by typing a simple calculation - say:

```
PRINT 25.42 - 25.43
```

at the QL keyboard. The result printed is not exactly 0.01. This is not a bug, but a natural consequence of the use of binary arithmetic - similar results occur on all computers that use fast binary arithmetic, including the IBM PC. By way of contrast, try:

```
PRINT 2542 - 2543
```

So long as you work in whole units, using up to nine digits, you will get exact results, although the SuperBASIC interpreter only displays a maximum of seven digits.

PRINT_MONEY and INPUT_MONEY get around the problems of decimal input and output by the use of strings to store decimal values.

PRINT_MONEY expects two parameters. It prints the second value to the channel specified by the first value, without advancing to the next line. In order to give neat results, a minimum of four characters is always printed, so that, for example, the value '0' would be printed '0.00'.

```
DEFine PROCedure PRINT_MONEY(channel,amount)
LOCal money$
  money$=amount
  IF amount<100 THEN money$="0" & money$
  IF amount<10  THEN money$="0" & money$
  PRINT #channel;money$(1 TO LEN(money$)-2);
  PRINT #channel;".";money$(LEN(money$)-1 TO);
END DEFine PRINT_MONEY
```

INPUT_MONEY, listed overleaf, will accurately read monetary amounts from a specified channel. The code is rather long when compared with a simple INPUT, but it does incorporate extensive error checking. A normal INPUT would cause the program to stop if incorrect characters were entered (but see the 'Input Validation' trick introduced in Chapter 5).

INPUT_MONEY assumes the existence of a procedure COMPLAIN which is called if an error is detected. The function checks for five possible errors, rejecting very brief entries, those which contain non-numeric characters, more than one decimal point, more than two digits after the point or more than seven before the point. The result is returned in pence.

```
DEFine FuNction INPUT_MONEY(channel)
LOCal pence, pounds, digit, sign, pointpos, money$
  INPUT #channel;money$
  IF LEN(money$)=0 THEN COMPLAIN "Entry too brief."
  IF money$(1)="-" THEN
    sign=-1
    IF LEN(money$)=1 THEN COMPLAIN "Entry too brief."
    money$=money$(2 TO)
  ELSE
    sign=1
  END IF
  FOR digit=1 TO LEN(money$)
    IF NOT (money$(digit) INSTR ".0123456789") THEN
      COMPLAIN "Invalid character in entry."
    END IF
  END FOR digit
  pointpos="." INSTR money$
  IF pointpos=0 OR pointpos=LEN(money$) THEN
    pounds=money$
    pence=0
  ELSE
    IF "." INSTR money$(pointpos+1 TO) THEN
      COMPLAIN "More than one point."
    END IF
    pounds=money$(1 TO pointpos)
    pence=money$(pointpos+1 TO)
    IF pence>99 THEN
      COMPLAIN "Invalid number after point."
    END IF
  END IF
  IF pounds>9999999 THEN COMPLAIN "Number too large."
  RETurn (pounds*100+pence)*sign
END DEFine INPUT_MONEY
```

The accuracy of arithmetic in a compiled program is never less than in its interpreted counterpart. Paradoxically, compiled output may sometimes look less accurate, since answers are displayed more precisely - this means that very small errors caused by the conversion to binary can be seen, where previously they were obscured by rounding.

When the interpreter performs arithmetic, it converts all values into floating-point (decimal) form as they are encountered. This means that extreme values (from 10E-615 to 10E+615) can be processed, but it imposes a considerable processing overhead. The internal routine used to compute the total of 2 and 2 must also be able to add 1234.5 and -6.7859, or any other values.

Integer variables are allowed by the interpreter, but it does not process their values any more quickly since they are converted into floating-point form. Integers use less memory (two bytes rather than six) as they can only record whole numbers between -32768 and 32767.

SUPERCHARGE is capable of performing true, fast integer arithmetic. This means, for instance, that the result X%*Y% can be computed with a single processor instruction, rather than the scores of instructions needed for a floating-point multiplication. The same is true for other operations such as comparison, addition, subtraction and division.

Integer arithmetic is inherently faster and more concise than floating-point arithmetic. It is important to use integer arithmetic as often as possible, especially in loops and when computing array subscripts, in order to obtain the maximum possible increase in program performance when compiling.

However, the restricted range of integer arithmetic can cause problems of incompatibility between compiled and interpreted programs - especially in the obscure one-in-a-million circumstances which compiler-writers dread!

Consider this example:

```
10 PRINT A%+1
```

The compiler recognises that A% is an integer variable, and '1' is an integer value. It consequently generates a single, fast instruction to add the two integers. But if the value of A% happens to be 32767 (the maximum integer) the addition fails, since it produces a result which is outside the permissible range of integers.

When this statement is executed by the interpreter both A% and '1' are converted into floating-point values as soon as they are encountered. A floating-point addition is used, giving the correct result of 32768.

Of course, this is an unlikely example. If the line were:

```
10 B%=A%+1
```

both the interpreter and the compiler would report an error if the statement was started with A% holding the value 32767 - the interpreter would recognise that the result 32768 could not be stored in an integer variable.

However, more intricate traps can crop up. Consider this integer calculation:

```
10 B%=A%*3/2
```

In this case the compiler may detect an error when executing the statement. If A% has a value greater than (32767/3) the temporary result of the multiplication would exceed the range of valid integers. In this case, the ordering of the expression is important. The line:

```
10 B%=A%/2*3
```

would work correctly for values of A% up to (32767/3*2) - at which point the result would be too great to fit in B%.

SUPERCHARGE does not enforce this ordering when it evaluates expressions, since the order of execution has an effect on the accuracy of the result in such instances. The division is performed using integer arithmetic - discarding the remainder - so that the value of B% calculated by the compiler is always a multiple of 3. In contrast, when the interpreter is used, the remainder after division is taken into account, giving a more accurate result.

Although there are some pitfalls associated with the use of integer variable arithmetic, they crop up rarely in practice and can be easily corrected by the use of floating-point variables in specific instances. An integer value can be forcibly converted to floating-point, in interpreted or compiled programs, by raising it to the power of 1.

(c) Array and String handling

With typical vagueness the QL User Guide specifies that 'under certain circumstances' a name may be used to reference more than one element of an array. These circumstances are a little more restricted under the compiler than they are under the interpreter.

Supercharge only supports array 'slicing' - the specification of more than one element of an array with a single reference - for the last dimension of strings. SUPERCHARGE only allows arrays to be passed as parameters in the case of one-dimensional character arrays (strings). The maximum length of such strings is the length of the parameter text. The default 'end' slice is taken to be the latest LEN of a string, unless this is 0, when the physical length is used. The slicing of implicit strings (eg; "FREDDY"(2 TO 4)) is not supported. When a SuperBASIC program is interpreted, undimensioned strings are allowed to have any length up to 32766 characters. As characters are added to the string, memory is allocated and de-allocated as required. This causes 'storage fragmentation' - used and unused memory become muddled together - which is why interpreted BASIC programs that use arrays tend to 'grow' in size as they run.

Compiled programs must occupy a static amount of memory if they are to be executed as multi-tasking jobs, so they cannot use memory in the same way. In order to keep things tidy, all undimensioned strings in a compiled program are assigned a fixed maximum length of 256 characters. The effect is just as if a DIM $(256) statement appeared at the start of the program for every otherwise-undimensioned string.

If you find the limit of 256 characters restrictive, you can over-ride it by adding an explicit DIM statement to the compiled program. Thus, if you need to be able to store up to 4,000 characters in the string PARA$, use the command:

```
DIM PARA$(4000)
```

at the start of your program. This command does not disturb the interpreter, so you can test your program as normal.

It is important not to assign more space than you need in this way, since each DIM increases the size of the compiled task in memory; if you use a lot of very large strings you could make it impossible for the task to run at all. The compiler can cope with strings of up to 32,764 characters.

NOTE 1: The CLEAR command makes all arrays undefined; this includes strings, so you must re-define ALL strings with explicit DIM statements if you wish to use them after a compiled CLEAR command.

NOTE 2: Under some circumstances the interpreter will return the length of a string when asked for its zeroth element! SUPERCHARGE doesn't do this.

The total number of elements in an array must not exceed 65,535. The limit allows the compiler to access array elements much more quickly than would be the case if larger arrays were allowed. This 'restriction' will only affect those with expanded QLs and very large storage needs, as it limits the size of a floating-point array to a mere 393K!

The total number of elements of each dimension of an array (one more than the number given in the DIM statement, since subscripts start at 0) must not be greater than 32,767. The last value in the DIM for a string array is treated as a maximum length rather than a dimension, so there may be 32,767 strings in a compiled array (not just 32,767 characters). A single compiled string element may hold up to 32,764 characters.

(d) Nesting of structures

Procedure and Function definitions must not be 'nested' in compiled programs. In other words, the end of one definition must preceed the start of the next.

This rule helps to simplify the compiler, without imposing any restrictions on the power of compiled programs. The interpreter simply ignores the nesting of definitions, and determines the 'scope' of variables at run-time.

Control structures - FOR, IF, REPEAT and SELECT statement groups - must be 'properly nested'. In other words, the end of a structure must not appear within another structure unless that first structure also began within the second.

Two examples should make this clear:

```
10 REMark Legal nesting
20 FOR I=1 TO 10
30    REPeat X
40    END REPeat X
50 END FOR I

10 REMark Illegal nesting
20 FOR I=1 TO 10
30    REPeat X
40 END FOR I
50    END REPeat X
```

The indentation in the second example emphasises the error.

It might be possible to make the second listing run under the interpreter, if GO TO statements were scattered around so that the loops didn't cross one another at run-time.

Even if the GO TOs were present, the second example would not be allowed by SUPERCHARGE. The compiler cannot follow GO TOs and other transfers of control, working out the flow of control. It would have to 'execute' every line with all possible values for such a check to be exhaustive. This would make SUPERCHARGE complex and extremely slow, if not completely unworkable; consequently program lines are only considered in the order in which they occur.

It follows (by implication) that each structure must have a single END. You may have as many EXITs, NEXTs and RETURNs from a control structure as you wish.

You don't need to specify the ENDs of 'single-line structures' - the 'short forms' specified in the QL User Guide. Some errors in the interpreter make the interpretation of these short-forms unreliable. In particular, the interpreter treats GO SUB calls as structure terminators. The bugs are not present in compiled programs.

(e) Computed line references

The SuperBASIC interpreter allows a calculation to appear anywhere a line number is expected. It matches a line reference to the first program line with a number equal to, or greater than, that specified.

In compiled programs, you are only allowed to specify a constant number where a line reference is required; furthermore, it must be the exact number of a program line.

This rule affects the GO TO, GO SUB and RESTORE statements. It allows the compiled program to be shorter and faster (since a table of line addresses is not needed at run-time) at a small cost in flexibility. If you really need to compute a line reference, you should use ON..GO TO or ON..GO SUB rather than calculate a line number. Alternatively you might use the SELECT construct, which is a more general and secure way of coding a choice between several alternatives.

In a compiled program, SELECT or IF must be used in place of a computed RESTORE, since SuperBASIC has no ON..RESTORE statement.

(f) Program editing

The SuperBASIC program editor is built-in to the interpreter. This is sensible when programs are being interpreted and tested but pointless once they are compiled as there is no 'program text' for the editor to manipulate.

The commands AUTO, DLINE, EDIT, LIST, LOAD, MERGE, MRUN, RENUM and SAVE are not supported by the compiler since it leaves no program text for them to operate upon.

The commands CONTINUE and RETRY are not supported since they are designed for use while interactively debugging a program. This is not possible from within a compiled program!

(g) Calculations in DATA

Sinclair SuperBASIC is unusual among BASIC implementations in that it allows calculations to be specified in DATA statements.

SUPERCHARGE does not allow expressions in DATA statements, since their processing would complicate the generation and execution of compiled code. Only string and numeric constants are allowed, although an exception is made for the four unary operators: +, -, NOT and bitwise negation, which may be applied to numeric constants. Explicit assignments must be used to obtain the effect of calculations in DATA.

The bugs in the interpreter's handling of READ, DATA and RESTORE statements do not crop up in compiled programs.

(h) Channel numbers

The QL User Guide does not state a maximum value for the channel numbers used in statements such as OPEN and CLOSE. In fact, when the SuperBASIC interpreter is used, the maximum channel number depends upon the amount of free memory. Compiled programs must run within a limited area of memory, so SUPERCHARGE only allocates space for 16 channels, numbered from 0 to 15. This should be ample for almost all purposes - remember that channel numbers may be re-used once previously associated files have been closed.

(i) Parameter locality

When SUPERCHARGE is used, parameters (variables declared in DEF statements) are local to the definition in which they appear. Changes in their values do not affect external variables. In other words, changes to a parameter never affect the variable from which the parameter was derived.

SUPERCHARGE always passes parameters by value, whereas the interpreter uses 'call-by-value' or 'call-by-reference', depending upon the format of the calling statement. This is mentioned in the QL User Guide, 'Concepts' section: 'Functions and Procedures'. The original QL documentation was very vague about parameter correspondence, so that SUPERCHARGE was written without making use of variables passed by 'reference'. When we found the flaw we looked at the code needed to fix it, and discovered some snags.

At present SUPERCHARGE invokes procedures by evaluating the parameters (if any) and jumping to the start of the procedure, which contains code to set up corresponding local variables. This is the mechanism used in most compiling languages, which require that the form of parameter correspondence is defined in the heading of the procedure. But in SuperBASIC the format of the calling statement determines how parameters correspond; a parameter might be passed by reference once, then by value, in two calls to the same procedure. If SUPERCHARGE allowed reference parameters, the 'set up' routine would have to be part of every call, rather than part of the procedure. This could make the compiled program much longer.

We reluctantly decided that the cost, in terms of delayed launch and reduced performance, would be greater than the compatibility to be gained by supporting reference parameters in SUPERCHARGE.

If you find that a program which you wish to compile passes values back from procedures or functions via the parameter list, you should use a 'global' variable - one which exists outside the routine which is being called - to pass and store the value, rather than a parameter. This is inelegant and may require that you add a few lines to your program, but it will ensure that it works efficiently and compatibly whether it is interpreted or compiled.

A similar mechanism is used to pass values to machine code procedures or functions, so they may not return values via their parameters either. Very few routines try to do this.

'BUGS' FIXED BY THE COMPILER

Although SuperBASIC is, in general, a sophisticated and flexible programming language, there are a number of faults in the interpreter built in to the QL ROM. We have taken the liberty of correcting many of these faults through SUPERCHARGE, although the corrections make the compiler slightly less compatible with the interpreter than would otherwise be the case! This list summarises the more important faults which are corrected by SUPERCHARGE.

(1) Interpreted BASIC can crash if more than nine parameters or local variables are used in a single procedure or function. SUPERCHARGE can cope with any number of parameters or local variables without problems.

(2) Mathematical results are only displayed to a maximum of seven decimal places by the interpreter, even though it works internally to a greater precision. The compiler displays nine decimal places (enough to show quantities of up to 9,999,999.99 pounds, rather than 99,999.99). The discussion of floating-point maths earlier in this chapter indicates the steps you should take to avoid rounding errors, which affect SUPERCHARGE and SuperBASIC equally.

(3) SuperBASIC stops with a 'buffer full' error if more than 128 characters are read in response to INPUT, on QL versions AH and JM. The compiler allows up to 32767 characters to be read, although this does depend upon the amount of free memory available.

(4) The interpreter does not allow multi-tasking of BASIC programs. Many compiled programs may run at one time, so long as there is enough room for them all in memory.

(5) The interpreter cannot handle SELECT statements for integer or string variable values. Integer and string SELECT works perfectly when compiled with SUPERCHARGE.

(6) Variables passed as procedure parameters may not be used in SELECT statements (unless they were the last parameter), when a program is run on a version JS QL. The compiler does not impose this restriction.

(7) The BASIC function RESPR, used to reserve memory, does not work under the interpreter if a multi-tasking job is running. The compiler always allows RESPR, since it allocates memory from within the task's environment rather than the resident procedure area, which cannot expand while the 'adjacent' transient program area is in use.

NOTE: add-on commands should be loaded from BASIC, not from within compiled programs. This ensures that they are correctly linked into the interpreter, rather than into the compiled program (which won't know what to do with them).

(8) The interpreter does not process single-line (short form) FOR loops properly if they contain the GO SUB command - the GO SUB stops the loop. This bug is fixed when the compiler is used.

(9) The CALL statement, used to invoke machine-code routines, crashes the interpreter if it is used in programs longer than 32K bytes, on QL versions up to JM. The compiler handles CALL correctly on any version of the QL.

(10) Integer FOR loops (e.g. FOR I%=1 TO 2) are not allowed by the interpreter. SUPERCHARGE can generate code for integer FOR loops; unfortunately current versions of the SuperBASIC editor do not allow such lines to be entered, so this correction will not have any practical effect until the editor and interpreter are both changed to correct the bug. In the meantime REPEAT loops, with explicit counting, should be used when fast integer loops are needed. This is discussed in Chapter 7, 'Getting the most from SUPERCHARGE'.

(11) The value of the identifying variable of a FOR or REPEAT statement is set to zero by the interpreter when the statement is encountered. For example, the commands:

```
LET T=3
FOR T=T TO 6:PRINT T
```

print values from 0 to 6, rather than from 3 to 6. This bug is fixed in SUPERCHARGE, although it should be noted that it is poor programming style to use one variable for two logically-distinct purposes. In the example, T is used first as a limit, then as an iteration count.

WHAT COMPILERS CAN AND CAN'T DO

SUPERCHARGE increases the speed of most BASIC programs by a large factor, simply by rationalising the steps needed to produce a given effect. In general a speed-up factor of between 5 and 20 times can be expected on short programs. Long programs are often accelerated much more, since (contrary to advertising claims) the SuperBASIC interpreter does get steadily slower as program sizes increase, while the speed of compiled programs is constant (and faster).

In this chapter we explain what the compiler can and cannot do. We quantify the speed advantage SUPERCHARGE gives to some 'standard' BASIC programs, and look at simple ways in which programs can be made even faster.

Supercharging your hardware

SUPERCHARGE is just a program (albeit a very sophisticated program), so it cannot change the speed of the QL hardware. If the time taken to execute your programs is dominated by data transfer delay (on the microdrives, display, or RS-232, for example) it is unlikely that compilation will make a large difference. After all, SUPERCHARGE would not be very useful if it attempted to 'speed up' RS-232 transfer rates to 96,000 baud, or spin the microdrive tapes at a rate of 10 metres per second - the hardware just couldn't cope!

SUPERCHARGE can give dramatic speed improvements on programs that copy data, especially on semi-random devices such as disks and microdrives, since it reduces the delay between reading and writing so that data can be packed more densely. The only way to discover whether or not your program falls into this category is to compile it.

Even if SUPERCHARGE does not dramatically improve the speed of your program it may increase its utility. It might be argued that speed is almost irrelevant, since compiled programs may multi-task - so you can get on with something else while your compiled code runs 'in the background'.

Compiled programs generally load more quickly than their interpreted counterparts; the difference in speed may amount to a minute or more for long programs. Compiled code is more 'secure' in that it cannot easily be listed or modified (except by those with the original BASIC). The precision of numeric results is also increased.

TRICKS, TYPES AND IN-LINE CODE

In most cases we are not solely concerned by the speed at which a program runs. Unless you are Sebastian Coe it usually doesn't matter whether a process takes half a second or a hundredth. But the time may come when you need to increase the speed of a program by a factor greater than that offered automatically by SUPERCHARGE.

In this section we explain how you can obtain speed-increases of up to 100 times by making small changes to your BASIC program. If you are writing a new program it may be useful to bear these guidelines in mind as you work.

Loops, arrays and function-calls

A few lines often consume the majority of the execution time of an entire program. A useful rule of thumb is the idea that ten per cent of a substantial program is executed for ninety per cent of the time (and vice versa). It follows that well-chosen, localised changes to a program can have a dramatic effect on its overall run-time.

In general there is little point in trying to accelerate sections of a program which wait for input or do not form part of a loop. 'Nested loops' are most important - these are loops within loops. You need only optimise the innermost loop to obtain most of the possible speed-up, since that code is executed most often. It is especially important to reduce the code at the heart of a loop to the minimum. Often some steps are performed in loops when they might just as well appear outside. Let's take an example:

```
FOR I=0 TO PI STEP PI/180
A(J)=A(J)+SIN(I)*COS(THETA)
END FOR I
```

This loop is performed much more quickly if the COS calculation is moved outside the loop. At present the value of COS(THETA) is worked out 180 times more than is really necessary, and the array A() is indexed needlessly. This code runs more than twice as fast:

```
TEMP=COS(THETA)
TOTAL=A(J)
FOR I=0 TO PI STEP PI/180
TOTAL=TOTAL+SIN(I)*TEMP
END FOR I
A(J)=TOTAL
```

Note that there's no point replacing PI/180 with a constant or variable, since the Start, End and Step values used in a FOR loop are only worked out once, when the FOR is found.

SUPERCHARGE cannot do much to accelerate the rate at which complex mathematical functions are evaluated, since most of the interpreter's time is spent working out the result - the formula is so complex that this delay swamps the delay while the interpreter decides the action to be performed.

In this case SUPERCHARGE, and all other QL compilers, is limited by the 68008 microprocessor. The algorithms used are efficient. The only way to speed evaluation would be to reduce the precision required or to fit special-purpose hardware, such as the 68881 arithmetic co-processor. The Intel 8087, available as an option for the IBM PC and other machines, is not a great deal faster than the QL at floating-point maths.

You can obtain useful results by compiling your own low-precision mathematical functions (by summing suitable series) with SUPERCHARGE. This is because the fundamental floating-point operations of multiplication, division and so on are performed quite fast, especially when the overhead imposed by interpreted execution is not present. This is not the place to explain the way in which mathematical functions can be approximated using simple steps; some readers might be relieved to hear that! Masochists & mathematicians are advised to read Chapter 11 .... Alternatively, you might be able to replace calls to resident functions with variable or array-references. This technique is useful if you use the same values of a function again and again. SUPERCHARGE is much faster than the interpreter at accessing arrays of any number of dimensions, and in any case complex functions (LN, SIN, SQRT etc) take much longer to evaluate than array subscripts.

It can also be useful to reduce the number of calls to machine-code functions in compiled programs. In order to preserve exact compatibility with the interpreter, and keep the size of compiled programs within bounds, many of these are executed using calls to QL ROM routines. This process by-passes the QL's slow expression-evaluation and interpretation, but the code used is inevitably more 'general' than it need be in some circumstances.

This is only really worth considering if that part of your program needs to run at top speed. The principle is most relevant where string-handling is concerned, as we discuss in the next section.

Fast string handling

SUPERCHARGE and the SuperBASIC interpreter differ in the schemes that they use to represent strings. The compiler stores strings in two parts - a fixed-length part, containing the string address and the length, and a variable-length part containing the text. The interpreter stores the length and the text together, in a separate area. This makes access to strings other than the 'most recent' difficult, and massively reduces the speed of string concatenation (&) - putting it simply, the length information 'gets in the way' of the interpreter.

The upshot of this is that SUPERCHARGE is extremely fast at slicing and appending strings, but the advantage is reduced if lots of comparisons and function-calls are mixed in - in such cases the compiler has to re-organise its store to suit the BASIC interpreter. This is done quickly and efficiently, but it still imposes an 'overhead' which should be avoided by programmers seeking maximum speed. Comparisons are slowed by the complex rule used to sequence strings, which takes special account of embedded numbers.

Integer arithmetic

When the interpreter processes numbers it goes about it in a rather long-winded way. The variables and constants normally used in SuperBASIC are 'floating-point' - in other words, they can represent a vast range of values, to an accuracy of 31 bits (about one part in two billion). The 68008 processor has simple, fast instructions to add and subtract values, and even to perform multiplication and division, but these instructions can only cope with a much smaller range of values.

Normal SuperBASIC adds 2 and 2 in a number of stages. It works with the magnitude (or exponent) and digits (or mantissa) separately. It fetches both pairs of values, checks that they are both of about the same magnitude, and adjusts them if need be. It adds the digits, makes another adjustment if the magnitude has changed in the process, and stores the result, in two lumps. The process for multiplication and division is even more complicated, and in any case it is much slower than those one-step instructions.

You can take advantage of the QL's fast instructions by specifying that some variables in your program are 'integers'. The word means 'whole numbers'.

In SuperBASIC integers are confined to the range -32,768 to +32,767. Integer variables are denoted by a per cent sign at the end, just as string variables are denoted by a dollar sign. Any fractional part assigned to an integer variable is ignored. Thus INTVAR%=10/3 leaves INTVAR% with the value 3 (not 3.33333333).

For many 'counting' applications, such as keeping football statistics, writing games, searching arrays and so forth, the limited range and accuracy of integers is not a problem - and the greater speed of integer arithmetic is very alluring. Most 'machine code' games would proceed at a snail's pace if the programmer was not able to take advantage of integer arithmetic.

The snag is that Sinclair's SuperBASIC interpreter contains very little support for the processing of integers. It can fetch them from integer variables and store them in integer variables, but that's about it. If you put the line:

```
SUM%=COUNT%+1
```

into an interpreted BASIC program, it takes LONGER to be executed than the line:

```
SUM=COUNT+1
```

This is because the interpreter only knows one way of adding. It can add a floating-point value to another, giving a floating-point result, but it doesn't know how to add an integer to an integer.

When the first example is executed, the interpreter starts by fetching the value of COUNT% and the value 1. It can only add floating-point numbers, so it laboriously converts the value of COUNT% before it performs the multi-step addition. The value 1 has been stored in floating-point form all along, even though it is a valid integer value. Finally, once the floating-point result is known, the interpreter turns it into an integer (equally laboriously) so that it fits into the integer variable SUM%.

This is obviously a silly way of doing things, although it is sadly common in micros - the Commodore 64, amongst other machines, does just the same thing. The upshot is that integer arithmetic is no faster than floating-point - in fact it may even be slower, due to all the conversion that has to take place. The 'advantage' is that two bytes are used to store the values, rather than six - this is only significant if you use large arrays.

SUPERCHARGE comes to the rescue at this point, because it allows proper, fast integer arithmetic. A large chunk of the compiler is concerned with keeping track of the types of values, so that integer arithmetic is used wherever possible. The compiler is limited by the need to remain compatible with the interpreter, and to process mixed floating-point and integer expressions, but it can still process integers five to ten times faster than the (carefully optimised) floating-point routines.

When you program in SuperBASIC you probably ignore the availability of integer variables at present. If you adjust your programs so that integers are used whenever possible you will find a substantial speed improvement - long BASIC programs which use integer variables may run a hundred times faster when compiled.

For best results you should use integer variables whenever the limitations of range and accuracy are not important. Be sure to use integer variables when accessing arrays or slicing strings. This does not restrict you, as subscripts must be whole numbers, and SUPERCHARGE imposes an upper subscript limit of 32767, for precisely this reason.

You can often get good results by changing the type of just a few variables, remembering the rule that a few lines are usually responsible for most of the execution time of a program. Experimentation and close examination of the program is your best guide here, although it can be useful to break into an interpreted program whenever a pause occurs, noting the appropriate line-numbers.

There is a problem with fast integer loops, since Sinclair do not allow an integer FOR statement on current QLs. If you try to type in a line like:

```
FOR LOOP%=1 TO 10: PRINT LOOP%: NEXT LOOP%
```

your computer may reject it as a 'bad line'. If your QL allows this command, you're in luck, and the rest of this page will not concern you. Otherwise, you will not be able to write integer FOR loops, since the compiler can only process lines which are correctly loaded into the QL. The solution is to use an REPEAT IF loop instead - this is rather slower, but still about three times faster than using a floating-point loop - even more so if you intend to use LOOP% as an array subscript.

```
LOOP%=1:REPEAT L1:PRINT LOOP%:LOOP%=LOOP%+1:
   IF LOOP%>10 THEN EXIT L1:END REPEAT L1
```

In-line code and threaded code

68008 machine code is, by nature, extremely verbose - it takes four bytes of code, for instance, to compare a register value with a character! There is an inevitable trade-off between concise and fast code, as the SuperBASIC interpreter demonstrates - if the authors had more space in which to write it they could doubtless have made it faster (although more time would doubtless not have gone amiss!). The more concise your code, the more general each routine must be and - in large programs - the slower it runs.

This trade-off is no less true of compiled code than it is of the laboriously hand-crafted variety. SUPERCHARGE recognises this, so it gives you the option of 'manual' control over the 'style' of the code it generates.

If you ignore the control the compiler uses its own 'autopilot', so this is an option you can safely leave alone if you wish. If you're really keen you can improve the performance of compiled programs this way, but the results are only marginal and you should expect an increase in the size of the compiled task as a result.

The 'style' control is intended for fine-tuning - its effects are not great and can often be achieved more economically by re-coding the BASIC carefully or by applying the other advice in this chapter. But, when you need to squeeze that last ounce of performance out of SUPERCHARGE, 'in-line code' may do the trick.

Most of the time the compiler produces a compact, fast code called 'threaded code' - this is a code which contains a mixture of data and routine addresses, rather than data and machine-code routines. The code is executed by jumping to each address in turn. In in-line code, by contrast, all the routines are written out in full and executed in a continuous stream.

SUPERCHARGE uses an extremely fast linkage for threaded code, and the complexity of the routines is carefully judged so that the 'overhead' of jumping from one to the next is extremely small.

The advantage of threaded code is that it is very concise - even if a given operation has to be performed several times, it only need appear in the file once. BASIC programs contain many repeated operations, so threaded code can give a dramatic reduction in task size.

Indeed, the original SuperBASIC version of SUPERCHARGE would only just fit into a standard 128K computer, while the compiled version leaves space for a substantial BASIC program as well.

The second part of the compiler (the code-generator) assembles 68008 machine code for each operation as required by the first part (the parser). It only adds a routine if it is definitely needed, so short programs can be compiled into quite small tasks - as the hacks put it, there is no 'library overhead'. A few routines to set up the task, handle errors, open windows and so forth appear in all SUPERCHARGED programs.

You can force the code-generator to use in-line, rather than threaded code, by putting a special REMARK statement around the lines you want generated as in-line code. Put:

```
REMARK +
```

before the lines you want extra-fast, and:

```
REMARK -
```

after them. The REMARK does not have to be at the start of the line, but it obviously must be the last statement (since nothing after a REM is executed). It is a good idea to put these on lines of their own, however, since this makes them easy to find later. It doesn't matter whether you enter a space between REMARK and the '+' or '-' sign.

Tasks which use in-line code are larger and marginally faster than their threaded counterparts. The option is only really useful on already fast integer code - it has next to no effect on complex graphical or mathematical programs since the time spent between one such threaded operation and another is relatively tiny (a fraction of one per cent).

If you try to compile the whole of a large program into in-line code you may find that the resultant task is too large for the compiler to process. SUPERCHARGE imposes a 64K limit on the code of compiled programs, although the amount of space used for data is limited only by the available memory. In threaded code this represents an enormous program - perhaps 5,000 lines of SuperBASIC - but in-line code is generated at a prodigious rate and may well exceed the limit after only a few hundred lines. This confirms that the 'style' control is a fine-tuner rather than a a brute tool to boost compiler performance.

Memory games

A better weapon for those determined to squeeze the ultimate performance out of their QL is an expansion memory board. The processor inside the QL is slowed by the fact that it shares memory with the video display electronics - if the video chip is busy, the processor has to wait.

External memory can safely ignore the video chip, so that the processor can access it more readily. Programs run measurably faster in external memory. The QL always puts tasks and BASIC into add-on memory unless that is full.

In practice SuperBASIC is not slowed much by the internal memory, since all of the interpreter code is in ROM, which can be accessed fast. Compiled programs are slowed a little more, but they may reap extra benefit in relative terms, since the interpreter tends to use slow memory for string handling, even on expanded systems. The exact effect depends upon the memory board used, although all the ones we have tested work faster than the built-in memory.

Some comments on style

The SuperBASIC interpreter has an excellent range of 'structured' control constructs, but it executes these rather slowly. In particular, loops, procedure and function calls have an elaborate set-up sequence and use line-numbers as reference points rather than machine addresses, so they become steadily slower as programs get longer. A 'long' reference can be 100 times slower than one over a few lines, in a large SuperBASIC program.

This property of the interpreter means that routines at the start of a program are found more quickly than routines later on - it also means that long programs run disproportionately slowly. SUPERCHARGE generates very fast code for loops and jumps - the compiler may accelerate EXIT, NEXT, REPEAT and GO TO by a factor of over 1,000!

Consequently, some of the tricks needed to obtain fast execution in interpreted programs are irrelevant to the compiler. You should not be afraid to make extensive use of structured loops, procedure and function calls in your programs since they are compiled into fast code. There is no advantage to be gained by using archaic statements such as GO TOs and GO SUBs in preference to structured commands.

BENCHMARK TIMINGS

The table shows the results obtained when standard 'benchmark' test programs were run on the QL, with and without SUPERCHARGE. The programs are listed on the next page.

The leftmost column contains the timings obtained when Personal Computer World magazine tested the QL SuperBASIC interpreter. The other columns contain the timings for SUPERCHARGE, using floating-point and integer arithmetic. In each case the 'speed-up' ratio is shown. All timings are in seconds.

BENCHMARK TEST NO.	S/BASIC TIMING	S/CHARGE -FLOATING-POINT-	RATIO	S/CHARGE ----INTEGER----	RATIO
1	2.1	0.25	8.4x	0.065	32x
2	6.4	0.29	22.0x	0.125	51x
3	10.7	1.24	8.6x	0.320	33x
4	10.3	0.94	11.0x	0.290	36x
5	13.2	1.04	12.7x	0.340	39x
6	26.1	2.56	10.2x	0.665	39x
7	61.8	4.13	15.0x	1.060	58x
8	25.8	8.64	3.0x	N / A	

(1) The SuperBASIC timings appeared in the June 1984 issue of Personal Computer World, and are reproduced with permission. They also appeared in 'QL USER - The Complete Dossier' in July 1984. There is some variation in speed between individual QLs and different versions of the hardware - these are discussed in the July and August 1985 issues of QUANTA, the Independent QL User Group newsletter.

(2) Timings were averaged over 10-50 runs of the standard programs, running as independent QL tasks. The tests were conducted with version AH, JS and MG ROMS - no significant speed difference was found. The build-number of the testing computer was D06, with a Simplex Data expansion RAM card fitted. Other hardware configurations may give slightly different results.

(3) There is no integer timing for Benchmark 8. This benchmark tests intrinsic mathematical functions (raising to a power, sines and logarithms) which do not have integer equivalents.

BENCHMARK PROGRAM LISTINGS:

```
100 REMARK Benchmark 1
200 PRINT "S"
300 FOR K=1 TO 1000
400 NEXT K
500 PRINT "E"
600 STOP
```

```
100 REMARK Benchmark 2
200 PRINT "S"
300 K=0
400 K=K+1
500 IF K<1000 THEN GO TO 400
600 PRINT "E"
700 STOP
```

```
100 REMARK Benchmark 3
200 PRINT "S"
300 K=0
400 K=K+1
500 A=K/K*K-K+K
600 IF K<1000 THEN GO TO 400
700 PRINT "E"
800 STOP
```

```
100 REMARK Benchmark 4
200 PRINT "S"
300 K=0
400 K=K+1
500 A=K/2*3-4+5
600 IF K<1000 THEN GO TO 400
700 PRINT "E"
800 STOP
```

```
100 REMARK Benchmark 5
200 PRINT "S"
300 K=0
400 K=K+1
500 A=K/2*3-4+5
510 GO SUB 900
600 IF K<1000 THEN GO TO 400
700 PRINT "E"
800 STOP
900 RETURN
```

```
100 REMARK Benchmark 6
200 PRINT "S"
300 K=0
400 K=K+1
500 A=K/2*3-4+5
510 GO SUB 900
520 FOR L=1 TO 5
540 END FOR L
600 IF K<1000 THEN GO TO 400
700 PRINT "E"
800 STOP
900 RETURN
```

```
100 REMARK Benchmark 7
200 PRINT "S"
300 K=0
350 DIM M(5)
400 K=K+1
500 A=K/2*3-4+5
510 GO SUB 900
520 FOR L=1 TO 5
530   M(L)=A
540 END FOR L
600 IF K<1000 THEN GO TO 400
700 PRINT "E"
800 STOP
900 RETURN
```

```
100 REMARK Benchmark 8
200 PRINT "S"
300 K=0
400 K=K+1
500 A=K 2
600 B=LOG(K)
700 C=SIN(K)
800 IF K<1000 THEN GO TO 400
900 PRINT "E"
```

Notes on the Benchmarks (continued):

(4) The speed of compiled FOR loops (Benchmarks 1, 6 and 7) is a little misleading, as SuperBASIC FOR loops are allowed to be considerably more sophisticated than those in 'standard' BASIC. Substantially faster floating-point results are obtained if FOR loops are re-coded using explicit ranges, such as:

```
FOR I=1,2,3,4,5
```

in place of the implicit range used in the test programs:

```
FOR I=1 TO 5
```

(6) The benchmark programs are very short, so they do not show the maximum possible speed-up. The larger your program is, the more SUPERCHARGE can accelerate it.

COMPILERS AND MACHINE CODE

Although the compiler gives a large speed-up factor it does not produce programs as fast as hand-written machine code. We explain this apparent flaw in this section.

Diagnostic help

Compiled programs perform extensive error-checking while they run. They ensure that array references are correct, and that parameters such as colours, channel numbers or print positions have sensible values. This testing is performed efficiently but it inevitably reduces the speed of compiled programs. The compiler also keeps track of the BASIC line-number corresponding to the code being executed.

These diagnostic checks are considered essential, since it would be very difficult to detect obscure bugs if they were not present, and there would be an increased risk that the machine would crash when errors occured. Multi-tasking crashes are especially unwelcome as they can affect programs running concurrently.

Machine-code programmers are used to machine crashes; they have a detailed knowledge of the code they are running and are willing to expend a great deal of effort tracking down bugs. Compiler users do not have this detailed knowledge, and would not want to use it anyway - they choose to save their own time, rather than that of the computer.

Compatibility

SUPERCHARGE must be compatible with the SuperBASIC interpreter, or its great advantage over other QL compilers - the ease of interactive testing - is lost. This means a number of trade-offs:

(a) The compiler can only offer the simple data-structures of SuperBASIC - 16 bit integers, strings and 44 bit floating-point variables (4 bits of every 6 byte floating-point number convey no information about the value). The other data-types available to the machine-code programmer - bits, bytes, addresses and 32 bit integers - cannot be used because of the need for compatibility with the interpreter.

Integer graphics co-ordinates would give greater speed, but SuperBASIC requires that floating-point values are used, so that SCALE can work under all circumstances. The speed of memory-addressing commands (such as PEEK and POKE) is also limited by the use of floating-point variables.

(b) The addressing conventions imposed by SuperBASIC are not as efficient as they might be. To give one simple example, programs would run more efficiently if character arrays (strings) had elements numbered from zero, rather than one.

(c) The speed of printing to the display is hamstrung by all the options which may be used - windowing, different MODEs, CSIZEs, INK, PAPER, FLASH, OVER and so forth. These parameters can be changed within a program, so SUPERCHARGE cannot make assumptions about them without running the risk of incompatibility. The compiler does take steps to ensure that groups of characters are printed more efficiently than by the interpreter.

(d) Since SuperBASIC is an extensible language, the compiler must make some attempt to support future extensions, including those which were not even mooted when SUPERCHARGE was designed and written.

We aim for the maximum compatibility that can be given without greatly compromising the efficiency of compiled code. Some people may feel that compatibility should have been greater, while others might complain at the cost of what compatibility has been provided. Like all compromises, our decision cannot please everybody.

Special cases

It is true that the compiler cannot take account of special cases in the same way that an experienced human programmer can. To some extent this reflects the limitations of digital computers and of our understanding of thought, but there is still scope for much improvement to SUPERCHARGE before these limits are approached.

A more practical limitation is the amount of memory available for the compiler's 'intelligence'. The authors could not encode their entire knowledge of programming into 128K bytes - let alone leaving space for intermediate code and a substantial BASIC program!

As generated programs become more efficient, the time needed to produce them increases. The speed of code produced by SUPERCHARGE is limited by the fact that compilations must be performed as quickly as possible.

Even if the compiler had the information and analytical power needed to generate code as good as a human programmer, it might end up working as slowly as the human. At present SUPERCHARGE is much faster than even the most prolific hacker.

Demonstrable correctness

The 'simple-minded' code produced by the compiler has one other advantage over the hand-crafted variety - it usually works first time.

In recent years there has been a great deal of academic research into the task of proving the correctness of programs; the conclusion, so far, is that this is extremely difficult (without exhaustive experimentation). The difficulty seems to increase disproportionately with the intricacy of the code being analysed.

A compiler capable of producing very efficient code for many special cases is likely to be inherently less reliable than a simpler creation. We have tried to pitch the complexity of SUPERCHARGE at a level at which it offers useful results without a great sacrifice of reliability. This is the main reason why the complex 'parser' of SUPERCHARGE was written in SuperBASIC and then compiled.

Programming 'tricks' and optimisations

It is important to weigh up the pros and cons before using 'trick' techniques in your programs - they may make the code less clear and harder to debug. With care, these three tips can increase the speed of all SuperBASIC programs:

(a) Ordered SELECT
(b) Lazy IF evaluation
(c) Array re-dimensioning

All of the techniques work by reducing the amount of effort required to obtain a specific effect.

(a) Ordered SELECT.

If the instances (or 'cases') in a multi-statement SELECT are re-ordered so that the most likely values come first, a useful speed improvement may result. This trick works because the only way such statements can be executed is to test for each case; later cases do not have to be evaluated if an earlier one matches. SUPERCHARGE can't do this for you because it can't guess which cases are the most common. If the values tested for can easily be converted into a continuous integer range, ON GOSUB is often faster than SELECT, since one 'test' handles all instances.

(b) Lazy IF evaluation.

If a test contains the AND keyword, BOTH tests are always performed, even though the result is known as soon as one has been evaluated. This must happen, in case either test contains a function-call which has 'side effects' on other variables. You can save time, if your code doesn't do this, by writing IF A THEN IF B THEN instead of IF A AND N THEN. The first test should be the one most likely to fail. You should be careful to keep END IFs matched properly.

(c) Array re-dimensioning.

The computer must do a lot of work when an array is re-dimensioned - the old memory must be 'reclaimed' and new space found. If LOCAL arrays are re-dimensioned SUPERCHARGE will not reclaim the old memory until the procedure or function is finished. Re-dimensioning is always the fastest way to clear an array, as long as you don't alter the size.

POWERFUL PROGRAMMING

In this chapter we present a few programs which give some indication of the power of SUPERCHARGE. Doubtless you will think of many more useful applications. Ultimately, the scope of SUPERCHARGE is limited only by the imagination and ingenuity of its programmers (This Means You!).

YET ANOTHER 'CLOCK'

For about a year the only proof that the QL user had of the machine's multitasking capability was the 'Clock' program listed in the User Guide. Only the luckiest users got a User Guide that matched their machine, in any case. For historical reasons, therefore, and for those who never got the listing in the 'Concepts' section of the User Guide to work, we list a 'Clock' program written for SUPERCHARGE.

```
10 REPEAT tick: AT 0,0: PRINT DATE$: PAUSE 50
```

Adjust the size of window 1, or the values after AT, to alter the position at which the clock appears. Add INK, PAPER and PAN statements (etc) to taste. Load the task with EXEC, and savour the thrill of multi-tasking as the pioneers knew it. Use REMOVE TASK to get rid of it when you get bored (this shouldn't take long - the clock will tell you exactly how long). Alternatively you can use SET_PRIORITY to reduce the priority of the task to 1 so that it doesn't waste too much processor-time. The clock can be set and adjusted with the SuperBASIC commands SDATE and ADATE.

A ONE-LINE SPOOLER

It is often useful to be able to copy files from one device to another while you get on with some other task. This simple but very useful program lets you do just that:

```
10 INPUT "Copy from "!F1$,"Copy to"!F2$:COPY F1$ TO F2$
```

Compile this program in the usual way, then EXEC it. Select the device (and filename, if any) from which information is to be taken, then specify the destination device or file. The program copies data between the source and destination while you get on with doing something else. Set the task priority as required. Use DATASPACE (Chapter 4) to set the buffer size: SUPERCHARGE uses all 'free' memory within a task as a buffer for COPY. The report 'IN USE' appears if you try to access a file or device while it is busy.

HOW SUPERCHARGE WAS BORN

Work on SUPERCHARGE began at the end of June 1984, as the first 'working' QLs became available. At first the compiler was intended for Southampton software house Quicksilva, but they lost interest in the QL market when they were bought out by the Argus Press magazine group. The project was eagerly taken over by Freddy Vachha's upcoming QL software house, Digital Precision.

SuperBASIC was potentially a very powerful, expressive programming language, but it suffered because of rushed implementation, dreadful documentation and some last-minute changes intended to make it more compatible with its forerunner, ZX Spectrum BASIC. Several nice features were discarded en route, and much of the new code was untested.

The aim of the SuperBASIC compiler was to correct the weaknesses of the interpreter, without affecting the power of the language or its greatest asset - its expandability. Supercharge (as it was later dubbed) should compile the vast majority of existing SuperBASIC programs without alteration. Incompatible code should be clearly indicated so that it could be corrected with the minimum of fuss.

Supercharge should run on all versions of the QL, and a program compiled on one system should run - without alteration - on other versions. The code should be efficient in its use of memory, fast, and capable of multi-tasking. These requirements, and the complexity of the language, meant that SUPERCHARGE had to be much more sophisticated than other micro BASIC compilers.

When design work began few technical details of SuperBASIC were available. It was far from clear how SuperBASIC was meant to work. The QL User Guide was incomplete, and programming tools were rare - the only 68008 assembler available was a simple and slothful SuperBASIC program.

Work on the 'parser' of the compiler was started nonetheless. The program was written in SuperBASIC, for ease of testing and experimentation - the slow speed of the interpreter did not matter since the aim, from the start, was that the compiler should eventually compile itself. The syntax of SuperBASIC was distilled into a 'grammar', so that a small set of routines could be used to analyse any valid SuperBASIC program. An intricate expression evaluator was developed to keep track of the QL's powerful but potentially inefficient 'coercion' features, and a library of diagnostic routines was developed to simplify testing.

In April 1985 SUPERCHARGE compiled its first program - a simple recursive factorial calculator. Rather than generate machine-code directly, the original compiler produced macros which were assembled later. This temporary arrangement made it easy to check the compiler's output by eye, and provided a useful 'bridge' between readable SuperBASIC and incomprehensible compiled code.

The compiler was then adapted to generate a concise, binary intermediate code which could be passed to a purpose-built native code generator. Gerry Jackson was given the job of writing this program. He also devised the structure of the 'template library' which holds 68008 code routines before they are compiled - in the conventional sense - into an executable program. A wide range of SuperBASIC programs was flung at the compiler, in order to check that it worked correctly and gave sensible and helpful error messages when asked to do the impossible.

The final stage of the development of SUPERCHARGE involved the compilation of the compiler itself. Modifications were made to ensure the greatest practicable degree of support for 'extension' procedures and functions. DATASPACE was written and the SUPERCHARGE utility commands were developed and tested.

THE AUTHORS

SIMON N GOODWIN BSc worked for Racal for three years, developing Business and Computer Aided Design systems, before he went freelance. He is the author of a dozen microcomputer packages, ranging from compilers to utilities and games. He has lectured on compiler and interpreter design. Simon Goodwin is a prolific communicator, author of over 100 articles for computing and electronics magazines since 1979, and an experienced broadcaster on TV and Radio. At present he is Personal Computer World magazine's 'agony aunt', employed to solve the (computer-related!) problems of readers through the monthly 'Computer Answers' column.

GERRY JACKSON MSc has spent 15 years with a major British computer company, where he become one of the few Britons to design and make both an original microprocessor chip and a 2,000 integrated-circuit minicomputer. He is also a Computing Course Tutor with the Open University. In the field of software Gerry Jackson has written processor simulators and implementations of the Forth programming language for the Dragon and QL computers.

CREDITS

We wish to acknowledge the skill of Jan Jones and Tony Tebby, designers of the QL SuperBASIC interpreter. Tony Tebby's QDOS notes and Jan Jones' book, 'QL SuperBASIC - the definitive handbook' were the source of much useful information. The handbook is recommended to all serious SuperBASIC programmers - it is published by McGraw Hill. Tony's notes are published by almost everyone - we favour the 'QL Advanced User Guide' (Adder) when disassembling and 'The Sinclair QDOS Companion' (Sunshine) when assembling.

SUPERCHARGE was designed and written by Simon N Goodwin, between July 1984 and September 1985; additional code and memos were contributed by Gerry Jackson. Mission control was by Freddy Vachha. Thanks are also due to Julie Butler, Mary Cassidy, Chas Dillon, Mike Gottlieb, Andy Pennell, Tony Tebby and A.H Rom for help and/or encouragement.

SUPERCHARGE was developed on standard QL systems, fitted with the CST floppy disk system, supplied by Computamate of Scotia Road, Burslem, Stoke on Trent; Tel. 0782 811711. Simplex Data expansion memory was also used. Both products are highly recommended on grounds of quality, efficiency and reliability.

IMPORTANT NOTICE

While all reasonable steps have been taken to ensure that SUPERCHARGE is bug-free and accurately documented, the purchaser should be aware that it is impossible to test any compiler under all possible circumstances. It is the responsibility of the purchaser to ensure that SUPERCHARGE is fit for any specific purpose. No responsibility or liability is accepted for loss of business caused, or alleged to be caused, by its use.

Gerry Jackson's SUPERFORTH

FORTH is a structured, intermediate-level programming language, which combines the the speed of machine-code with high-level control constructs and interactive testing facilities.

SUPERFORTH is a complete, ultra-fast implementation of the Forth '83 standard. In addition to the standard commands SUPERFORTH includes a plethora of extra features:

* Support for all the QL's features including sound and graphics. Input and Output may be re-directed at will.

* Full 32 bit integer arithmetic, allowing lightning-quick calculations to nine digits of precision.

* All floating-point arithmetic operations are supported, including Logarithmic and Trigonometric functions.

* Multi-tasking (demo supplied) with full job control for SUPERFORTH and machine-code programs. SUPERFORTH itself runs as a task, so other programs may run concurrently.

* A powerful screen editor allows Forth blocks to be edited and saved in standard format on disk or microdrive. Named files may be used instead of Forth blocks - these files may be prepared with any text editor, including Quill.

* The comprehensive 80-page manual supplied with SUPERFORTH serves as a reference guide language and a tutorial for those new to the language.

PLUS... PLUS... PLUS... PLUS... PLUS... PLUS... PLUS...

The SUPERFORTH package also includes an extremely powerful implementation of the classic boardgame REVERSI. This superb program is written entirely in SUPERFORTH, and the well-written source-code is included for you to study.

REVERSI offers nine levels of play, with near-instantaneous response on levels 1 and 2. It beat the Spectrum champion, MOI Othello, 10-0 in a supervised match - in fact we have yet to find any program capable of beating REVERSI on equal time - or any human capable of beating it at its top skill level. Many options enable you to exchange sides, retract moves, set up positions and display evaluations at will.

Digital Precision SUPERFORTH + REVERSI: both for £29.95.

GLOSSARY

ABORTED Stopped because it was completely impossible to continue.

ADD-ON COMMANDS Resident procedures & functions in machine code linked in to SuperBASIC to make them invokable just like ordinary SuperBASIC commands (available when the QL is switched on).

ARRAY An ordered collection of a number of elements of the same type, the position of each element being uniquely defined by an ordered set of discrete index values. The number of index values required to specify the position of an element is equal to the number of dimensions of the array. One dimensional arrays are often called vectors or strings: two dimensional arrays are called matrices or tables.

ASSEMBLY LANGUAGE A convenient notation for expressing machine code in a form (using alphanumeric mnemonics) easily comprehensible by humans.

ASSIGNMENT A statement within most languages (including SuperBASIC) where a new value is given to a variable.

BASIC An acronym for Beginner's All purpose Symbolic Instruction Code. BASIC is a mid-1960s programming language developed from FORTRAN. It is the most common microcomputer language. Despite being subject to an ANSI standard, there are many variants of it: SuperBASIC is almost certainly the most powerful & flexible of them.

BENCHMARK A test or series of tests designed to measure the performance of a system of software & hardware, for comparative purposes. Typically used to test speed.

BINARY Expressed with a number base of 2 (ie; allowable digits 0,1).

BITWISE OPERATOR A logical operator that works on binary digit structure rather than on Boolean (True/False) principles.

BUGS Localised errors in a program or system. The SuperBASIC interpreter possesses them in large quantities.

CS (We've heard so many that we will leave this space blank for you to fill in.....)

CALL To transfer control to a subroutine, function or procedure, with the provision to return to the instruction following the call instruction at the end of executing the called code.

CALL BY REFERENCE To call a procedure or function with one or more parameters with the specification of the parameters such that changes to their value within the procedure or function causes corresponding changes to their value outside it.

CALL BY VALUE To call a procedure or function with one or more parameters with the specification of the parameters such that changes to their value within the procedure or function has no effect on their value outside it. This is done by making an exact copy of the parameters at the time of the call, & discarding the copy on return.

CASE Character format applicable to the alphabetic set only: can either be UPPER (ASCII 65 to 90) or lower (ASCII 97 to 122). In olden times (before computers or Eddie Shah) type was assembled using two cases (as in boxes) - the one on top used for capitals, & the lower one for little letters. Hence Case. Interesting tidbit.

CENTRAL PROCESSING UNIT The principal operating part of the computer, comprising an arithmetic/logic unit & a timing unit to control communications of all sorts. On the QL the central processing unit(CPU) is the Motorola M68008 microprocessor.

CHANNEL A route for transmitting data to devices.

CODE GENERATOR The part of the compiler that operates last of all, to generate native code for output to a device.

COMPATIBLE In the case of software: the ability of a piece of software to accurately reproduce the behaviour of its predecessor (with particular reference to accepting the same input formats). SUPERCHARGE is compatible with SuperBASIC.

COMPILER A program to translate a symbolic high level language into native code automatically. In the case of SUPERCHARGE, the high level language is Sinclair SuperBASIC & the native code is M68008 machine code.

COMPILE-TIME As opposed to run-time, the actual time at which a high level language is translated into native code. SUPERCHARGE shifts many operations which under the interpreter would be performed in run-time to compile-time to be done once & for all.

CONCURRENCE The progress of two or more tasks in parallel on a computer.

CONDITIONAL A logical statement comprising a condition which, if satisfied, will result in specific action (in the case of a conditional branch, a transfer of program control) being performed.

CONTIGUOUS Immediately adjacent.

CORRUPT When applied to programs or data (as opposed to suppliers of computer hardware & software): in some way changed (possibly to the point of unusability or irretrievability). This is to the dismay of the programmer/operator, who has forgotten to make a backup. Corruption can be caused by a failure of the machine or of the storage medium, or by an errant program. SUPERCHARGE owners who actually use microcartridges for file storage will be all too familiar with the symptoms.

CURSOR A symbol on the display screen indicating the active position & usually signalling the computer's readiness to accept input.

DEBUGGED PROGRAM A program in a state in which Sinclair could reasonably be expected to discontinue it.

DEVICE INDEPENDENCE The ability within a programming language to use exactly identical commands to control input & output from vastly differing devices, the device name being a sufficient identification for the operating system to do the needful. SuperBASIC & SUPERCHARGE are both device independent.

DIGITAL PRECISION Us.

DIMENSIONING Operation performed in defining an array to specify the number of dimensions it is to have & also the maximum value of the index corresponding to each dimension.

DISCRETE Not varying & not capable of varying in a continuous way, instead only in steps.

ELEMENT A member of an n-dimensional array, referenced by an ordered set of n indices.

EPROM An acronym for Erasable Programmable ROM - a memory chip capable of being repeatedly reprogrammed by the user by controlled exposure to UV radiation.

EXPRESSION A collection of numbers, strings, variables & parentheses connected by operators & capable of being evaluated.

EXTENSIONS Another word for add-On commands.

EXTERNAL REFERENCE A reference within a program to another program outside it. Until exact data on the location etc of the second program is available to the first one, the reference is said to be unresolved.

FLOATING POINT A representation of real numbers enabling numbers within a wide range of magnitudes to be concisely expressed & compactly stored.

FUNCTION A subroutine which returns a value & which can be invoked from within an expression.

GLOSSARY You are looking at one now!

HARDWARE Unlike software, those portions of a computer system that are capable of being kicked. For example, a microcartridge tape is capable of being kicked (we suspect it often is!) & hence is hardware, while the program on it is software.

HEX Expressed with a number base of 16 (ie; allowable digits 0-9, A-F).

HIGH LEVEL LANGUAGE A programming language in which both control & data structures reflect the needs of the programmer, rather than the specific design & hardware of the computer. In the QL, SuperBASIC is the high level language - the SUPERCHARGE compiler translates it into native code.

IDENTIFIER Posh name for Name.

INLINE CODE As opposed to threaded code, code that does not involve calls to a series of subroutines.

INTEGER A whole number in the range -32768 to +32767 (inclusive) as far as SuperBASIC is concerned.

INTERACTIVE A mode of working in which there is an immediate or near-immediate response to commands once they are input. The SuperBASIC interpreter allows interactive debugging - it is principally these commands which are not supported by SUPERCHARGE. It seems very reasonable to expect users to have done most or all of the program debugging using the interpreter before invoking the compiler.

INTERMEDIATE CODE A very concise definition of a program produced as an interim information store during actual compilation.

INTERPRETER A program to analyse a unit of code (in SuperBASIC, a statement) in a high level language & then to carry out the actions specified within that unit, rather than produce a machine code translation for subsequent execution. In the case of the QL, the SuperBASIC interpreter is supplied free with the machine. By & large, the structure & syntax of SuperBASIC do not make interpreting it essential or even preferable to compiling it - the decision to supply an interpreter rather than a compiler was that it is easier to write an interpreter!

INTERPRETER DATA STRUCTURE The information maintained by an interpreter so that it can keep track of the modification & the execution of a program.

JOB A collection of tasks & their related data. This term is sometimes used as a synonym for task, which is confusing.

KLUDGE Affectionate (?!) term for a Sinclair EPROM.

LINE An ordered sequence of one or more statements in a BASIC program, referenced collectively by a line number.

LINKING The combination of one or more machine code programs, resolving all external references between them. Linking is used either to add on prewritten subroutines or to build up a complex program step by step from simpler ones. SUPERCHARGE does not support linking. This is because SUPERCHARGE already supports add-on commands, & because SuperBASIC does not give any specification whatsoever for linking (remember, SUPERCHARGE compiles SuperBASIC & not some language we dreamed up on the spur of the moment!!). External programs can always be CALLed or EXECed instead.

LOOP A sequence of instructions within a program that is repeated until a prescribed condition is satisfied.

LOW LEVEL LANGUAGE A programming language (typically assembly language or machine code) in which control & data structures are direct reflections of central processing unit architecture.

MACHINE CODE The operation code of a machine, specific to its own central processing unit. In the QL, the machine code is Motorola M68008 code which is identical to M68000 except in the number of clock cycles some instructions take to execute.

MATHS STACK The stack where intermediate calculation data is stored.

MICROPROCESSOR A semiconductor chip with a fixed operation code set. It is characterized by its speed, internal word length, external word length & architecture. The M68008 inside the QL has a 7.5MHz clock, internal word length of 16/32 bits, an 8 bit external data bus & 32 bit architecture.

MOTOROLA The US manufacturer of the 680XX family of microprocessors. Motorola is a communications corporation - much of their turnover comes from mobile wireless communications, car radios & the like. They are pretty big.

MULTITASKING The concurrent execution of a number of distinct tasks. Despite claims to the contrary, SuperBASIC cannot in any sense multitask. SUPERCHARGE, however, produces code that can multitask with any combination of other SUPERCHARGE tasks / machine code programs / code generated by other multitasking high level QL language compilers / the interpreter task & any other task you can think of.

NAME A sequence of alphanumeric characters & underscores, the first character of which must be non-numeric.

NATIVE CODE Program code in a format suitable for direct execution by a central processing unit: ie; code containing no symbolic references & no unresolved external references. A synonym is absolute code.

NESTING The embedding of a syntactic structure within an instance of another syntactic structure. The term is often used in connection with loops.

OBVIOUSLY We're not quite sure about it, & can't prove it anyway.

OPERATING SYSTEM The collection of software that controls both system resources & the processes utilising these resources on a computer. On the QL the operating system is QDOS.

OPERATION CODE The collection of instructions for specifying the operation of a particular central processing unit, defining the repertoire of operations it can perform.

OPERATOR An entity, usually represented by a symbol, that can be applied to one or more operands (constants, variables, functions or expressions) so as to yield a result. A unary operator is one that is applied to just one operand (eg; NOT): a binary operator is an operator applied to two operands (eg; +).

OBJECT CODE The output of a compiler - in the case of SUPERCHARGE, directly executable M68000 machine code.

OPTIMISATION — The production of object code that in some way makes best use of the resources of the computer: usually minimisation of execution time is the objective (rather than reduction of object code size, which is less relevant when memory is abundant). Optimisation can either be global (sequences are reordered, invariant operations are moved outside loops, loops are merged, etc) or local (code is adapted to exploit machine architecture & quirks, redundant operations are excluded, etc). SUPERCHARGE performs a great deal of local optimisation & a certain amount of global optimisation. Time & Space optimisation can be selected on a statement to statement basis using the REM+ & REM- facility on SUPERCHARGE.

PARAMETER — Information passed to a procedure, subroutine or function.

PARENTHESES — Posh name for brackets.

PARSING — The process of deciding whether a sequence of input symbols is a syntactically & semantically valid sentence in a language. The parser i SUPERCHARGE has as its input a SuperBASIC program & as its output intermediate code.

PASS — A stage in the process of compilation, involving a complete scan through the program or the corresponding intermediate code. Compilation typically involves a number of passes.

PRECEDENCE — The conventional order or sequence in which multiple operators within an expression are to be evaluated. Both SuperBASIC & SUPERCHARGE follow the B(Bracket)U(Unary Op) M/D(Multiply & Divide)A/S(Add & Subtract) hierarchy.

PRIORITY — A numerical value used to dictate the proportion of system resources (typically but not always central processing unit time) to be devoted to a specific task.

PROCEDURE — A section of a program identified by a name, capable of being called from anywhere in the program (including from within itself) & able to carry out operations on data specified externally as parameters.

PROCESS — Another word for task.

PROGRAM — A set of statements that can be submitted to a computer & used to direct its behaviour. SuperBASIC programs on the QL are procedural programs: ie; programs where a very precise definition of the sequence of steps to be followed by the computer in order to obtain the desired results has been explicitly provided (rather than programs which only specify the constraints & which leave the computer to decide on what steps to take & in what sequence to take them).

QDOS — We've been reliably (ie; by Sinclair) informed that this is the QL's operating system. We've been more reliably informed that its more like a semi-random collection of machine code routines. Whichever it is, there is unfortunately no way to get at it via SuperBASIC.

QL — What the *?!*£$?'+! are you doing here if you need to look this up!

RAM — An acronym for Random Access Memory, which is a silly name because ROM is random access too. RAM is characterised by being read/write memory.

REGISTER — An entity used to store information within a computer system (usually within the central processing unit) for fastest possible access times.

RELOCATABLE — As applied to a program, one that can work anywhere in memory since it either contains no absolute memory addresses or contains memory addresses expressed relative to an origin somewhere in the program. SUPERCHARGE produces fully relocatable code.

REMARK — A comment in a high level language inserted to make the program less incomprehensible. SUPERCHARGE ignores remarks except for REM+ & REM- (for optimization).

ROM — An acronym for Read Only Memory - you need powerful electromagnetic fields, X-Rays, Quanta of UV light, nuclear bombs or real physical violence to change the contents of ROMs.

RUN-TIME — The time at which a program is executed, as opposed to the time it was written or compiled.

SEMANTICS — The part of a language definition concerned with the specification of the actual meaning or effect of a text written in compliance with the language's syntax. Semantics involve the analysis of symbols taken in context & not individually.

SINCLAIR RESEARCH The people who made it all possible - & then very nearly made it impossible. What do they get up to in the daytime?! Despite all their help, SUPERCHARGE has finally seen the light of day.....

SITE LICENCE That which you are required by Digital Precision to have if you want to use SUPERCHARGE to produce programs (or parts of programs) or to use SUPERCHARGED programs, AND if you wish to sell or distribute them commercially. The Licence is for software houses or other entities (hereon collectively referred to as software houses) that sell or otherwise distribute programs: a programmer does NOT need a Site Licence (unless he/she starts commercially selling or distributing SUPERCHARGED programs, in which case he/she becomes a 'software house'). The Site Licence is a legal document obtainable from Digital Precision by mail for just £250 (incl VAT): a VAT invoice will be provided. The software house need only purchase ONE Site Licence from Digital Precision irrespective of the number of DIFFERENT SUPERCHARGED programs being sold. The Site Licence is of indefinite duration & will not be revoked. It relieves the software house of ALL future requirements to pay Digital Precision royalties on the proceeds of sale by it of all SUPERCHARGED programs. Failure by a software house engaged in selling or other distribution of SUPERCHARGED programs to obtain the required Site Licence in advance will render it liable to action under the civil & criminal law. Digital Precision shall in such cases seek to recover full legal costs, back-royalties computed on estimated sales as well as punitive damages from the software house. Further, Digital Precision may seek a Court injunction prohibiting the software house from selling or distributing SUPERCHARGED programs until a Site Licence is obtained. Programmers are notified that SUPERCHARGED programs will always contain certain codes & constructs to make them identifiable as having been generated by SUPERCHARGE. If a programmer sells or otherwise transfers rights in a SUPERCHARGED program to a software house, the programmer is strongly advised to inform them that SUPERCHARGE has been used. It is quite likely that the software house will already have a Site Licence for SUPERCHARGE. If they do not, the programmer is strongly advised to inform them about this condition & to instruct them to obtain a Site Licence immediately. If the programmer does not do this, & Digital Precision sues the software house, the software house is extremely likely to seek indemnity from the programmer or at the least to withhold royalties payable to the programmer.

continued...

To sum matters up - to encourage the use of SUPERCHARGE, we've kept the Site Licence fee very low, & made it a once-only payment (unlimited duration, no further royalties). You only need one Site Licence no matter how many copies of how many different SUPERCHARGED programs you will be marketing. We are being very reasonable. Please do not force us to become unreasonable!

SLICE A part of an array obtained by restricting the range of values of one or more of the indices of the array to a set of contiguous discrete values which is a subset of the set of original values of the index or indices defined when dimensioning the array. One or more of the indices may be eliminated altogether.

SOFTWARE Those portions of a computer system which cannot be kicked - the intangible bits like programs.

SOURCE CODE The form of a program input to a compiler for translation into object code. In SUPERCHARGE, the source code is a debugged SuperBASIC program.

STACK A linear LIFO (Last-In,First-Out) data list where all accesses, removals & insertions are made at one end (the top). There are at any time a number of stacks operating within the QL's memory.

STATEMENT The sentence-like unit which is the building block for a high level language.

STRING An ordered sequence of characters.

STRUCTURE When applied to a program: its overall form, with emphasis both on its component parts & on the interrelationships between these parts. In a well structured program, the breakdown into components has been carried out in a consistent & recognised way, & the interfaces between components are easy to follow & well-defined. SuperBASIC is one of the BASICs which allows fully structured programming. SUPERCHARGE supports ALL the structured program constructs of SuperBASIC.

SUBROUTINE A section of code to which control is transferred by a call & on whose completion control reverts to the instruction after the call. In SuperBASIC, subroutines with names are referred to as procedures. Subroutines often save space & make a program more readable.

SUBSCRIPT A means of referencing an element in an array, by appending to the name of the array the indices which uniquely identify the element.

SUBSTRING A string which is contained exactly within a main string: ie; the main string contains somewhere within it the same characters as the substring in the same sequence & without any intervening characters.

SUPERBASIC The programming language supplied free as the host environment on the QL. An advanced, enhanced & very well-structured variant of BASIC.

SUPERCHARGE The State of the Art Digital Precision SuperBASIC Compiler of which you are now a proud owner.

SUPERCHARGED As applied to a program, one that has been produced using SUPERCHARGE.

SYNTAX The set of laws governing & defining the permitted sequences of characters in a programming language. Unlike semantics, syntax is not concerned with the meaning of the constructs but only with their form.

SYSTEM That which is obtained by combining software & hardware.

SYSTEM VARIABLE A value held by a system on a semi-permanent basis to store statuses, defaults & other data on the utilisation of system resources.

TAG An identifier used to discriminate between variants of the same type.

TASK A unit of system activity. A task is composed of a set of program instructions, a workspace area & a descriptor defining current statuses of any system resources allocated to it. SUPERCHARGE produces tasks which may be run concurrently.

TASK IDENTIFIER A pair of integers, the first an index number of the task & the second a tag. Together they uniquely identify the task.

THREADED CODE A code containing a sequence of entry points for routines. An unconditional branch is made to a routine whose address is indicated by a word in the code: on completion the routine is terminated by another unconditional branch to the new entry point indicated by the next code word.

TOKENISATION The conversion of a program in text form into a less verbose binary form consisting of units called tokens. Strangely enough, tokenisation on the QL increases rather than decreases verbosity - for example, the value 0 is tokenised into SIX bytes!!

TRAP	A system state triggered by a signal to the microprocessor that the current sequence of instructions is to be abandoned & a sequence appropriate to the interruption commenced in its place (such sequence often lying within the operating system). Traps are used for all sorts of error handling & for invoking just about everything.
VARIABLE	A value stored somewhere within the computer & capable of being changed. Variables are referenced by their name.
WORKSPACE	A reserved area for a task or other program to read from, write to, manipulate & generally use.

****
*
* THE GLOSSARY WAS WRITTEN BY FREDDY VACHHA BSc *
*
****

Skip this chapter unless you have a mathematical bent AND need fast computations of advanced math functions.

The advanced mathematical functions (trigonometric, inverse trigonometric, logarithmic & exponential) are computed with great accuracy (far more than is almost ever needed) and consequently very low speed by the interpreter. SUPERCHARGE supports these functions to the same level of accuracy (compatibility is the object):it is hence also slow, albeit faster than the interpreter. In this chapter we discuss faster methods of dealing with these functions - speed is very much an issue whenever the drawing of graphics is concerned: it is fair to point out that when large figures are to be drawn, the time taken to actually plot the pixels - as opposed to the time taken to calculate WHERE they should be plotted - may be substantial. If this is the case we would say that the processor is I/O bound - in plain English, SUPERCHARGE will have less effect in such cases.

The recommended way of dealing with such functions is with a look-up table. An array of 90 elements, for example, could be used to hold the values of the sines of integer angles from 1 to 90 degrees. The array is filled only once, at the beginning of the program (either from a file, a set of DATA statements or as a result of computation). Linear interpolation will then give reasonably accurate results for non-integer angles: eg; to find SIN(22.71DEG) we use SIN(22.71DEG) = SIN(22DEG) + .71 * (SIN(23DEG)-SIN(22DEG)) - this gives a result which is 99.997% accurate (ie; good to four decimal places - more than enough).

However, you may not wish to use look-up tables. Space may be short, or the range of values to be covered may be too large, or the rapidity of variation in some regions (ie; as with TAN near PI/2) may make interpolation impossible. In such cases a quite acceptable approximation is obtained by summing the first few terms of a suitable convergent series for the function.

Here are the most important of these series - in most cases, summing them to 3 terms will give at least 2 digits of accuracy, which is good enough for many graphical purposes. If you want more accuracy, use more terms. Note that all angles are represented in radians. Always convert angles (using relationships like SIN (PI-X)=SIN(X)) to suitable ranges of values (eg; between -PI/2 & +PI/2 for SIN).

```
SIN(X) = X - X^3/3! + X^5/5! - . . . .
COS(X) = 1 - X^2/2! + X^4/4! - . . . .
TAN(X) = X + X^3/3 + 2*X^5/15 + 17*X^7/315 + . . . .   [X^2<PI/4]
ARCSIN(X) = X + 1/2*X^3/3 + 1/2*3/4*X^5/5 + . . . .        [X^2<1]
ARCCOS(X) = PI/2 - ARCSIN(X)          < exactly >          [X^2<1]
ARCTAN(X) = X - X^3/3 + X^5/5 - . . . .                    [X^2<1]
          = PI/2 - 1/X + 1/3/X^3 - . . . .                 [X^2>1]
SINH(X) = X + X^3/3! + X^5/5! + . . . .
COSH(X) = 1 + X^2/2! + X^4/4! + . . . .
TANH(X) = SINH(X)/COSH(X)             < exactly >
ARCSINH(X) = LN ( X + SQRT(X^2+1) )   < exactly >
ARCCOSH(X) = LN ( X + SQRT(X^2-1) )   < exactly >          [X>=1]
```

```
ARCTANH(X) = 1/2*LN((1+X)/(1-X))     < exactly >          [X^2<1]
LN(1+X) = X - X^2/2 + X^3/3 - . . . .                     [X^2<1]
LN(1-X) = - X - X^2/2 - X^3/3 - . . . .                   [X^2<1]
EXP(X) = 1 + X + X^2/2! + X^3/3! + . . . .
A^X = e^(X*LN(A))                    < exactly >
```

When defining such functions make sure that the function name isn't a SuperBASIC reserved word (examples of which are LN, EXP, SIN etc)! Also, make the maximum use of integers in the function definitions (or you may end up with the 'speeded-up' function working slower than the SuperBASIC one!).

An example of a very fast function to calculate SIN(X) is given below. It assumes values for X are in radians & are in the range 0 to 2*PI (you could put checks on this into the definition). If you study it carefully you will be able to write similar definitions for all the advanced functions listed earlier in this chapter.

```
DEFine FuNction SINE_FAST(X)
LOCal Y%
  IF X>1.5707963 THEN
    IF X<3.1415927 THEN
      X=3.1415927-X
    END IF
  END IF
  IF X>=3.1415927 THEN
    IF X<4.712389 THEN
      X=9.424778-X
    END IF
  END IF
  IF X>=4.712389 THEN
    X=X-6.2831853
  END IF
  Y%=100*X
  Y%=Y%-Y%*Y%/125*Y%/480+Y%*Y%/125*Y%/160*Y%/160*Y%/3750
  RETURN .01*Y%
END DEFine SINE_FAST
```

The computation of Y% looks complicated because of the several restrictions stated on page 49.
i> Clearly Y% will always lie in the range from -158 to +157 (X has been made to lie between -PI/2 & +PI/2, and Y%=100*X). We must make sure that no intermediate answer can ever lie outside the interval -32768 to +32767: hence expressions like Y%*Y%/125*Y% appear. The "worst" case is 158 * 158 / 125 * 158 = 31554 (or 31442 if integer multiplication is being carried out) which is less than 32767 in magnitude. Note that as we explain on page 49, the -32768 to +32767 range is a limit imposed not by SUPERCHARGE but by the integer size available on the microprocessor.
ii> In order to make the result as accurate as possible, we divide by as small a number as we can early on. For example, the second term has as a denominator 3!*100^2 = 60000. 125 is the smallest factor of 60000 which prevents an overflow. 480 is, of course, 60000 / 125.

iii> The third term should have a denominator of 5!*100^4 = 1.2E+10. In this case we have been able to factorise it & choose suitable factors - if no suitable factors could be found, we would have made each denominator besides the last one the smallest integer that would prevent overflow. We would then use as the last denominator the integer closest to the number required to make the product of the denominators correct. For example, if in the above case the overall denominator was 1.1E10 instead of 1.2E10, the 4 denominators to be chosen would be 121,158,158 & 3642 (1.1E10 / 121 / 158 / 158 = 3641.6). The rounding is fine - we are only after 2 digits of accuracy, and in any case the third term is always smaller than the two before and hence less significant.
iv> In the penultimate line of the listing, note we have chosen to multiply by .01 instead of dividing by 100. We are aware that there will be a tiny loss of accuracy (refer to pages 45-46) but the improvement in speed given by multiplication over division is more than adequate compensation!

```
**********************************************************************
*                                                                    *
*          CHAPTER 11 WAS WRITTEN BY FREDDY VACHHA BSc               *
*                                                                    *
**********************************************************************
```

INDEX

Page

Page

Page